Buckminster

Eclipse Buckminster

The Definitive Guide

Henrik Lindberg, Cloudsmith Inc.
Thomas Hallgren, Cloudsmith Inc.

Eclipse Buckminster: The Definitive Guide

by Henrik Lindberg and Thomas Hallgren

0.6 - include/exclude patterns explained for group and action. New Examples — building product, building legacy sites. New Troubleshooting chapter. Minor updates of reported issues. Buckminster command 'install' added in headless install instructions.

Dedication

This guide is dedicated to all software developers who have voiced their frustration with manually putting software build systems together, and to all early adopters of Buckminster that have voiced their frustration over the lack of examples and documentation when trying to construct an automated system.

Don't worry, I have my hat on, and I think this is what the documentation suggested...

Stew has his first build tool experience.

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

Software development is becoming software assembly, with components sourced from around the world and based on a wide range of implementation technologies. The Eclipse Plug-In Development Environment (PDE) does a great job of streamlining development componentized plug-ins and feature-sets when using the Eclipse IDE interactively. However, the PDE manages only those components implemented as Eclipse plug-ins, and uses a different way of building when automating builds in “headless fashion”. There is also only limited support in Eclipse for materializing the project workspace per se — i.e. fulfilling all external and internal component dependencies.

Buckminster’s objective is to leverage and extend the Eclipse platform to make mixed-component development as efficient as plug-in development, and to make automated building as simply a choice of invoking the one and only build definition from within the graphical user interface, or from the command line. To accomplish this, Buckminster:

- introduces a project-agnostic way of describing a development project’s component structure and dependencies
- provides a mechanism for materializing source and binary artifacts for a project of any degree of complexity and
- builds the end result by orchestrating the execution of built-in and user-provided build and test actions.

Why use Buckminster?

As a developer, you want to stay focused on the construction of your code, you expect it to be built interactively giving you instant error feedback. Once your code compiles, you expect to instantly be able to run/debug it — and when you make changes to the code it hot deploys into the running instance. At some point the edit/debug cycle is over — you have a set of components, and unit tests.

But you're not really done, of course. You still need to share what you've done so it can be integrated and built on a build server, tested, fixed, rebuilt, retested etc. The vision for Buckminster is simple — the system should just take care of all this for you automatically!

Most of the information needed is already formally expressed in your code, so Buckminster can figure out a lot about the components and how things should be put together. There are certain choices you made as a developer that are almost impossible for Buckminster to figure out on its own. So, a little work is still required on your part. But hopefully a lot less. Another important set of benefits comes from Buckminster's ability to run the same actions both interactively in the IDE and headlessly on a server. This is particularly useful for organizations implementing continuous build integration and test automation, as well as for open source development where anyone should be able to build the source.

Why read this book

We've attempted to make this book a clear, concise and definitive reference. We've tried to cover the bases regarding using Buckminster in the most typical usage scenarios. We've also tried to provide enough detail to serve as a starting point for more specialized scenarios, including customizing Buckminster itself. Following are the key topics we address:

- **The general nature of Buckminster.** Not everyone wants to learn Buckminster from the bottom up by working through the XML schemas. So we will quickly get you up to speed on Buckminster's architecture and what it can do for you.
- **How to get and install Buckminster.**

- **How to get software components from various sources.** Buckminster provides the mechanisms to get software components in source and binary form from a variety of sources such as source code repositories, Eclipse p2 update sites, and Maven.
- **How to invoke actions** that perform builds and other common tasks.
- **Best practices** when working with Eclipse plug-in projects, and when building RCP applications.
- **Publishing** the built result so it can be consumed by users.
- **Solution cookbook** with examples of how to solve various common issues when building software.
- **Setting up continuous integration with Hudson and Buckminster.**
- **Unit testing.**
- **Extending Buckminster.**
- **Reference documentation.**

This book’s audience

We expect that most readers have familiarity with Eclipse in general. When describing Buckminster features that directly related to developing Eclipse plugins, OSGi bundles in general, writing complete RCP applications, managing p2 repositories, or using Buckminster for C++ development, we expect the reader to have an understanding of development using the respective technology. Although we do provide introductions to the technologies surrounding Buckminster, as it would otherwise be difficult to understand the full picture, these introductions are by no means intended to serve as anything but starting points for further explorations.

Conventions used in this book

Most books show you all the conventions used, but there are only a few things that needs to be mentioned...

Manually inserted line breaks

Examples in XML tends to get quite wide, and line breaks must be inserted or the lines will be truncated. When this is the case, we include a ↵ where the line is broken, and one or several ¬ characters on the subsequent line to denote that what follows is a continuation of the previous line. Here is an example:

```
http://somwhere.outthere.com:8080/with/long/path/and/parameters/like↵
¬?thisOne=withAValue&andThisOne=withAnotherValue↵
¬&thisThirdParameter=withYetAnitherValue↵
¬&soForth=untilTheLineNeedsToBeBrokenUpAgain&andThenSome=extraStuffAtTheEnd
```

If you type in one of these examples, you should remove everything from the ↵ to the last ¬ (inclusive) on the next line and have no line breaks.

Replaceables

Replaceables denote text that is variable in nature — the replaceable part is something you would type, or that is generated by the system. We use the guillemots characters « and » around the part that should be replaced e.g. `copy «fromName» «toName»`.

Getting examples from this book

The examples in this book can be obtained from the Buckminster source code repository. Up to date information is found at the general Buckminster project page at Eclipse.

The Buckminster project page is located at http://www.eclipse.org/buckminster.

Request for comment

Please help us improve future revisions of this book by reporting any errors, bugs in examples, confusing or misleading statements, or examples that you would like to see included.

Please report issues with this book in the Eclipse Bugzilla under the category Tools → Buckminster → documentation. The Eclipse Bugzilla is found at https://bugs.eclipse.org/bugs/.

Acknowledgements

Buckminster has been in development for quite some time. A precursor to Buckminster was developed in 2000, at the company Frameworx Inc. with the purpose of supporting the company's distributed development of a "software as a service" framework. Ironically, the component that resonated the most with the company's clients was the possibility to use this internal component, for traditional development. Many thanks to Kenneth Ölwing, who at Frameworx was the driving force behind this system.

BEA Systems, and more specifically the BEA Java Runtime Products Group, home of the JRockit JVM (now part of Oracle), is an early adopter of Buckminster. They have developed (and continue to develop) a set of tools for Eclipse and needed a convenient way to execute headless builds of these tools in orchestration with building the JVM itself. BEA sponsored development of Buckminster for a period of two years and provided real world production issues that helped increase the usability, stability and overall quality of Buckminster. Special thanks to Marcus Hirt for his enthusiasm, and continued support.

Bjorn Freeman-Benson helped write the first introductions to Buckminster, and had the courage to use Buckminster when assembling the update site for the Ganymede release. The interest in the "Ganymatic" helped increase the awareness of Buckminster and we noted a lot more traffic on the Buckminster newsgroups and we got many new users.

Early adopters among the Eclipse projects include STP, and ECF. Many thanks to Oisin Hurley, and Scott Lewis for the confidence in letting Buckminster build their projects.

Oisin Hurley, and Marcus Alexandre Kuppe are worth special thanks as they have never missed an opportunity to get hurt by the latest experimental Buckminster features and thereby helping us sort out the useful from the stuff that never should have been written. Thanks for all the great feedback!

David Williams, the Galileo build master, is using Buckminster's repository aggregation functionality to assemble and verify the repository for the Galileo release. We are grateful for the confidence and the help we received from David tracking down the cause of issues, which is not an easy task in a system as large as Eclipse. Also, special thank you to the p2 and PDE teams in resolving issues when things were getting rough.

Finally, without all the valuable feedback from the Buckminster user community in form of bug reports and patches — a big thank you! Buckminster would not be what it is without your help.

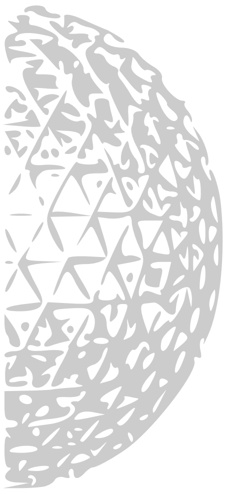
We are also very grateful to Cloudsmith Inc, our current employer, and its investors for making it possible for us to work on Buckminster.

TO-DO: It is not yet possible to acknowledge those that helped putting this documentation together...

Part I. Introduction

This part is intended as a quick introduction to Buckminster’s functional domain which includes provisioning, building, sharing, testing and publishing software components.

Central concepts such as the Eclipse workspace and target platform, OSGi and the Eclipse Plugin Development Environment (PDE), and the Eclipse provisioning platform (p2), are explained and put into context. Specifically, this chapter discusses how Buckminster works.



1 *Eclipse*

This chapter contains a brief overview of selected Eclipse technology and how it relates to Buckminster's domain of composing component based systems.

An overview of Eclipse concepts such as the workspace, target platform, component types such as plugins and features, is also found in this chapter.

You will also find a brief introduction to ANT, although not strictly Eclipse technology, it is still used by Eclipse and Buckminster.

Eclipse technology

A selection of Eclipse Technology explained.

Equinox

Equinox is the name of the OSGi runtime underlying the Eclipse IDE. It is a general purpose OSGi runtime. Equinox is (among many things) responsible for the loading (and unloading) of components. It functions as the container for the rest of the system.

For more technical information about OSGi — see http://www.osgi.org/About/Technology.

Platform

The Eclipse Platform provides the core frameworks and services upon which all plug-in extensions are created. It also provides the Equinox runtime in which plug-ins are loaded, integrated, and executed. The primary purpose of the Platform is to enable other tool developers to easily build and deliver integrated tools.

Java Development Tools (JDT)

Java Development Tools (JDT) is the set of tools build on top of the Eclipse platform for developing in the Java programming language. It includes a rich set of functionality for editing, compiling, debugging and running java code.

When used alone, created projects are "plain java" and management of dependencies is handled in a manual fashion and with this comes all the classic java issues with specifying a class path, and making sure all the required parts are available when running the code.

You will find more information about using Buckminster with "plain java" in Chapter 17, *POJO Projects*.

Plugin Development Environment (PDE)

The Plugin Development Environment (PDE) is a set of tools built on top of the Eclipse platform and JDT for developing Eclipse Plugins as well as more general OSGi bundles. PDE has a rich set of functionality to work interactively with the additional meta data found in plugins and bundles and supports all required operation from construction to publication.

The relationships between Eclipse plugins, features, and OSGi bundles is further addressed in the section called “The Eclipse component types”.

PDE also includes PDE-build, which consists of generation of ANT scripts that are then used to build software headlessly.

Buckminster provides a much more convenient way of invoking the various build actions in PDE than the script based PDE-build, as Buckminster does not generate scripts.

Rich Client Platform (RCP)

The Rich Client Platform (RCP), is the name for the Eclipse technology that makes it possible to write general purpose applications based on the Eclipse platform. The term “RCP application” is often used to denote the top level product such as the Eclipse IDE. Two well known open source applications built on RCP are the bittorrent client Vuze (Azureus), and the RSS reader RSS Owl. There are also many smaller RCP application in the Eclipse family, such as the p2 and Buckminster installers, the p2 agent, i.e. small independently packaged utilities with a user interface.

Many companies build their internal applications using Eclipse RCP.

Buckminster provides support for building complete RCP-products with a minimum of effort.

p2

Equinox p2 is the relatively new provisioning platform (introduced in Eclipse 3.4 *Ganymede*), designed to be a platform for many different kinds of provisioning solutions, and specifically designed to be a replacement for the Eclipse Update Manager. In Eclipse 3.5 *Galileo* p2 is both functionally rich and well tested with over a year of use, and with close to 2000 unit tests having been constructed. In 3.4 it existed in parallel with the Update Manager, and in 3.5, p2 has replaced it completely.

As p2 is heavily used by Buckminster, and p2 also defines the format of the typical end result (an installable system, or plugins to such a system) we have included a somewhat longer description in Chapter 2, *p2* as we believe this technology to be new to most users.

The Eclipse component types

The Eclipse system contains several types of “components”; OSGi bundles, plugins, features, fragments, and products. In this section we present an overview of what they are, and the role they play in the composition of a software system built on Eclipse.

Plugins, features and OSGi bundles

The terms “plugin”, “feature”, and “bundle” (short for OSGi bundle) refers to Java components that contains meta data information that makes it possible to manage their life cycle. The terms “plugin” and “feature” are specific to the Eclipse platform, and “bundle” is the generic software component handled by an OSGi runtime. Since Eclipse is built on the Equinox OSGi runtime, it can make use of all three types; bundles, plugins, and features.

Bundle

A *bundle* is the fundamental type. In addition to being the container for the code it has meta data describing its dependencies on other bundles, and requirements on packages expected to be present when using the bundle.

Plugin

A *plugin*, is also an OSGi bundle. What makes it special is that it also can contain information that makes use of the Eclipse extension mechanism — a declarative way to define that a bundle contains code that extends functionality in some other bundle.

Feature

A *feature*, is a grouping of plugins and other features. It defines a unit of what should be installed together. The feature is a configuration — a bundle may specify that it requires that a certain java package must be present, but the bundle says nothing about where this package should come from. This can be specified in the feature. This separation allows a bundle to be used in different configurations without requiring that the bundle itself needs to be changed.

Fragments

A *fragment* is a special kind of bundle with what could be called a “reverse dependency” on a host bundle. Fragments are typically used to implement optional code that is included in a configuration, often filtered on parameters like installed language, operating system, hardware architecture and user interface technology. As an example, a fragment could contain code that is only needed during testing or debugging, contain features available only on a particular platform, or for a particular language.

A fragment can also have normal dependencies — these come in effect if the fragment is selected for inclusion.

Fragments are included in a configuration by requiring them in a feature.

Products

A *product* is a special grouping mechanism used to define a “top level” product (such as the Eclipse IDE itself). Unfortunately, the tools that help maintain the group aspect of the product definition are somewhat lacking (in comparison to the same functionality for features), and we recommend that the product definition is used only to define the product aspect, and that all grouping is defined in a single feature that is referenced by the product. An examples of how to do this is found in Chapter 16, *Building RCP Products*.

When a product definition also acts as a grouping mechanism, it is referred to as a “bundle based product”, and when it refers to feature(s) (we recommend using only one) it is said to be “feature based”.

In addition to referring to the feature(s) or bundles being the configuration for the content of the product, the product also has a reference to a “branding bundle” that contains items such as the splash screen and icon for the product.

The Workspace

The Eclipse Workspace contains projects. These projects can be specialized (i.e. plugin project, feature project etc.). When you are looking at content in the Eclipse Navigator, or Package Explorer you are looking at content in projects.

You can get content into the workspace by:

- creating new projects and importing files manually
- importing a complete projects from somewhere on disk
- importing one or several projects from a source code repository
- linking to content in the correct format somewhere outside of the workspace
- importing from a “team project set” file, which contains a list of projects to check out from a source code repository.
- importing from source bundles (this is primarily used for debugging and patching).

As you can see, there is only one option that is suitable for automation — using the team project set. Many set up their projects to include such a file in a “meta project” and users begin by checking out this project and then importing using the team project set.

The pitfalls is that the team project set must be maintained manually. As dependencies are added or removed, the set of files required in the workspace may differ, and there is no way to control loading some projects from a branch or a tag.

Solving this particular problem was actually one of the very first requirements for Buckminster — as you will see later, Buckminster provides convenient population of the workspace for the typical case, and it is quite easy to load particular parts from branches and tags.

The Target Platform

The *target platform* is a definition of the set of features/plugins to use when running the code being built. You can say that the code is built for a particular target platform. By default, the target platform is defined to be the same as the Eclipse IDE — this means that when you are running your code in the self hosted environment you will not encounter missing bundles. When however you export and run the code separately, you will almost certain be hit by surprises.

Prior to Eclipse 3.5 there was no good way of managing a target platform in the IDE. A target platform was simply an Eclipse configuration in a directory.

In 3.5 the functionality to handle management of the target platform has been added. Multiple *Target Definitions* can be created. A definition can be saved to file (for later loading). It is also possible to make one definition the active target platform. The new *Target Definition* defines a set of locations. Each location can be one of:

Directory	A directory in the local file system.
Installation	An installation (such as an Eclipse SDK) in the local file system.
Features	One or more features from an installation.
Software Site	Downloads plug-ins from a p2 repository.

The preferred way of handling target platforms in 3.5 is to create one (or several) with the IDE and then save the definition to a file. Buckminster can use such definitions, and you can also materialize a target platform using Buckminster.

Launch configuration

A launch configuration is a definition of how to launch/run/debug something from within Eclipse. There are multiple classes of launchers for Eclipse covering running plugins, OSGi frameworks, tests, etc. Launch configurations can also launch servers or just run external commands.

Don’t confuse launch configuration with target platform. The (eclipse type) launcher launches the active target platform definition with the configuration specified in the launch configuration. This makes it possible to switch target platforms, build for that target, and then launch what was built for testing.

Many developers use the Eclipse IDE itself as the target platform, and then define the set of features/plugins to run in the launch configuration. In Eclipse 3.5, where target platform management has been improved, it is better to define an exact target platform and then have a simpler launch configuration that just use what is in the workspace and everything enabled in the target platform. This separation of concerns is valuable as the target platform definitions are reusable across many products/launchers, and makes it easier to migrate components to newer targets.

ANT

Apache ANT is a Java based build tool that is both well known and widely spread. ANT is integrated with Eclipse, and Buckminster.

The Buckminster integration consists of:

- Buckminster actions can be implemented as ANT scripts.
- It is possible to invoke Buckminster actions from ANT-scripts.

2 *p2*

An introduction to the Eclipse provisioning platform

Equinox p2 is the relatively new provisioning platform introduced in (Eclipse 3.4 *Ganymede)* designed to be a platform for many different kinds of provisioning solutions, and specifically designed to be a replacement for the Eclipse Update Manager. In Eclipse 3.5 *Galileo* p2 is both functionally rich and well tested and it has now replaced Update Manager completely.

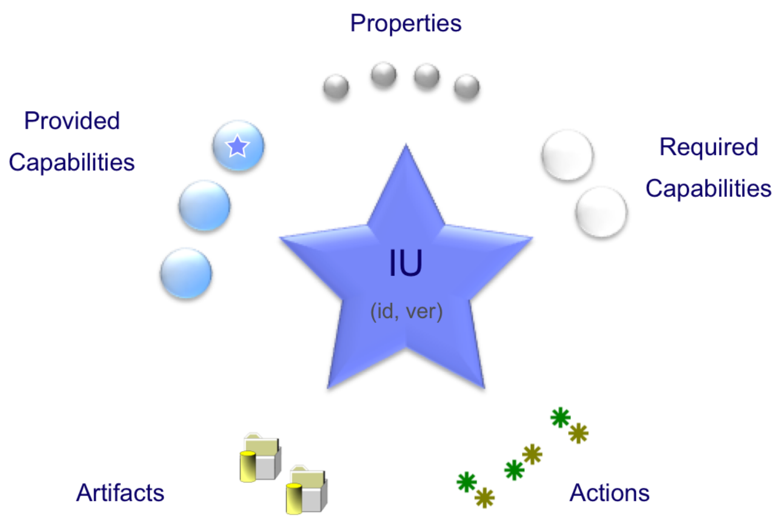
Equinox p2 is still new technology, and does not yet have an official API, and much work remains in utilizing its full potential. Replacing the use of the Update Manager in Buckminster with p2 was a big task since Buckminster and p2 in part are overlapping in functionality. Buckminster also steps outside of the OSGi domain which is the primary focus in p2 and PDE. The transition has been very successful thanks to the cooperation of the p2 and PDE teams — all three projects have gained from this; Buckminster can now use the much richer p2 to perform provisioning, and have access to PDE functionality via APIs — gone are the mysterious paths taken to overcome deficiencies in the old Update Manager. Equinox p2 has gained increased generality by the addition of the Omni Version capability which makes it possible to use p2 to resolve dependencies for non OSGi components. The Omni Version is covered in Appendix C, *Omni Version Details*

This chapter is only a brief introduction to p2 meant to establish the key concepts.

The Installable Unit

The central concept in p2 is the Installable Unit (IU). It is an entity named in a name-space having a version e.g. the `org.eclipse.equinox.bundle` named `org.myorg.helloworld` having version `1.0.3`.

Figure 2.1. Anatomy of an IU



Dependencies are handled by declaring *required capabilities* which are matched with *provided capabilities* also declared in a IU. Specifically, all IUs have a declaration that they provide themselves as a capability. This makes it possible for one IU to require another. The dependency mechanism is very flexible as it allows addition of new capability types. Capability types for Eclipse related types (i.e. plugins, bundles, features, java packages, etc.) have already been defined and are used by p2.

An IU's artifacts — i.e. the content the IU is describing, is referenced via name and type, and when the artifacts are needed, they are looked up in a p2 *artifact repository*.

The IU also contains *touchpoint instruction*; actions that are invoked in specified phases of a provisioning job e.g. when installing of uninstalling. The instructions can be things like copying files, unziping an archive, changing startup parameters etc.

If an IU requires special installation instructions these must naturally be installed before an attempt is made to install the IU itself. A mechanism called *meta requirements* allows an IU to declare these, and can then trust p2 to handle resolution and installation of these when an installation of the IU itself is requested.

Metadata repository

The meta data describing components — i.e. the IUs, are stored in a p2 meta data repository. Technically, a meta data repository is an interface and there are several implementations delivered with p2.

- A simple meta data repository stored in a file system directory
- A composite meta data repository that references other meta data repositories
- An Update Site based repository (i.e. the structure used by the older Eclipse Update Manager)
- Specialized repositories that enable the current installation (among other things) to be used as the meta data repository.

Artifact repository

An artifact repository contains the contents of IUs such a files, zip archives, jar files, etc. Technically an artifact repository is an interface and there are several implementations delivered with p2. The available repository implementations are similar to the meta data repositories (i.e. simple, composite, update manager based, and special).

There are many advanced options such as controlling how artifacts are physically stored and sent over the wire; verbatim, packed, or as a delta.

Combined / co-located repositories

Although p2 is capable of handling that meta data and artifact repositories are stored in completely different locations (anywhere addressable by a URI), the most common set up (and the only one supported from the Eclipse SDK's user interface) is a combined (or co-located) repository where a meta data repository and an artifact repository is addressed via a single URI.

Profile

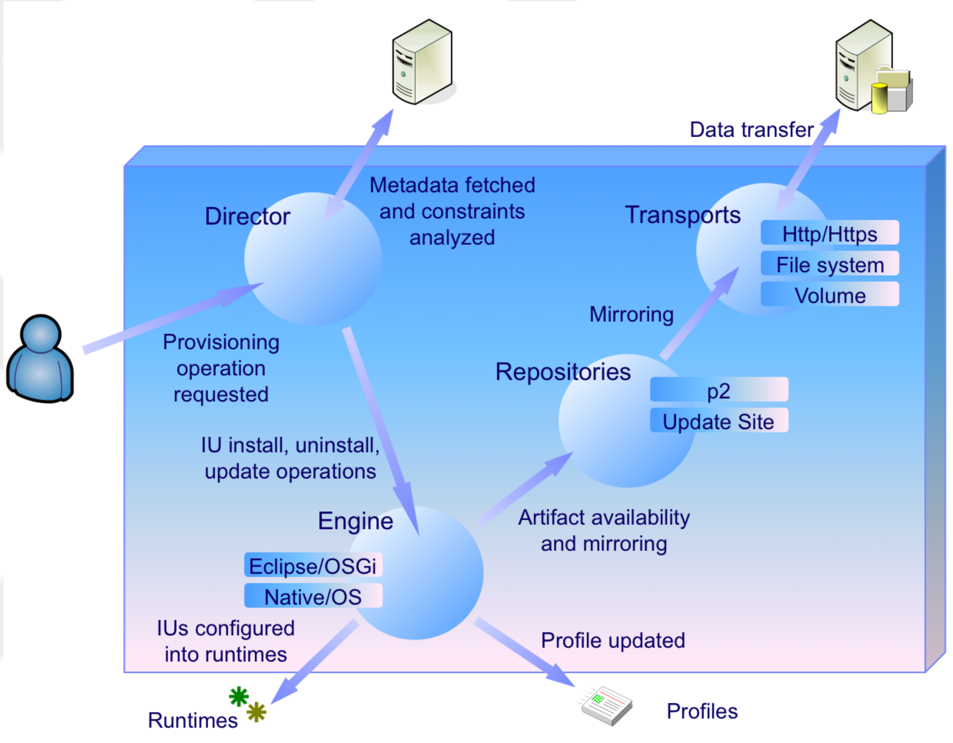
The p2 *profile* is a central concept — an installation of a product is described by a profile. It contains the meta data for everything that is currently installed. Thus, installation always takes place into a p2 profile.

A profile maintains a history, and it is possible to roll back to a previous configuration. As you may guess, a profile can also function as a repository, making it possible to "copy" parts of an installation from one profile to another.

p2 internals

Internally, the provisioning work is divided up between p2's major parts. The *director* handles provisioning requests such as installing or uninstalling one or several IUs. The director performs the work by using the meta data available in a profile, combined with the meta data in associated meta data repositories (those that have been used to install components from, or repository references just about to be added to the profile). This information is then fed to the *planner* which is responsible for resolving all requirements (dependencies). The resulting plan is fed to p2's *engine* which executes the work in *phases* (in simple terms — it collects items, downloads/mirrors artifacts, installs, and then configures them).

Figure 2.2. p2 in action



Data transfer
Director
Metadata fetched and constraints analyzed
Transports
Http/Https
File system
Volume
Mirroring
Provisioning operation requested
Repositories
p2
Update Site
IU install, uninstall, update operations
Engine
Artifact availability and mirroring
Eclipse/OSGi
Native/OS
IUs configured into runtimes
Profile updated
Runtimes
Profiles

The planner uses SAT4J to handle the complicated NP-complete problem of resolving requirements. It is interesting to note that there is a guarantee that if there is a solution, it will be found, and it will be an optimal solution (i.e. optimal in the sense of a defined set of weights such as “later version is better”). The use of SAT4J is a major leap forward compared to the old Update Manager (may it rest in peace).

Categories

From an end user perspective, an important part of p2 is the handling of categories. They are used to group related features and arrange them in a structure that makes sense for a human installing software. The features (although one level up above the (to a user) almost incomprehensible very technical plugin names) are still often quite technical in naming, and it can be very difficult for a user to understand the purpose of a particular feature. You have probably already seen the use of categories, as they enable you to browse the content under labels like “java development”, and “modeling” as opposed to just seeing a long alphabetical list with project names.

Prior to Eclipse 3.5, categories were authored in the Update Manager’s `site.xml` file stored in an update site. Such category information is read by p2 when it reads an older update site. When producing new p2 repositories however, the category information needs to be authored differently. In Eclipse 3.5 there are three ways; use the new *Category Editor* which creates a file that PDE makes use of when exporting to a p2 repository, use the (provisional) *p2 publishing advice* which is stored in a `p2.inf` file in the component being published, or use Buckminster which supports definition of categories in build properties.

Publishing

The act of making components available for consumption by p2 is referred to as “publishing”. It is an area that overlaps with three key technologies; p2, PDE, and the Eclipse platform, and if you look under the covers, you will see that they work in close cooperation.

PDE understands the source components, the meta data that makes the java projects be plugins, features or products. These are translated into the p2 form (IUs), containing information and instructions that makes it possible to install them and control the startup of the equinox environment.

Publishing components as p2 repositories does not require any additional authoring of p2 specific artifacts. More specifically, you do not need to author the IUs — this is done by the PDE specific publisher.

Prior to Eclipse 3.5, publishing was done by first producing an update site, and then generating the p2 combined repository from the output. This is basically what p2 does when it encounters an old style Update Manager site — “publishing” if you like, the update site on the fly. Although this interpretation of update sites is still supported, the recommended way of publishing is via the p2 publisher as it has more information available. (As you will see later, Buckminster provides a very convenient mechanism to execute p2 publishing).

Installing

Installing from p2 repositories (or update sites adapted by p2) can be done by a user of the Eclipse SDK directly in the SDK’s “install new software” dialog. With update manager, this was the only (managed) choice — most experienced users simply dropped the required files into the Eclipse installation folder structure (and this worked most of the time). Now, with p2, an install is fully managed to ensure that all requirements are met and that needed actions such as setting startup levels, and modifying initialization parameters take place during installation. This ensures that things actually have a chance of working, be updated, and eventually uninstalled.

With p2, the options are many, especially since p2 does not require that the system being installed into is active when performing the install — it can be done by an external p2 “agent” (there is a utility application called the “p2 agent” which is one example of such an agent. The “p2 installer” is another such example, and the SDK itself also has such an agent).

Note

The various agents all share the same p2 code — the difference is that they are designed to be used in different situations, and thus they expose only information required to support the particular task they were design to handle.

Since users have become accustomed to “dropping in” things that should be installed, this is also supported in p2, but the plugins and features are now dropped in a special folder that is monitored by p2. When it encounters new material in this folder, p2 will perform the same type of managed installation as when installing from repositories. There are several caveats when using drop-ins to install, and it is not the recommended approach as the higher quality meta data provided by publishing is unavailable.

The SDK agent

The p2 SDK agent manages installations into the SDK when used from the user interface. But the functionality of this agent can also be accessed from the command line to perform installation as an external agent. This is referred to as “running the embedded director app”.

Users of Eclipse will typically not use this embedded agent, and instead perform installation work via the user interface. The user interface and backing functionality can also be used in RCP applications, and there are many configuration option available to cater to different installation and update policies (on demand, automatically on startup, completely hidden from the user, update only (no new install, no uninstall), lock down of used repositories, etc.

The SDK agent allows the user to add and remove repositories (under *Eclipse* → *Preferences*), or directly in the “install new software” dialog. The user can see what is installed, select new features to install from selected repositories, perform the installation, and much more.

Since the SDK agent is designed to install into the running SDK itself, many of the advanced features, such as installing into an arbitrary profile, control advanced repository layout through the use of bundle pooling and shared installs are not present in the user interface. One of the other agents should be used for this purpose.

The director application

The director application is part of every Eclipse SDK and can be invoked from the command line. The director app is also packaged as a separate headless product with a reduced footprint. The headless director application is maintained by the Buckminster project. (See Appendix A, *Installation*, for how to obtain it).

The director application makes it possible to control the more advanced features in p2, while still having convenient command line options available for the most common operations.

We will shown examples where the headless separate director application is used and how to get it is explained in the section called “Installing the Headless Product”.

The p2 Installer

The *p2 installer* should be seen as an exemplary implementation of an installer, its user interface is quite unsophisticated, and it lacks many production grade qualities such as detailed progress information, and error reporting. That said, it is still a very useful utility when a user interface based installer is wanted.

The p2 installer in its default configuration is designed to install the Eclipse SDK. It is pre-configured with all the parameters, and when invoked after downloading, all that is required by the user is to tell the installer where it should install the SDK.

It is however possible to feed the p2 installer a different set of parameters by providing a properties file with the information regarding what to install from where, and then modifying the startup of the installer to override the built in default. This requires far less effort than creating a custom installer and may be sufficient for many smaller applications.

The p2 installer is used in one of the examples to install a RCP application — see Chapter 16, *Building RCP Products*.

The EPP wizard

Finally, the Eclipse Packaging Project (EPP) has written an application called the EPP-wizard, a RCP application with a RAP user interface which is driven by meta data to allow a user to select between high level EPP packages such as “Eclipse classic”, or ”Web development”, and then add support for optional technologies (such as Buckminster).

At the end of the process, the EPP-wizard provides the user with a configured p2 installer, that when downloaded and invoked will install exactly what the user picked from the available options.

The Buckminster installer

The Buckminster project also provides an experimental installer. It is designed to be started via Java Web Start or via a Java applet and it gets its initial parameters indirectly via a URL. Originally this installer used Buckminster’s provisioning capabilities and before p2 this was one of very few options available when an external, web startable installer was wanted.

The Buckminster installer also includes a JSON client, and is capable of engaging in a dialog with an smart repository and thereby present more information about what is being installed, manage a sign-in dialog, branding, and much more.

The Buckminster installer is however not yet considered released — its API may need further changes to be suited for general use, and testing is limited. Using this installer requires setting up the server side correctly and this part is not included in the installer, and no documentation is provided.

Shipping

By *shipping* we mean making the published material available to the intended consumers. You may think of this as “publishing” (i.e. making something publicly available), but this term is already used to mean making the internal meta data found inside projects public to the outside world in the form of p2 repositories.

In fact, there is no support in Eclipse to handle the steps required once such publishing has taken place. The resulting folder structure with files in them are simply written to disk, and there everything ends.

The most common way of shipping is making the published result available on a web site. And in cases when what is shipped is supposed to be installed into the Eclipse SDK, or consists of plugins for some other RCP application, this is as simple as just copying the result written to disk by the publisher to the appropriate directory where a web server picks it up.

If creating a complete application however there are more to consider. Users will typically not have the application installed to begin with, so user must start by downloading something. As seen in the section called “Installing” there are several installers available that can serve as a starting point — from the headless director application, to the interactive Buckminster installer. The benefit of using these is that there is no need to ship the complete application pre-configured for different platforms — as this is handled by the installer. Unfortunately, as the various installers were all created for a specific purpose, and some being more “exemplary”, you may find that they may not suffice if you are going to ship a more high profile application, and you may want to write your own installer.

Your options for shipping includes:

- Pre-configured installations per platform. To do this, you typically run the headless director app — telling it to install for one particular configuration (operating system, window system, architecture, language, etc.) into a location on disk. The result is then zipped-and-shipped.
- An installer configured to install the application from a remote repository. This has advantages as the initial download is small, and the bulk of the installation is performed by p2 which supports parallel downloading, selection of mirrors, and compressed artifacts. It is also very simple to add download of newer versions as everything is stored in a central repository.
- Zipping up a p2 repository with everything and a configured installer. The benefit is that the user will download everything that is needed to local disk, and can perform the install while not being connected to the Internet. The downside is that the repository contains components that are never used on the platform where it is installed.

  This form is suitable if you are shipping on a CD/DVD.
- Delivering application via a Linux package manager such as RPM creating a read only and shared installation that is then extended via an embedded p2 agent.
- Hybrid form, where the basic application is downloaded using one of the above mechanisms, but where bulky extras are installed via a p2 agent embedded in your application (like the Eclipse SDK p2 agent), or via an external installer.

In addition to deciding on how to ship — you must also decide on how you want to compose the required repositories. Your options include:

- Creating a composite repository with a reference to the main Eclipse repository for everything that is used from the Eclipse platform. This has the advantage that “your site” is always up to date with the latest repository content, and you do not have to store copies of everything in your repository.
- Creating an aggregated meta data repository that contains the meta data from the Eclipse main repository as well as your site(s), but uses the existing artifact repositories via a composite artifact repository. This has the advantage over the simplest form in that all of the meta data is obtained in a single download, and since you are reconstructing the meta data, you also have more control over the categorization of features.
- Mirror everything you need to your repository and then deliver everything from your servers. The benefit is that you have full control, but you do not make use of the Eclipse mirrors, and you must periodically update your mirrors.

Buckminster has support for aggregating sites — this functionality has been used in the Eclipse 3.5 Galileo release to compose the final Galileo repository. The Buckminster site aggregator is not described in this book.

Summary

Equinox p2, is a provisioning *platform* and as such has a rich and flexible feature set. Being rich and flexible also means that it is complex. It is complex in itself as it is solving a very difficult problem, and it is doing so with OSGi technology that under the covers need to perform complex tasks so developers can focus on the functionality instead of the mechanics of configuring a dynamic system — all in order to provide consumers of the resulting software with a high quality software provisioning experience — simply click install, and run automatic updates.

In the following chapters we will show how Buckminster, p2 and PDE work together, and how you can used Buckminster to handle some of the complexities.

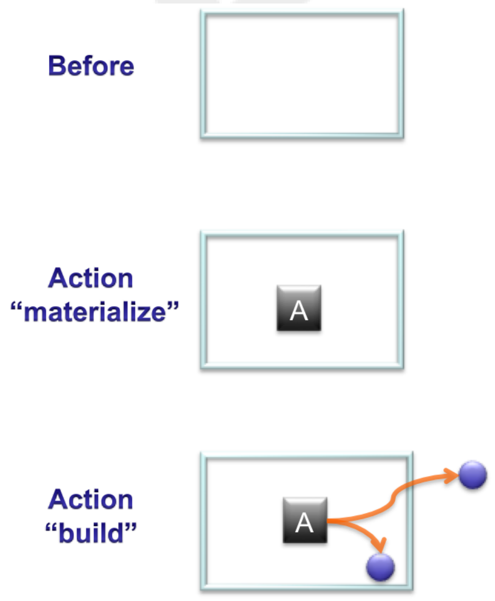
3 *Buckminster Introduction*

This chapter is an overview of the functionality in Buckminster. You probably want to read this chapter before continuing with the second part of this book.

Functional Overview

The highest level description of what Buckminster does is simply as follows. You want to build something, and have nothing of the material you want to build. You tell Buckminster to *materialize* the *component* you are going to build, and then you tell Buckminster to build it. This produces output within your workspace, or somewhere on disk.

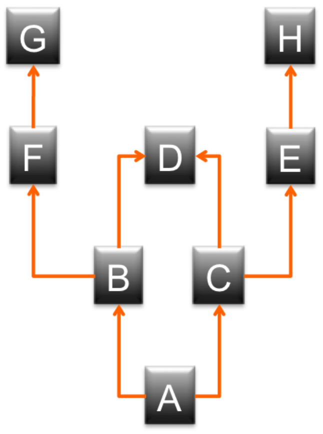
Figure 3.1. Buckminster from 10.000 ft



Materialization fetches components so they can be worked on. *Actions* such as build can then be performed.

When you request the component to build (*A*, in the example above), Buckminster will not only fetch this component, but also resolve all of its dependencies transitively.

Figure 3.2. Transitive Materialization

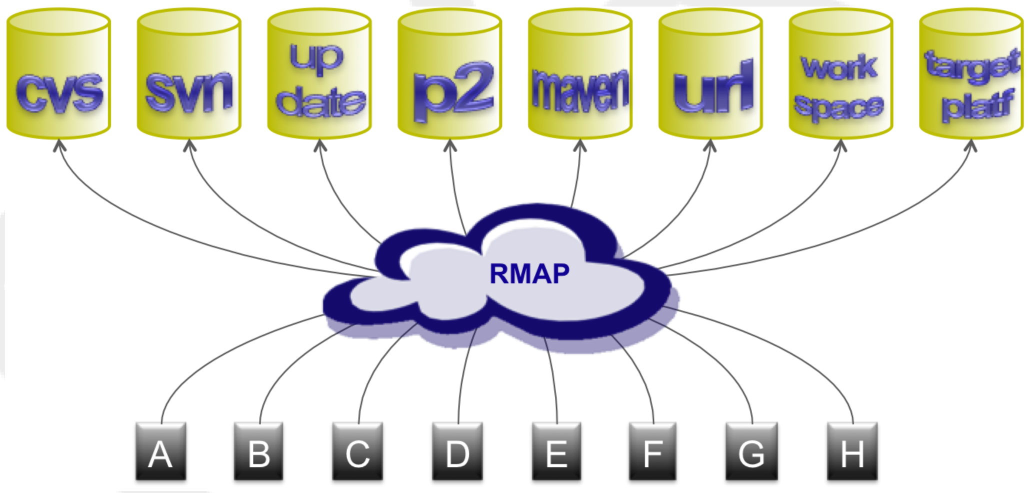


When requesting component A, it in turn requires B, and C — they both require D, B requires F, and F in turn requires G, similarly C, requires E and H.

Getting Components

The first two questions are usually, *Where does Buckminster get the components?* and *Where does Buckminster store them?*

Figure 3.3. Resource Map and Repositories



Components are looked up in a resource map (RMAP) which holds the rules for accessing different types of repositories.

When Buckminster needs a component, a lookup is performed in a Buckminster resource map (RMAP). The RMAP contains rules how to translate a request for a component of some particular type and version to a repository location of a particular repository type, and how to address the component within this repository.

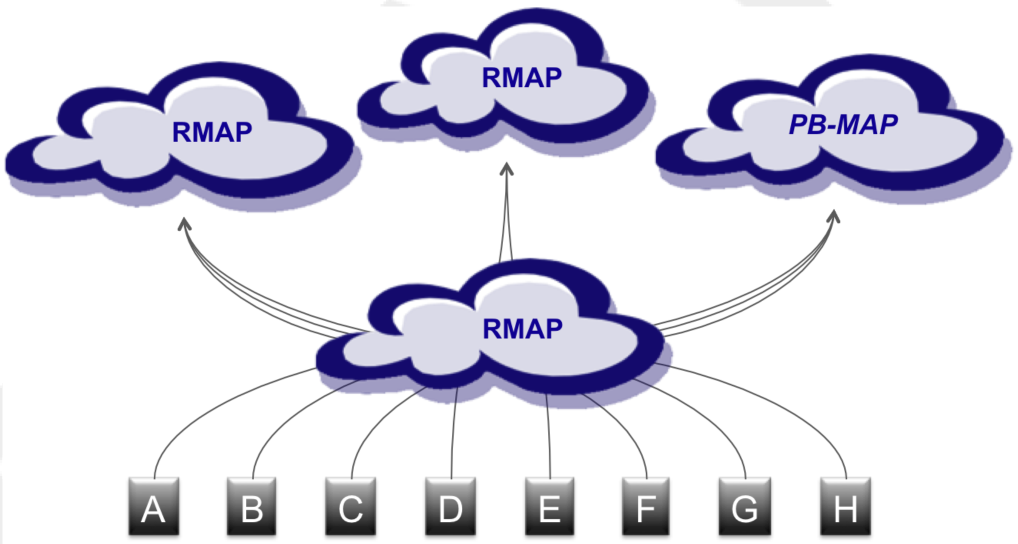
Buckminster supports a wide variety of repositories, and it is possible to extend Buckminster with new types.

- **CVS** — it is possible to reference components found in HEAD, in branches and via timestamps.
- **SVN** — it is possible to reference components found on trunk, branches, and named tags.

- **Update Site** — components published on a Eclipse Update Site in the format specified by the Update Manager (in use up to Eclipse 3.5). In Buckminster for Eclipse 3.5, update sites are read via p2.
- **p2** — components available in a p2 repository can be fetched.
- **Maven** — components stored in a maven repository can be fetched.
- **URL** — a single component can be fetched from a given location.
- **Workspace** — the components currently in the workspace (probably in source form) are also available to Buckminster's resolution process — naturally there is no need to actually fetch them, but their presence may override resolving to the same component in binary form in some other repository.
- **Target Platform** — the components in a target platform are available to Buckminster's resolution process — these are also not fetched, but affect the resolution process.

The resource map does not have to be a single map. It is possible to reference other maps.

Figure 3.4. Federation of Resource Maps

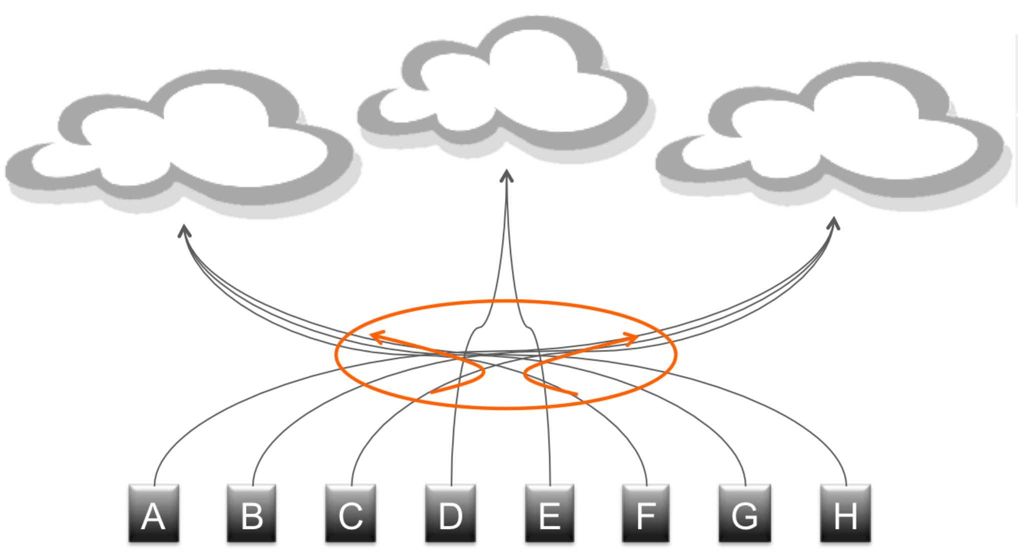


A federation of resource maps, including a platform base builder map.

It can be useful to organize the overall resource map in a distributed fashion. You may want that different projects maintain a map of their components — which is especially important if projects are following different naming standards, and when they are performing refactoring of repositories. An important feature for projects at Eclipse is that the platform base builder maps are directly supported. This is important because many Eclipse projects include components from the Eclipse Orbit repository and a platform base builder map is provided for this repository, and it can be directly used. Some projects, that are currently building with the platform base builder naturally also benefits as it is easier to transition to Buckminster by directly being able to use existing maps[1].

[1]Although not required, if you are using the platform base builder maps it is recommended that you switch to a Buckminster resource map as it is easier to maintain if you are following typical project naming standards.

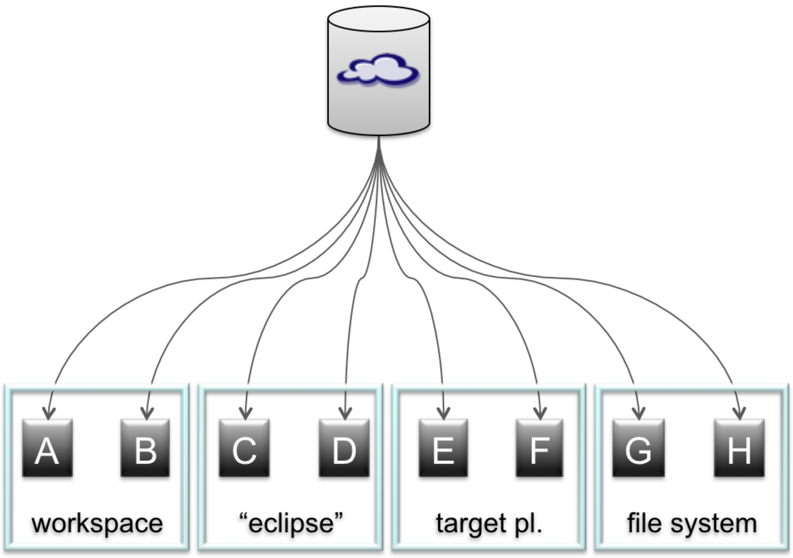
Figure 3.5. Resource Map routes



Resolution can take different routes depending on rules and parameters.

When Buckminster resolves a component it can take parameters and rules into account when selecting the route through the map (single map, or federation). You can for instance organize the maps so that users looks up components from a local repository rather than always going to a central repository, and you can organize the map so this can be done when the component name matches a regular expression (and much more).

Figure 3.6. Materialization Types



Buckminster can materialize (store) fetched components in different types of locations.

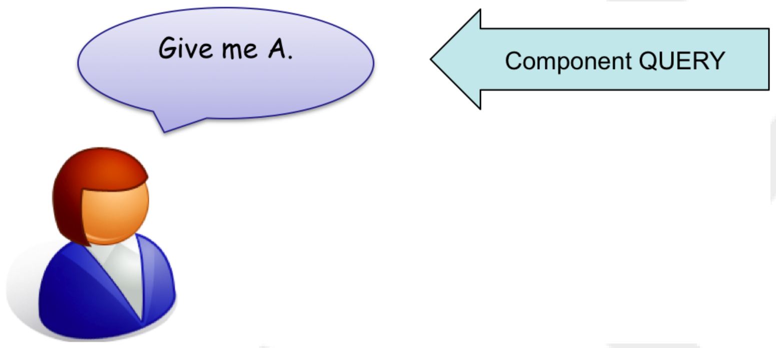
When Buckminster materializes components, they can be stored in different types of locations. Buckminster supports Eclipse related locations, and the file system, but can be extended with other types of locations.

- **Workspace** — typically projects are materialized to the workspace, but it is also possible to bind binary components (this was common practice prior to Eclipse 3.5 because of difficulties with managing the target platform)
- **Eclipse** — i.e. installing tools into an Eclipse based product such as the Eclipse SDK or an RCP application. Prior to Eclipse 3.5, this was done by using the Update Manager. Since 3.5 this is performed using p2.

- **Target Platform** — i.e. installing into a definition against which components are built. Prior to Eclipse 3.5 the target platform had to be created separately, and then referenced in later operations. In 3.5, a target platform can be dynamically created and installed into.
- **File System** — i.e. storing the component on disk.

Now you have seen how Buckminster gets components, and where they are stored when materialized. But how do you tell Buckminster what you want?

Figure 3.7. Telling Buckminster what to get



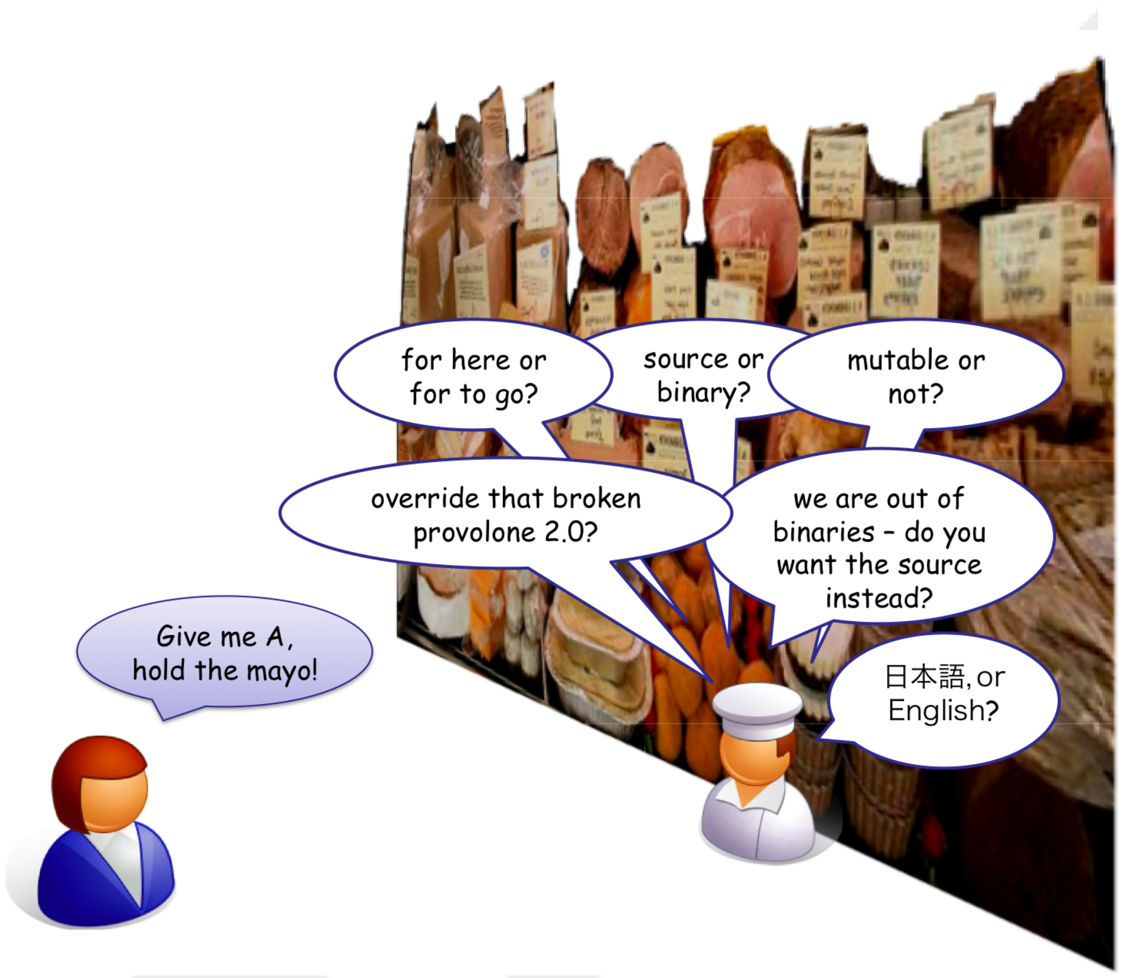
Getting things is done by submitting a Component Query.

Figure 3.8. Ordering at “Bucky Burger”

Telling Buckminster what you want can be as easy as ordering a meal at Bucky Burger...

Most of the time, the only thing needed is to state the name of the component you want. Buckminster will then find the latest version of the component. But sometimes you may have very detailed requirements on your meal.

Figure 3.9. Ordering at the Bucky Deli


for here or for to go?
source or binary?
mutable or not?
override that broken provolone 2.0?
we are out of binaries - do you want the source instead?
Give me A, hold the mayo!
日本語, or English?

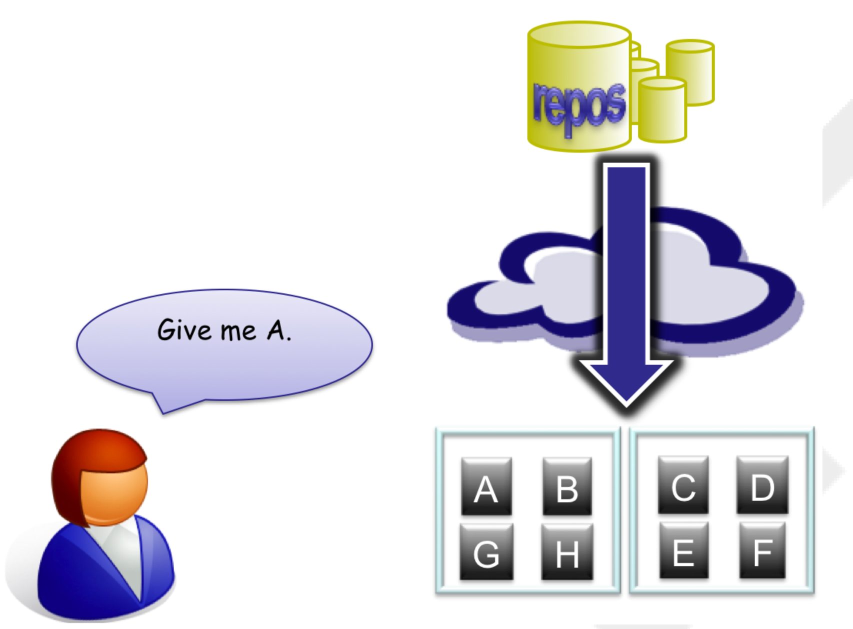
Getting picky at the Bucky Deli. (Are you sure that pepper is south Brazilian?)

As you will see later, Buckminster has a very powerful query mechanism where you can specify many options:

- Do you require source, or prefer source, but can work with binary, or only require binary form.
- Do you require source that can be modified and checked in (given that you have authority to do so naturally).
- Do you want to load some components from a branch or tag and override the default.
- Do you want to override certain component-version combinations even if requirements in the components say otherwise.
- You may want to specify that a search should use a particular path in the resource map for certain components — perhaps loading them from a central repository instead of a local mirror.
- You may want some components from a release repository, but some should be picked from a nightly build repository.

Buckminster component queries are entered and edited in a CQUERY-editor — which is explained in detail in the section called “The CQUERY Editor”.

Figure 3.10. Getting components — summary



Summary of getting a component — a query is resolved and components fetched from repositories, and materialized into different locations.

Component

We have already introduced the term *Component* without any further explanation. Now is the time to look a bit closer at what is meant by a component in Buckminster.

Figure 3.11. Component

Component is an abstraction — a named and versioned piece of content.

A component is an abstraction of a unit in a software system having a name, a type and a version. A component typically has content [2]— and it can exist in multiple forms — such as source or binary. When Buckminster obtains the definition of a component, and subsequently its content, a *Component Reader* matching the component instance's physical shape is used to interpret the component's meta data and translate it into a Buckminster *Component Specification* (CSPEC). This translation takes place each time the component is requested — there is no need to save the result. This has some important benefits:

- No round trip engineering is required. The meta data at the source is used directly.
- Does not require restating already expressed facts such as dependencies.

A Component is not tied to any particular implementation technology — Buckminster works just as fine with Java, C, PHP, as with just a collection of files. Even if it is possible to turn just about anything into a leaf component, in order to be really useful however, there must be some meta data available that describes the component and its dependencies.

[2]A component without content functions as a configuration or grouping mechanism.

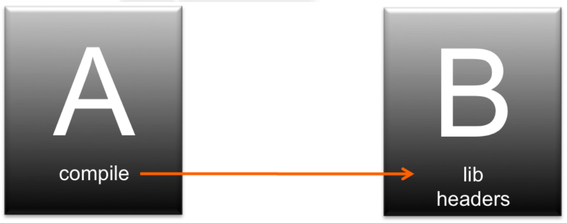
Buckminster have component readers for several meta data formats, and it is possible to add extensions for additional types. And in case you wonder, it is possible to combine different types of repositories with different types of component readers (although some combinations are nonsensical as certain type of meta data may only exist in certain types of repositories). Here is a list of available component readers:

- Eclipse types: plugin, feature, product, fragments
- OSGi types: bundle
- Maven: maven POM (version 1 and 2)
- Buckminster: Buckminster's CSPEC and Component Specification Extension (CSPEX). Both which are further explained below.

Component attributes

Components have attributes which are similar to concepts like member variables of a class, or the properties of a bean. The attributes represents either static data, or are dynamically computed by an action.

Figure 3.12. Component Attributes



Component A's compile action requires the lib and headers attributes from component B.

The type of an attribute is always an *path group* which represents a (possibly empty) collection of files. So, when the lib attribute in Figure 3.12, "Component Attributes" is read in the compile action it will get a collection of the lib files in component B.

It is also possible to declare an attribute to be an aggregation of other attributes (and actions, as you will see in the next section).

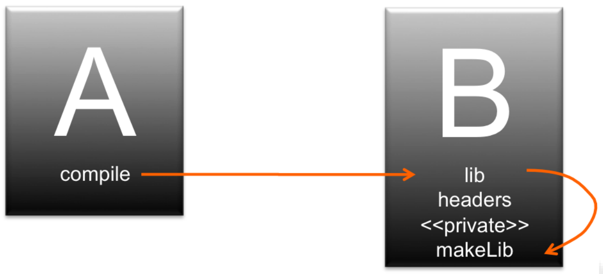
Component actions

Components have actions which are similar in concept to methods of a class. The return type is always a collection of files. From the requester's view there is no difference between a static attribute and an action, its value is simply requested — there is no need to know if the returned list of files is a static list, or a computed value.

Using the example in Figure 3.12, "Component Attributes" — the attribute lib could simply by turned into an action that computes the list of files to return instead of being a static declaration.

Using the *group* mechanism makes it possible to do advanced constructs that includes the result of both static attributes and results from invoking actions.

Figure 3.13. Component Actions



Private action invoked to produce the value of the lib attribute.

Attributes can be marked to be private, which means that they can only be used from other attributes in the same component. Public attributes can be read, and public actions can be invoked (i.e. read) by other components. In Figure 3.13, "Component Actions" the `compile` action in A get the value of `lib` in A. The attribute `lib` is declared to be a group containing the value of the action `makeLib`. As a result, the compile action gets the list of files produced by `makeLib` without knowing it was invoked. The `lib` attribute could also include the result of other actions, or static attributes.

The ability to encapsulate private actions is very important. Most build technologies do not provide this distinction and this creates maintenance headaches as it is almost impossible to know where and how build actions may be used. The end result is usually that no-one dares to make any changes to the build system because they don't know what they will break.

Actors

You may already have wondered how the body of an action is implemented — *What language are they written in?* The answer is that an actions body is made up of an Actor, and Buckminster has several types of actors available. Additional actor types can be added as extensions.

Buckminster has several types of actors:

- Java — compile, jar, etc.
- PDE — build bundles, features, fragments and products, pack, sign
- p2 — build a repository
- General — fetch files and execute system commands
- ANT — invoke ANT tasks. This is very useful for integrating with existing build systems written using ANT.

Turning something into a component

As you have seen earlier, there is nothing you have to do if the software unit you are interested in already has meta data for which there is a component reader available (as it is for all the Eclipse related types; bundle/plugin, feature, and product).

When this is not the case what you need to do depends on if there is meta data available at all, and if the meta data is rich enough to be useful — if that is the case, you are probably best of by adding a new component reader by extending Buckminster. For more information about how to extend Buckminster see Appendix B, *Extending Buckminster*. If however, the meta data is missing, or is poor, or you just don't want to create an extension, it is possible to use Buckminster's native CSPEC XML format. The Buckminster Reader expects to find a CSPEC file inside the component in a particular location. The file is created with the Buckminster CSPEC editor. This is explained in detail in the section called "The CSPEC editor". There is also a hybrid solution possible, for some reason it may not be possible to insert the meta data into the actual component, then you can construct an extension that still uses the CSPEC format, but where it is stored in an external location. This is much easier to implement than a full reader, as it only requires handling the association between the two — the meta data format and parsing can be reused.

Note

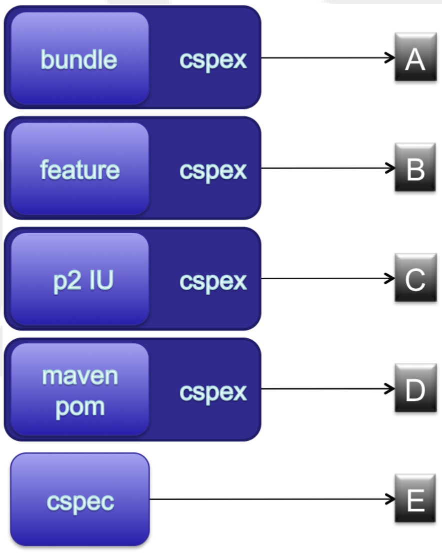
It is only in fairy tales a frog turns into prince by a mere kiss.

Decorating a component with additional advice

Buckminster has an extension mechanism for component specifications that allows you to decorate a component with additional advice. This is useful in several situations:

- adding additional actions to the component
- overriding faulty meta data
- adding dependencies to underspecified components
- hooking actions that should be executed as a component is materialized
- wrapping existing action to some additional work before or after the original action

Figure 3.14. Component Specification Extension — CSPEX



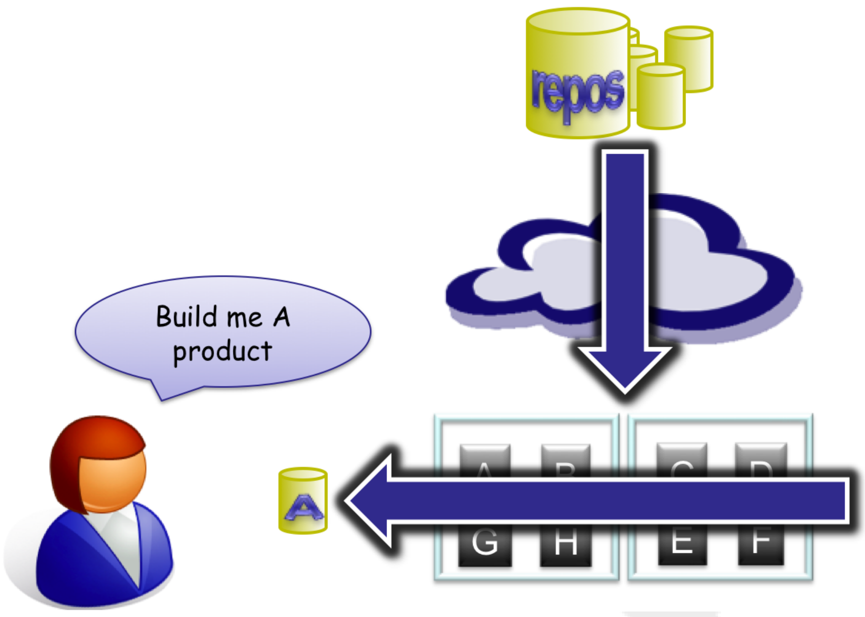
All component types can be extended with a CSPEX.

An extension is made by storing a CSPEX file inside the component — all available component readers are capable of handling this extension. The CSPEX is using the same format as the CSPEC.

Summary

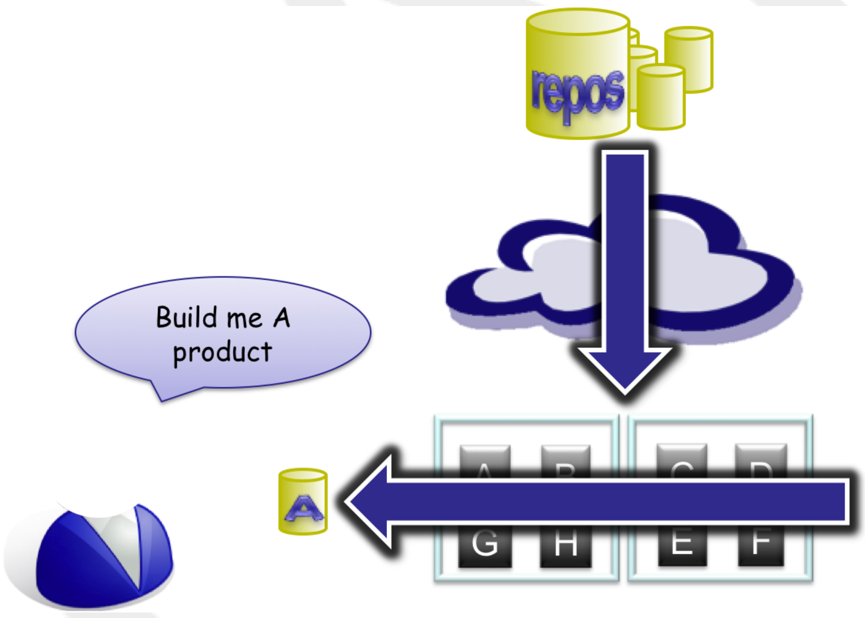
Buckminster gets units of a software system called components from repositories by looking them up in a resource map, reading and translating them into a common form, and then materializing them into different locations such as the workspace or target platform. When the components have been materialized Buckminster runs actions defined in the components such as building a product or a repository of plugins.

Figure 3.15. Buckminster Summary



Buckminster builds a product.

Figure 3.16. Buckminster Headless



Look Ma — No head!

The Figure 3.16, “Buckminster Headless” illustrates the most important feature of them all — the ability to build exactly the same thing in a headless configuration without having to do any additional authoring!

Reading on. You have now seen an overview of Buckminster and how it relates to other Eclipse technologies. You should now have a high level understanding of the capabilities. The rest of this book is not intended to be read from start to finish (although you may still want to), but instead provide detailed drill down in the various parts, as well as examples, and reference material.

Part II. Buckminster

In this part, we take a deeper look into Buckminster. The chapters are not intended to be read in sequence, although we try to follow a logical sequence — starting with the resource map, as that is probably the first thing you want to set up. Alternatively, you may want to start by installing Buckminster as described in Appendix A, *Installation*, and then running through some of the examples in Part III, “Examples”.

4 Resource Map

In this chapter we take a closer look at the resource map and how its features can be used to map component names to resources in different types of repositories.

The resource map is one of the first things you are required to set up. Without a resource map, Buckminster can only find resources already in your Eclipse IDE.

You can find several examples of resource maps in Part III, “Examples”, and some of these may be immediately useful to you as they map many of the components found at Eclipse.

The work required to create the resource map for your project or or organization depends to a large degree on how well naming standards have been enforced, and in what type of repository components are found. If your components are in p2 repositories, update sites, or in CVS, SVN or Perforce, and you have followed recommended repository layout, and named your projects after the component names, then the work is quite straight forward.

On the other hand, if you are pulling in components by downloading them from download pages found somewhere on the Internet, and you need to scrape the HTML content returned in order to find the correct URL for a particular version, platform, language, etc. then you naturally have more work setting up the map. Luckily — you also have the most to gain in automating such a manual task.

The search for the component

In the resolution process Buckminster finds that an already found component requires some other component. As an example say that the component `org.myorg.hello` requires the component `org.myorg.world`. Buckminster must now find that component to get its requirements (and so forth), and starts the process by looking up `org.myorg.world` in the resource map. In addition to the component name, the component also has a type, and possibly a version range that further constrain the search.

The objective is to find a *reader type* (how to read the content of the repository), and a *component type* (how to interpret content), and then a location to visit to get the actual content.

Buckminster tries to find these by searching through a list of specified *locators*. The locators are searched in the order they are defined. The locator has a pattern that is used to match against the name of the component. If the name is matched by the pattern in the locator, the search continues in a named *search path*. The search path specifies a list of component *providers*.

A *provider* specifies that it is capable of looking up components using a particular *reader type*, and *component type* from a particular *name to location transformation* (specified by a *URI* with additional rules).

Let‘s take a look at what that can look like in a RMAP[1].

Example 4.1. locator and search path

```
<searchPath name="dash">❶
    <provider readerType="cvs"❷
        componentTypes="osgi.bundle,eclipse.feature,buckminster"❸
        mutable="true"❹
        source="true">❺
        <uri format=":pserver:anonymous@dev.eclipse.org:/cvsroot/technology,org.eclipse.dash/{0}">❻
            <property key="buckminster.component" />❼
        </uri>
    </provider>
</searchPath>
❽
<locator searchPathRef="dash" pattern="^org\.eclipse\.eclipsemonkey([\.\-].+)?"/>
<locator searchPathRef="dash" pattern="^org\.eclipse\.dash(\..+)?" />
<locator searchPathRef="dash" pattern="^org\.mozilla\.rhino" />
```

❶ A *search path* is declared and named “dash”.

❽ Then, look at the *locators* — the search path “dash” is reused by all the locators. Different patterns are needed to match the different component names found in the dash repository. In case you are wondering about the patters; `org.eclipse.dash.somepart.hello` could be the name of a component.

❷ A *provider* is declared with a ‘`cvs`’ reader type

❸ The *reader types* are declared — we are interested in OSGi bundles (plugins, and plain bundles), eclipse features, and components that use Buckminster meta data.

❹ The attribute *mutable* is set to `true` because we want searches for mutable source (i.e. projects checked out from CVS that can be worked on and checked in again) to also use this search path. Note that in this example we are using an anonymous user so in order to be able to actually check things in again, someone with write access would have to use the IDE’s team CVS functionality to relocate the projects with their user id once they have been materialized to the workspace. You will see later how to create entries using authentication — see the section called “Providers and authentication”.

❺ The attribute *source* is set to `true` because we do want the source (as opposed to some binary incarnation of the component — we are perhaps even running an older version of the component).

❻ The *uri* specifies the location of the component name under the `org.eclipse.dash` root in the eclipse technology project’s CVS repository. Note the {0} at the end, which specifies the use of a parameter.

❼ The *property* `buckminster.component`, (which always contains the name of the component currently being looked up), is used as an argument to the uri in ❻.

Creating a RMAP

A resource map is defined in an XML file. The extension ‘`.rmap`’ is used to make it decorated with the appropriate icon when handling this file in eclipse.

You can naturally start by copying an existing RMAP that contains something similar to what you want, or you can start from scratch.

The Buckminster User Interface, has defined *New File Wizards* for the Buckminster artifacts. So you can use *File* → *New* → *Other...* → *Buckminster* → *Resource Map file* to create a ‘`new_rmap.rmap`’ in a project of your choice. The created file contains XML name space declarations, but is otherwise empty.

Once the file has been created, it is opened for editing.

[1]the XML name space declarations, and use have been omitted to increase readability.

Editing a RMAP

There is unfortunately no graphical editor available, so editing of the resource map is done using an XML editor for Eclipse (or naturally some other XML editor of your choice).

As with every XML artifact used in Buckminster, the RMAP is described by an XML schema (see Buckminster XML Schemas). If you make an XML editor aware of the location of the schema, it will be able to help you with automatic code completion, validation, and valid attribute values (see the section called “Configuring Eclipse for XML Editing”).

Designing a RMAP — some advice

When you are creating your first RMAP, you are probably going to just “hack away” until you have something that works for your project. As you are doing so you are getting to learn how the RMAP can be used.

Our experience is that organizations (and individuals) over time has created lots of components where strict adherence to naming standards has not been followed. We have come across things like:

- Some users thought the names were too long — “I don‘t want my Eclipse project to be called `com.mycompany.someroot.myproj.titanic.module-a.mybundle`, I want to name them after *me*, and then the name of the bundle!”
- Well, the product was called “titanic” earlier, but marketing did not like that name, so we changed it to “titan”, we modified the name of the root in the source code repository, but did not bother with all the other names — except in some parts of the repository.
- “Some of our projects have misspelled project names, and it is just hell to change all the scripts. Oh, and some have misspelled components too :)”

So — faced with reality, what should you do? Should you try to create maps that map everything in a repository, and deal with all the peculiarities in this repository across different projects? Should you delegate the work to the respective project and compose a master resource map out of what the projects provide? Should you undertake a large “naming standard hygiene” project to clean up all the past sins and mistakes?

Well, only you can tell what is appropriate for you. Fixing the odd mistake in naming in a smaller project is probably doable at a low cost and risk and well worth doing in order to reduce complexity. However, in an environment with many components handled by a geographically distributed engineering organization and in many repositories, and with a multitude of handcrafted build scripts — well, you could always try to get a budget...

A pragmatic balance is probably the best advice — map what you need, focus on getting your project’s build automated and leave the rest unchartered. Pick things to automate where you have the most to gain and then continue with the next. Once things are automated, it is much easier to change the bad naming if you want to perform some cleanup. In some cases (depending on the source code management system used), it can be difficult to move things around so you may just have to live with having to handle the complexities in the mapping. The good news is that once projects are automated, users can rely on Buckminster’s resolution to do the work for them, rather than having to ask a colleague where in the repository a particular component may be located. (“You were looking for version 3... I thought you said 4 — well, the project was called “titanic” back then, and it was before we cleaned up the references, and oh, yes, it is in the old source code control system — let me see if I remember the URL and the branch name we used for maintenance on the released 3.5a...”).

Locators

The locators are the entry point into the map — the patterns you provide for matching on component names controls where the lookup continues.

The absolutely simplest locator would direct everything to a single search path:

```
<locator searchPathRef="everything.found.here" />
```

Omitting the pattern is perhaps not very useful on its own, but becomes useful when you want to continue trying with more locators as shown in the section called “Fail on error”.

How to write patterns

The locator patterns are based on Java regular expressions. If you need a primer, or more information about Java regular expressions look at this tutorial [http://java.sun.com/docs/books/tutorial/essential/regex/].

Lets look at an example:

```
<locator ... pattern="^org\.eclipse\.eclipsemonkey([\.\-].+)?"/>
<locator ... searchPathRef="dash" pattern="^org\.eclipse\.dash(\..+)?" />
```

Both patterns start with a `^` which means that the matching is anchored at the start of the input (and the input in this case is a component name). The pattern then continues with explicit name parts where period delimiters in the name are escaped with \ since a `.` otherwise means “any character”.

At the end of the second pattern you see `(\..+)?` which means zero or more occurrences of a literal period followed by a sequence of one or more characters. This is a good pattern to use when projects (i.e. component containers) are named after the component names and period is the only separator used.

At the end of the first pattern you see the construct `([\.\-].+)?` which accepts a hyphen or a period as separator.

This rule was created because the component `org.eclipse.eclipsemonkey-feature` could not be matched with the simpler rule `(\..+)?` since that rule requires a period after `eclipsemonkey`.

Tip

Although your patterns only have to be precise enough that they discriminates between the providers, you may later want to compose larger maps and it is a good idea to make sure that your patterns exclude what is outside of your map’s domain. Start your patterns with `^` and your unique prefix (e.g. `org.yourorg`...).

Fail on error

When a locator has directed to a search path, a component will either be found by one of the providers on the path, or the lookup will fail with an error. By default, the search will stop on an error, but it is possible to tell the RMAP that the search should continue with the next locator. Let’s look at an example, where we look things up in the Eclipse Galileo[2] and Orbit[3] repositories.

[2]Galileo is the name for the Eclipse 3.5 release, and the Eclipse Galileo repository contain the official coordinated release.

[3]Orbit is the name of the Eclipse repository of external (3d party) components that have been approved for use and redistribution from eclipse.org. i.e. components with acceptable license and pedigree.

Example 4.2. fail on error

```
<locator searchPathRef="myWay" pattern="^org\.myorg(\..+)?"/> ❶
<locator searchPathRef="org.eclipse.galileo" failOnError="false" /> ❷
<locator searchPathRef="orbit" /> ❸
```

❶ Everything beginning with `org.myorg` is directed to `myWay`. If not found, the search fails.

❷ Everything else is directed to the path `org.eclipse.galileo` (a path that looks things up in the Eclipse Galileo repository). Here `failOnError` is set to `false` as we don't want to set up patterns for Galileo and/or the Orbit repository. (As the `eclipse.import` reader caches an index of the repo, the omission of the pattern does not have a negative effect — it can quickly determine if a component is in the repository. You only need a pattern if you wanted to exclude some components from being looked up by this locator).

❸ If we did not find the component in the Galileo repository, the search continues with the orbit search path (there is no pattern). This is our last stop before giving up so we want to fail on error (the default setting).

Parameterized locator

So far, we looked at static declarations that picks a search path based on only the component name. But what if you want to pick up certain components from one repository such as a nightly build, and get the rest from the release repository? Clearly, you could insert a new locator with a pattern to match the particular component and direct it to a search path for the nightly build, but this is discouraged as it requires you to actually change the RMAP, and is specific to a particular case. The next time around you may need several components, and some other user will be needing another mix.

A parameterized locator is simply a locator that references a search path based on a property value. As you will see later, it is possible to set property values, and associate property values with individual (or groups of) components at the time when you are requesting them (see the section called "Properties"). In simple terms, this means, that instead of requesting "Give me component A", you will request "Give me component A, but pick B from nightly repo".

Example 4.3. locator with parameterized search path

```
<locator searchPathRef="myWay" pattern="^org\.myorg(\..+)?"/>
<locator searchPathRef="org.eclipse.platform.${useBuild}" failOnError="false" />❶
<locator searchPathRef="org.eclipse.galileo" failOnError="false" />
<locator searchPathRef="orbit" />
```

❶ The `${useBuild}` inserts the value of the property `useBuild` in the search path name.

As you see in Example 4.3, "locator with parameterized search path", you can include property values in the string that is a reference to the search path. What you need to do is simply to set up one search path for each alternative repository you are interested in.

A common setup is to have repositories that reflect the software process. The projects at Eclipse typically set up a series of repositories for *nightly*, *integration*, *milestone* and *release* builds — so suitable values for the useBuild property could be `NBUILD`, `IBUILD`, `MBUILD`, and `RBUILD`.

As an example, the search path `org.eclipse.platform.MBUILD` would be set up to access the platform's milestone build repository.

Redirects

As mentioned earlier, it is possible to reference one RMAP from another and thus build a federation of maps. This is done by using a `redirect` element instead of a `locator` — it works like the locator, but instead of continuing in a search path, it imports a referenced RMAP (referenced with a `href` attribute), and continues with the imported map's locators. You can only have one redirect, and it must appear last.

Note

You can use parameterized references for redirects as well.

Example 4.4. Using redirects

```
<!-- Example A ->
<locator searchPathRef="myWay" pattern="^org\.myorg(\..+)?"/> ❶
<redirect href="http://www.myorg.org/maps/ourmap.rmap" failOnError="false" /> ❷

<!-- Example B -->
<locator searchPathRef="myWay" pattern="^org\.myorg(\..+)?"/>
<redirect href="http://www.myorg.org/maps/eclipsemap.rmap" /> ❸
```

❶ A locator for my things in my project
❷ A redirect to a map on myorg's web server that maps all projects at myorg.
❸ A redirect to a map on myorg's web server that maps all the eclipse projects.

Locators summary

The list of locators is the entry point in the map and they are used to direct the search to a search path with declared providers or to a different RMAP via a redirect. It is possible to parameterize the search and thus create support for common use cases such as:

- Select between repositories like nightly, or release, globally, or on a per component basis.
- Select between different maps based on user's location by having a property that selects an appropriate repository mirror.
- Let a committer property control if you get mutable source or source bundles for debugging.
- Make per developer overrides when experimenting by using a local RMAP that ends with redirect to the official RMAP for the project or organization.

Search paths

A search path is a reusable element in a RMAP that defines how a component name is looked up in a repository and transformed into useful meta data. The search paths can be declared in any order — the search is conducted in the order specified by the locators.

The search path consists of one or several provider declarations where the provider defines a combination of a reader type (how to access the content in the repository), a component type (how to interpret the content found), a location to the repository, and a means to locate a component within the repository.

A search path is in itself quite simple — in addition to having a name, and a list of provider elements, it can have an optional documentation element where a description of how the path is supposed to be used, what it references etc. can be included.

A search path must have at least one provider. A search path typically contains one provider, but it is possible to specify more than one — in which case the provider capable of returning the component with the highest matching score compared against the request will be used. If two providers return the same score, the provider declared first is used. When making a request, options can be set that are compared to attributes set in the provider declarations to reach the score.

Request options. The request options indicate the wanted shape (mutable source, source, or binary), and if the request can be relaxed (i.e. if source is not mutable, is it ok with unmutable source, and if source is not available at all, is it ok with a binary result). The request options are set in *advisor nodes* when making the request. This is explained in detail in the section called “Advisor nodes”.

Providers

A *provider* declares a combination of a *reader type* (how to access the content in the repository), a *component type* (how to interpret the content found), a *location* to the repository, and a means to locate a component within the repository. A provider also declares attributes that are used when calculating a matching score used when determining which provider to use among several in a search path.

Example 4.5. provider

```
<provider readerType="eclipse.import" ❶
    componentTypes="osgi.bundle,eclipse.feature" ❷
    mutable="false" ❸
    source="false"> ❹
    <uri ❺format="http://download.eclipse.org/eclipse/updates/3.4?importType=binary"/>
</provider>
```

❶ The `readerType` attribute contains the name of a reader for a particular type of repository (a connector to a particular repository type such as CVS, SVN, or p2, or something specialized like the `eclipse.import` reader used in this example).

❷ The `componentTypes` attribute contains a comma separated list of component types this provider can handle. Naturally, this provider will not be considered if a request is made for some other component type.

❸ This is a declaration that this provider is incapable of producing mutable components (i.e. source that can be modified and committed back to a source code repository).

❹ This is a declaration that this provider is incapable of producing buildable source for the component.

❺ This is a declaration of the location of the repository. A `format` attribute contains a string where parameter replacement can take place. Since we are using the `eclipse.import` provider, the protocol is given, the only thing needed is the URI to the location, and an option in the URI that indicates `importType=binary`.

Selection of a reader type, and component types is straight forward, and so is the specification of mutable and source. It starts to get complicated when it comes to the combination of a reader/component type, and the specification of the location. For well structured content with rich meta data, it is as simple as in the example, but it can also get quite complex with very detailed mapping for something available via a download URL.

Reader type

The `readerType` attribute specifies the reader that Buckminster should associate with this provider. The value of this attribute must specify the fully qualified name of a reader[4].

The Buckminster framework provides reference implementations[5] for the following reader types[6] (all which are explained in more detail in subsequent sections).

Available Reader Types

cvs	A reader capable of navigating and reading CVS repositories.
svn	A reader capable of navigating and reading Subversion repositories. The SVN repository reader assumes that any repository contains the three recommended directories trunk, tags and branches.
p4	A reader capable of navigating and reading Perforce repositories.

[4]The name originates in the plugin that provides the reader.
[5]meaning that these implementations could be replaced by other plugins providing a compatible implementation.
[6]The name in bold is the name of the reader as it should appear in the readerType attribute.

maven, **maven2**	The `maven` reader can read Maven 1 repositories, and `maven2` reader has support for reading Maven 2 repositories. Both readers are capable of navigating and reading Maven repositories. It is based on the `url.catalog` reader.
(~~site.feature~~)	Deprecated in Eclipse 3.5. Use `eclipse.import` instead.
eclipse.import	A reader capable of reading anything that can be read by p2 (i.e. p2 repositories, and the older eclipse update sites), as well as being capable of reading a PDE map-file. This reader is replicating the Eclipse SDK capability to import features and plugins from an arbitrary site. NOTE that the type of import (binary, or source) is controlled using a URI parameter in the repository locator.
url	A reader capable of reading one single file appointed by an URL (typically a jar, dll, or other pre-compiled artifact).
url.catalog	Reads files from a specific catalog (folder) appointed by a URL. Currently only supports the `file` URL scheme.
url.zipped	Reads individual files from a zip archive appointed by a URL.
local	The local reader is capable of reading existing components (i.e. previously materialized) just using the component meta-data.

If you are interested in extending Buckminster with a new reader type — there is more information in the section called “Extending Reader Type”.

Note

The different reader types have different capabilities, and use the location URI in different ways. Please consult the specific section for each of the reader types.

cvs reader

The `cvs` reader is capable of reading content from a CVS repository. This reader is integrated with the team CVS support in Eclipse. This means that authentication is integrated, and you have several different options how to handle passwords as described in the section called “Providers and authentication”.

You must have the Buckminster *cvs feature* installed to use this reader.

Here is an example:

```
<provider readerType="cvs"
    componentTypes="eclipse.feature,osgi.bundle,buckminster"
    source="true"
    mutable="true" >
    <uri format=":pserver:anonymous:secret@example.org:/cvsroot/test,teststuff/{0}" > ❶
        <bc:propertyRefkey="buckminster.component" />
    </uri>
</provider>
```

❶ The `cvs` reader can use the `:pserver` protocol to talk to CVS. Here the user named `anonymous`. with password `secret` connects to the CVS root `/cvsroot/test`. It prepends all component names with `/teststuff/` to get to the location. Although it is usually possible to add elements to the root (e.g. `/cvsroot/test/teststuff,{0}`), it is not always possible to do the reverse (e.g. `/cvsroot,test/teststuff/{0}`) as there may be restrictions on access to the directory stated as the root.

SVN reader

The `svn` reader is capable of reading content from a Subversion (SVN) repository. This reader is integrated with the team svn support in Eclipse. This means that authentication is integrated, and you have several different options how to handle passwords as described in the section called “Providers and authentication”.

You must have one of the Buckminster SVN features installed to use this reader. Your choice depends on if you are using Subclipse, or Subversive as your SVN client. See Appendix A, *Installation* for details regarding SVN installation and configuration.

Example 4.6. using svn provider

```
<provider readerType="svn"
    componentTypes="osgi.bundle,eclipse.feature,buckminster"
    mutable="true” source="true”>

    <uri format="http://dev.eclipse.org/svnroot/tools/org.eclipse.buckminster/trunk/{0}↵
¬?moduleAfterTag&amp;moduleAfterBranch">
        <bc:propertyRef key="buckminster.component" />
    </uri>
</provider>
```

The SVN repository reader assumes that any repository contains the three recommended directories `trunk`, `tags`, and `branches`. A missing `tags` directory is interpreted as no tags being available. A missing `branches` directory is interpreted as no branches being available.

Different organization choose to handle the structure under branches and tags differently — i.e. either you find the named things (*modules*) under trunk, tags, and branches, or you find trunk, tags, and branches under the named module. A single project repository typically has trunk, branches, and tags at the top level. Repositories with many top level projects typically use the top level projects as the first level in the repo, with trunk, branches, and tags under each top level project. The SVN connector can not figure this out on its own — it needs a bit of help.,

The URL used as the repository identifier must contain the path element `trunk`. Anything that follows the `trunk` element in the path will be considered a *module* path. The repository URL may also contain a query part where the order of module vs. trunk/branches/tags can be declared. The query part may have four different flags:

moduleBeforeTag	When resolving a tag, put the module name between the `tags` directory and the actual tag (e.g. .../`tags/`*`module`*`/taggedByMary`).
moduleAfterTag	When resolving a tag, append the module name after the actual tag (e.g. .../`tags/taggedByMary/`*`module`*).
moduleBeforeBranch	When resolving a branch, put the module name between the `branches` directory and the actual branch (e.g. .../`branches/`*`module`*`/marysBranch`).
moduleAfterBranch	When resolving a branch, append the module name after the actual branch (e.g. .../`branches/marysBranch/`*`module`*)

A fragment (#) in the repository URL will be treated as a sub-module. It will be appended at the end of the resolved URL (e.g. `/trunk/x?moduleBeforeTag#y` becomes `/tags/x/myTag/y`).

Credentials. The SVN connectors support putting the user name and password directly in the URL using the standard URI notation.

```
http://yourname:yourpassword@example.org/svnroot/...
https://yourname:yourpassword@example.org/svnroot/...
svn://yourname:yourpassword@example.org/svnroot/...
svn+ssh://yourname:yourpassword@example.org/svnroot/...
```

You can naturally use parameters for name and password

```
<uri format="https://{0}:{1}@example.org/svnroot/trunk/{2}";>
    <bc:propertyRef key="example.user" /> ❶
    <bc:propertyRef key="example.password" />
    <bc:propertyRef key="buckminster.component" />
</uri>
```

❶ The `example.user` is a property that is passed in via one of the available mechanism for settings properties. See Chapter 10, *Properties*.

Perforce (P4) reader

The perforce (`p4`) reader is capable of reading content from a perforce repository. You must naturally have the Buckminster perforce connector as well as Perforce itself installed. See Appendix A, *Installation*, for details regarding installation of Perforce support.

```
<uri format="p4://{0}:{1}@example.org/depot/module/{2}";>
    <bc:propertyRef key="example.user" /> ❶
    <bc:propertyRef key="example.password" />
    <bc:propertyRef key="buckminster.component" />
</uri>
```

❶ The example.user is a property that is passed in via one of the available mechanism for settings properties. See Chapter 10, *Properties*.

Maven 1 and 2 readers

The maven reader is used to read binary artifacts that have Maven meta data, in the form of a maven POM file. Use the maven2 reader if the repository is using the Maven 2 format. You must have the Buckminster Maven connector installed to use a Maven reader.[7]

Advanced maven mapping

The maven reader can be given extra information in the provider element to handle mappings between component names and the *maven dual identifier form*[8]. Components mapped from Maven are given a component name where group id and and artifact id are concatenated with a separating /. Normally there is no need to perform mapping between the two forms — but in some cases, like when a component exists in source form with a flat structured name (like x.y.z), it is not possible to automatically map this as it is impossible to determine where the group id ends, and the artifact begins — (it could be `x/y.z` or `x.y/z`).

The advanced mapping requires a maven provider extension kept in a separate XML schema, so the RMAP document should contain the following name space declaration:

```
<rm:rmap
    <!-- other name space declarations go here -->
    xmlns:mp="http://www.eclipse.org/buckminster/MavenProvider-1.0";
>
```

To use these, the provider element itself must be declared to use:

```
xi:type="mp:MavenProvider"
```

(as the normal `provider` element does not allow the maven extension elements as children), then to add the mapping — place a `mp:mappings` entry in the provider, with child `mp:entry` elements, one per mapping. (In case there is any doubt, the `mp:entry name` attribute is the component name).

```
<provider
    xsi:type="mp:MavenProvider"
    readerType="maven"
    componentTypes="maven,osgi.bundle" mutable="false" source="false">
    <uri format="http://repo1.maven.org/maven2"/>
    <mp:mappings>
        <mp:entry
```

[7]A contribution to Buckminster from the Maven project to give Buckminster full maven 2 is currently stuck in Eclipse IP review.
[8]Maven uses a group identity and an artifact identity as the unique component identity.

```
                name="org.apache.activemq.core"
                groupId="org.apache.activemq"
                artifactId="activemq-core"/>
            <mp:entry
                name="slf4j.log4j12"
                groupId="org.slf4j"
                artifactId="slf4j-log4j12"/>
        </mp:mappings>
        <!-- more stuff ... -->
    </provider>
```

Eclipse import reader

The `eclipse.import` reader can read anything that can be read with Equinox p2 (see Chapter 2, *p2*). In its standard configuration, p2 is capable of reading both p2 repositories and older update sites. It is also possible to extend p2 with other types of repositories, but this is transparent to Buckminster.

Note

The type of import (binary, or source) is controlled using a URI query parameter in the repository locator.

An example is shown in Example 4.5, “provider”. Note that the URL uses an URL query to define if the import is binary or source (`importType="source"`, or `importType="binary"`).

The `eclipse.import` reader mimics the behavior of the Eclipse IDE’s ‘*import features or plugins*’. The result of the import is that a project is created in your workspace and the class path of that project is set up to include any jar file included in the project. If the plugin has source, the source will be unfolded into the project.

The `eclipse.import` reader is also capable of reading PDE maps that use the the map file formats referred to in PDE-build as: “*Map file entry for CVS*”, “*Map file entry for other repositories*” and “*ANT ‘`GET, url`’-format*”[9], and treat them as a repository. One such map is found at:

```
http://download.eclipse.org/tools/orbit/downloads/drops/R20090529135407/orbitBundles-R20090529135407.map
```

It has content that looks like this (partial sample):

```
plugin@com.ibm.icu,3.6.1=GET,http://download.eclipse.org/tools/orbit/downloads/↵
¬drops/R20090529135407/bundles/com.ibm.icu_3.6.1.v20080530.jar
plugin@com.ibm.icu,3.6.0=GET,http://download.eclipse.org/tools/orbit/downloads/↵
¬drops/R20090529135407/bundles/com.ibm.icu_3.6.0.v20080530.jar
plugin@com.ibm.icu,4.0.1=GET,http://download.eclipse.org/tools/orbit/downloads/↵
¬drops/R20090529135407/bundles/com.ibm.icu_4.0.1.v20090415.jar
plugin@com.ibm.icu,4.0.0=GET,http://download.eclipse.org/tools/orbit/downloads/↵
¬drops/R20090529135407/bundles/com.ibm.icu_4.0.0.v20081201.jar
plugin@com.ibm.icu,3.8.1=GET,http://download.eclipse.org/tools/orbit/downloads/↵
¬drops/R20090529135407/bundles/com.ibm.icu_3.8.1.v20081217.jar
plugin@com.ibm.icu,3.4.5=GET,http://download.eclipse.org/tools/orbit/downloads/↵
¬drops/R20090529135407/bundles/com.ibm.icu_3.4.5.jar
```

This behavior in the `eclipse.import` reader is triggered if the URI ends with ‘`.map`’ .

URL reader

The `url` reader is used in situations where you want to get a single artifact. You can either refer directly to the component with the `uri` element, or use a `matcher` element to parse/scrape the content of the URL to get a list of possible URLs to components to match against. A good example is a ftp URL to a directory. (Contrast this with that you would have to specify a separate provider/reader for each separate file).

[9]You can read more about PDE map files in this Eclipse Help file for Eclipse 3.5 called PDE Build Advanced Topics/Fetching from Repositories [http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.pde.doc.user/tasks/pde_fetch_phase.htm]

The matcher is quite smart — an attempt is made to read the content as valid XML (many sites deliver what is stated to be XML or XHTML, but it may still not be valid), and if this fails, the content is treated as “rogue” HTML and is scanned as flat text. In both cases (structured valid XML, or flat text), a search is made for `<a href="...">` elements, and the referenced URLs are collected into a list of “catalog content”.

You still have to pick something from the resulting list of URLs, and this is done with a `matcher` element. A `matcher` is a powerful mechanism[10], and is explained in more detail in the section called “Handling indirection”.

Example 4.7. url reader

```
<provider readerType="url" ❶
    componentTypes="jar" ❷
    mutable="false" ❸
    source="false"> ❹
    <uri format="${matchedURL}"/> ❺
    <matcher ❻
        base="http://mirrors.ibiblio.org/pub/mirrors/apache/myfaces/binaries/" ❼
        versionFormat="..."> ❽
        <match name="matchedURL" ... /> ❾
    </matcher>
</provider>
```

❶ The reader type is set to `url`

❷ We expect the found URLs to lead to `jar` files

❸❹ We set both `source` and `mutable` to false as we do not expect to find references to source projects that we can bind to the workspace, nor make changes and commit them back.

❺ We expect to be reading the component at a URI that gets its value in the parameter `matchedURL` — see ❾.

❻ A `matcher` element is used — it defines the base URL (i.e. the “page to read”) and how a resulting version string should be turned into a processable version.

❼ The `base` is the URL to the “page” to read — here a page from ibiblio.org, containing a mirrored apache repository. A sample is show in the figure below, but you can visit the URL in a browser to see its full content.

Figure 4.1. Sample content from an ibiblio page

```
myfaces-commons12-1.0.0-bin.tar.gz.asc        15-Sep-2008 19:57  197
myfaces-commons12-1.0.0-bin.zip               15-Sep-2008 19:57  1.2M
myfaces-commons12-1.0.0-bin.zip.asc           15-Sep-2008 19:57  197
myfaces-core-1.1.6-bin.tar.gz                 16-Aug-2008 13:01  2.9M
myfaces-core-1.1.6-bin.tar.gz.asc             16-Aug-2008 13:01  197
myfaces-core-1.1.6-bin.zip                    16-Aug-2008 13:01  4.7M
myfaces-core-1.1.6-bin.zip.asc                16-Aug-2008 13:01  197
myfaces-core-1.1.7-bin.tar.gz                 05-Jun-2009 15:06  2.9M

// ...
```

❽ In this example we expect the versions used on the page to use a non OSGi version format. See Chapter 9, *Versions*, for more information about version formats. If no version format is specified, the version strings must be in a format that can be directly parsed by omni version — i.e. OSGi or omni version’s raw format.

❾ We use a `match` element to search through the list of generated links found at the matcher’s `base` URL — the final URL that is matched becomes the value of the property declared in the match element’s `name` attribute — i.e. `matchedURL`. (The body of the match element is not shown here, only how the result of a `match` is returned to ❺. In fact, to really parse the page in this example, we need to use something more powerful than the `match` element — but the principle is the same. The real matching is shown in later examples).

[10]Which is a nicer way to say that they are a bit complex...

URL catalog reader

The `url.catalog` reader is used in situations where you have a content in your file system[11] in a format suitable for interpretation by one of the available component types.

Note

The `uri` attribute should be a reference to a directory.

URL zipped reader

The `url.zipped` reader is used in situations when the component you are interested in is an item inside a zip file. The content of the zip file should be in a format that can be understood by the selected component type.

Note

The `uri` attribute should be a reference to the zip file.

Local reader

The `local` reader is used in situations when the material you are interested in already exists in the file system in the form you want. In this case, Buckminster does not materialize anything (i.e. no downloads or copying takes place — contrast this with the `url.catalog` reader). Instead, Buckminster simply reads the meta data available in the appointed location and binds the location into the workspace as a project.

You can use this mechanism for different purposes — here are some of the things we know this has been used for:

Using an unsupported Source Code Control System. Buckminster currently supports CVS, SVN and Perforce, but you may already be using some other source code control system and do not want to implement support for your tool. What you can do is to use your tool to create the wanted layout in your file system (i.e. check things out, or whatever the terminology is in the tool you are using), and then simply point to that location using the local reader.

Performance Optimization. When Buckminster is used as part of a larger build systems, other actions may already have materialized many of the things needed. Materializing them again just wastes cycles when you are sure the material is already there. It does not matter how the material ended up in the location — it could have been Buckminster that materialized them earlier.

Structured Download. You are using material downloaded from different sites such as individual jar files. You can point to them directly in your RMAP, but this means that they will be downloaded from the original site each time they are requested (and not in your cache). Instead, you may want to set up a separate Buckminster materialization of the things you will be using in your organization. The materialized result can then be made available in different forms. Many use a distributed file system — and can hence be accessed with the local reader.

Note

The `uri` attribute should be an absolute file system path without the scheme.

Providers and authentication

Some providers/readers may require authentication to connect to the repository. You have different options in this area and the solution to use depends on your security requirements, and the type of credentials required.

[11] Currently, only the file: scheme is supported.

Interactive use. When Buckminster is used from within the Eclipse SDK the authentication is handled the same way as when the repository is used directly. If you have already connected to a repository (say using the team CVS or SVN functionality) and choose to have Eclipse remember the credentials then you will not be prompted again for the same repositories. If the Buckminster action triggers request for credentials, it will be handled and remembered the same way as when using the team functionality. In essence, there is nothing extra/special that you need to do.

Credentials in the RMAP. You can store the credentials directly in the RMAP. It is not as crazy and unsecure as it may sound since this is very useful in cases where you do need to authenticate, but the user and password are a publicly known — i.e. “anonymous/anonymous” or similar. Also, if you take care protecting the RMAP it may still be a viable solution even if the RMAP has user name and password in clear text.

Credentials in template. You can pre-populate a workspace and keystore by using the SDK, and then distribute this “template configuration” to the servers where you are building headlessly. You still have to protect against someone using these templates to run unauthorized actions. What you need to include in the template may wary as readers may be different in their handling of storing credentials.

Credentials in properties. You can store the credentials in properties and use property values in the RMAP. This way the RMAP is reusable, and different users can supply their own properties.

Component types

As you may recall, *component types* allows you to specify a list of the types of components a provider is capable of producing. Each component type is a fully qualified name of an implementation that provides translation of the native/external meta data associated with the component to the internal format used by Buckminster.

It is possible to extend the component types as described in the section called “Extending Component Type”.

Buckminster has reference implementations[12] for the following component types (see the reference guide “Component Types“ for details):

component types

osgi.bundle	Standard component type for software assets maintained with Eclipse (i.e. something that has an Eclipse `.project` or `plugin.xml` files). Essentially, an Eclipse plugin.
eclipse.feature	An Eclipse feature component.
(~~eclipse.site~~)	Deprecated.
jar	Refers to components that are JAR files and can be treated as components in their own right. Buckminster will generate a CSPEC that has no dependencies. This type is intended to be used when you want to depend on a JAR known to be found using a common URL.
maven	Basically an extended JAR type but assuming Maven repository dependency information contained in a maven POM.
buckminster	Refers to software assets that have no derivable component specification information, or where a plugin for the particular component type has not been created. Assumes that the own-

[12]these are referred to as reference implementation since they can be replaced by a compatible implementation.

er of the asset has added a manually created (and maintained) CSPEC inside the software asset. The reader will expect an included `buckminster.cspec` to contain the meta data.

(~~site.feature~~)

Deprecated.

bom

Refers to a component which is a Buckminster *Bill of Materials* (BOM) artifact as produced by the Buckminster resolution process. When this is used, the resolution process accepts the referenced BOM as the resolved solution.

This is very useful in situations where a dynamic resolution is unsuitable. As an example the component '`org.sloppy.enfant.terrible`' may be difficult to resolve as many special paths needs to be taken through the RMAP. With a pre-resolved "static" BOM there is no need to repeatedly specify the complicated advisor nodes and property settings required to make the poorly specified enfant.terrible resolve — just because it is required by other components. When (eventually) the bad component is fixed, it is easy to switch back to a dynamic approach again since the change takes place only in the RMAP.

Another example is when some other system is producing a configuration, and it should be used "as is" — rather than trying to transform this system's meta data into Buckminster component specifications and then letting Buckminster repeat the resolution process, it is instead possible to directly produce the BOM. This can be especially useful when interacting with or migrating from a legacy dependency management system.

Tip

Use the `bom` type when you don't want dynamic resolution of everything.

unknown

Indicates components for which no dependency information can be inferred or has been made available. The component is still useful, but it only has a name.

Advice regarding components with no meta data

If you need to handle components where there is no meta data, or the meta data is not in a form that can be handled by Buckminster, you still have a few options.

If you are the owner of the component, or can persuade the owner — the meta data can be added where it should be added — at the source. It does not matter which meta data format the component owner adds — it could be OSGi, or Maven, just as well as Buckminster meta data.

If it is not possible to add the meta data at the source. The component type `unknown` can be used in a provider, but a separate component is needed to keep track of the dependencies. In many cases, you probably have a configuration of such components that should go together, so you can probably create a feature/grouping component to reference the component with unknown meta data, and then use this group component elsewhere in your system.

You can naturally also repackage a component, include the required metadata, and then distribute it instead of the original. Many do this even if it requires maintaining an internal version of the same component (with the obvious problems if it is mixed with external packages that does not use the internally repacked version).

Version converter

When Buckminster gets components from a source code repository and interprets their version, there are three cases to consider:

- There is meta data in the component that specifies the version to use (e.g. `osgi.bundle`), and you want to use the component reader’s ability to set this version as the version of the component.
- There is meta data, but you want the name of a branch or a tag to reflect the version of the component
- There is no meta data in the component, and the only choice is to derive it from a branch or a tag.

You can handle this by using a version converter in the provider specification. You have to decide if you want the transformation to be based on a branch or a tag, and the version format (i.e. version type). You then have to specify how a version such as triplet `3.0.1` is translated into a branch or tag name (perhaps to `v3_0_1`). You also need to specify the reverse — how to transform a branch or tag name into a valid version string for the selected version type — e.g. how to go from `v3_0_1` to triplet `3.0.1`.

Example 4.8. version converter

```
<versionConverter type="branch" versionFormat="..." > ❶
    <transform ❷
        fromPattern="\." ❸
        fromReplacement="_"
        toPattern="_"
        toReplacement="." ❹
        />
    <transform ❺
        fromPattern="^(.*)$" ❻
        fromReplacement="v$1" ❼
        toPattern="^v(.*)$" ❽
        toReplacement="$1" ❾
        />

</versionConverter>
```

❶ A version converter is declared to convert versions on a `branch` — the `versionFormat` attribute is not needed if the version is of OSGi type (or if the version happens to be in the Omni Version raw format). Otherwise, a version format should be used — see Chapter 9, *Versions*.

❷❺ Two transformer elements are used — the first handles transformation between ‘.’ and ‘_’ as separator, and the second transforms between a prepended ‘v’ and “no v”.

❸ Since this is a regexp pattern the literal period ‘`.`’ must be escaped with `\.` When doing the reverse at ❹, the replace string is not a regexp — and the escape is not needed.

❻ This pattern makes everything; ‘`.*`’, between the beginning `^` and the end `$` captured in a regexp segment (the ‘`( )`‘)

❼ The segment from ❻ is used in the replacement string (i.e. ‘$1’)

❽❾ This pattern should look familiar — everything is matched in a segment except the ‘v’, and the replacement is the matched segment.

Tip

The terms ‘to’ and ‘from’ are quite confusing as the transformation is bidirectional. What you have to repeat to yourself are “I am converting ***from*** *a version* to a branch/tag-name using `from`”, and “I am converting ***to*** *a version* from a branch/tag-name using `to`”.

You can extend Buckminster with additional version converter types (if branch and tag are not enough). See the section called “Extending Version Converter”.

Handling indirection

As you have seen in the examples up to this point the locations of repositories (or a single file as in the case of the `url` reader) have been known, and we have simply entered the location in the `uri` element parameterized with the component name. But there are many situation where the only known address is to a web page where downloadable items are listed. In this section we take a look at how you can handle this situation without having to periodically and manually revisit the web-page with the listing and then manually update the RMAP. (If you are looking for a mechanism to handle indirection to source code, see the section called “PDE map — extended provider”).

Buckminster has a *matcher* that operates as a “content-scraper”, picking out URLs and matching them against patterns to determine if they represent a component.

The primary intended use for the matcher is for picking a URL to a single artifact, but it can also be used if you find yourself in the odd situation where the only way to get a repository URL is via an indirection.

You can ignore this entire section if you are just skim reading, it is all about details how to write patterns that pick out the interesting parts from URLs — perhaps looking something like this:

```
http://someForge.org/downloads/download.php?project=eggnogg&file=eggnogg_1_0_0-osx-x86-en.tar.gz
```

Content is matched using a `matcher` element that contains a regular expression composed out of a structure of `match` and `group` elements.

The matcher

The `matcher` element defines the content to scan, and the version format to use when converting a found version string into a version. The version format can be omitted if an OSGi versioning scheme is used (or if the version string happens to already be in the Omni Version raw format). For other formats see Chapter 9, *Versions*.

```
<matcher base="http://someForge.org/downloads/view.php?project=eggnogg"
    versionFormat="..." >
    <!-- match and group elements go here -->
</matcher>
```

The nested `match` and `group` elements are used to compose a regular expression.

Although it would be possible to write a single regular expression to do all the matching, the resulting expression containing many segments would be very hard to write, and even harder to read — it would also require figuring out the segment indexes to use when putting the mapped pieces together.

Figure 4.2. matcher principle

```
<matcher ... >
    <match pattern "a" />
    <match pattern "b" />
    <group name="theNumbers" >
        <match pattern "1" />
        <match name="theTwo" pattern "2" />
    </group>
</matcher>
```

In Figure 4.2, “matcher principle”, a very simple regular expression is constructed that matches a string containing “ab12” — the construction is equivalent to the regular expression `ab(1(2))`, and in addition the property `theNumbers` gets associated with the segment matching `12`, and the property named `theTwo` is associated with the segment matching the digit `2`.

So, the `matcher` element, as well as the `group` element, creates a regular expression pattern out of its children by simply concatenating them (in the order they are stated).

The match element. The `match` element defines part (a fragment) of the regular expression. It also has a convenient way to declare literal prefix and postfix string matches where special characters does not have to be escaped.

Figure 4.3. match element

```
<match
    name="aPart" ❶
    pattern="[a-z0-9_]+" ❷
    optional="true" ❸
    prefix="http://somewhere.org/downloads/download.php?file=" ❹
    postfix="&format=binary" ❺
/>
```

❶ `name` can be used to make the matched result available in a property — here the value will be available in the property `aPart`.

❷ The `pattern` is a required attribute (all others are optional) and it used for a regular expression pattern fragment. Special characters must be escaped with \ if they are to be used literally.

❸ The `optional` attribute indicates if the the entire match defines an optional part in the overall matching pattern it is part of. The default is `false`.

❹ The `prefix` is a literal prefix. The effect of using `prefix` is the same as if the prefix string was at the beginning of the pattern, but with all special characters escaped (i.e. there is no need to escape special characters in the `prefix`). In the figure, the first part of the URL is matched this way.

❺ Similar to `prefix`, but at the end. In the figure, an extra parameter in the URL is matched.

The group element. The `group` element is basically just a grouping that may have a name. The resulting pattern is the concatenation of the `match` and nested `group` elements contained in the group.

Extracting meta data from the URL. As a scan for a URL to a component takes place, a mechanism is needed to determine if the URL potentially is a match (with respect to version, architecture, os, language, etc.) without having to read all the contents and look for meta data. It may even be the case that there is no meta data inside the component, so all the search mechanism has is the URL string. So, these elements have dual use — they act as part of the textual match (*is this a URL of interest at all*), and the resolution (*should this component be selected*). The metadata-extracting elements can only be used in a `group` element, and their individual names can not be set. Several of the elements define values for a *target filter* — see Filters. Even if these elements have predefined meaning, their patterns *must* be defined.

meta data extracting elements

arch	Matches and sets the target filter for *architecture*. Example `x86`.
os	Matches and sets the target filter for *operating system*. Example `macosx`.
nl	Matches and sets the target filter for *natural language*. Example `en_US`.
ws	Matches and sets the target filter for *windowing system*. Example `cocoa`.
name	Matches and sets the *name* of the matched component candidate.
version	Matches and sets the *version* of the matched component candidate.
revision	Matches and sets *revision information* for the component candidate. The revision is used when a request is made for components in a particular revision.
timestamp	Matches and sets *timestamp information* for the component candidate. The timestamp is used when a request is made for components having a particular timestamp.

Note that using matcher’s `revision` and `timestamp` extractors is equivalent to what takes place in the source code control repository readers when a file is found —there the meta data regarding timestamp and revision is always available and is always provided to the resolution process. If selection is made based on these values is a different issue.

PDE map — extended provider

The `PDEMapProvider` is a provider extension that allows a PDE releng source map to be used as an indirect declaration of `cvs` and `svn` readers. To use this extension, the XML schema must be declared.

```
<rm:rmap
    <!-- other name space declarations go here -->
    xmlns:xi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:pmp="http://www.eclipse.org/buckminster/PDEMapProvider-1.0"
>
```

Example 4.9. Using PDEMapProvider

```
<provider xi:type="pmp:PDEMapProvider" ❶
    readerType="cvs" ❷
    componentTypes="osgi.bundle,eclipse.feature" ❸
    mutable="false" source="true"> ❹
    <uri format=":pserver:anonymous@dev.eclipse.org:/cvsroot/rt,↲
        ¬org.eclipse.ecf/releng/org.eclipse.ecf.releng.maps" ❺
    />
</provider>
```

❶ The `provider` element is declared to be an extended element of type `PDEMapProvider`.
❷ The `readerType` is a reference to the reader used to read the *releng map*.
❸ The `componentTypes` refer to the component types looked up in what is *mapped in the releng map*.
❹ The `mutable` and `source` attributes refers to the content *mapped in the releng map*.
❺ The `uri` refers to the location of the *releng map* — the releng map has a format defined by PDE and is a map from component names to source code locations in CVS and SVN.

Note

The `PDEMapProvider` extension can be used with any reader type capable of producing a single file (e.g. `cvs`, `svn`, `p4`, and `url` readers).

Properties

We have already used properties throughout this chapter, but there is much more you can do with properties. Property capabilities are shared across RMAP and CQUERY artifacts, and are therefore covered in a separate chapter. See Chapter 10, *Properties*, for details.

Properties are declared at the top level in the RMAP document, but the properties declared there are not the only properties available when the map is used by the Buckminster resolution process — the system properties, properties defined in the CQUERY being resolved, etc. can all be used.

As an example, you can set a property like this:

```
<property key="buildType" value="RBUILD" />
```

The RMAP XML document

The RMAP XML document needs to be declared to be an XML document, and the schemas used should be declared along with a namespace. Here is a declaration that includes all schemas (the `xi`, `mp`, `pmp` are optional).

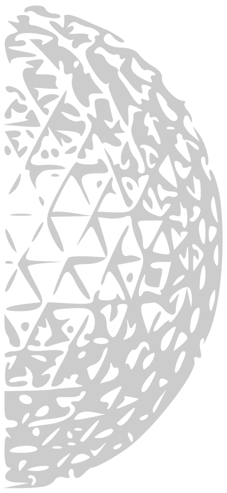
```
<?xml version="1.0" encoding="UTF-8"?>
<rm:rmap
    xmlns:xi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rm="http://www.eclipse.org/buckminster/RMap-1.0"
    xmlns:mp="http://www.eclipse.org/buckminster/MavenProvider-1.0"
    xmlns:pmp="http://www.eclipse.org/buckminster/PDEMapProvider-1.0"
    xmlns:bc="http://www.eclipse.org/buckminster/Common-1.0"
>
<!-- The definition of the RMAP -->
</rm:rmap>
```

For more information about schemas see Buckminster XML Schemas.

Summary

As you have seen in this chapter, setting up an RMAP can be as simple as entering a handful of lines in XML, at the same time as the RMAP has a lot of expressive power to deal with the real world complexities of locating components, handling underspecified and erroneous meta data, and handling the required flexibilities when requiring different configurations of a particular set of components taken from tags or branches in source code repositories, or from different binary repositories.

You will find complete annotated RMAP examples in Part III, “Examples”.



5 Component query

In this chapter we take a closer look at the Buckminster *Component Query* (CQUERY), and how it specifies the request to get components — covering both the simple bucky-meal way ("I am hungry, never mind where the cheese came from..."), as well as the the deli-counter equilibrist meal order ("...only include cheese from milk produced from macrobiotically grazed sheep with ISKCS — Q, G or R karyotypes, please").

You can start issuing component queries as soon as you have installed Buckminster. The queries will then be resolved against the content in your Eclipse workspace, and target platform. This is possible since Buckminster scans the environment, and keeps itself updated as your configuration or workspace content changes. Buckminster can not however correctly guess where everything came from, nor does it have any knowledge about any repositories. In order to make your queries resolve against repositories (source code or binary), you need to construct a Buckminster *Resource Map* (RMAP) as described in Chapter 4, *Resource Map*.

One query to get them all...

Running a CQUERY is what starts off the resolution and provisioning process carried out by Buckminster. A CQUERY is expressed in XML and is stored in a file ending with '`.cquery`'. You can either use an existing file, perhaps one published on a web site, or you need to create a new.

When querying, you always query for a (single) root component, and Buckminster will resolve all of its dependencies. As soon as you entered the query, you can execute it, it is possible to preview and modify the result, you may also save the resulting resolution in a *Bill of Materials* (BOM) file that can be used in various ways — see Chapter 7, *Bill of Materials (BOM)*. As the final step the query can also carry out the materialization step — i.e. perform the creation of material in your workspace, install into your Eclipse IDE, or target platform, or place files in your file system.

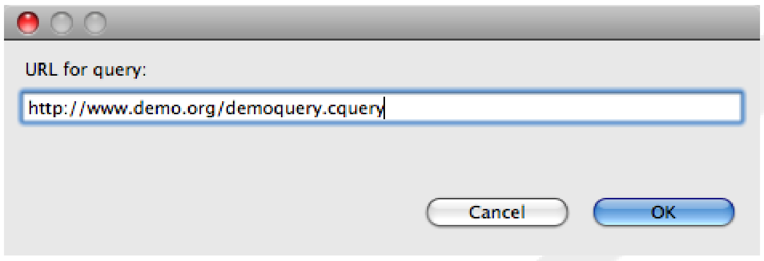
Most of the time, there is already some master feature that defines your product, or main feature that includes everything that is required. But sometimes, you may need to construct this extra configuration component. It does not matter which one of the supported component types you use for such a configuration, but we recommend the use of a plain java project with Buckminster meta data, or an Eclipse feature-project depending on the circumstances.

Opening an Existing CQUERY

You can open and execute existing CQUERY files. When Buckminster is installed in your Eclipse IDE, it understands that files with the suffix '`.cquery`' should be opened with the CQUERY editor[1]. This

[1]The CQUERY file itself is in XML format, and may be edited with an XML editor, but you should not have to do that unless you have a craving for a dose of XML.

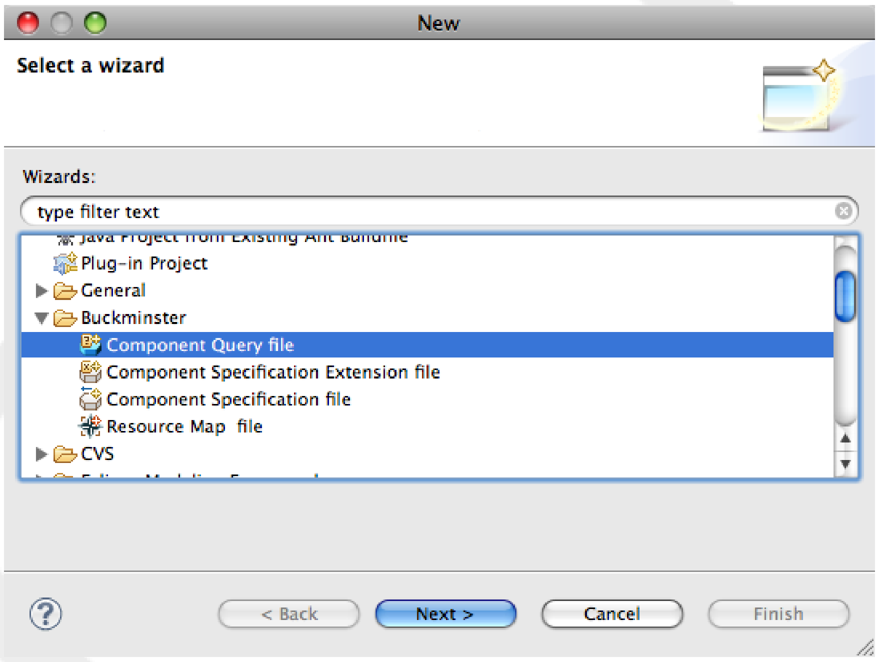
applies to files in your workspace, in your local file system, and general URLs[2]. If you have a URL to a CQUERY file you use the *File* → *Open a Component Query...* which opens a small dialog where you can enter the URL. (If you are running on Windows, you can use *File* → *Open File...* as it also opens a URL).


URL for query:

http://www.demo.org/demoquery.cquery

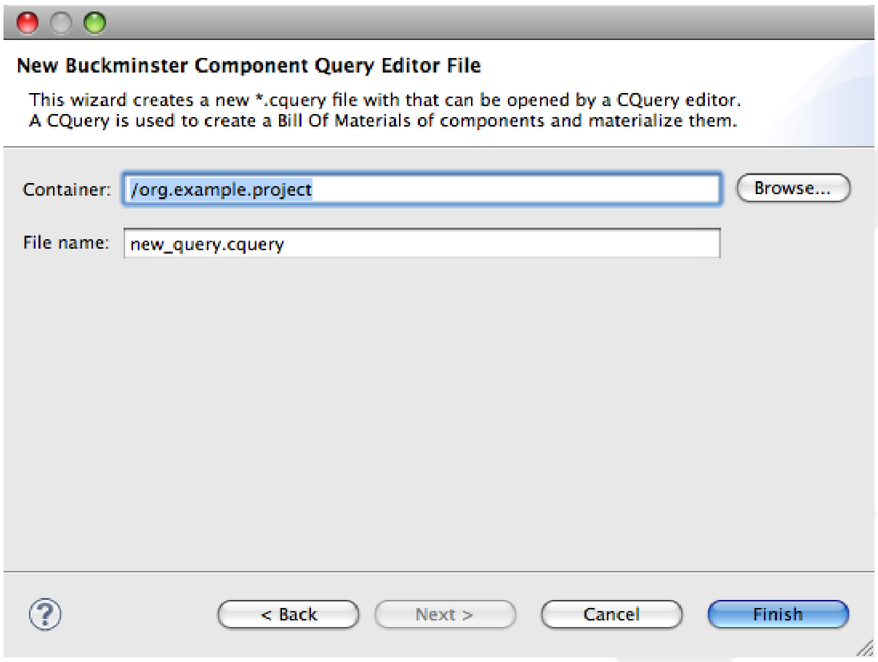
Creating a new CQUERY

A new CQUERY is created by using *File* → *New* → *Other...* → *Buckminster* → ######### ##### #### .



Click on *Next*, and enter (or browse the workspace and select) the location for the file. You can name the file anything, but you must keep the extension '`.cquery`', or Buckminster will not associate it with the correct editor.

[2]i.e. you do not have to separately download them to disk, and then open them as files.


New Buckminster Component Query Editor File

This wizard creates a new *.cquery file with that can be opened by a CQuery editor.
A CQuery is used to create a Bill Of Materials of components and materialize them.

Container: /org.example.project

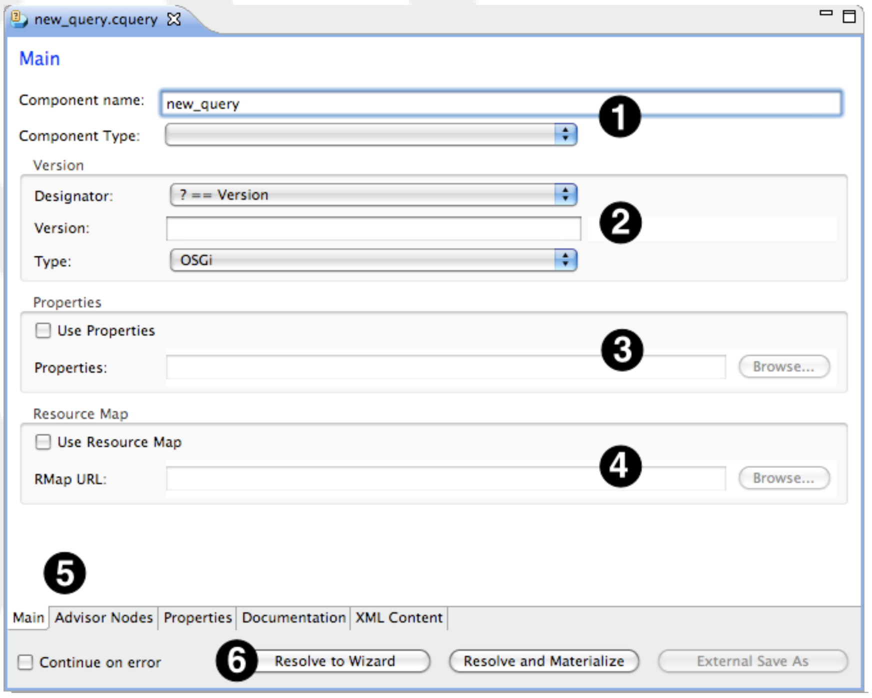
File name: new_query.cquery

The CQUERY Editor

The CQUERY editor allows you to edit all aspects of the CQUERY, and is also used to execute the query. The editor is associated with files ending with '`.cquery`', and open automatically as the default editor for such files.

When executing the query, it is possible to preview the result, edit the result, and also to save choices made regarding edits, and installation destinations.

The editor main tab


new_query.cquery

Main

Component name: new_query

Component Type:

Version

Designator: ? == Version

Version:

Type: OSGi

Properties

Use Properties

Properties:

Resource Map

Use Resource Map

RMap URL:

Continue on error

❶ **Name and Component type.** This is where you enter the name of the (top/root) component, and select the component type from the drop down list. (When the editor opens for a newly created file, the *Component name* reflects the filename (here `new_query`), which you need to change to the name of the component you want to materialize).

❷ **Version, range and version type.** In this section you can enter a version, or version range for the component you are requesting. The drop down has entries for ==, >=, and four different ‘between’ entries (i.e. if *from* and *to* should be inclusive or not). When values for both *from* and *to* are required, an extra field appears. If you leave version empty, the default search is for the latest available version.

❸ **Property File.** There are several ways to set properties (see Chapter 10, *Properties*). This mechanism allows you to set properties from a properties file. The properties set this way override properties in the CQUERY itself. If the properties file does not exist, it is ignored (but the issue is logged). One use of this could be to create a “dormant” override system; with a reference to the properties like `${user.home}/.projectX.properties`, then users can provide their property settings (like user name and passwords to repositories) in that file. The reference is a URI, so it is possible to reference a properties file using a URL.

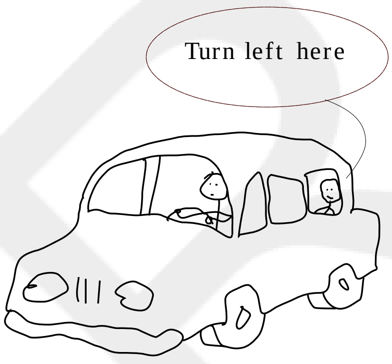
❹ **Resource Map.** This section allows you to reference a resource map (RMAP) that maps between component names and repositories. It is possible to enter a URI, and system properties like `${user.home}` can be used. If a RMAP is not used, the resolution takes place against the content of the workspace and target platform.

❺ **Editor Tabs.** The tabs along the bottom gives access to editing the details of the query — the final tab allows the resulting XML to be viewed. All the tabs are explained in more detail in the later sections.

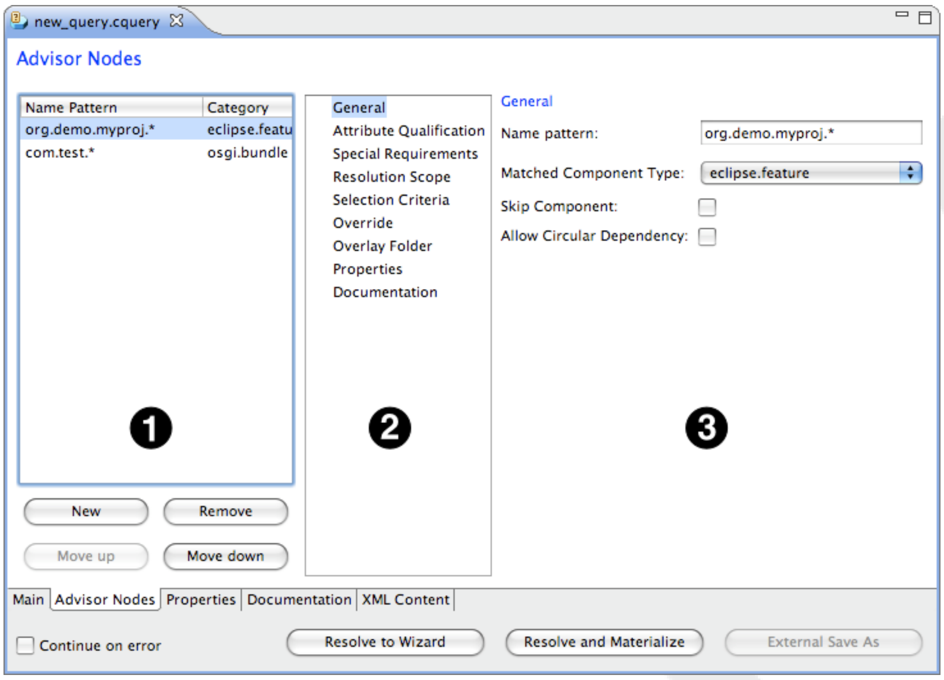
❻ **Processing.** The processing section allows you to run the query — all in one go, or interactively. There are also option to save the result. A check box allows you to control if the process should stop as soon as an error occurs, of if the process should continue to the end anyway.

Advisor nodes

The Advisor Nodes tab allows you to edit the CQUERY *Advisor Nodes*. They are so named because they provide advice to the Buckminster resolution process.



You only need to provide information in advisor nodes if you want to modify how an individual component (or groups of components) are handled. Examples include setting properties to control that some components should be picked from a nightly build repository rather than the release repository, overriding dependency metadata (widening or constraining ranges, skipping an unwanted version, etc), selecting a subset of a component’s full dependency graph, and much more.



Advisor node tab parts

❶ **Node list.** The node list shows a list of advisor nodes. Each entry is identified by the name pattern the node is set up to match — in this example you see `org.demo.myprog.*` and `com.test.*` which tells you there are two nodes. The category column reflects component type (e.g. `eclipse.feature`, `osgi.bundle`, `buckminster`, `maven`, etc.). You can create a new node, remove a node, or rearrange the order of the nodes with the buttons at the bottom of the list.

❷ **'Attribute group selector'.** The attribute group selector lets you see a group of advisor node attributes at a time — as you can see there are 8 different entries, and there are several options behind each — simply click on an entry, and the corresponding fields and values are shown to the right of the list. In the example the 'General' attributes group was selected. (Note that all attributes are for the currently selected advisor node).

❸ **Node Attributes.** This section shows the attributes in the selected attribute group. In the example the 'General' attributes are selected, and here you see the *name pattern* (a regular expression pattern), *matched component type*, if matching components should be *skipped*, and if *circular dependencies* should be allowed or not.

General attributes

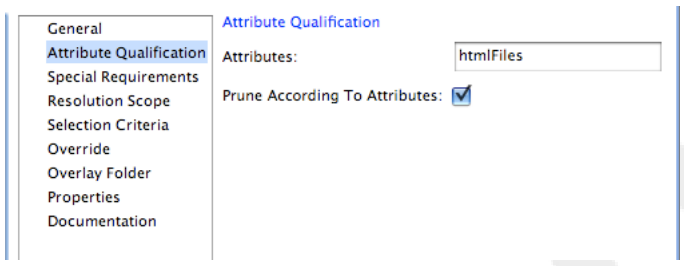
You can see a screenshot of the general attributes in 'the section called "Advisor nodes"', at 'Node Attributes'. The general attributes are used as follows:

Name Pattern	This is a regular expression pattern that is used to select the components that should receive the advice provided in this advisor node. You need to consider the order — the first matching advisor node is used, and the remaining nodes are not consulted.
Component Type	You can also match on component type — a drop down list lets you select from the list of component types known to your current Buckminster configuration. (If you provide extensions to buckminster, make sure the extensions are installed when editing, or you will not be able to select your extensions). If you leave this blank, the node will match any component type.
Skip Component	If you check skip component, then any component that matches the name pattern, and component type (if any), will be exclud-

	ed from the resulting resolution, and the excluded component's dependencies are not resolved. This is useful when a configuration brings in a component with broken/faulty/old dependencies (and you are going to fix the problem), or when a component is brought in and you know it is not used.
Allow Circular Dependency	Normally, circular dependencies are treated as an error as it makes it impossible to determine the correct build order. By checking this box, a dependency that references an ancestor will simply be ignored.

Attribute qualification

The *attribute qualification* allows you to only resolve the subset of dependencies that are required by the stated attributes. By default, all dependencies are resolved. This is useful when you only need the value of a particular attribute from a component, and do not want to materialize everything the component depends on (e.g. runtime parts, test data). You can enter several attributes separated by comma.



Attributes: htmlFiles
Prune According To Attributes: ✓

Attributes
: A list of attributes for which the dependencies should be resolved.

Prune According to Attributes
: An advanced setting that results in a pruned component specification. Do not check this unless you know exactly what you are doing.

Advanced Topic — Attribute Pruning

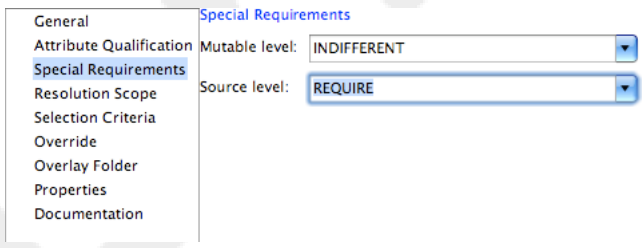
Pruning attributes means that the resulting component specification CSPEC's content is reduced to only include requested attributes and the dependencies required by those attributes.

Note that materialization to a workspace in combination with pruning is quite meaningless since the component meta data is kept in sync with the component's actual content — the pruning performed in the resolution process is thereby lost.

Unless you have very special needs, you should not use pruning.

Special requirements

The *special requirements*[3] lets you control how to deal with source vs. binary, and the shape of the source (just available, or as mutable/'modifiable' projects).



Mutable level: INDIFFERENT
Source level: REQUIRE

Mutable level
: Controls how the resolution should make a choice between mutable and non mutable components.

Source level
: Controls how the resolution should make a choice between component in source or binary form.

[3]This is a really bad term — there is nothing special about these requirements at all — think of them as 'source requirements'.

The possible values for mutable level and source level are:

INDIFFERENT
: The resolution process is allowed to pick whatever it thinks is best. (This is the default, and the default if you have no advisor node at all).

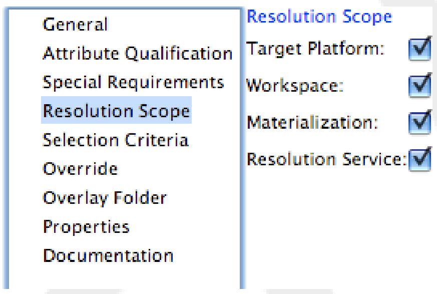
REJECT
: The resolution process is not allowed to select a component with this trait (i.e. mutable or source form).

DESIRE
: The resolution process should value a component with this trait higher (i.e. deliver mutable/source if it is available).

REQUIRE
: The component must have this trait. The resolution will fail if such a component is not found (even if it exists without the trait).

Resolution scope

The Resolution Scope lets you control what resources the resolution process should consider when performing the resolution.



Target Platform: ✓
Workspace: ✓
Materialization: ✓
Resolution Service: ✓

Target Platform
: Should components found in the target platform be used.

Workspace
: Should components found in the workspace be used.

Materialization
: Should Buckminster materializations be used (i.e. things previously downloaded).

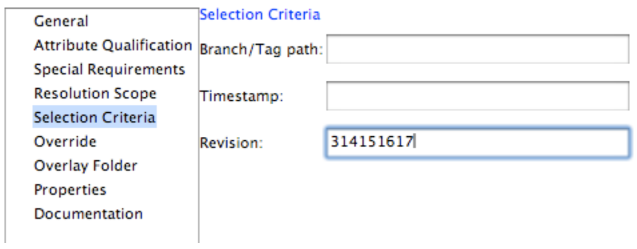
Resolution Service
: Should Buckminster talk to a map service (see below).

The default is to include all in the resolution scope. Some common scenarios where it is important to control the scope are: — when you do not want to find binary versions in the target platform when you are working on code that should go into the platform and need to have them in binary form in your workspace — skip earlier materializations in preference of contacting repositories again.

Resolution Service. Buckminster can talk to a RMAP service via a JSON protocol. It is possible to turn the resolution service on/off in Buckminster preferences, see the section called “Preferences”, and to specify a provider of such a service. The default preference setting for resolution service is ‘off’.

Selection criteria

The selection criteria attributes lets you control if components should be picked from the default branch in repositories or from a named branch, tag, timestamp or revision. Not all repositories are capable of this — there is for instance no notion of branches or revisions in a p2 repository.



Branch/Tag path:

Timestamp:

Revision: 314151617

Branch/Tag path
: This is a search path specification for branches and tags to be searched.

Timestamp
: Enter a string in timestamp format.

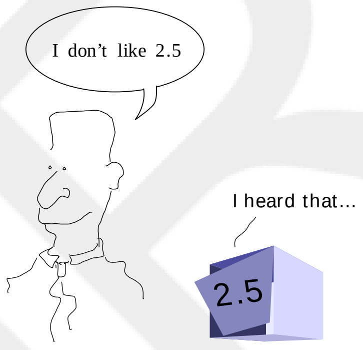
Revision
: Enter the name of a revision

Branch/Tag path
: The branch tag path is used to define a search path. The branches and tags in this comma separated list are searched in the specified order. Branches are entered by simply stating their name, and tags are entered with a leading slash '`/`' character. The special keyword '`main`' is used to refer to the repositories notion of *main branch* (e.g. 'trunk' for SVN, 'head' for CVS, etc.). As an example '`bug17,/release3,main`' would first look for the component on branch 'bug17', and if not found there, look in the tag 'release3', and finally if not found there either, look in the repository's main branch.

Timestamp
: The timestamp is always in UTC and should be entered in a format corresponding to Java `DateFormat.getDateTimeInstance(SHORT, SHORT)` for the current Locale. For the US locale that would be '`M/d/yy h:mm a`'.

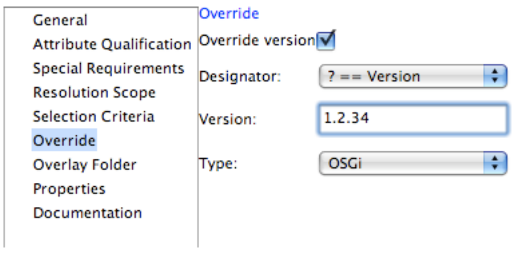
Revision
: Specifying a revision means that only content with a revision smaller or equal to the specified revision should be considered. The support for revision can be different in different repositories, and the revision identifier translates to the closest concept. For SVN the *repository revision number* is used, and for P4 the *change set identity* . In CVS, the revision is ignored, as CVS does not support identifiable revisions. Also note that SVN and P4 can combine branch/tag with revision.

Override (version)



The override allows you to override the requested version/version range of components matched by the advisor node. As an example if you want all requests for the component '`X`' to use the range 3.0 to 4.0 irrespective of what is stated in the components that have a dependency on `X`, you create an advisor node that matches `X` and specify the override for the range 3.0 to 4.0.

You have to be careful when creating the pattern — if you specify something like '`.*`' and `1.0.0` you have stated that every request for every component should request version 1.0.0. It is best to specify the full name of the component you want to override to avoid future surprises.



Override

Override version ✓

Designator: ? == Version

Version: 1.2.34

Type: OSGi

Override version
: Turns override on if checked.

Designator
: Select if version should be ==, >= or between two versions (inclusive/exclusive).

Version
: The version to request. (Two fields open for the range cases).

Type
: Select a version type from the drop down list.

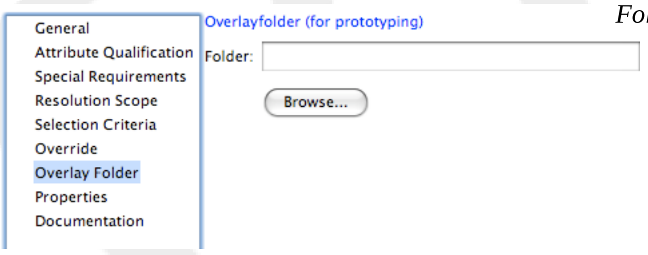
The version(s) should be entered in accordance with the selected version type format. See Chapter 9, *Versions*

Overlay

It is possible to specify an *overlay folder* (in your file system) that will overlay the found component during resolution — files in or under the overlay folder are used instead of the corresponding file in the component. This is mostly intended for experiments or temporary workarounds, but can be used to solve some tricky issues.

Advanced Topic — Meta Data Patching with Overlay

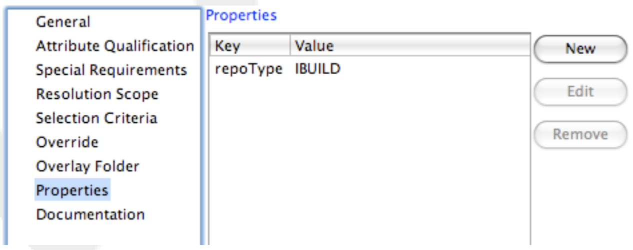
A very advanced use of overlay is to first materialize a patch that fixes meta data problems in some other component, and then use the materialized patch location as the overlay when materializing the component in need of patching.



Folder
: The path to an overlay folder.

Properties

This part allows you to modify the properties while a component matching the advisor node selection is being processed. See more information about properties in Chapter 10, *Properties*



Key	Value
repoType	IBUILD

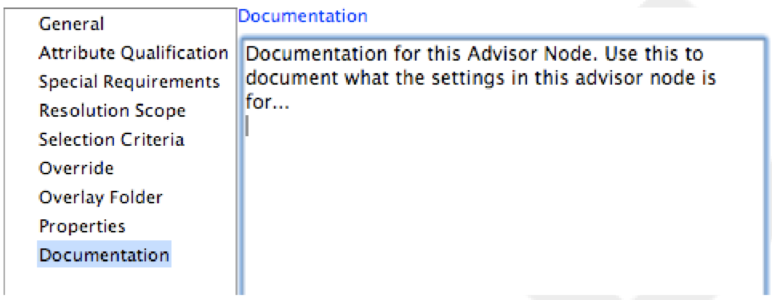
Property List
: Shows the properties that will be set for matching components.

New / Edit / Remove
: For adding, changing or removing entries in the list.

A very common use of property settings in an advisor node is to set values that are used to select a particular repository in the RMAP. See the section called “Parameterized locator” for an example in the RMAP. In the screenshot above you can see an example of `repoType` being set to `RBUILD`.

Documentation

The documentation part can be used to document what the advisor node does. This is valuable when a complex query is constructed and it can be hard to understand the particular purpose of a node. The documentation is currently not used anywhere but in the CQUERY editor.



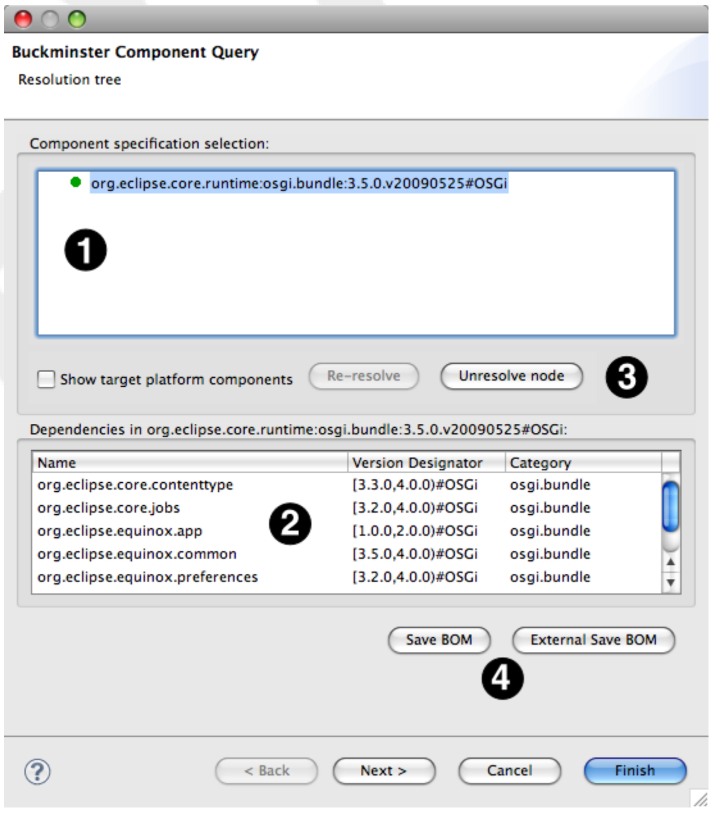
Documentation

Documentation for this Advisor Node. Use this to document what the settings in this advisor node is for...

Materialization wizard

The *Materialize to Wizard* runs the resolution and materialization under the control of the *materialization wizard*, where you can influence the process. It also gives you the opportunity to save the settings you are making for future use. (Your other choice is to select “Resolve and Materialize” which will run the entire process in one step — see the section called “Resolve and materialize”). This wizard is also used when materialization is performed as an *import* of a MSPEC, BOM, or CQUERY, i.e. using *File* → *Import...* → *Other* → *Buckminster* → *‘Materialize from a Buckminster MSPEC, CQUERY or BOM’*

Figure 5.1. Materialization wizard’s first page



Component specification selection:

● org.eclipse.core.runtime:osgi.bundle:3.5.0.v20090525#OSGi

Dependencies in org.eclipse.core.runtime:osgi.bundle:3.5.0.v20090525#OSGi:

Name	Version Designator	Category
org.eclipse.core.contenttype	[3.3.0,4.0.0)#OSGi	osgi.bundle
org.eclipse.core.jobs	[3.2.0,4.0.0)#OSGi	osgi.bundle
org.eclipse.equinox.app	[1.0.0,2.0.0)#OSGi	osgi.bundle
org.eclipse.equinox.common	[3.5.0,4.0.0)#OSGi	osgi.bundle
org.eclipse.equinox.preferences	[3.2.0,4.0.0)#OSGi	osgi.bundle

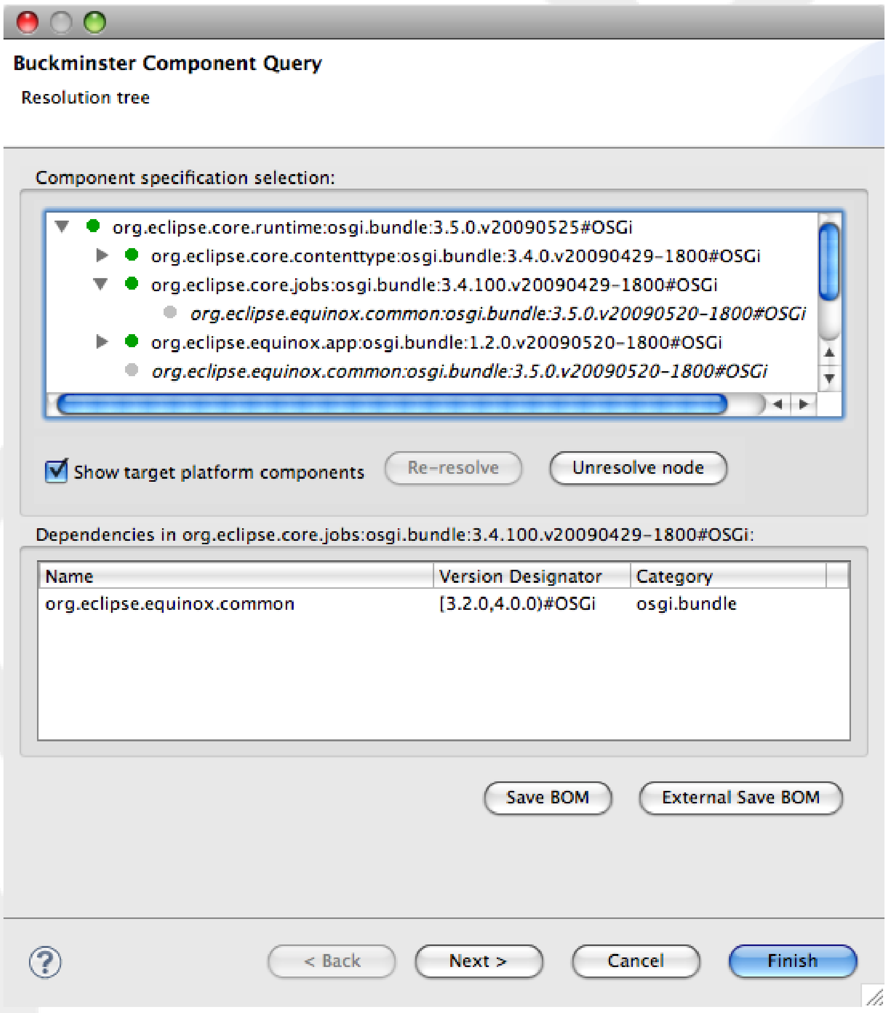
❶ This area shows the result of the resolution. By default components from ‘target platform’ are not shown in the resolution tree. In this example though, we explicitly asked for a component from the platform, so it still shows up.

❷ This list shows the dependencies in the selected component. You can see the requested range, and component type.

❸ In this section you can select if you want to see all components from the target platform in the tree above. The buttons ‘*re-resolve*’ and ‘*unresolve*’ lets you retry the resolution. If you had turned on ‘continue on error’ you can see red dots for unresolved nodes, and you may try to re-resolve them. This is useful when facing network issues with some components, or where you are repeatedly tweaking some information in the repository where the component is supposed to be found. The ’*unresolve*’ simply forgets the previous resolution for the component and lets you perform a ‘*re-resolve*’.

❹ These buttons allow you to save the resolution result as a Buckminster *Bill of Materials* (BOM) file. You can save it in a project in your workspace, or somewhere external (in the file system). You can read more about the BOM file in see Chapter 7, *Bill of Materials (BOM)*.

Figure 5.2. Wizard with target platform components shown


Buckminster Component Query
Resolution tree

Component specification selection:

```
▼ ● org.eclipse.core.runtime:osgi.bundle:3.5.0.v20090525#OSGi
    ▶ ● org.eclipse.core.contenttype:osgi.bundle:3.4.0.v20090429-1800#OSGi
    ▼ ● org.eclipse.core.jobs:osgi.bundle:3.4.100.v20090429-1800#OSGi
          ● org.eclipse.equinox.common:osgi.bundle:3.5.0.v20090520-1800#OSGi
    ▶ ● org.eclipse.equinox.app:osgi.bundle:1.2.0.v20090520-1800#OSGi
      ● org.eclipse.equinox.common:osgi.bundle:3.5.0.v20090520-1800#OSGi
```

[✓] Show target platform components

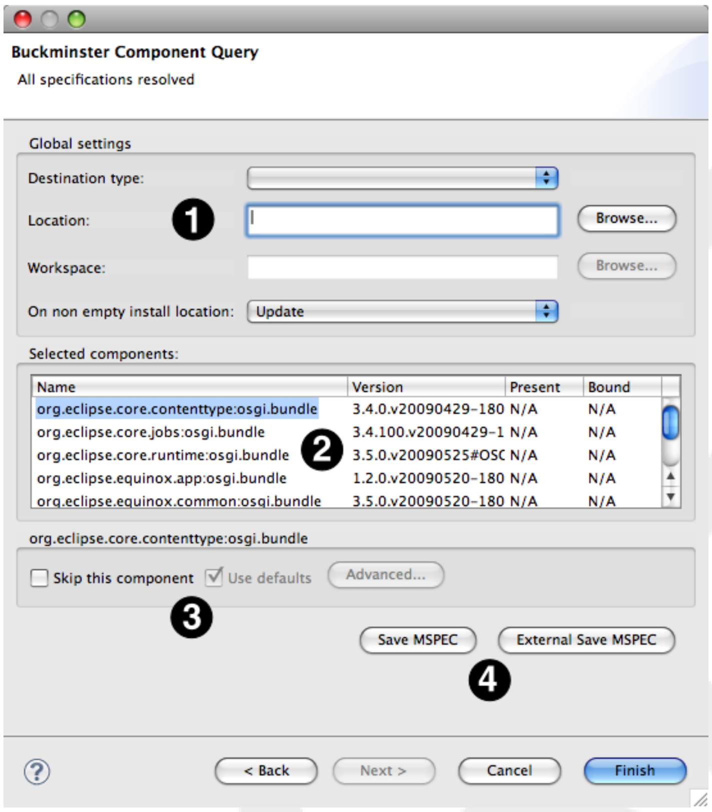
Dependencies in org.eclipse.core.jobs:osgi.bundle:3.4.100.v20090429-1800#OSGi:

Name	Version Designator	Category
org.eclipse.equinox.common	[3.2.0,4.0.0)#OSGi	osgi.bundle

Here, target platform components are also displayed in the tree. *Green* dots means resolved, and *gray* dots means that the component was first resolved to satisfy some other dependency. *Red* dots (not shown here) indicate that the resolution failed.

Once you are happy that the resolution contains what you wanted, you continue to the next step where you can specify where components should be materialized/installed.

Figure 5.3. Wizard’s materialization page



Buckminster Component Query

All specifications resolved

Global settings

Destination type:

Location:

Workspace:

On non empty install location: Update

Selected components:

Name	Version	Present	Bound
org.eclipse.core.contenttype:osgi.bundle	3.4.0.v20090429-180	N/A	N/A
org.eclipse.core.jobs:osgi.bundle	3.4.100.v20090429-1	N/A	N/A
org.eclipse.core.runtime:osgi.bundle	3.5.0.v20090525#OSC	N/A	N/A
org.eclipse.equinox.app:osgi.bundle	1.2.0.v20090520-180	N/A	N/A
org.eclipse.equinox.common:osgi.bundle	3.5.0.v20090520-180	N/A	N/A

org.eclipse.core.contenttype:osgi.bundle

☐ Skip this component ☑ Use defaults

❶ Here you control the ‘global’ settings where the materialization should go, and what should happen if the selected destination is not empty when the materialization takes place. These settings apply to all the components except those that are handled individually in the section below.

Available materializers

file system	A location in your file system.
p2	This materializer is used to create a platform agnostic target platform. (This is not the same as performing a p2 install, as such an installation is always for a particular platform). The location is a directory in your file system for the p2 artifact repository. The p2 materializer essentially performs the same task as the PDE `repo2runnable` ANT-task, but with more advanced selection criteria (the Buckminster resolution process vs. copy entire repository).
(~~site.mirror~~)	Deprecated in Eclipse 3.5. Used when materializing using the now deprecated Update Manager. Still supported in the editor for older artifacts.
(~~target platform~~)	Deprecated in Eclipse 3.5, and is now an alias for p2 materializer. Use the p2 materializer instead. Still supported in the editor for older artifacts.
workspace	The materialization will go into a workspace. If *location* and *workspace* fields are both empty then the current workspace is used (this would be the normal case). If *only location* is specified the materialization treats the location as a workspace. If *both location and workspace* are stat-

	ed, then the materialization is made to location, and the location is *linked* to the stated workspace.
'on non empty install'	Here you can control what should happen when the destination is not empty when the materialization takes place, you select between *update*, *fail* (i.e. report an error), *replace* (remove before materialization), and *keep* (use what is already there, do not update).

❷ The list shown is the result of the resolution — the *Bill of Materials* (BOM). You see component name, version, and the two columns '*Present*' (if the component has been materialized), and '*Bound*' (if it is bound to the workspace). The text '`N/A`' indicates that the component is a *fixture* — it can not be materialized — it exists in a form/location that is simply used.

❸ Here you can control if the selected component should be *skipped* — i.e. not materialized, and if you want to use the default location (as set in the area (1)). In the example, this button is grayed out because a component from the target platform is selected, and its location can not be changed. Unchecking '`use default`' enables the '`Advanced...`' button, and the settings made in the dialog that appears are used instead of the default.

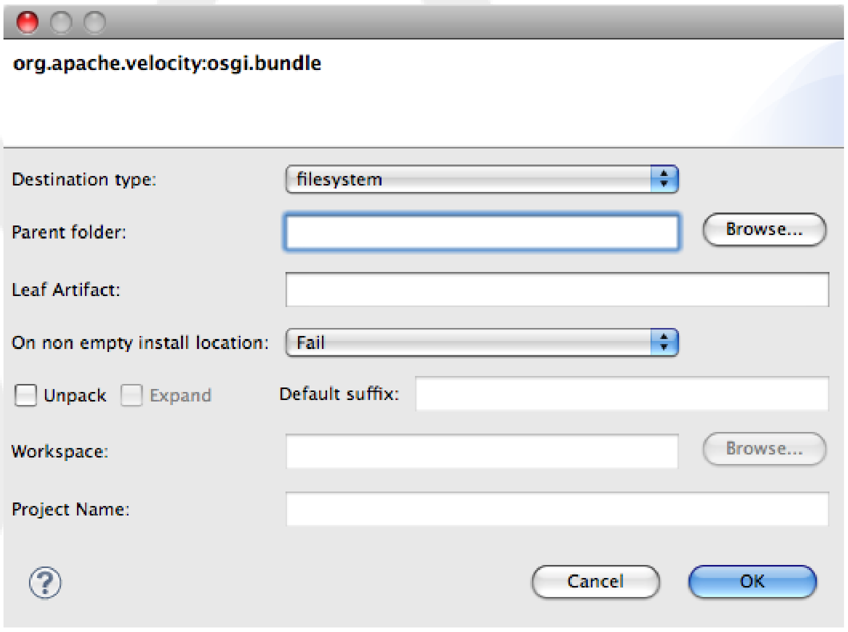
❹ The two buttons allows you to save the resulting *Materialization Specification* (MSPEC) in a workspace file, or in a file somewhere in your file system. The MSPEC is a Buckminster XML artifact that adds the final pieces of information in a materialization; i.e the *what goes where*-information that is edited on the wizard page. The MSPEC is described in more detail in Chapter 8, *MSPEC — Materialization Specification* .

Note

As a side of effect of saving a MSPEC, a BOM may also saved, with the same name as the MSPEC, but with the extension '`.bom`', and a reference is made in the saved MSPEC to this file. (If the file referenced in the MSPEC is already available, the BOM does not get written. See more details in Chapter 8, *MSPEC — Materialization Specification* .

Advanced settings

The *Advanced Settings* for selected components, is simply individual settings that do not use the default settings. The dialog that appears looks like this:


org.apache.velocity:osgi.bundle

Destination type: filesystem
Parent folder:
Leaf Artifact:
On non empty install location: Fail
Unpack Expand Default suffix:
Workspace:
Project Name:

Destination type	This is the same list of destination types as in the default settings (i.e. file system, workspace, etc.).

Parent folder	This is the same as *location* in the default/'global' settings — i.e. the folder where the component will be materialized.
Leaf Artifact	You can rename the file/folder that will be created by the materialization by entering the new name in this field. (When used in combination with *unpack*, this is the name of the created folder).
On non empty install location	This is the same choice as in the default settings (i.e. update, fail, etc.).
Unpack	Unpack implies two things: *deflate* which turns a '`x.tar.gz`' into a '`x.tar`', and *expand* which turns '`x.tar`' into a folder '`x`'. When selecting *unpack*, *deflate* is always implicit and *expand* is enabled by default. You can disable the expand if you only want deflate.
Expand	When a component is unpacked, it can also be expanded. See *Unpack* above.
Default suffix	The default suffix is used when it is impossible to automatically determine the file name (and hence the suffix) of the remote file. The suffix is only used to determine the *content type* of what is being read (it does not directly affect the resulting name of files or folders written in your file system). You can enter suffixes with multiple parts, i.e. '`tar.gz`' as this is important if you are doing both a *deflate* and *expand* of the content (see *Unpack*).
Workspace	This is the same as the workspace setting in the default/'global' settings.
Project name	When materializing into a workspace a project name can be stated (if you want it to have a different name than the component). Ignored for other types of materializations.

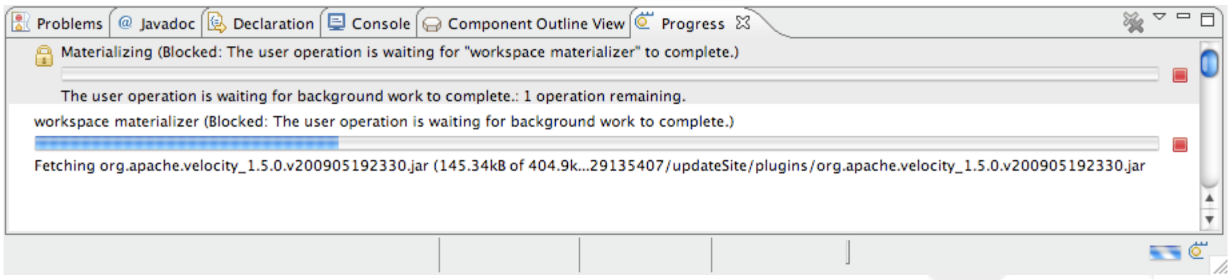
Example 5.1. Default Suffix and Renaming

The combination of *default suffix* and *renaming* can be a bit confusing, so here is an example to clarify. If you are downloading `monkey.zip`, it will be expanded into a folder called `monkey` in the location folder (no surprise). If you are downloading `123xe4a56_45-4` (and this is the content of the monkey zip-file, but where you only see the key used by the download service) you set the default suffix to `zip`, and the leaf artifact to `monkey`.

Watching the paint dry...

The final step of the process is to materialize the result of the query[4], possibly controlled by additional specification regarding skipped components, special treatment per component etc. You will see progress reported like this:

[4]Although, you do not have to perform this step if the purpose of running the wizard was to create a BOM or MSPEC for later use.



Materializing (Blocked: The user operation is waiting for "workspace materializer" to complete.)

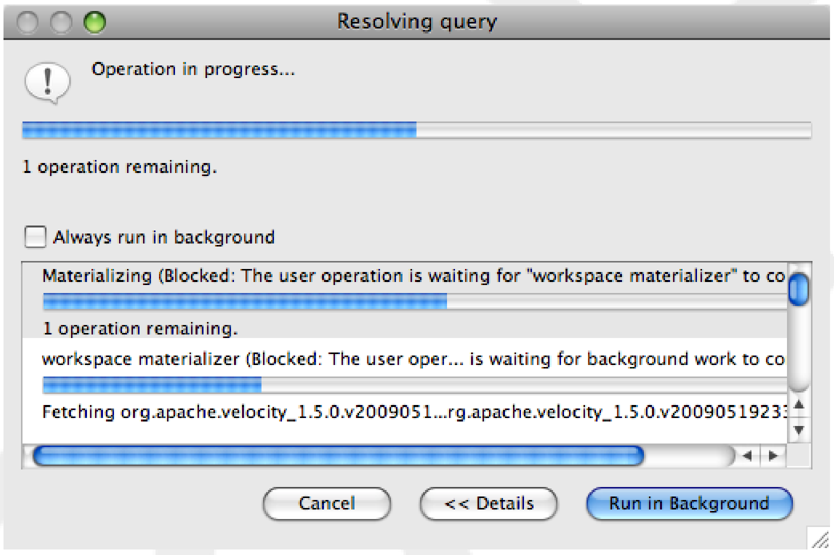
The user operation is waiting for background work to complete.: 1 operation remaining.

workspace materializer (Blocked: The user operation is waiting for background work to complete.)

Fetching org.apache.velocity_1.5.0.v200905192330.jar (145.34kB of 404.9k...29135407/updateSite/plugins/org.apache.velocity_1.5.0.v200905192330.jar

Resolve and materialize

The *Resolve and Materialize* runs the entire process in one step. Use this when you are happy with the defaults, and have no need to save intermediate results or settings. (See the section called “Materialization wizard” if you want more control). You will see a progress dialog that looks something like this:



Operation in progress...

1 operation remaining.

Materializing (Blocked: The user operation is waiting for "workspace materializer" to co

1 operation remaining.

workspace materializer (Blocked: The user oper... is waiting for background work to co

Fetching org.apache.velocity_1.5.0.v2009051...rg.apache.velocity_1.5.0.v20090519233

Summary

In this chapter you have seen the details of the CQUERY, how a query is created and edited with the CQUERY-editor resulting in a file that can be directly executed to materialize components, or to create more specialized artifacts (a *Bill of Materials* (BOM), or a *Materialization Specification* (MSPEC) for later more specialized use).

We did not show you the XML schema details of the CQUERY. The only reason to deal with the XML directly would be if you are generating queries or have specialized editing/refactoring needs. Please refer to the Part IV, “Reference” for the details.

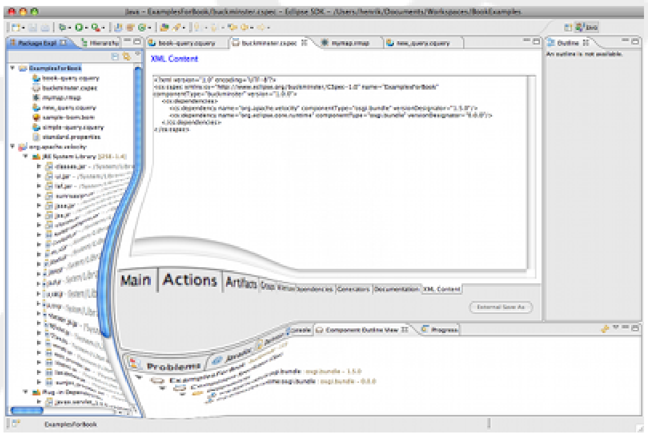
6 Components

In this chapter we take a closer look at Buckminster's view of *Components*, what they are, how they come into existence, and how they are used to manage configurations and building them.

Central to Buckminster's description of a component is the XML artifact *Component Specification* (CSPEC), and its extension mechanism CSPEX. These are explained in detail in this chapter.

Depending on what you are working with, you may not ever need to deal with authoring a CSPEC since for many component types (e.g. OSGi bundles, Eclipse features and products) the availability of the specification is immediate and automatic, and you can simply make use of the component's actions.

Figure 6.1. Secret revealed — where components come from



```
<?xml version="1.0" encoding="UTF-8"?>
<cs:cspec xmlns:cs="http://www.eclipse.org/buckminster/CSpec-1.0" name="ExamplesForBook"
componentType="buckminster" version="1.0.0">
    <cs:dependencies>
        <cs:dependency name="org.apache.velocity" componentType="osgi.bundle" versionDesignator="1.5.0"/>
        <cs:dependency name="org.eclipse.core.runtime" componentType="osgi.bundle" versionDesignator="0.0.0"/>
    </cs:dependencies>
</cs:cspec>
```

Your IDE is already expecting...

You can find several examples of CSPEC and CSPEX use in Part III, "Examples", and some of these may serve as templates for things you may want to do.

In the simplest cases, components *are just there*, but you may want to author new configurations, add or override actions in automatically created CSPECs, reuse existing actions in ANT, or in some external build system.

The component's anatomy

To Buckminster, a *Component* is described by the following meta data:

name	The *name* of a component is the primary identifier and is considered to uniquely identify a component when combined with component type.

type
: The *type* of component. See the section called “Component types”.

version
: The *version* of the component using an Omni Version as described in Chapter 9, *Versions*.

dependencies
: A component contains declarations of dependencies on other components. A dependency is expressed in terms of *component name*, *component type*, *version range*, and optionally include a filter that defines the applicability of the dependency in a particular environment (e.g. a dependency may only be valid when resolving for a particular operating system).

documentation
: A *short description* (typically one line of text suitable for display in a list), as well as a longer description where XHTML can be used is included in a CSPEC.

attributes
: An attribute of a component as seen from the outside is a named list of references to files . On the inside, an attributes is defined using one of the following:

  artifacts
  Used for static lists of artifacts.

  actions
  Used for dynamic/computed attributes. Typically some sort of build that produces new artifacts. Some actions are capable of producing more than one result where results needs to be independently reachable. To handle this, an action can declare additional attributes that corresponds to such results.

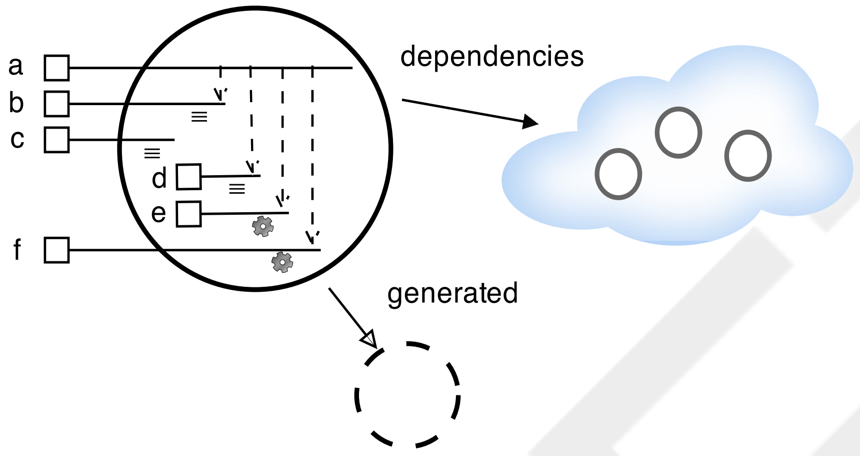
  groups
  Used to aggregate other attributes (i.e. artifact, action, or other groups).

filter
: A component’s filter is used to determine the component’s applicability/inclusion in a resolution in a particular environment (e.g. for a particular operating system, CPU architecture, etc.).

Advanced topic — Generators

generators
: A component can act as a generator of “virtual” components. This is useful when a component is brought into existence by an action/build step and it is impossible to locate such a component via the RMAP.

Figure 6.2. Component anatomy



A component with four public attributes `a`, `b`, `c`, `f`, and private attributes `d`, `e`, dependencies on other components, and a generated component. Illustration also shows that attribute `a` is a group consisting of attribute `b`, `d`, `e` and `f`. The attributes `b`, `c`, and `d` are simple static lists of artifacts, whereas `e` and `f` are generated.

CSPEC and CSPEX

Buckminster standard configuration includes support for several component types. Such an adapter interprets the existing meta data in its original form, and translates it into the CSPEC model. All computations done on components by Buckminster are done in terms of CSPECs. The actual CSPEC data is not persisted — it is created each time it is needed (although technically it may be cached for performance reasons). Even if the generated CSPEC is not persisted by default, it is still possible to generate the CSPEC in XML form for viewing, printing, or possibly for interchange with other systems.

A component that does not have any meta data to translate, or where there is no adapter for this particular component type can use Buckminster's CSPEC XML format for meta data to describe the component. This is done by placing a `buckminster.cspec` file in the component's root.

It is possible to extend/decorate an *automatically* generated CSPEC by placing a file called `buckminster.cspex` in the component's root.

Warning

Although technically possible to also extend a component that is described with a `buckminster.cspec`, such a construction is *not* recommended as it just makes it more difficult to author the meta data.

Both CSPEC and CSPEX are expressed in XML (see Buckminster XML Schemas), and the CSPEX is based on the same schema as the CSPEC, but adds capabilities to replace and remove information by using various `alterXXX` and `remove` XML elements. Elements in a CSPEX that are not marked with `alter` or `remove` are additions to the referenced CSPEC.

The CSPEC editor

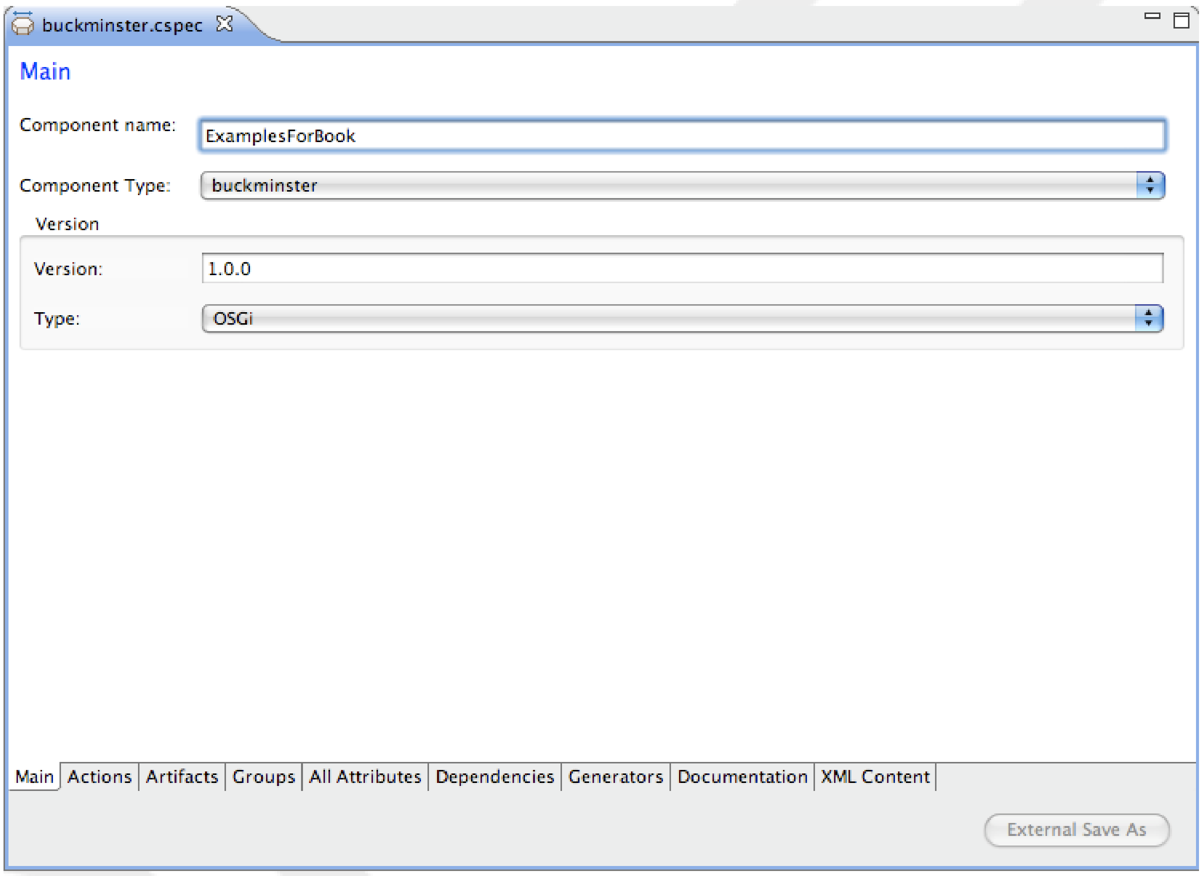
Buckminster includes a graphical CSPEC editor/viewer. As a *viewer* it is capable of showing the resulting component model (i.e. the combination of the automatically generated CSPEC and a CSPEX). As an *editor* it can be used to edit a CSPEC artifact.

Note

There is currently no graphical editor for a CSPEX. Editing is done using an XML editor. See the section called “Configuring Eclipse for XML Editing”. One alternative approach is to create a CSPEC and edit it with the graphical editor, and then modify the resulting XML file into a CSPEX — which may save you some time if you are new to Buckminster, and have a lot to author.

When the CSPEC editor is used as a viewer, it is in read-only mode with editing functions turned off. You can see that the editor is in view mode by looking at the title in the editor tab (it says “(read only)”), and in the File menu all operations that saves are disabled.

The editor is a multi-tab editor where different parts of the CSPEC are shown/edited on separate tabs. Here is a screenshot of what it looks like when the editor is opened:



Component name: ExamplesForBook

Component Type: buckminster

Version

Version: 1.0.0

Type: OSGi

The editor opens with its main tab selected (for editing the fundamental information; name and version). Along the bottom, you see all the tabs that takes you to different parts in the editor.

In this book we have taken the approach to explain the editor concept by concept, showing how the editor works, together with an explanation of the resulting CSPEC XML, and how it can be extended with a CSPEX (as opposed to dealing with the same concept multiple times — we hope this saves time jumping between sections).

Viewing a CSPEC

To view the resulting CSPEC for a component you have several options. The CSPEC editor can be opened in view mode (read only) on the resulting CSPEC of a component by:

- Selecting *File* → *View a selected CSpec...* which opens a dialog with a list of all components known to Buckminster in your current workspace. Select the component you want to view.
- Right click on the project folder or any file within the project in the Eclipse Package Explorer, or the Eclipse Navigator views. In the context menu that appears select *Buckminster* → *View CSpec...* and the CSPEC for the associated component is opened.
- A CSPEC can be opened from two of Buckminster's views — the *Component Outline View*, and *Component Explorer*. They are both found by selecting *Window* → *Show View* → *Other...* → *Buckminster*, and then the respective view. The Component Outline View shows information about the component currently selected, and the Component Explorer shows information about all known components. In either view, the context menu for a Component node presents the choice *Open*, which will open the CSPEC.

Note

It is not currently possible to open components via dependencies.

If you want to open a CSPEC (or CSPEX) for editing, you should locate the file in the workspace and then open it for editing as you normally open other files for editing. The "View a CSpec..."-actions always open the editor in view-mode (read-only).

Creating a CSPEC, or CSPEX

To create a CSPEC, or a CSPEX, simply invoke *File* → *New* → *Other...* → *Buckminster*, and then select one of → *Component Specification file*, or → *Component Specification Extension file*, which will prompt you for the name of the file, and then create it with the required XML declarations.

The resulting CSPEC looks like this initially (using OSGi version1.0.0 by default):

```
<?xml version="1.0" encoding="UTF-8"?>
<cs:cspec
    xmlns:cs="http://www.eclipse.org/buckminster/CSpec-1.0"
    name="org.demo.ExamplesForBook"
    componentType="buckminster"
    version="1.0.0"
/>
```

As you are editing, the editor will add needed name space declarations. If you edit the file using an XML editor, you need to add the namespace declarations yourself. See Buckminster XML Schemas for more information.

And the resulting file for a CSPEX should look like this:

```
?xml version="1.0" encoding="UTF-8"?>
<cspecExtension
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:bc="http://www.eclipse.org/buckminster/Common-1.0"
    xmlns="http://www.eclipse.org/buckminster/CSpec-1.0"
    >
<dependencies>
    <!-- Place your Dependencies here -->
</dependencies>

<generators>
    <!-- Place your Generators here -->
</generators>

<artifacts>
    <!-- Place your Artifacts here -->
</artifacts>

<actions>
```

```
    <!-- Place your Actions here -->
</actions>

<groups>
    <!-- Place your Groups here -->
</groups>

<alterDependencies>
    <!-- Place your Dependency alterations here -->
</alterDependencies>

<alterArtifacts>
    <!-- Place your Artifact alterations here -->
</alterArtifacts>

<alterActions>
    <!-- Place your Action alterations here -->
</alterActions>

<alterGroups>
    <!-- Place your Group alterations here -->
</alterGroups>

</cspecExtension>
```

Note

You should only create a CSPEC for a component that is not already translated into the CSPEC model automatically. Also note that you should only create a CSPEX for components that are automatically translated, and where you need to extend it. If you are unsure of how this works — see the section called “CSPEC and CSPEX”.

Name and version

When opening the CSPEC editor, it opens with its main tab selected. A screenshot of this can be seen in the section called “The CSPEC editor”.

The main tab has the following content:

Component name	This is the name of the component that together with the component type is the unique identifier for the the component. A typical name reflects the organisation creating the component (e.g. `org.eclipse.buckminster.core`).
Component Type	This is one of the supported types (e.g. `osgi.bundle`, or `eclipse.feature`). See the section called “Automatically generated meta data” for a brief overview, and the reference guide “Component Types” for all the details.
Version	This describes the version of the component. In the user interface the version is broken up into two parts; the version string in clear text, as displayed in the version field, and a named version format. The selection of the version format determines how the digits, strings, and delimiters in he version string are translated into the Omni Version instance used internally. For information about versions and version types see Chapter 9, *Versions*.

Tip

If you are working with OSGi bundles, Eclipse features or plugins you should always use OSGi versioning. When the choice is yours, we recommend also using OSGi.

CSPEC XML

In XML, name and version are written like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<cs:cspec
    xmlns:cs="http://www.eclipse.org/buckminster/CSpec-1.0" ❶
    name="ExamplesForBook" ❷
    componentType="buckminster" ❸
    version="1.0.0" ❹
    ~~versionType="Triplet"~~❺
>
```

❶ Namespace declaration for CSPEC.
❷ The name of the component.
❸ The component type.
❹ The version.
❺ Deprecated. The version type is supported for backward compatibility. **Should not be used for new artifacts**.

Note

You do not have to change you pre Eclipse 3.5 CSPECs as the formats `OSGi`, `Triplet`, `String`, and `Timestamp`, and `OSGi` are handled. If you however have created your own versioning scheme you must switch to using Omni Version.

CSPEX XML

A CSPEX can override all of these except the component's name. A CSPEC is bound to a CSPEC via inclusion in the component's root, and it will always extend the component in which it is embedded — as a consequence it is not possible to alter the components name. The corresponding section for a CSPEX looks like this in XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<cspecExtension
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:bc="http://www.eclipse.org/buckminster/Common-1.0"
    xmlns="http://www.eclipse.org/buckminster/CSpec-1.0" ❶
    componentType="buckminster" ❷
    version="1.0.0" ❸
    ~~versionType="Triplet"~~❹
>
```

❶ Namespace declaration for CSPEX is identical to CSPEC.
❷ Overrides the component type.
❸ Overrides the version.
❹ Deprecated. Overrides the version type. Should not be used in new CSPEX, the version attribute should be in Omni Version format. Supported for backward compatibility for the types `OSGi`, `String`, `Triplet`, and `Timestamp`.

Attributes

There are three types of component attributes; *artifacts*, *actions* and *groups*. These are explained in the following sections. Several concepts are common to all attributes — these are described here.

Visibility. Attributes are declared to be either *private* or *public*. The private attributes can only be referenced from within other attributes in the same component. The public attributes can be referenced from other components, and public actions can also be invoked from the user interface, or command line.

In XML the visibility is declared like this:

```
<artifacts>
    <public name="..." />
    <private name="... />
</artifacts>

<actions>
    <public name="..." />
    <private name="..." />
</actions>

<groups>
    <public name="..." />
    <private name="..." />
</groups>
```

This means, that in each section, an attribute of the particular type is created with public or private.

Note

Since the syntax for `public` and `private` elements is identical (except for the actual public/private element name) the following sections are only explaining the syntax using public elements.

Value. The value of an attribute is always an array of *path groups*, where a path group is a collection of paths referencing files or directories/folders. The paths are typically expressed as relative to the path group's *base* (which by default is the component's root, or specified explicitly), but paths can also be absolute. A relative base is relative to the component's root.

A path that ends with a slash '/' is a reference to a directory/folder. Absolute paths begin with a '/' (on Windows an absolute path starts with either '\', '/', '*drive letter*:\', or '*drive letter*:/').

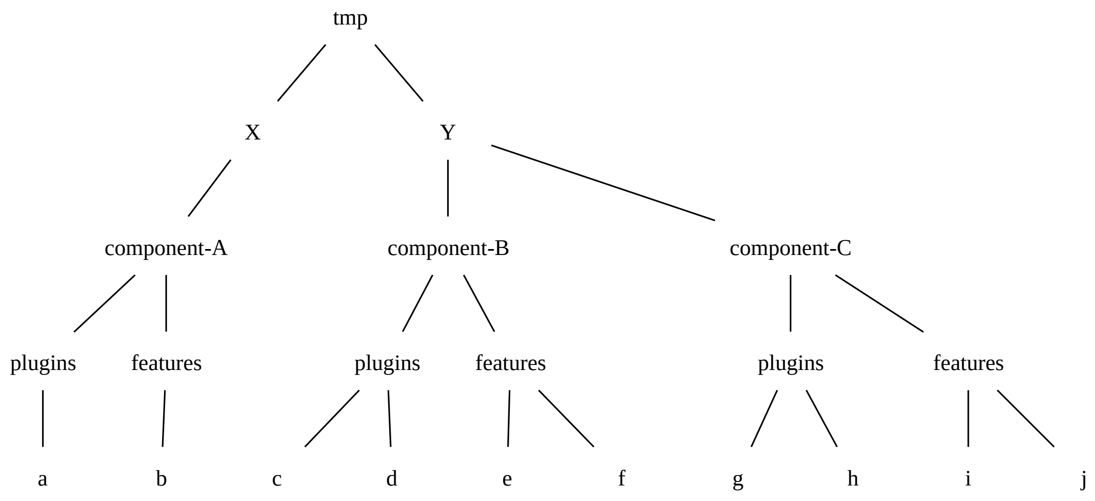
Tip

You can always use '/' as the path separator as it works on all platforms, and creates fewer issues as a '\' is often used as an escape character.

The *artifact* and *action* attributes always return an array with a single path group, and the *group* attribute returns an array with all values from the grouped attributes. As you will see later, a *group* can also manipulate the base path.

As the description is quite abstract — here is an illustration and some examples.

Figure 6.3. A file tree with components



As an example, if you want an attribute in `component-C` to include the artifacts ‘`g`’, and ‘`h`’, you can declare the attribute’s base to be an empty string (`base=""`), and use the paths ‘`plugins/g`’, and ‘`plugins/h`’. You get the same result if you instead set the base like this: `base="plugins"`, and set the paths to just ‘`g`’ and ‘`h`’. Does it matter which you use? The same files are referenced in both cases. The answer is: yes, it matters when you are copying the result. Let’s say you copy the result to a location ‘`Z`’. In the first case you would get ‘`Z/plugins/g`’ and ‘`Z/plugins/h`’, and in the second case you would get ‘`Z/g`’, and ‘`Z/h`’.

If you in each of the components `A-C` create attributes that represent plugins and features you can easily create a group that merges all plugins (with all paths relative to component root) into a plugins folder, and all features to a features folder. If you however declared all the plugins and feature attributes to be relative to `plugins` and `features` respectively, then the group would copy all of the `a-j` files into the same location.

Reference to the component itself

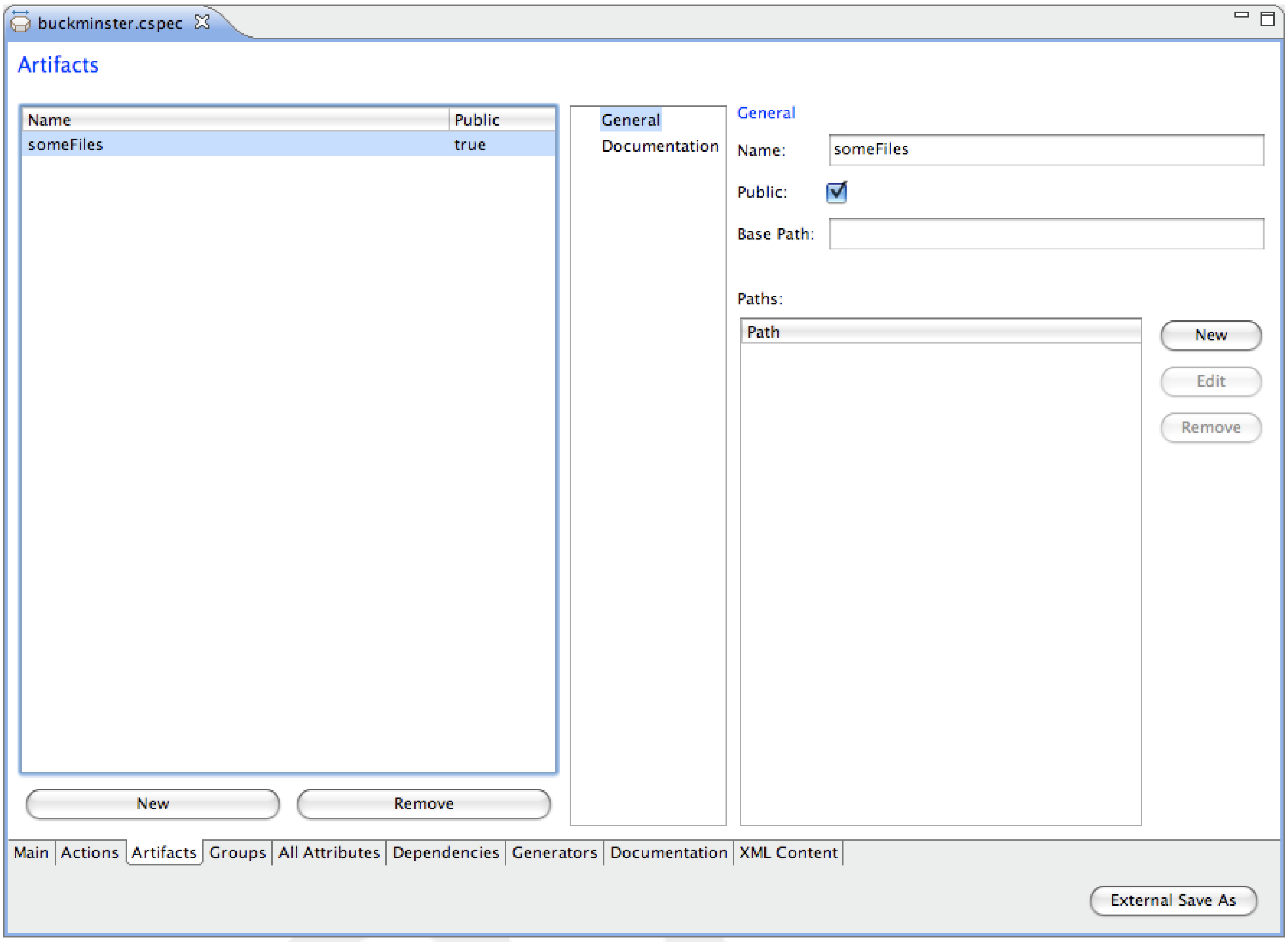
All CSPECs have an implicit attribute named `buckminster.component.self`. The *path group* of the `buckminster.component.self` can be in one of two forms depending on if the component is a directory or a file; if the component is a directory, then the base of the path group is equal to the location of the component and the array of paths is empty, and if the component is a file, then the base appoints the directory that contains this file and the path array has one path which is the file relative to that base.

Whenever a combination of component and attribute can be specified as a reference to an attribute, the default component is `buckminster.component.self`.

The `buckminster.component.self` attribute removes the need for you to create an additional attribute just to reference the static content of the component, and makes it easy to reference a components content in another component as the name is always the same.

Artifacts

An attribute implemented using *artifacts* is a static path group; a list of paths stated in the CSPEC. The CSPEC editor tab for editing artifacts looks like this:



On the left, there is a list of declared artifact attributes, their name and visibility is displayed. Buttons at the bottom allows for adding and removing artifact elements. To edit an element select it in the list, and then change its values on the right. There are two sets of values to edit; *General* (displayed above), and *Documentation* (not shown). The Documentation set consists of a single field where documentation for the artifact can be entered (XHTML is allowed) — this documentation is for the implementor/user of the CSPEC.

Name	This is the name of the attribute. This name is used to reference the value. Examples of typical names are jarfiles, documentation.html, documentation.pdf, headerfiles.
Public	When the checkbox is checked, the artifacts attribute is declared to be public. If not checked it is (not surprisingly) private. (See Visibility).
Base Path	This is the base for all paths in the *Paths* section. If *Base Path* is empty, the component's location is used as the base.
Paths	This is a list of paths, relative paths are relative to *Base Path*, and absolute paths may be used. Buttons on the side allows adding, removing and editing entries.

Note

There is no need to create an artifacts attribute for everything included in the component. The `buckminster.component.self` attribute always refers to the component itself. See the section called “Reference to the component itself ”.

CSPEC XML

In XML the declaration looks like this:

```
<artifacts>
    <public
        name="..."
        base="..."
        path="..." >
        <documentation>
            <p>This is documentation</p>
        </documentation>
        <path path="..." />
        <path path="..." />
        </public>

    <public ... />
</artifacts>
```

A short-hand notation can be used if there is only one path.

```
<artifacts>
    <public
        name="..."
        base="..."
        path="..."
    />
</artifacts>
```

Everything except the `name` attribute is optional.

CSPEX XML

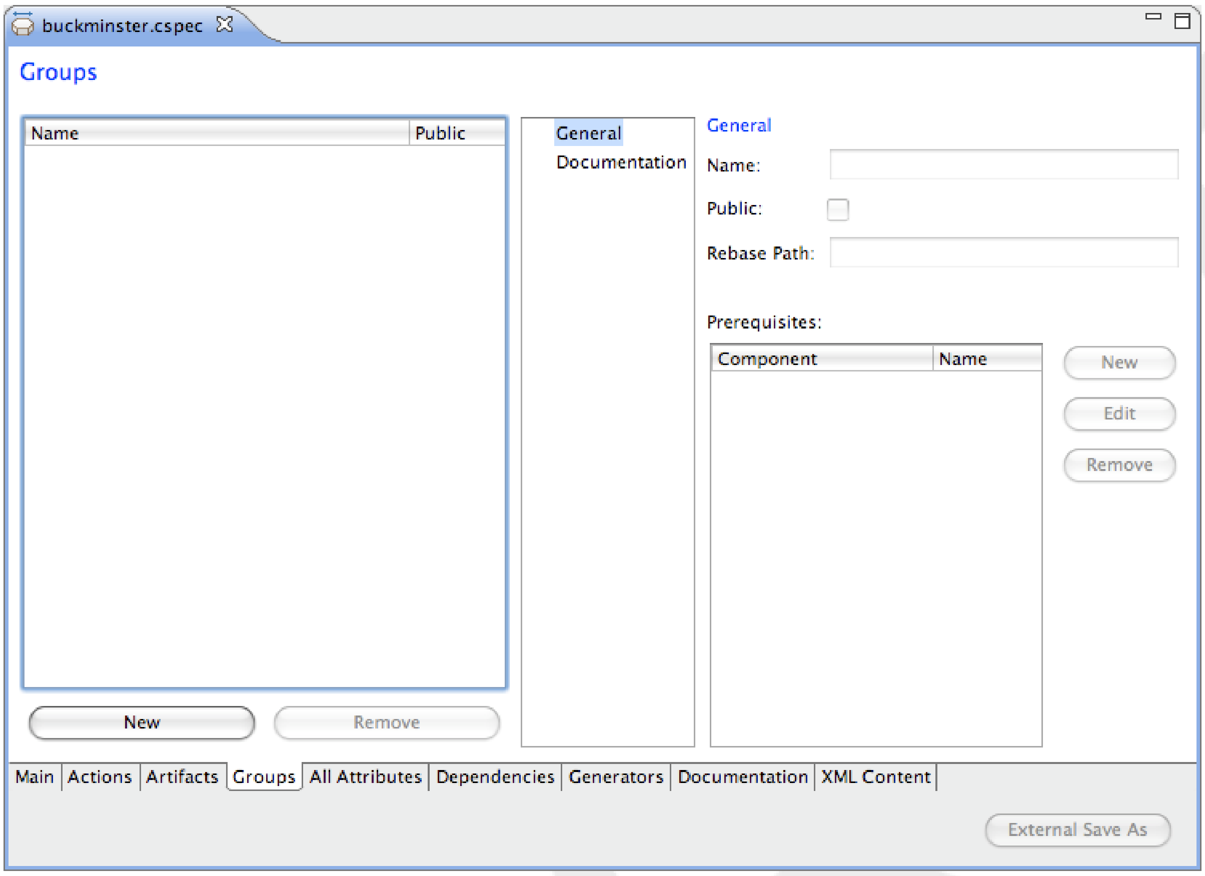
The CSPEX can extend an artifact declaration. Additions are made using the same declaration as in the CSPEC. Alterations are made in an `alterArtifacts` element. This is what it looks like in XML:

```
<alterArtifacts>
    <public name="..." > ❶
        <path path="..." /> ❷
        <removePath path="..." /> ❸
    </public>
    <public ... /> ❹
</alterArtifacts>
```

❶ The artifact referenced by name is overridden with the values declared in this element.
❷ This path is added
❸ This path is removed
❹ An alteration to an additional attribute.

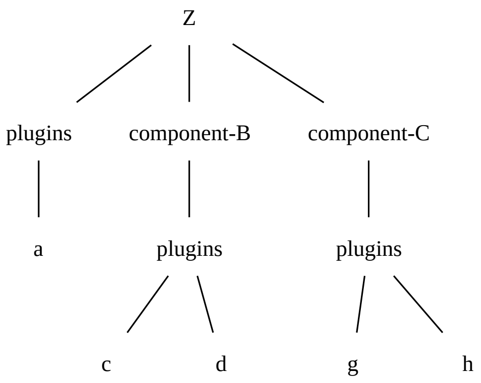
Groups

An *attribute* implemented using *groups* is a group of other attributes from the same, or other components. The CSPEC editor tab for editing groups looks like this:



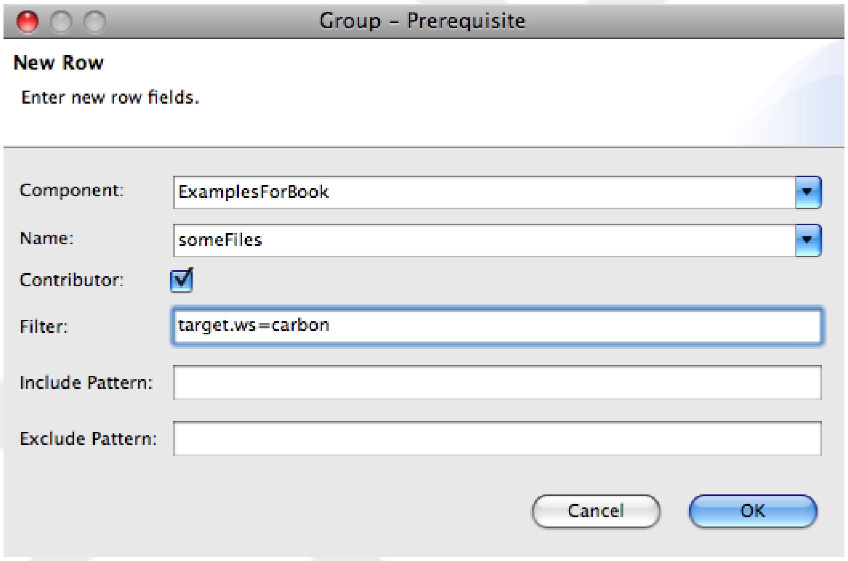
On the left, there is a list of declared *group* attributes, their name and visibility is displayed. Buttons at the bottom allows for adding and removing *group* elements. To edit an element select it in the list, and then change its values on the right. There are two sets of values to edit; *General* (displayed above), and *Documentation* (not shown). The Documentation set consists of a single field where documentation for the group can be entered (XHTML is allowed) — this documentation is for the implementor/user of the CSPEC.

Name	This is the name of the *group*. This name is used to reference the value. Examples of typical names are jarfiles, documentation.html, documentation.pdf, headerfiles.
Public	When the checkbox is checked, the *group* attribute is declared to be public. If not checked it is (not surprisingly) private. (See Visibility).
Rebase Path	By setting the rebase path, you can connect various path groups using a common base, making all relative paths in any path group be relative the new rebase path. The paths that are relative to bases that are not under the new rebase path are unaffected. As an example — look at Figure 6.3, “A file tree with components” and say you grouped the plugins from components `A`-`C` (all are relative to their respective component root). With a rebase path of `Y`, and a copy of the result to `Z` the result looks like this:



Prerequisites

This is a list of references to attributes in the same, or other components. Buttons on the right allows adding, removing and editing entries. When adding, or editing, the following dialog is shown:



Group – Prerequisite

New Row

Enter new row fields.

Component:	ExamplesForBook
Name:	someFiles
Contributor:	✓
Filter:	target.ws=carbon
Include Pattern:	
Exclude Pattern:	

Component
: The drop-down list shows all component that are added as dependencies. It is not possible to group an attribute from a component that is not among the dependencies. If Component is left empty — it means using attributes from the component itself.

Name
: This is the name of the attribute from the selected component. The drop down list shows the available attributes from the selected component. If the name is left empty, it means to use the referenced component's default attribute (i.e. self).

Contributor
: Should be checked if the result of the referenced attribute should be included in the group. If unchecked, the attribute is still polled for a value and can thus trigger actions. As an example, this is useful when an action produce unwanted artifacts like a log-file that should not be included.

Filter

A prerequisite can have a filter (see the Filters reference guide) which makes it possible to conditionally include the prerequisite in the result. The prerequisite is included if the filter is empty, or evaluates to true.

Include Pattern

The include/exclude patterns are regular expressions that are applied to the transitive scope of attributes in the prerequisite. The transitive scope can be thought of as a list of «*`component name`*»#«*`attribute name`*» entries. If you use an `includePattern`, *only* those entries that match that pattern will be included. If you use an `excludePattern` matching entries will be excluded. The `excludePattern` takes precedence in case an entry should match both patterns. The patterns are applied to each attribute in the transitive scope where the attribute is represented on the form «*`component name`*»#«*`attribute name`*» (or just «*`component name`*» in case self reference has been used). The match must be a full match for the expression. Partial matches does not count. (If you don't see the include and exclude patterns in the user interface you have a version that is too old — see Eclipse Bug 283936 [https://bugs.eclipse.org/bugs/show_bug.cgi?id=283936]).

Note

Action prerequisites (i.e. the input to actions) are not included in the transitive scope. If you need to perform include/exclude input to actions you need to do that in the respective action. If an action produces additional named attributes (in addition to its normal product value) then these are also included in the transitive scope (if referenced).

Exclude Pattern

See *Include Pattern* above.

CSPEC XML

A group is declared in XML like this:

```
<groups>
    <public
        name="..." ❶
        rebase="..." ❷
        >
        <attribute ❸
            name="..." ❹
            component="..." ❺
            contributor="false" ❻
            filter="..." ❼
            includePattern="..." ❽
            excludePattern="..." ❾
        />
        <attribute ... /> ❿
        <documentation> ... </documentation> ⓫
</groups>
```

❶ The group’s name
❷ The rebase path as explained for the editing field with the same name.
❸ The reference to the attribute to include in the group
❹ This is the name of the attribute to include from the specified component, or from ‘self’ if no component is specified.
❺ This is the name of the component to get the specified attribute from. May be empty.
❻ If the value of the attribute should be included in the group, the contributor should be set to true (which is the default if contributor is omitted).
❼❽❾ The attributes `filter`, `includePattern`, and `excludePattern` as explained for the same fields in the editor.
❿ The second attribute to be added to the group
⓫ Documentation for the group — can use XHTML.

CSPEX XML

A group can be extended in a CSPEX. Adding groups is done by using the same syntax as in the CSPEC. Alterations are done in XML using `alterGroups` like this:

```
<alterGroups>
    <public
        name="..." ❶
        rebase="..." >
        <alterAttribute ❷
            name="..."
            component=" "
            contributor="..."
            filter="..."
            includePattern="..."
            excludePattern="..."
            />
        <remove name="..." /> ❸
    </public>
    <public ... /> ❹
</alterGroups>
```

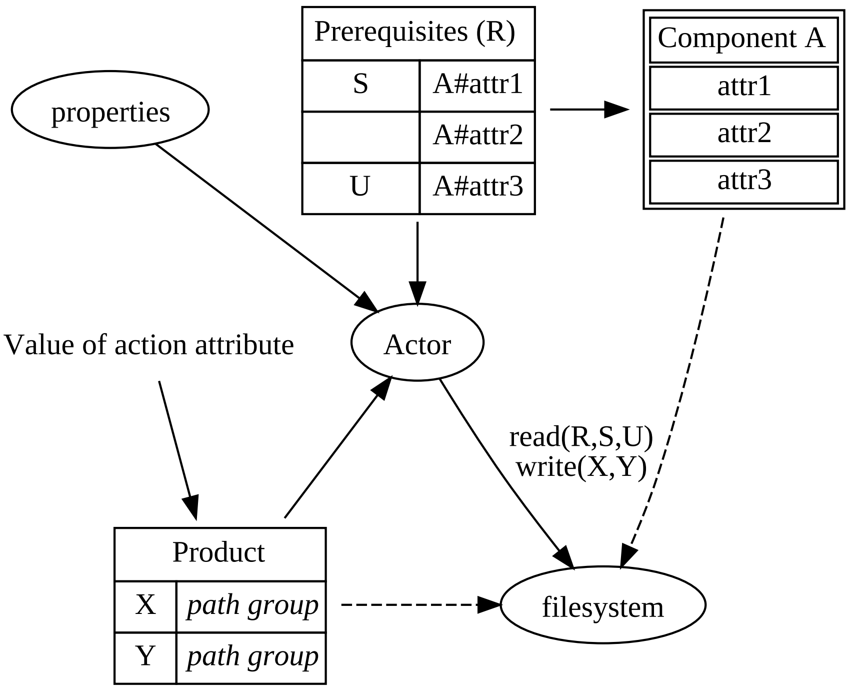
❶ The name attribute is used to select the group to alter. The attributes of this element overrides the attributes in the selected group.
❷ An inner `alterAttribute` is used to alter one of the entries in the group, the `name` selects the entry to modify, and the additional values override the corresponding values in the selected entry.
❸ An entry in the group is removed by using `remove`, the `name` selects the entry to remove.
❹ Here an additional group is altered.

Actions

An action is a dynamic attribute. It has the same type of value as other attributes — the difference being that it can compute the value when the value is requested. Actions may also be used for the purpose of only invoking them for their side effects (i.e. something the action does is valuable rather than a resulting file like some log-file it may produce). An action may also produce more than one named result.

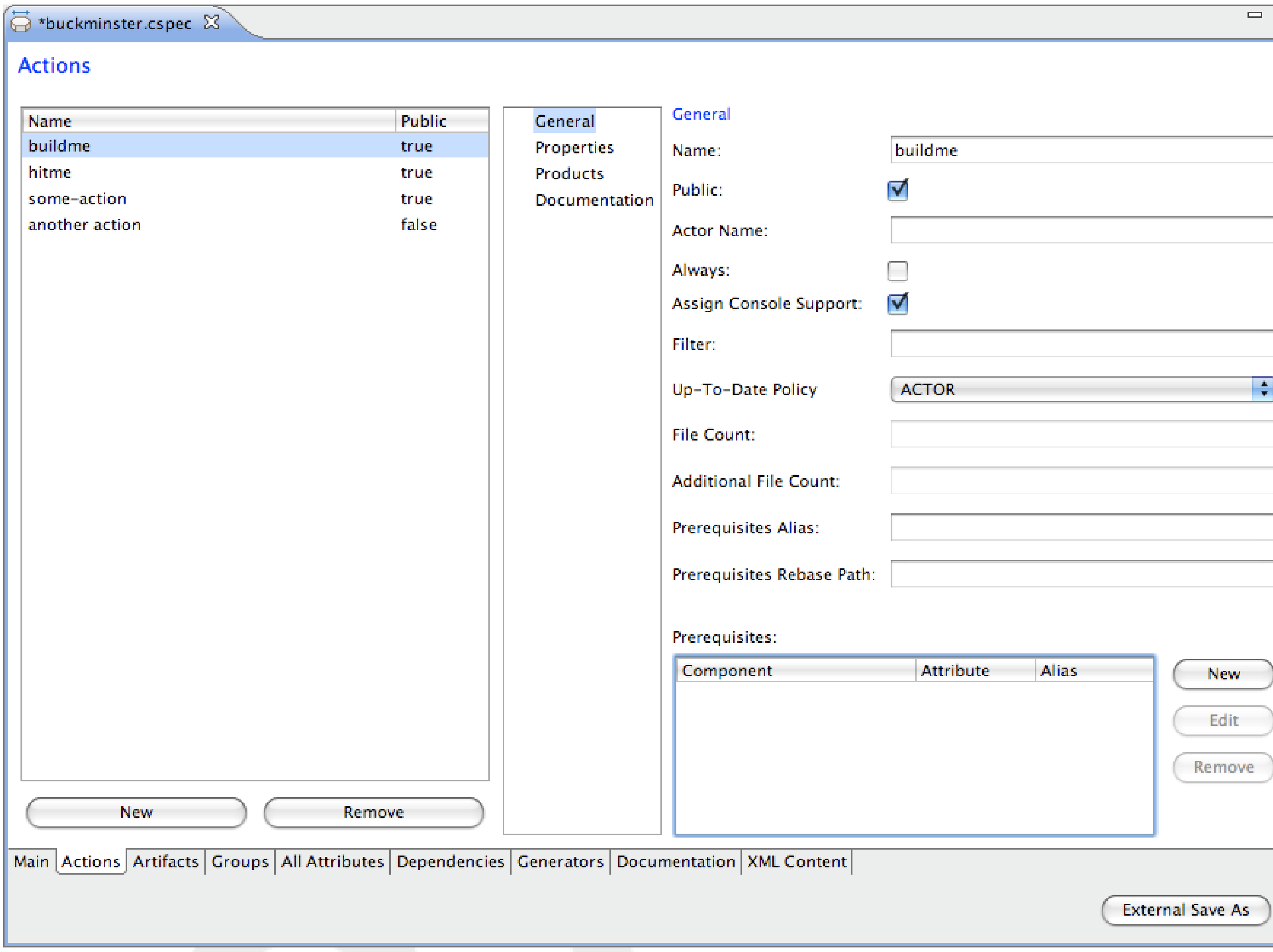
An action attribute is a quite powerful mechanism, and there are many options and details. Here is an illustration of the actions’s main parts (that is slightly more complicated than the simplest possible where there is a single input, a single output, and no required properties).

Figure 6.4. Action anatomy



Internally, an action has an *Actor* that handles the execution (a plugin, or script). The actor gets three things as input; *properties* (to initialize the actor, and control what it is supposed to do), *prerequisites* (the input to the action), and *product* (the result/output *path group* of the action). The illustration shows aliases (R, S, U, X, Y) — these, together with the properties can be thought of as the arguments to the actor. The actor finds the paths to the data via the aliases; the *input* is found in the prerequisites' aliases (R, S, U) which references attributes in components (which in turn, eventually, will refer to *path groups* that references files in the file system) — the *output locations* are found in the product's aliases (X,Y) which are *path groups* that references the output/result location(s) — most of the time a single path.

The CSPEC editor tab for editing actions looks like this:



Name	Public
buildme	true
hitme	true
some-action	true
another action	false

Name: buildme
Public: ✓
Actor Name:
Always:
Assign Console Support: ✓
Filter:
Up-To-Date Policy ACTOR
File Count:
Additional File Count:
Prerequisites Alias:
Prerequisites Rebase Path:
Prerequisites:

Component	Attribute	Alias

On the left, there is a list of declared *action* attributes, their name and visibility is displayed. Buttons at the bottom allows for adding and removing *action* elements. To edit an element, select it in the list, and then change its values on the right. There are four sets of values to edit; *General* (displayed above), *Properties*, *Products* (shown further on), and *Documentation* (not shown). The Documentation set simply consists of a single field where documentation for the action can be entered (XHTML is allowed) — this documentation is for the implementor/user of the CSPEC.

The General Section

Name	This is the name of the *action*. This name is used to reference the action’s value. Pick a name that is meaningful to a user when invoking it from a list of actions (e.g. `build.javadoc`).
Public	When the checkbox is checked, the *action* is declared to be public. If not checked it is (not surprisingly) private. (See Visibility).
Actor Name	This is the name of the *actor type*. Currently, there are several actor types that come with Buckminster. There is one general purpose actor that invokes ANT scripts (this actor is called `ant`), and several special purpose actors for specific tasks. It is possible to extend Buckminster with other types of actors. See Actors reference guide for all the details.

Always	This checkbox is used to control the up to date policy. When checked, this action will always be executed (i.e. it things of the output as never being up to date). If unchecked, the specified up to date policy is used.
Up-to-date-policy	The up to date policy indicates when the actor should consider the output to be up to date (and not run the action). This setting is ignored if *Always* is checked. The possible values are: *ACTOR* It is up to the actor to determine if output is up to date. *DEFAULT* Folders are never considered to be up to date, and output files must be younger than input files. *MAPPER* Each prerequisite is matched to a corresponding output/product. The match is verbatim or using a regular expression pattern and a replacement. All files must match and each product file must be younger then its respective match to consider the product up to date. The Mapper policy can be combined with Count policy to indicate that there are more files in the output (that are not present in the prerequisites) A count that denotes less then the number of mapped files has no significance. *COUNT* At least the specified number of files in the product must be younger then the youngest artifact in the prerequisites to consider it up to date. *NOT EMPTY* The output/product is considered up to date if it has content.
File Count	This value is used with the *COUNT* up to date policy (see above).
Additional File Count	This value is used with the MAPPER up to date policy (see above).
Assign Console Support	By default, actions are given access to the console (standard input/output and error streams). This can be turned off for an action that produces lots of unwanted output.
Filter	The filter is used to enable an action only when the filter expression evaluates to true. This is useful when certain actions should be ignored on certain platforms. The filter is written using LDAP filter syntax (the same way filters are expressed throughout Eclipse). See Filters for more information. Leave the filter field empty if you want the action to always be enabled. As an example, the filter expression `target.os=win32)` would only enable the action when running on 32 bit windows.
Prerequisites Alias	The *prerequisites* defines the action's input. They are similar to a *group*, but this group does not have a name that is useful. The *prerequisites alias* allows you to give the prerequisites group a name that can be used in the action (e.g. `input`, `source`,

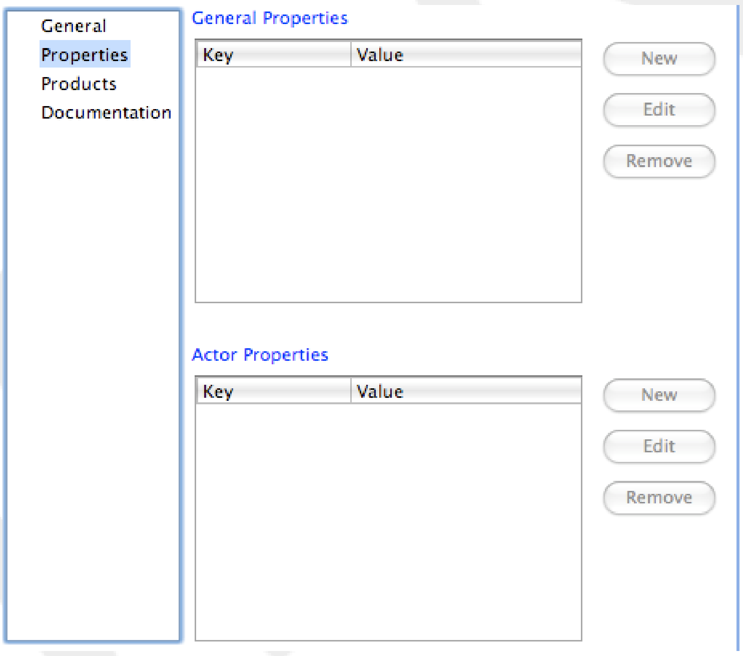
source-files, etc.). It is also possible to set aliases for individual attributes in the prerequisites group — see "*Prerequisites*" below.

Prerequisites Rebase Path
: The *prerequisites* defines the action's input, and is similar to a *group*. The *prerequisites rebase path* makes it possible to set the *rebase path* of this group. See "Group, rebase path" for an explanation of how rebase works.

Prerequisites
: This is a list of prerequisites — buttons on the right allows, adding, removing and editing entries. The prerequisites defines the *input* to the action — and by input, we mean both a concrete set of files, as well as any side effects that must have occurred prior to executing the action. A difference between prerequisites and a regular group is that is possible to specify aliases for the individual attributes. Setting aliases is useful as the actor would otherwise have to know the full names of the attributes it is processing. See the section called "Groups" for an explanation of the content of prerequisistes, and the dialog that appears for adding/editing.

The Properties Section

The Properties section is used to edit actor proprieties — there are two sets; properties that are used when the action invokes the actor (called *General Properties*), and properties that define parameters to initialize the actor (called *Actor Properties*[1]). This is what this looks like in the editor:



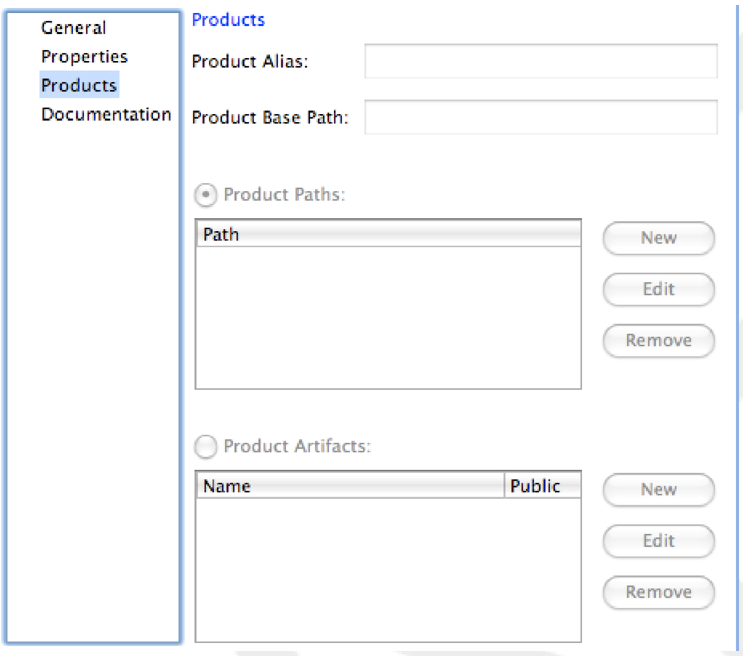
Each list is a list of property entries (a key-value pair). The Actor properties are used to initialize an instance of the specified actor type (e.g. an ant actor). The actor properties are specific to the type of actor being used (e.g. the ant actor has properties that references the ANT build file to use). See the Actors reference guide for information about each actor type's properties.

On the right of each list there are buttons for adding, removing, and editing entries. When adding and editing, a dialog pops up where key and value are entered. (A screenshot is not included).

[1]Which is a really bad name, since all of the properties are for an actor.

The Products section

The *Products section* is used to define the *output* of the action. The output (in addition to what is written to files as a consequence of running the action) is the *path group* value of the action attribute (or in special cases, multiple attributes). This section looks like this:



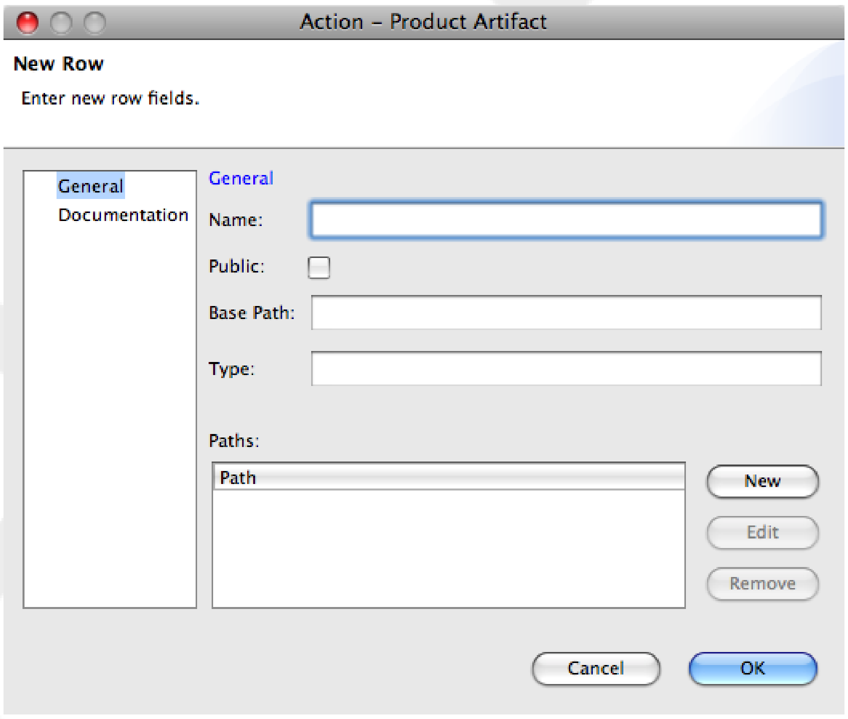
Product Alias	The *product alias* makes it possible to give the resulting output a name that can be used in the action/actor. Without this name, the actor would have to know the name of the component and the name of the action. Examples of aliases could be `result`, or `output`.
Product Base Path	The output/product is similar to a group, and just as in a group, it is possible to set the *base path* of the group using *Product Base Path*. See Group, rebase path for more information about the *base path*. The product base path is typically a reference to the directory where the actor has written its output (e.g. the compiled files). In a product, an empty base is equal to the variable `${buckminster.output}` which is the designated output folder for the build of the component. **Note** This is different from an empty base in an artifact. There, it defaults to the value of the variable `${buckminster.home}` which is the component location.
Product Paths	If *Product Paths* is selected, the output consists of a path group (i.e. a list of paths relative to the Product Base Path). The buttons on the side of the list allows adding, removing and editing entries. When adding or editing, a dialog pops up where the path can be entered. (A screenshot is not included in this book). *Product Paths* is mutually exclusive with *Product Artifacts*.

Product Artifacts

If *Product Artifacts* is selected, the output consists of a group of generated attributes. The main differences vs. *Product Paths* are that new individually addressable attributes are created, and that each such attribute has its own base path. This is very useful when an action produces more than one result e.g. compiled binaries and documentation. By making the two results available individually, some other group could include only the wanted subset, or an other action could have only the wanted subsection as a prerequisite. The mechanism also provides a separation of concerns between the action producing the result, and the result itself. At some later point you may want to refactor the actions so that compilation does not generate the documentation, and documentation is generated with its own action. By referring to the attributes that represents the result, you would not have to change anything where these results are used when splitting up the actions.

Product Artifacts is mutually exclusive with *Product Paths*.

The dialog for adding and editing *Product Artifacts* is similar to the dialog for editing attributes of artifact type.



See the section called “Artifacts”for an explanation of the fields.

CSPEC XML

The XML for actions looks like this:

```
<actions> ❶
    <public
        name="..." ❷
        actor="..." ❸
        >
        <actorProperties> ❹
            <property key="..." value="..." /> // *
        </actorProperties>
        <properties> ❺
```

```
            <property key="..." value="..." /> // *
        </properties>
        <documentation>...</documentation> ❻
        <prerequisites ❼
            rebase="..." ❽
            alias="..." ❾
            >
            <attribute ❿
                name="..." ⓫
                component="..." ⓬
                alias="..." ⓭
                filter="..." ⓮
                contributor="true" ⓯
                includePattern="..." ⓰
                excludePattern="..." ⓱
                />                // *
        </prerequisites>
        <products> ⓲
            <product ⓳
                alias="..." ⓴
                base="..." ㉑
                upToDatePolicy="..." ㉒
                filecount="..." ㉓ // NOT IN UI
                pattern="..." ㉔ // NOT IN UI
                replacement="..." ㉕ // NOT IN UI
                >
                <path path="..." /> ㉖ // Simple strategy
                // *
             </product>
        </products>
    </public>
    // *
</actions>
```

❶ Actions are written within an `actions` element. In this element, the individual actions are entered using either a `public` or `private` child element. The syntax is identical for both (except the difference in element name), and the example only shows a public element.

❷ A `public` element is used for a public action — its `name` attribute defines the action's name.

❸ The name of the actor is specified with the `actor` attribute.

❹ The *actor properties* are defined as `property` element children of an `actorProperties` element.

❺ The *general properties* are defined as `property` element children of an `properties` element.

❻ Documentation can be provided for an action using a `documentation` element. It may contain XHTML.

❼ A `prerequisites` element defines the input to the action.

❽ The prerequisites `rebase` path. See "Group, rebase path" for an explanation of how rebase works.

❾ Specifies an `alias` for the prerequisites that makes its content available to the actor.

❿ Each attribute that should be part of the prerequisites (i.e. the input) is stated with an `attribute` element. An `attribute` element references an attribute in a component.

⓫ The `name` attribute is the name of an attribute in some component.

⓬ The `component` attribute is the name of the component.

⓭ The `alias` attribute defines a name for the attribute that makes it possible for the actor to access this attribute separately.

⓮ The `filter` attribute makes it possible to specify a filter condition that dynamically determines if the prerequisites should contain this attribute or not.

⓯ The `contributor` flag can be set to true if an action is only included for its side effects. If contributor is false, the value of the attribute is not included in the result.

⓰⓱ `includePattern`, `excludePattern` — see the corresponding explanation for group prerequisites in the section called "Groups".

⓲ A `products` element defines the output/result of the action. An action can produce more than one product — the actions result is a group consisting of all such products. Each product may also be individually available as a separate “generated” attribute.

⓳ A `product` element is used to define a product.

⓴ The product alias makes it possible to reference the product from the actor. This is needed to make the actor aware of where output should be placed etc.

㉑ The `base` is the base path for the product.

㉒ The `upToDatePolicy` is the same as the corresponding setting in the user interface.

㉓ The `filecount` defines how many files in the product that must be present to determine that the product is up to date in relationship to the prerequisites. This value is used if the up to date policy is COUNT or MAPPER. When used with MAPPER, the `filecount` value is the number of files that must be present in addition to the mapped files. At present, a `filecount` field is not available in the user interface — see Eclipse Bugzilla Bug 283937 [https://bugs.eclipse.org/bugs/show_bug.cgi?id=283937].

㉔ The `pattern` is used in combination with `replacement`. This is however no yet implemented in Buckminster — see Eclipse Bugzilla Bug 283938 [https://bugs.eclipse.org/bugs/show_bug.cgi?id=283938]. The pattern/replacement are used with the MAPPER up to date policy. It is used to map input names to output names e.g. ‘`*.c`’ to ‘`*.o`’ or similar. Expressed as regexp, this could be expressed like this:

```
pattern="(.+)\.c$"replacement="$1.o"
```

The `replacement` would be applied to prerequisites paths relative to their respective bases.

㉕ `replacement` — see ‘`pattern`’ above.

㉖ A product has one or more `path` elements, or has one or more `public` elements if individually addressable attributes are wanted (see next example).

Alternatives for path in product:

```
<product ...>
    <public
        name="..."
        base="..."
        path="..." ❶
        />
    <public
        name="..."
        base="..."
        >
        <path path="..." /> ❷
        //...
    </public>
</product>
```

❶ A simple group with a single path can be defined with a `public` element with a `path` attribute. The `name` is the name of a created component attribute.

❷ A group with multiple paths can be defined with a `public` element with multiple `path` child elements. The `name` is the name of a created component attribute.

CSPEC XML

The XML for actions can be extended. Just like everywhere else in a CSPEX, things you want to add are just added using the same syntax as in a CSPEC, and the things you want to alter are handled in `alter`«XXX» elements.

```
<alterActions>
    <remove name="..." /> ❶
    <public name="..." actor="..." > ❷
        <alterProperties> ❸
            <remove key="..." /> ❹
            <property key="..." value="..." /> ❺
        </alterProperties>
        <alterActorProperties> ❻
```

```
            <remove key="..." />
            <property key="..." value="..." />
        </alterProperties>
        <alterPrerequisites ❼
            rebase="..."
            alias="..."
            includePattern="..."
            excludePattern="..."
            >
            <remove name="..." /> ❽
            <alterAttribute ❾
                name="..."
                component="..."
                alias="..."
                filter="..."
                contributor="..."
                />
        </alterPrerequisites>
        <alterProducts> ❿
            <removeProduct name="..." /> ⓫
            <removPath path="..." /> ⓬
            <path path="..." /> // add path ⓭
            <public ⓮
                name="..."
                base="..."
                path="..."
                >
                <path path="..." /> // add path ⓯
                <removePath path="..." /> ⓰
            </public>
         </alterProducts>
    </public>
</alterActions>
```

❶ A `remove` that removes the named action.
❷ Defines alteration of the named action, or is a new action.
❸ Defines alteration of the action's *general properties*
❹ A defined property is removed.
❺ A defined property is altered, or a new property is added.
❻ Alters the *actor properties* (the same way as the general properties are altered).
❼ Alters the *attributes* of the *prerequisites* — or if no attributes are stated, just indicates that alteration is wanted of the content of the `prerequisistes`.
❽ A `prerequisite` is removed.
❾ An attribute is altered, or a new attribute is added to the prerequisites.
❿ Specified to alter the `products`.
⓫ A `product` is removed.
⓬ A path is removed from a simple product.
⓭ A path is added to a simple product.
⓮ Alters or adds a product that defines a new attribute.
⓯ Adds a path to the enclosing attribute
⓰ Removes a path from the enclosing attribute.

Generators

The *Generators* tab is used to specify that the component generates other components. This is an advanced topic — normally components do not generate other components — see sidebar.

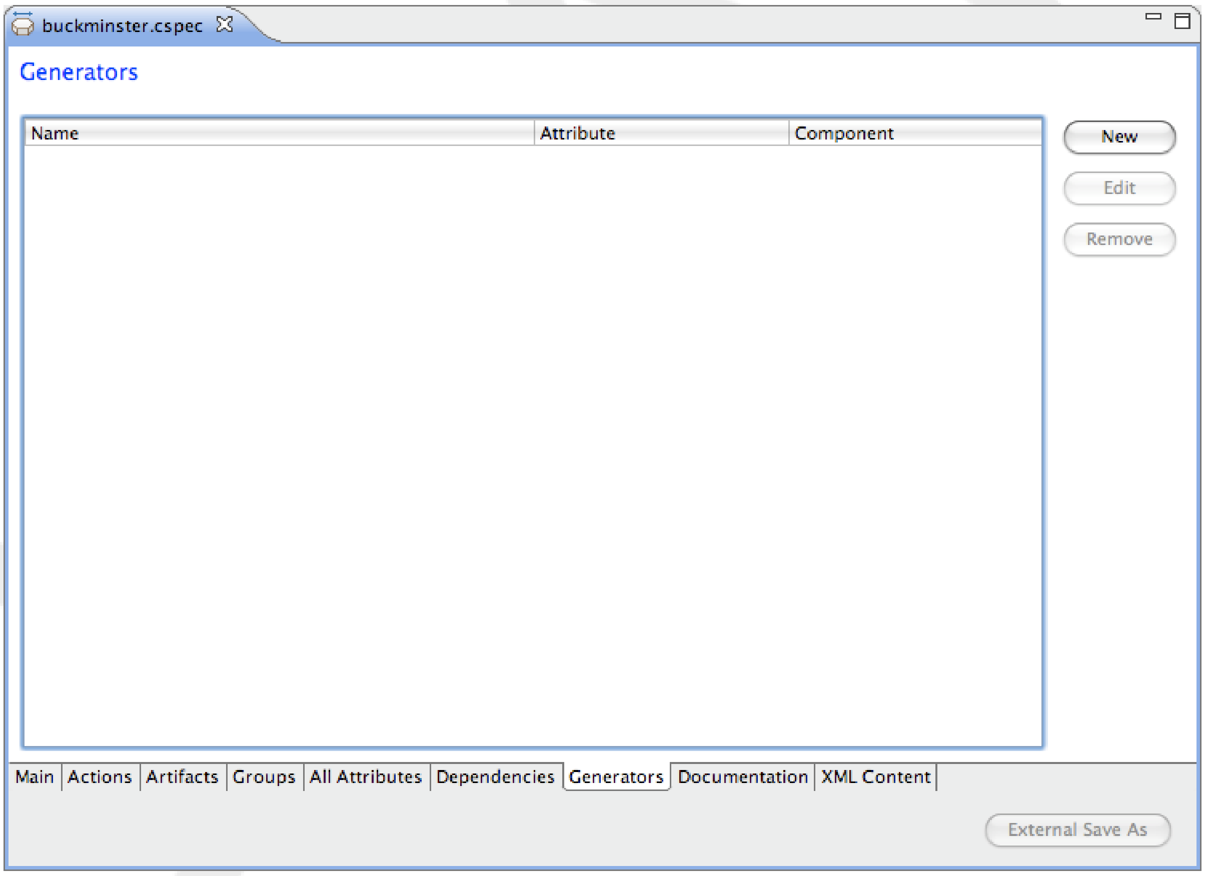
Advanced Topic — Generators

The ability to specify that one component generates others is very useful when components are created by executing actions. A common case is when using modelling and generating code. A component that contains the model can have actions that generates the java code, XML Schemas, code in some other language, a client component, a server stub component, etc. As these are all generated, they can not simply be located in some source repository, and those component that require the generated components needs to either specify a dependency on the component that generates what is needed as well as specifying a dependency on the generated component, or the configuration must already have resolved the component that generates what is required.

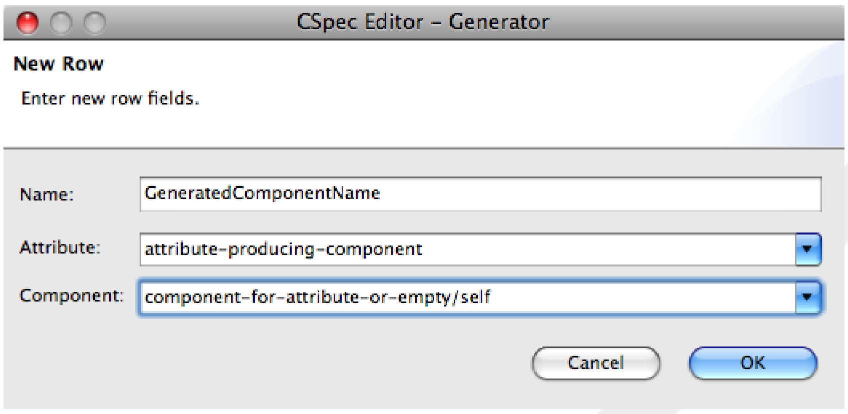
In addition to making it possible to work with generated components it also serves as a dependency indirection mechanism. Instead of being dependant on “X looked up in the RMAP” a dependency to a generated component becomes “X as made available by Y looked up in the RMAP”.

As an example A $\longrightarrow$ G, and X generates G, then either A $\longrightarrow$ (X, G), or C $\longrightarrow$ (X, G, A), in addition to A $\longrightarrow$ G, must be specified or the resolution will fail to find G.

The editor tab for editing Generators looks like this:



The *Generators* tab shows a list of defined generators, and buttons on the right makes it possible to add, remove and edit entries. When a generator is added or edited, the following dialog is used:



New Row

Enter new row fields.

Name: GeneratedComponentName

Attribute: attribute-producing-component

Component: component-for-attribute-or-empty/self

Name	The *name* is the name of the generated component.
Attribute	The name of the attribute that *produces* the generated component (typically an action, or a product of an action).
Component	The *component* is the name of the component of the attribute. If left blank, it means the component itself.

CSPEC XML

The XML for generators looks like this:

```
<generators>
    <generator ❶
        attribute="..." ❷
        generates="...." ❸
        component="...." ❹
        />
    // *
</generators>
```

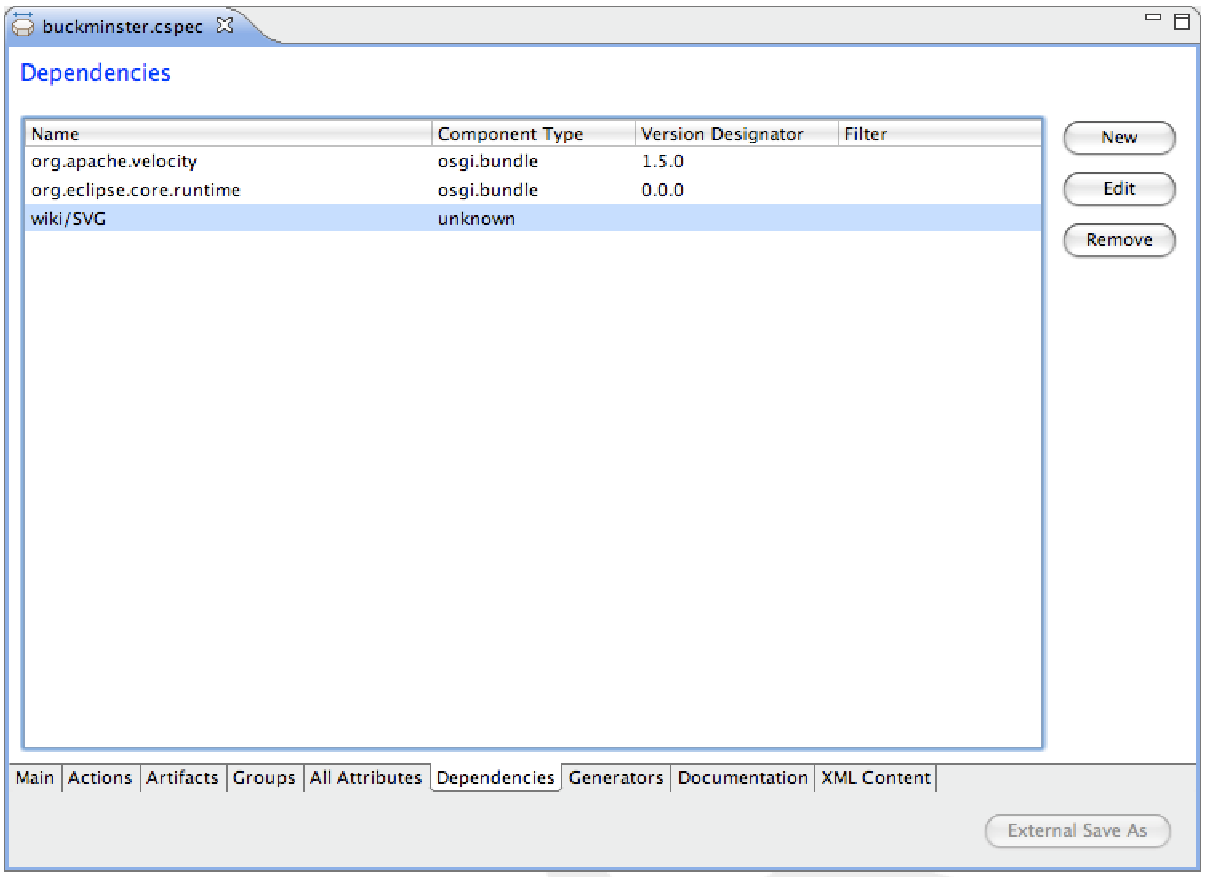
❶ The `attribute` is the name of the attribute in the *generating component* that produces the generated component.
❷ The `generates` attribute is the *name of the generated component*.
❸ The `component` is the name of the *generating* component.

CSPEX XML

The XML for generators is the same as in the CSPEC. There are no known component types that have generators so there is simply nothing to alter.

Dependencies

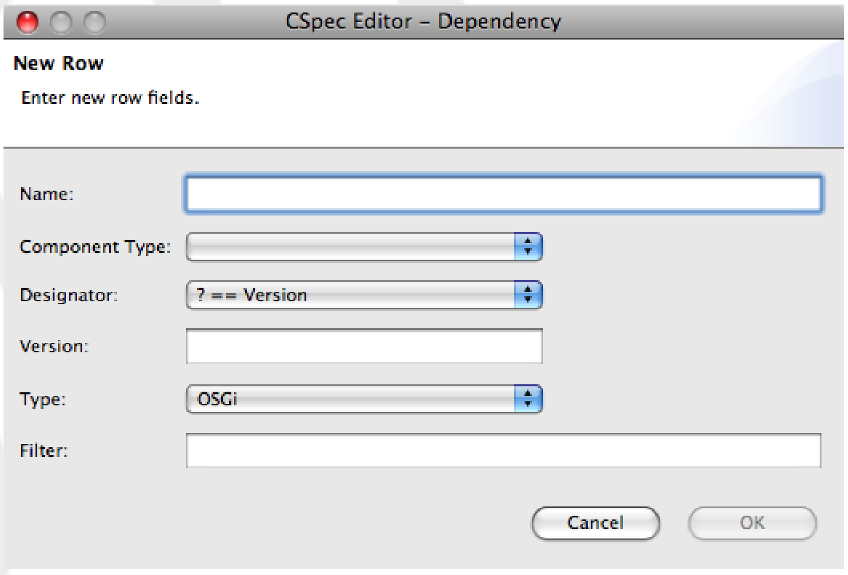
The *Dependencies* tab is used to define the component's dependencies on other components. The CSPEC editor tab looks like this:



Dependencies

Name	Component Type	Version Designator	Filter
org.apache.velocity	osgi.bundle	1.5.0	
org.eclipse.core.runtime	osgi.bundle	0.0.0	
wiki/SVG	unknown		

The tab shows a list of dependencies with columns for component name, component type, version/range, and filter. Buttons on the right allows adding, removing and editing entries. The dialog for adding and editing entries looks like this:



New Row

Enter new row fields.

Name:	
Component Type:	
Designator:	? == Version
Version:	
Type:	OSGi
Filter:	

The fields in this dialog are similar to what is entered in a CQUERY when requesting a particular component. You can think of a dependency as requesting the presence of another component if you like.

Name and Component type. This is where you enter the name of the requested component, and select its component type from the drop down list.

Version, range and version type. In this section you can enter a *version*, or *version range* for the component you are requesting. The drop down has entries for ==, >=, and four different 'between' entries (i.e. if *from* and *to* should be inclusive or not). When values for both *from* and *to* are required, an extra field appears. If you leave version empty, the default search is for the latest available version. See Chapter 9, *Versions* for more information about handling versions and version ranges.

Filter. The *filter* is used to specify when this dependency is valid. If you leave the field empty, the dependency is always valid. To restrict the validity, a filter is specified using LDAP filter syntax (just as filters are normally expressed throughout Eclipse). As an example, if the dependency is only valid on Mac OSx you would enter `target.os=macosx`). See Filters for more information. .

CSPEC XML

The XML for dependencies looks like this:

```
<dependencies> ❶
    <dependency ❷
        name="..." ❸
        componentType="..." ❹
        versionDesignator="..." ❺
        versionType="..." ❻
        filter="..." ❼
</dependencies>
```

❶ Dependencies are stated in a `dependencies` element.
❷ Each dependency is stated in a `dependency` element. Its attributes defines the dependency.
❸ The `name` is the name of a component this component depends on (requires).
❹ The `componentType` is the type of component required.
❺ The `versionDesignator` is a version or version range defining the constraints for the required component. See Chapter 9, *Versions*, for more information about how to enter versions and version ranges.
❻ The `versionType` is one of the supported version types. See Chapter 9, *Versions*, for information about version types.
❼ The `filter` allows specification of a filter that dynamically determines the applicability of the dependency. See the 'Filters reference guide' for more information about filters.

CSPEX XML

The XML for dependencies can be extended — it looks like this:

```
<alterDependencies> ❶
    <remove name="..." componentType="..." /> ❷
    <dependency /> ❸
</alterDependencies>
```

❶ Dependencies are altered within an `alterDependencies` element.
❷ A dependency is removed. Currently, it is not possible to specify the `componentType` which makes it impossible to alter a ambiguous dependency — see Eclipse Bug 283940 [https://bugs.eclipse.org/bugs/show_bug.cgi?id=283940].
❸ A new dependency is added, or an existing dependency is modified (if the name and component type match). The syntax is the same as in a CSPEC.

Automatically generated meta data

Buckminster standard configuration includes support for several component types. Such an adapter interprets the existing meta data in its original form, and translates it into the CSPEC model. All computations done on components by Buckminster are done in terms of CSPECs. The actual CSPEC data is not persisted — it is created each time it is needed (although technically it may be cached for performance reasons). Even if the generated CSPEC is not persisted by default, it is still possible to generate the CSPEC in XML form for viewing, printing, or possibly for interchange with other systems.

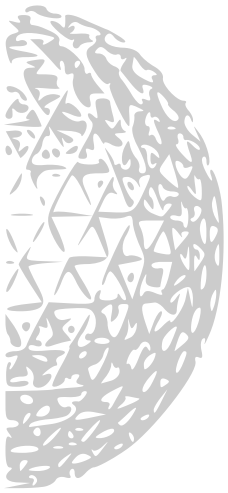
You will find all the details in the reference guide Component Types, but here is a brief overview:

osgi.bundle	This is a component type for OSGi bundles, and Eclipse plugins.
eclipse.feature	This component type understands Eclipse features.
jar	A component type that is a single jar file.
maven, maven2	The maven component type translates components with maven 1 meta data in a maven POM file. The maven2 component type handles the maven 2 POM format.
buckminster	A component using Buckminster's CSPEC XML as its metadata.
bom	A component of `bom` type is replaced by the top component in the BOM — this takes place in the resolution process and you will never see or interact with a component of this type.
unknown	An unknown component type is for components where there is no meta data whatsoever. The only information available is the component name, and its version (or possibly a null-version when a component is not versioned). An unknown component has no dependencies.

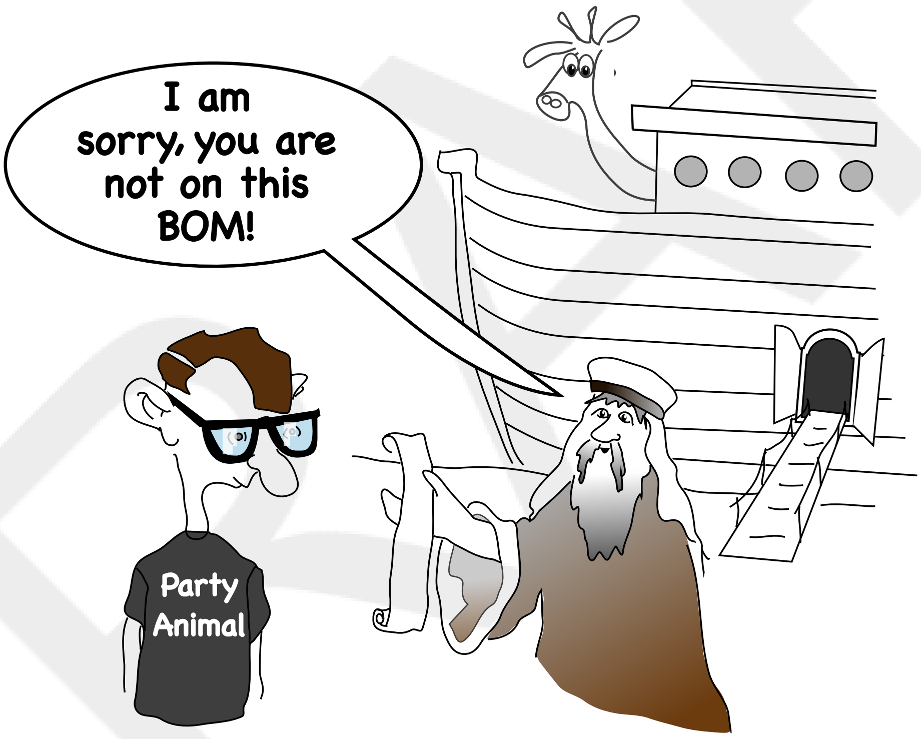
Bookmarks

A note about bookmarks

Buckminster supports including bookmarks containing information about web pages and RSS feeds in the component meta data since Eclipse 3.4. The mechanism is based on placing a special `buckminster.opml` file inside a component. Although still supported, our current recommendation is to only use this mechanism in components devised for building and publishing purposes. See Appendix D, *Bookmarks and OPML* for more information.



7 *Bill of Materials (BOM)*



The *Bill of Materials* (BOM) is an artifact containing a packing list consisting of the exact names and versions of all the components that were resolved by a CQUERY.

The BOM, in contrast to the other Buckminster artifacts is not something you should edit. It is generated by the resolution process, and can be used for different purposes:

- as input to the materialization process, either directly, or referenced by a MSPEC.
- as a manifesto for what was used to build a piece of software

- as input to visualization — a simple report, or a graphic dependency view
- as input to external tools — perhaps generating technical release documentation
- as a pre-made resolution of a component in a RMAP

At this point you have probably already figured out that this is going to be a very short chapter. It is however still useful to know what the BOM contains in principle, even if you are not going to construct one by hand.

The BOM's anatomy

The BOM contains the following:

- A list of all the components in the resolution
- For each component, a copy of that component's CSPEC as it was found/generated at the time of resolution. This means that all of the component's dependencies, attributes etc. are available in the BOM without further lookup.
- For each component, a reference to where it was found.
- A copy of each provider (from the RMAP) that were used to resolve an included component.
- For everything copied, all property references are replaced with the values these properties had a the time of resolution.
- A copy of the CQUERY that was used to produce the BOM.

Warning

Since all property values are expanded and included in the BOM you must be careful with properties that include user name and password information. They are stored in clear text inside the BOM.

Materializing a BOM

A BOM is indirectly materialized when executing a CQUERY, or when materializing a MSPEC that references a BOM (as described in the respective chapters covering CQUERY and MSPEC). The BOM can also be directly materialized[1] by importing it using *File* → *Import...* → *Other* → *Buckminster* → *'Materialize a MSPEC, CQUERY or BOM'* which opens the materialization wizard (see the section called "Materialization wizard").

Since the BOM contains a full snapshot (taken at the time of resolution) of all information required to materialize the components, there is no need for the materialization process to look things up in a RMAP.

Viewing a BOM

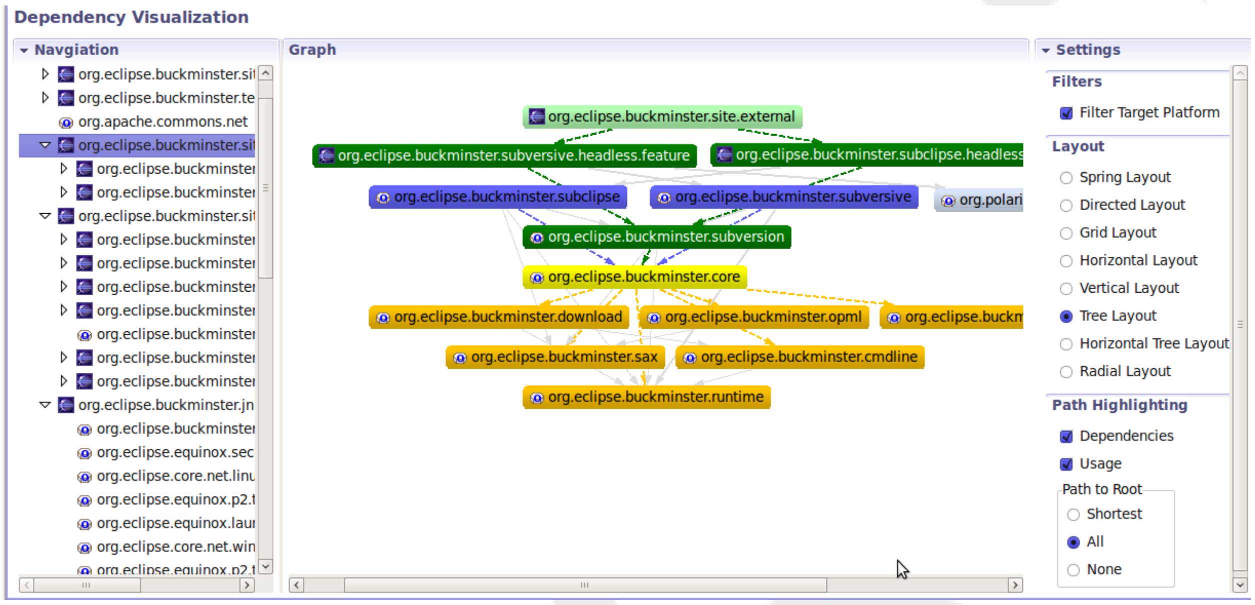
Buckminster keeps an up to date resolution of what is currently available in a running Eclipse SDK — in a way you can think of this a dynamic bill of materials. This set of components can be viewed in the Buckminster *Component Explorer* available via *Window* → *Show View* → *Other...* → *Buckminster* → *Component Explorer*, and it is also possible to see the resolution for the component that is currently selected in the workspace in the Buckminster *Component Outline View*, located next to the Component Explorer in the menu. These views however, do not show all the details in the BOM — you can however

[1] naturally a resolution must first have been produced and saved into a `.bom` file

see the names, versions and dependencies. If you want to see all the information you have to look at the XML directly.

There has been some exciting development lately of a graphic dependency viewer. You can read more about this viewer in the section called “BOM visualizer”. Here is a screenshot of what the graphic dependency visualizer looks like:

Figure 7.1. Dependency visualizer


Dependency Visualization
Navgiation
org.eclipse.buckminster.sit
org.eclipse.buckminster.te
org.apache.commons.net
org.eclipse.buckminster.sit
org.eclipse.buckminster
org.eclipse.buckminster
org.eclipse.buckminster.sit
org.eclipse.buckminster
org.eclipse.buckminster
org.eclipse.buckminster
org.eclipse.buckminster
org.eclipse.buckminster
org.eclipse.buckminster
org.eclipse.buckminster
org.eclipse.buckminster.jn
org.eclipse.buckminster
org.eclipse.equinox.sec
org.eclipse.core.net.linu
org.eclipse.equinox.p2.t
org.eclipse.equinox.laur
org.eclipse.core.net.win
org.eclipse.equinox.p2.t
Graph
org.eclipse.buckminster.site.external
org.eclipse.buckminster.subversive.headless.feature
org.eclipse.buckminster.subclipse.headless
org.eclipse.buckminster.subclipse
org.eclipse.buckminster.subversive
org.polari
org.eclipse.buckminster.subversion
org.eclipse.buckminster.core
org.eclipse.buckminster.download
org.eclipse.buckminster.opml
org.eclipse.buckm
org.eclipse.buckminster.sax
org.eclipse.buckminster.cmdline
org.eclipse.buckminster.runtime
Settings

Filters
- [✓] Filter Target Platform

Layout
- ○ Spring Layout
- ○ Directed Layout
- ○ Grid Layout
- ○ Horizontal Layout
- ○ Vertical Layout
- ● Tree Layout
- ○ Horizontal Tree Layout
- ○ Radial Layout

Path Highlighting
- [✓] Dependencies
- [✓] Usage

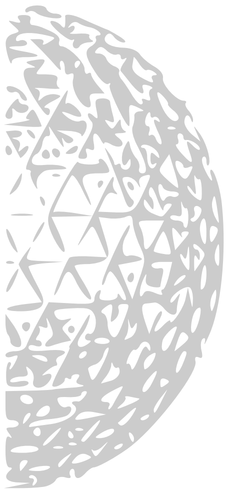
Path to Root
- ○ Shortest
- ● All
- ○ None

Summary

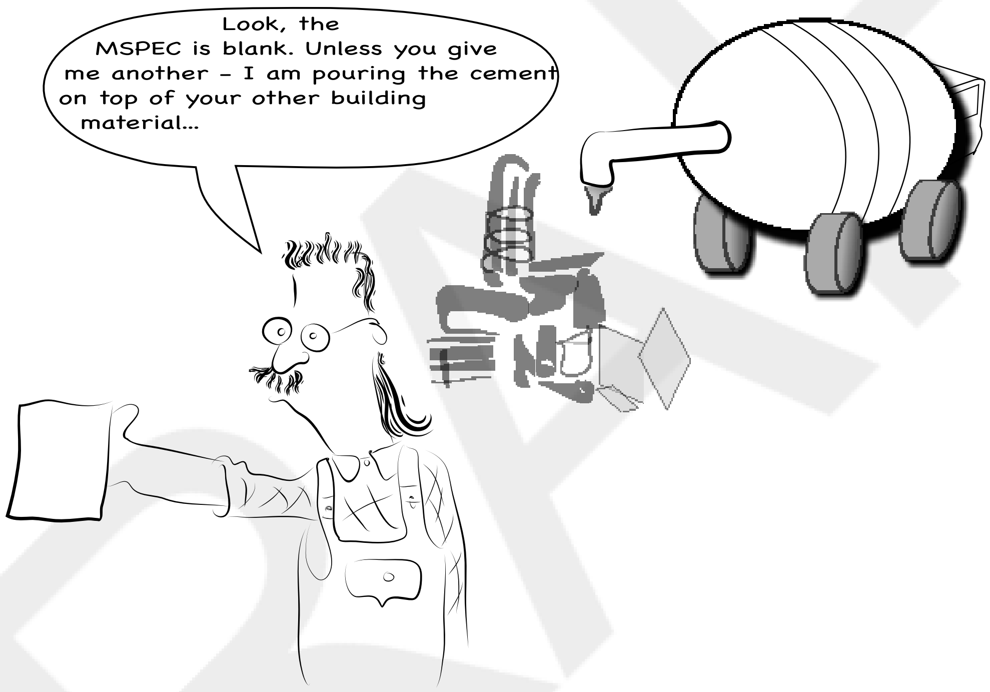
The Bill of Materials (BOM) is an immutable Buckminster artifact that contains a snapshot of all the information used at the time a CQUERY was resolved. When a BOM has been produced, it can be materialized many times with exactly the same result[2].

A BOM can also be used as input to a resolution process as a “pre-resolved component” which is useful in situations where a component is difficult to resolve and the details of its resolution is a concern that should be dealt with separately from those components that require it. (See the section called “Component types”, for more information about using a BOM in the RMAP — see section about the `bom` component type).

[2]provided that the components artifacts are still in the same locations and are unchanged



8 MSPEC — Materialization Specification


Look, the MSPEC is blank. Unless you give me another - I am pouring the cement on top of your other building material...

An MSPEC is just that — an order to the materialization ‘delivery agency’, instructing it about the details of what to put where and how — or put another way, it is like the assembly instruction you get with every flat package from IKEA.

By default (when executing a query with “*Resolve and Materialize*”), the materialization goes into the workspace. By instead using “*Resolve to Wizard*” you can modify the materialization settings and direct components to different locations (and types of locations). The wizard also lets you save the materialization settings in a MSPEC file. See the section called “Materialization wizard”.

Creating a MSPEC

A MSPEC can be created by saving it in the CQUERY *Materialization Wizard*. This will also create a BOM with the resulting resolution and the MSPEC will have a reference to this BOM.

You can also create a MSPEC manually, as a new XML file. Currently there is no “*New File Wizard*” for MSPEC. (As an alternative copy one created by the CQUERY wizard as a starting point).

A MSPEC should be saved in a file ending with '`.mspec`'.

Editing a MSPEC

There is no specific editor for the MSPEC artifact although the materialization wizard sort of functions as one, so you will most likely use use a XML or text editor. Using an XML editor is preferred as a good XML editor can be made to understand the MSPEC schema and thus validate what you write and also aid with code completion. See the section called “Configuring Eclipse for XML Editing”.

To edit a MSPEC in the wizard, you start by importing the MSPEC as the import will start the materialization wizard where you can modify the content and save it (you do not have to execute the actual materialization).

To import use either '*File → Import... → Other → Buckminster → Materialize from Buckminster MSPEC, CQUERY or BOM*', (or starting with the '*Import...*' command in the context menu over a MSPEC file), and the materialization wizard appears. You can read more about how this wizard works in the section called “Materialization wizard”.

The MSPEC*Modus Operandi*

A MSPEC can refer to a BOM, or a CQUERY. When the MSPEC is materialized and is referring to a BOM, it will use this information directly. When using a CQUERY, the query is first resolved and the resulting BOM is then materialized. (The terms *static-MSPEC*, and *BOM-MSPEC* are sometimes used to denote a MSPEC referencing a BOM, and *dynamic-MSPEC*, or *CQUERY-MSPEC* for the CQUERY case). In both the static and dynamic case, the result is a list of components, and these are the components that can be controlled with the MSPEC (i.e. you can not introduce an arbitrary component into the mix in the MSPEC, the component has to have been resolved).

(The *materialization wizard* understands both static and dynamic MSPECs, and if the MSPEC references a CQUERY the query is resolved as part of the process — this is very helpful as you want to make sure you don't have stale rules in the MSPEC as it references components by name/name patters).

The MSPEC describes default settings for the materialization (such as type of materialization, location and handling of conflict with existing files) — these are used if a more specific rule for a component does not state something different. The rules use a name pattern to match components and they provide the settings to use for components that match.

MSPEC in XML

Here is a very simple MSPEC

```
<?xml version="1.0" encoding="UTF-8"?>
<md:mspec xmlns:md="http://www.eclipse.org/buckminster/MetaData-1.0" ❶
    installLocation="/Users/henrik/TMP/" ❷
    materializer="filesysteme" ❸
    name="ExamplesForBook:1.0.0#OSGi" ❹
    shortDesc="This is a short description" ❺
    conflictResolution="update" ❻
    url="book-query.cquery" ❼
    />
```

❶ This is declared to be a MSPEC using the `md` name space. (See Buckminster XML Schemas).
❷ This sets the *default location* for the installation.

Note

If *not specified*, the current workspace is assumed by the workspace materializer, for `filesystem`, the file will be placed in a project called '`.buckminster`' (the name is configurable under *Eclipse* → *Preferences* → *Buckminster* → ‘*Buckminster project folder*’) in your workspace.

❸ This is the *materializer* to use (here ‘`filesystem`’).

Available materializers

file system	A location in your file system.
p2	This materializer is used to create a platform agnostic target platform. (This is not the same as performing a p2 install, as such an installation is always for a particular platform). The location is a directory in your file system for the p2 artifact repository. The p2 materializer essentially performs the same task as the PDE `repo2runnable` ANT-task, but with more advanced selection criteria (the Buckminster resolution process vs. copy entire repository).
(~~site.mirror~~)	Deprecated in Eclipse 3.5. Used when materializing using the now deprecated Update Manager. Still supported in the editor for older artifacts.
(~~target platform~~)	Deprecated in Eclipse 3.5, and is now an alias for p2 materializer. Use the p2 materializer instead. Still supported in the editor for older artifacts.
workspace	The materialization will go into a workspace. If *location* and *workspace* fields are both empty then the current workspace is used (this would be the normal case). If *only location* is specified the materialization treats the location as a workspace. If *both location and workspace* are stated, then the materialization is made to location, and the location is *linked* to the stated workspace.

A materializer must be specified.

❹ The name of the MSPEC can be declared — it is only used for human identification (and error messages). It is required, so declare it with a name that makes sense to you (the default name you get when creating a MSPEC with the materialization wizard is component name, version and type)

❺ A short description used for human understanding what the node is about (optional).

❻ The conflict resolution is specified; a choice of `update`, `fail` (it is an error if the location is not empty), `keep` (use what is there, do not update), and `replace` (remove existing first), can be made.

❼ The *url* attribute is a reference to either a BOM or a CQUERY that defines the set of components to materialize. (Here a CQUERY in the same location as the MSPEC is used so the URL is relative).

Using properties

It is possible to declare properties with `property`, or `propertyElement` in an `mspecNode`. This gives the ability to set default values in a rule, and to override them (in a query or on the command line, etc.) as described in Chapter 10, *Properties*.

Note

You have to edit the MSPEC with an XML editor to be able to define properties, as this is not supported by the materialization wizard. There is no restriction on using property values in he materializer wizard fields — it is only the setting of default values that is not supported.

Rules

The rules are very similar to the root of the MSPEC. Each rule is described with a child `mspecNode`. Here is an example

```
<mspecNode
    namePattern="org\.demo\.server\..*" ❶
    materializer="filesystem" ❷
    componentType="osgi.bundle" ❸
    resourcePath="useThisName" ❹
    exclude="false" ❺
    installLocation="/usr/local/server/test" ❻
    conflictResolution="update" > ❼
    <unpack ❽
        expand="true" ❾ // default == true
        suffix="tar.gz" ❿ // use if type not known
    />
</mspecNode>
```

❶ The *namePattern* is a regular expression. All components with matching names will be materialized in accordance with this rule. The first found rule that matches a component is used.

❷ The *materializer* to use (e.g. `filesystem`).

❸ The *componentType* is one of the support component types (e.g. `osgi.bundle`, `eclipse.feature`) — see the section called “Component types”.

❹ The *resourcePath* is the name of the resulting file or folder (depending on what is being materialized). A relative path is relative to the *installLocation*. This is typically used when a repository does not provide the real file name when reading a stream. When used with unpack this is the name for the folder into which the unpack takes place.

❺ A component can be skipped by setting exclude to true, the default is false.

❻❼ See the default setting with the same name.

❽ An unpack element is used to specify that an element should be unpacked, and optionally also expanded.

❾ If expand is set to true, an unpacked artifact is also expanded. The default is true. Example: an artifact may be in `tar.gz` format, and the unpack results in a `.tar` file, if also expanded, the content of the tar file becomes available.

❿ The suffix is used to specify the content format of the artifact. This is important if the repository does not set the name of the read artifact to reflect the type. As an example, suffix could be set to `tar.gz` to indicate a tar file that is then compressed with gz. The unpack can handle multiple uncompress, e.g. ‘`jar.pack.gz`’ first unpacks the gz, then, the pack, and finally (if specified with expand set to true), expands the jar.

Materializing a MSPEC

To materialize a MSPEC use either ‘*File → Import... → Other → Buckminster → Materialize from Buckminster MSPEC, CQUERY or BOM*’, (or starting with the ‘*Import...*’ command in the context menu over a MSPEC file). The materialization wizard appears. You can read more about how this wizard works in the section called “Materialization wizard”.

As with most of the Buckminster actions, you can also materialize a MSPEC headlessly. See Headless Commands.

Summary

As you have seen, the MSPEC is a quite simple concept “specify what goes where”, even if it first appears to be a bit daunting (“Yet another type of artifact, sigh...”) and with a somewhat roundabout way of editing (import), and using (also import).

The MSPEC is essential if you need to direct different parts to different places, but does not have to be used if you do not have this need.

9 Versions

Buckminster supports versions and version ranges from different versioning schemes. If you are working with Eclipse and OSGi based components, you are probably already familiar with how they work — and you can continue to use both versions and version ranges expressed just like they are expressed everywhere else in the Eclipse user interface. If you however step outside of the OSGi realm, there are many different versioning schemes in use. Buckminster's version handling is based on the *omni version* implementation found in Equinox p2.

In addition to the omni version implementation, Buckminster uses a version format naming and recognition scheme that makes it easier to handle different version formats in the user interface.

If you are only developing for Eclipse and OSGi, you will still benefit from the general overview of version and version ranges, and the handling of version qualifier substitution. If you are using Maven you will also need to learn about how to handle these types. Finally, if you are working on extending Buckminster, or if you want to use Buckminster in domains that use version formats that are (yet) unnamed in Buckminster, you need to understand more about the full omni version scheme.

The really detailed implementation details are found in Appendix C, *Omni Version Details*.

Omni Version introduction

The omni version is a canonical format. There is only one implementation and it is capable of describing versions in a wide range of versioning schemes. There is no central registry of version formats — each version or range instance carries the full specification. That each version carries the full definition means that versions can be transmitted between systems without risk of not functioning because of a missing definition. The fact that there is only one implementation means that there is no risk of not functioning because a particular implementation is not available in a system[1].

The omni version's canonical format is called the *raw* format, and it is constructed by parsing an *original version string* using a *format*. Since the raw format retains the format and original version stings, it is possible to recreate the input.

Omni versions always compare version based using the translated raw format. This creates a strict ordering of all versions across all versioning schemes.

[1]Earlier versions of Buckminster user an extension mechanism that required that a version type extension must be installed in order to parse and use a specification using the custom version type. In practice, this made it very difficult to define extensions if these were not contributed and accepted in the Buckminster code base. The omni version implementation in Eclipse 3.5 has solved this issue.

Example 9.1. An OSGi version expressed in raw

```
raw:1.0.0.'r1234'/format(n[.n=0;[.n=0;[.S=[a-zA-Z0-9_-];]]]):1.0.0.r1234
```

In Example 9.1, "An OSGi version expressed in raw" you can see what the OSGi version `1.0.0.r1234` looks like in raw format. Luckily, when using Buckminster, you don't have to use such strings in your input as you will see in the next section.

Buckminster and Omni Version

When specifying versions (and version ranges) in the various Buckminster XML based documents, the format pattern is referred to via a defined name. When using one of the graphical editors, the version format can be selected from a drop down list, and the original format string is entered in a separate field.

Internally Buckminster users omni versions for calculations.

New named formats can be introduced via a Buckminster extension point (see Appendix B, *Extending Buckminster*). No coding is required, but the extension must be provided by a bundle.

Note

Currently, the raw format is not available as a named format. See Eclipse Bugzilla Issue 282397 [https://bugs.eclipse.org/bugs/show_bug.cgi?id=282397], for status on this issue.

You can add new named formats by extending Buckminster — see the section called "Version type".

The new implementation in Buckminster for Eclipse 3.5 is backward compatible with respect to how input is specified. The only notable difference is that comparisons are now made on the canonical format and it is possible to compare versions using different formats[2].

Buckminster's named formats

Buckminster has the following named formats:

OSGi	The OSGi version format on the format `major.minor.micro.qualifier` where major, minor, and micro are numeric, and qualifier is a string. Major must be specified, but minor and micro defaults to 0 if omitted. The qualifier is optional.
Triplet	A version format used by Maven, and others, which is similar to OSGi, but where an empty qualifier compares as larger than any qualifier.
String	A single segment version using string comparison as performed by Java String.
Timestamp	A version format where the version is expressed as a timestamp and compared in ascending order.

For more details on the rules, and how these named formats are expressed, please see Appendix C, *Omni Version Details*.

Version ranges

Version ranges are expressed using the following syntax:

[2]This produced an error prior to Buckminster for Eclipse 3.5.

{ '[' | '(' } «*`lower-bounds`*» [',' | «*`upper-bounds`*»] { ']' | ')' }

Description:

- The range must start with a literal `[` or `(`, and end with a literal `]` or `)`.
- «*`lower-bounds`*» is a version in the specified format
- The use of `[` at the beginning means that the «*`lower-bounds`*» is included in the range.
- The user of `(` at the beginning means that the «*`lower-bounds`*» is excluded from the range such that any v > «*`lower-bounds`*» is included.
- Similarly, a `]` or `)` at the end specifies that the «*`upper-bounds`*» is included or excluded from the range.
- If the optional *`upper-bounds`* is not specified, the value of the «*`lower-bounds`*» is used here as well (and in this case both ends must be inclusive).
- The range must be well formed so that «*`lower-bounds`*» <= «*`upper-bounds`*».
- A single version «*`x`*» can be used where a range is expected — this means any v >= «*`x`*»

10 Properties

Revision History

Revision 3	July 27 , 2009	HL
Precedence table changed once again		
Revision 4	July 28 , 2009	HL
Faulty use of ${{0}} in format changed to warning.		

Buckminster's property functionality consists of providing access to system properties, the ability to declare properties, property expansion in attribute values, perform transformation of property values, a way to protect properties from being changed, and a mechanism defining a scope for property values.

It is possible to declare and transform properties in RMAP, CQUERY, and MSPEC artifacts. More specifically, the advisor nodes in a CQUERY can set property values that apply only when the advice is activated.

There are two ways properties can be set; by using the simpler `property`, or the more elaborate `propertyElement`.

Property expansion

Buckminster supports property expansion in almost all attribute values. A property referenced with `${«property name»}` is expanded, and the expression is replaced with the value of the property at the time of the expansion.

Advanced property expansion

If the value is also a property reference it will in turn be expanded. (A limit makes sure that property expansion does end if there is endless recursion). This can be used in different ways — here is an example:

```
<bc:propertyRef key="${aPropertyName}" />
```

Here, the *value* of the property `aPropertyName` will be used as the key to lookup the final value.

Warning

This however does *not* work:

```
<«element» format="/tmp/somewhere/${{0}}">
   <«function» ...>
</«element»>
```

This does not work because the `{0}` is expanded last by format.

Setting property value with “property”

Setting a property value with the element property is simple. Here is an example:

```
<property
    key="hatType"
    value="bandana"
    mutable="true"
/>
```

which sets the property `hatType` to the value `bandana` and allows the value to be overridden by properties set in contexts of higher precedence (see the section called “Precedence”).

Using “propertyElement”

The element `propertyElement`[1]has a lot more functionality than the simpler property element. The property value is constructed out of a concatenation of constants, the values of other properties, or transformations applied to such values.

Here is a simple example:

```
<propertyElement
    key="hatType"
    mutable="true" >
    <constant value="bandana" />
</propertyElement>
```

As you can see, the only difference from the simpler `property` is that the value is defined as a child element instead of an attribute. (The `constant` used in this example is just one of several functions to use when composing the value).

Property functions

The property functions are:

Property Functions

constant	Defines a constant value. `<bc:constant value="Hello" />`
propertyRef	Produces the value of a referenced property via name. `<bc:propertyRef key="some.property" />`

toLower	Concatenates the value of each child element and makes the result lower case.
	`<bc:toLower>`
	`    <bc:constant value="TAKE ME DOWN" />`
	`</bc:toLower>`

toUpper	Concatenates the value of each child element and makes the result upper case.
	`<bc:toUpper>`
	`    <bc:constant value="fix me up" />`
	`</bc:toLower>`

[1]Admittedly, a really bad name choice for this element.

format

Formats a resulting string based on a template string where parameters in the template are replaced with the values from the enclosed child elements.

```
<bc:format format="I am the {0}, of this {1}">
    <bc:constant value="result" />
    <br:constant value="example" />
</bc:format>
```

replace

Concatenates the value of each child element (except the special `match` element, that optionally is used to specify a more advanced matching), and replaces the result with a template string where matched segments can be included. Also see the section called “Replace function”.

```
<bc:replace pattern="."
    quotePattern="true"
    replacement="_" >
    <bc:constant value="Make my . become _, please." />
</bc:replace>
```

split

Concatenates the value of each child element, and splits the result based on a regular expression. The result appears as multiple child elements in an enclosing element. The split function must be placed last among the children in the enclosing element. Also see the section called “Split function”.

```
<bc:format format="Easy as '{0}{1}{2}'" >
   <bc:split pattern="[0-9]*">
       <bc:constant value="a34b8c9393" />
   </bc:split>
</bc:format>
```

Produces `Easy as 'abc'`as the result.

The property functions can be nested to arbitrary depth (except `constant`, and `propertyRef`).

Replace function

The replace function as seen in the section called “Property functions” replaces matched parts in the input with a replacement template. There are additional attributes that control the functionality, and a more advanced matching option available.

The boolean `quotePattern` attribute, when set to true will automatically quote all special characters giving them literal meaning. The default is `false`.

When using the replace function, either the two attributes `pattern` and `replacement` should both be specified, or one or more `match` child element should be used (for more advanced matching).

The match element. A match element performs a match/replace as directed by the attributes `pattern`, and `replacement`, and the pattern can be automatically quoted with `quotePattern=true`.

When multiple match elements are used, they are executed in the order they are defined, and the input to the first is the (non-match-element) siblings, and the input to subsequent match elements is the result of the preceding match.

```
<bc:replace>
    <bc:constant value="a.b#c-d" />
    <bc:match pattern="\." replacement="_" />
    <bc:match pattern="#" replacement="_" />
    <bc:match pattern="-" replacement="_" />
</bc:replacement>
```

The example produces `a_b_c_d`.

Split function

The split function as seen in the section called “Property functions” splits the input based on a pattern. There are two additional attributes that control the functionality.

style	This attribute can be set to `quoted`, `unquoted` or `groups`. The default is `unquoted`. Use `quoted` to automatically quote all special characters (unquoted requires quoting special characters if they are wanted as literals). The `groups` style require that segments are used in the pattern, and will use the matched segments as the resulting values.
limit	Limits the number of resulting elements.

Precedence

The context in which a property’s value is set defines the value’s precedence. In any given context, the property value with the highest precedence will be used, unless a property value is set with `mutable="false"`, which makes it the highest preceding value until the end of the context where it is defined.

The following table shows the property precedence during the materialization process (resolution, download, bind/install/import/copy/etc.).

Table 10.1. Materialization property value precedence

Priority	Context
1	Provider/Matcher rule in RMAP
2	*Advisor nodes* in CQUERY (if used)
3	CQUERY
4	MSPEC node (i.e. in a matching node, if used). Does not have any effect during resolution (materialization nodes are not matched until the end).
5	MSPEC (if used)
6	Properties file (passed on command line with `-P`, or set in the invoke action dialog).
7	Command Line (specified with `-D`«*xxx*»=«*yyy*»)
8	System Properties
9	Run/Debug Preferences — String substitution (in the IDE).
10	Target platform properties `target.`«*» automatically set during startup.
11	Static properties; `eclipse.home`, `workspace.root`, `localhost` automatically set during startup.
12	Resource Map

As an example — if the RMAP contains the following property definition:

```
<rm:property key="hatType" value="bandana" />
```

the CQUERY contains the definition:

```
<cq:property key="hatType" value="bowler" />
```

and an activated advisor node contains:

```
<cq:property key="hatType" value="top-hat" />
```

then, the value of the property when used as in `${hatType}` or using an element as in

```
    <bc:propertyRef key="hatType" />
```

in a resource map will be `top-hat`.

Note

An exception is thrown if a property has `mutable="false"` set and an attempt is made to override the value in a context with higher priority.

The precedence when component actions are invoked are the priorities 6-11 as shown earlier — but repeated in the table below for convenience. Note that property values used during the resolution are not remembered — that context is long gone when action are invoked.

Table 10.2. Action invocation property value precedence

Priority	Context
1	Properties file (passed on command line with `-P`, or set in the invoke action dialog).
2	Command Line (specified with `-D«xxx»=«yyy»`)
3	System Properties
4	Run/Debug Preferences — String substitution (in the IDE).
5	Target platform properties `target.«*»` automatically set during startup.
6	Static properties; `eclipse.home`, `workspace.root`, `localhost` automatically set during startup.

Typical property use

Properties are typically used for two purposes — passing values required when communicating with repositories such as user name and password, and to provide "routing" in the RMAP where components are picked from different repositories based on the value of properties.

The default values are typically set in the RMAP. Advisor nodes are used in CQUERY to override the defaults for specific components (those that match the rules in the advisor node). Properties set in the CQUERY itself overrides all the defaults in the RMAP. This can be used to make query statements like "I want all components to come from the milestone build repositories, except components matching org.myorg.hack.* which should come from the nightly repositories".

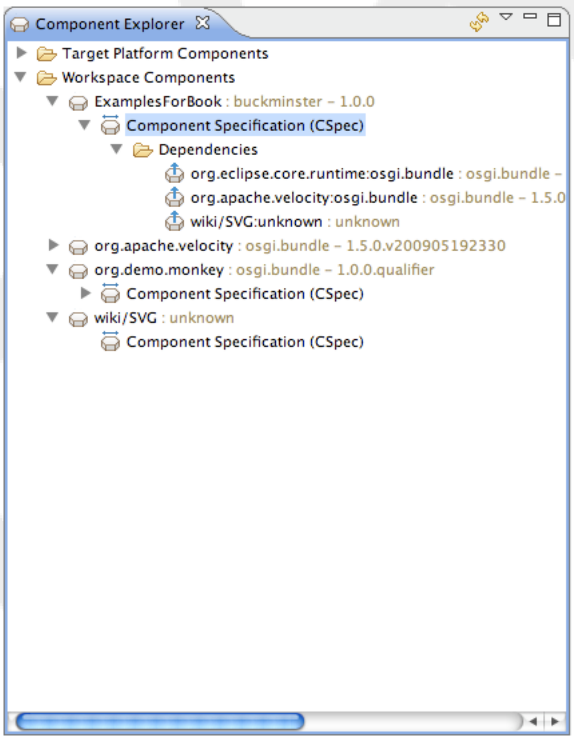
11

Buckminster User Interface

In this chapter we describe the Buckminster user interface. Buckminster is a bit “shy” in that it does not have much of a visual presence — you find Buckminster’s functionality in editors for the Buckminster artifacts, in popup menus over certain objects, and in some familiar places like the New File Wizard.

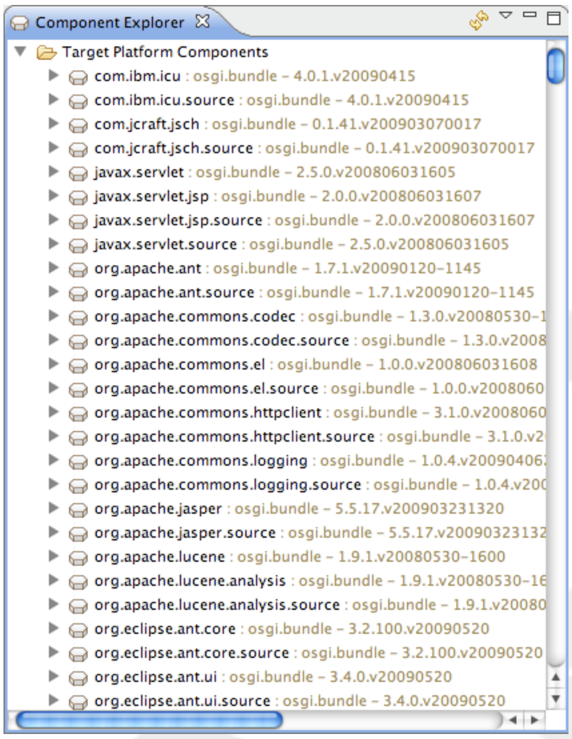
Component explorer

The Buckminster component explorer provides a view of all components known to Buckminster. You can explore all components in the workspace as well as those in the target platform.



```
Component Explorer
▶ Target Platform Components
▼ Workspace Components
    ▼ ExamplesForBook : buckminster - 1.0.0
        ▼ Component Specification (CSpec)
            ▼ Dependencies
                org.eclipse.core.runtime:osgi.bundle : osgi.bundle -
                org.apache.velocity:osgi.bundle : osgi.bundle - 1.5.0
                wiki/SVG:unknown : unknown
    ▶ org.apache.velocity : osgi.bundle - 1.5.0.v200905192330
    ▼ org.demo.monkey : osgi.bundle - 1.0.0.qualifier
        ▶ Component Specification (CSpec)
    ▼ wiki/SVG : unknown
          Component Specification (CSpec)
```

Showing content from the target platform:



```
Target Platform Components
  com.ibm.icu : osgi.bundle - 4.0.1.v20090415
  com.ibm.icu.source : osgi.bundle - 4.0.1.v20090415
  com.jcraft.jsch : osgi.bundle - 0.1.41.v200903070017
  com.jcraft.jsch.source : osgi.bundle - 0.1.41.v200903070017
  javax.servlet : osgi.bundle - 2.5.0.v200806031605
  javax.servlet.jsp : osgi.bundle - 2.0.0.v200806031607
  javax.servlet.jsp.source : osgi.bundle - 2.0.0.v200806031607
  javax.servlet.source : osgi.bundle - 2.5.0.v200806031605
  org.apache.ant : osgi.bundle - 1.7.1.v20090120-1145
  org.apache.ant.source : osgi.bundle - 1.7.1.v20090120-1145
  org.apache.commons.codec : osgi.bundle - 1.3.0.v20080530-1
  org.apache.commons.codec.source : osgi.bundle - 1.3.0.v2008
  org.apache.commons.el : osgi.bundle - 1.0.0.v200806031608
  org.apache.commons.el.source : osgi.bundle - 1.0.0.v2008060
  org.apache.commons.httpclient : osgi.bundle - 3.1.0.v2008060
  org.apache.commons.httpclient.source : osgi.bundle - 3.1.0.v2
  org.apache.commons.logging : osgi.bundle - 1.0.4.v20090406
  org.apache.commons.logging.source : osgi.bundle - 1.0.4.v200
  org.apache.jasper : osgi.bundle - 5.5.17.v200903231320
  org.apache.jasper.source : osgi.bundle - 5.5.17.v20090323132
  org.apache.lucene : osgi.bundle - 1.9.1.v20080530-1600
  org.apache.lucene.analysis : osgi.bundle - 1.9.1.v20080530-16
  org.apache.lucene.analysis.source : osgi.bundle - 1.9.1.v20080
  org.eclipse.ant.core : osgi.bundle - 3.2.100.v20090520
  org.eclipse.ant.core.source : osgi.bundle - 3.2.100.v20090520
  org.eclipse.ant.ui : osgi.bundle - 3.4.0.v20090520
  org.eclipse.ant.ui.source : osgi.bundle - 3.4.0.v20090520
```

Note

The explorer sometimes gets out of sync with the set of available components. The collection of the data can be time consuming and is not always up to date (some files can have changed). If you do not see what you expect, simply press on the refresh button.

To open the Component Explorer click *Window* → *Show View* → *Other...* → *Buckminster* → *Component Explorer*.

Component outline

The Buckminster component outline view shows the main CSPEC information known to Buckminster for the component associated with the currently selected resource (i.e. the component for the file that is active in an editor or view).

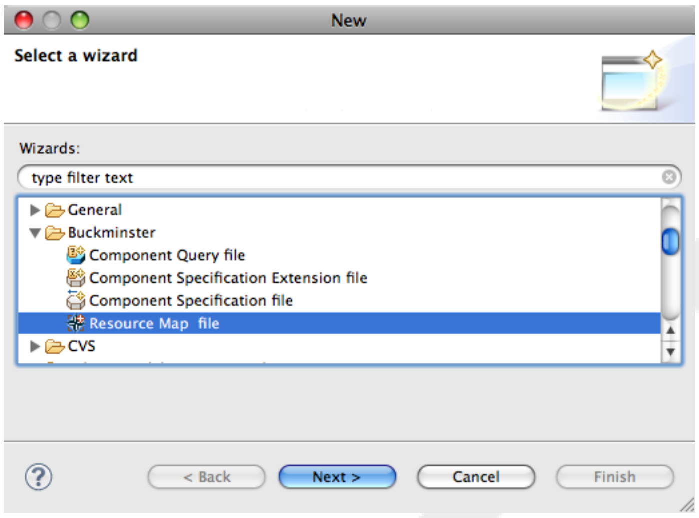
This is basically the same view as the Component Explorer, but for one component at a time.

You will soon notice that the component outline sometimes gets confused over what is currently selected — it will then show nothing. Simply selecting the component you are interested again makes it show up.

To open the Component Outline click *Window* → *Show View* → *Other...* → *Buckminster* → *Component Outline*.

New file wizards

The Buckminster wizards for creating new files are located at *File* → *New* → *Other...* → *Buckminster*. The dialog looks like this:



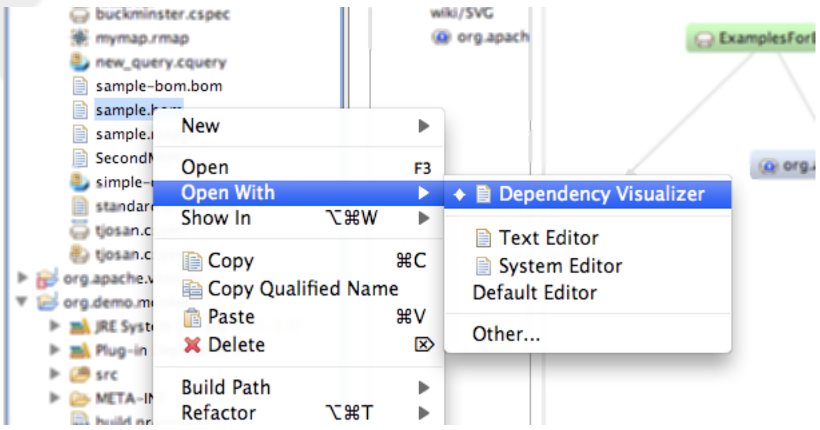
When using the wizard you will get help with the basic XML declarations, and in some cases a starting template. You can naturally also create files manually.

BOM visualizer

The BOM visualizer presents a graph view of a resolution[1]. It is possible to explore the dependencies and to get a detailed understanding on how all included components depend on each other. The visualizer offers drill down (see only part of the resolution), see shortest path to root, or all paths, focus on one component, and arrange the nodes using different layout algorithms. You can select if you want to see the components picked from the target platform or not. (In the screenshot below, we did include the platform).

The visualizer is available as an “editor” for a BOM file. So the steps you need to take are:

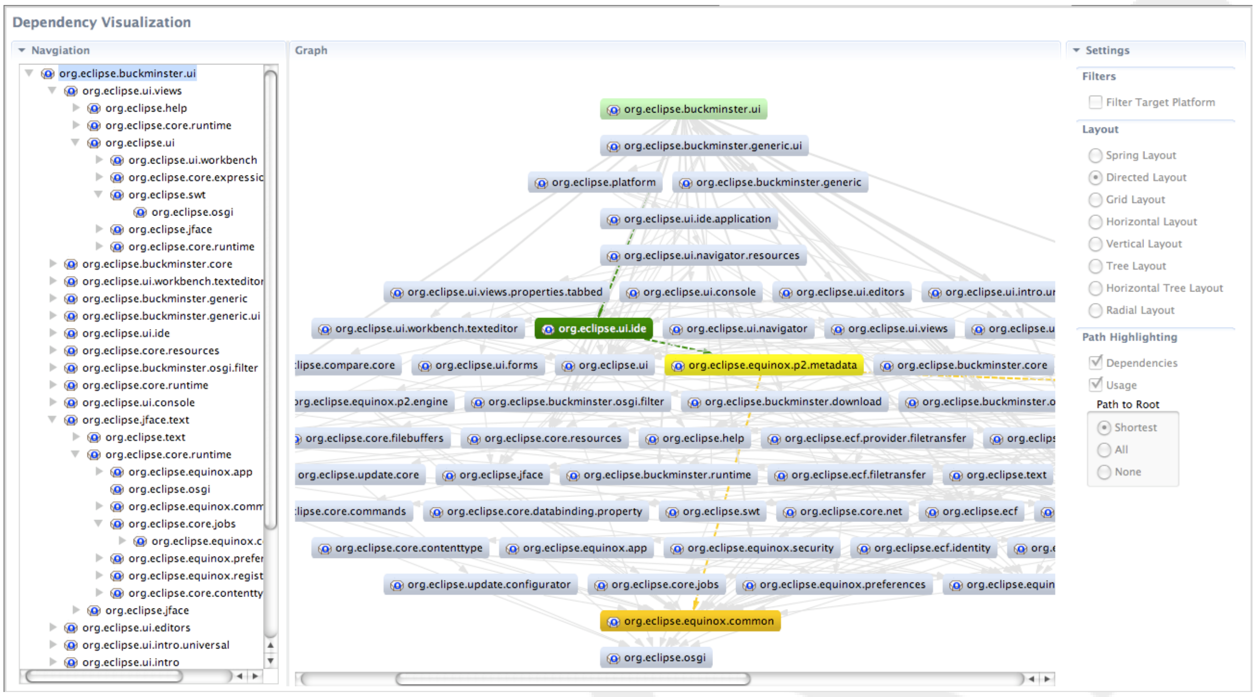
1. Use a CQUERY to query for the resolution you want.
2. Select “Materialize to Wizard”, and then “Save BOM” (You can then cancel the query as you only need the resolution meta data, and not the actual components).
3. In the context menu for the created BOM file, select *Open With* → *Dependency Visualizer*.



[1]The visualizer was not released as part of the original Eclipse 3.5 Galileo release, but is include in the Buckminster updates for 3.5.

Tip

You can visualize any component in the workspace or target platform — simply enter the component's ID in the CQUERY editor.


Dependency Visualization
Navgiation
Graph
Settings
org.eclipse.buckminster.ui
org.eclipse.buckminster.generic.ui
org.eclipse.platform
org.eclipse.buckminster.generic
org.eclipse.ui.ide.application
org.eclipse.ui.navigator.resources
org.eclipse.ui.views.properties.tabbed
org.eclipse.ui.console
org.eclipse.ui.editors
org.eclipse.ui.workbench.texteditor
org.eclipse.ui.ide
org.eclipse.ui.navigator
org.eclipse.ui.views
org.eclipse.ui.forms
org.eclipse.ui
org.eclipse.equinox.p2.metadata
org.eclipse.buckminster.core
org.eclipse.buckminster.osgi.filter
org.eclipse.buckminster.download
org.eclipse.core.resources
org.eclipse.help
org.eclipse.ecf.provider.filetransfer
org.eclipse.update.core
org.eclipse.jface
org.eclipse.buckminster.runtime
org.eclipse.ecf.filetransfer
org.eclipse.text
org.eclipse.core.databinding.property
org.eclipse.swt
org.eclipse.core.net
org.eclipse.ecf
org.eclipse.core.contenttype
org.eclipse.equinox.app
org.eclipse.equinox.security
org.eclipse.ecf.identity
org.eclipse.update.configurator
org.eclipse.core.jobs
org.eclipse.equinox.preferences
org.eclipse.equinox.common
org.eclipse.osgi

Filters

- [ ] Filter Target Platform

Layout

- ○ Spring Layout
- ● Directed Layout
- ○ Grid Layout
- ○ Horizontal Layout
- ○ Vertical Layout
- ○ Tree Layout
- ○ Horizontal Tree Layout
- ○ Radial Layout

Path Highlighting

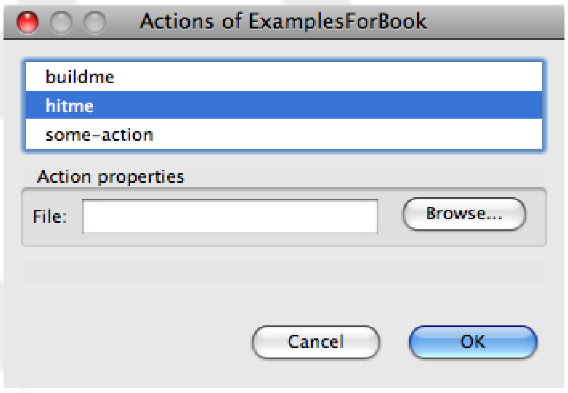
- [✓] Dependencies
- [✓] Usage

Path to Root

- ● Shortest
- ○ All
- ○ None

Invoking actions

You can invoke Buckminster actions (i.e. trigger actions in CSPECs) from the user interface. To do this, open the context menu for the component and select *Buckminster* → *Invoke Action...* and this dialog appears:


Actions of ExamplesForBook

- buildme
- hitme
- some-action

Action properties

File: [ ] Browse...

Cancel OK

The dialog shows all publicly available actionable attributes (i.e. public actions, and public groups that include actions). You can also pass properties to the action via a properties file. This is typically the properties file that you also use when running the same action in headless fashion. (The Buckminster invoke action dialog will remember the path to the last used properties file, as you will see when you open the dialog a second time).

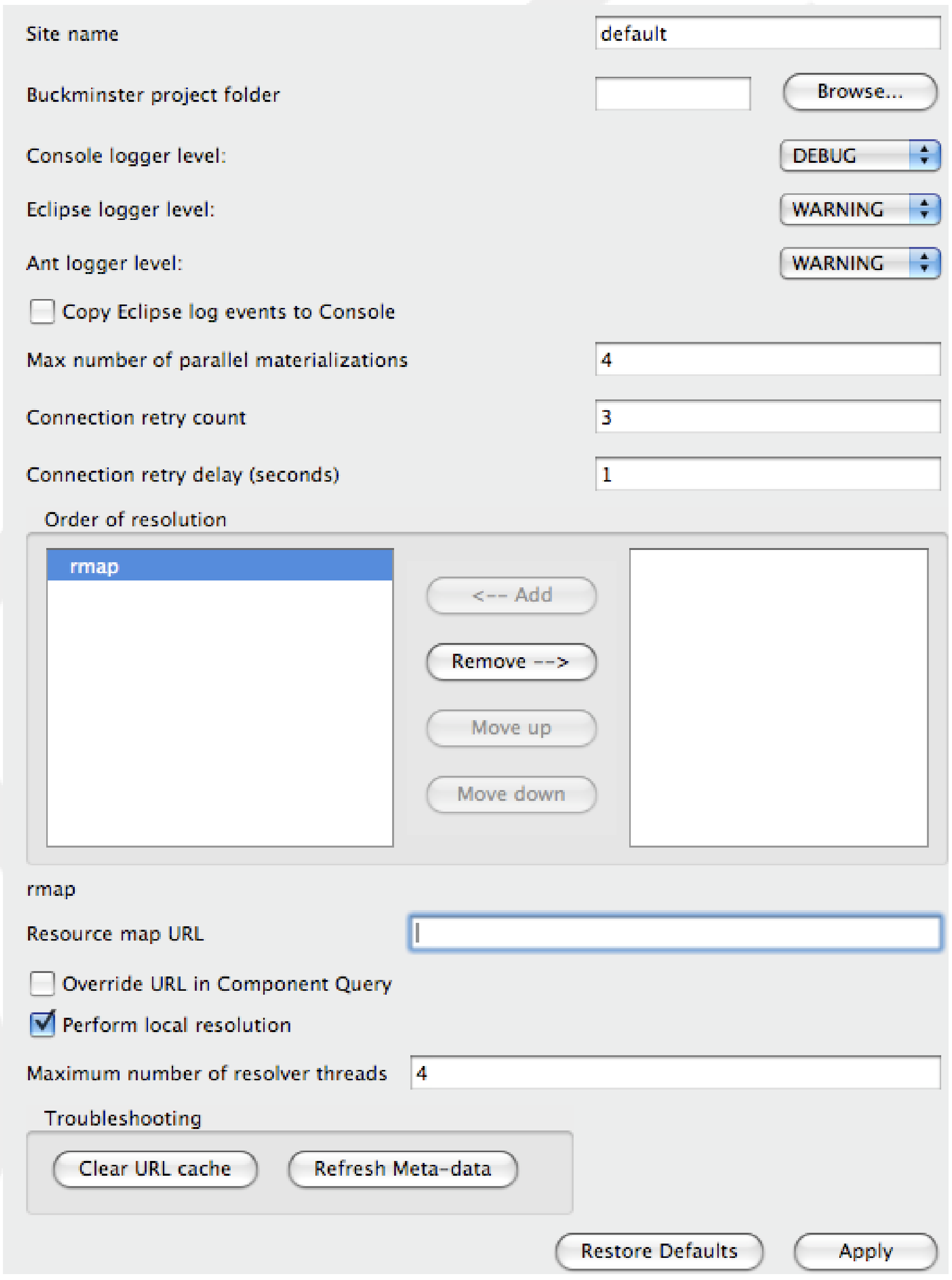
The screenshot above is just an illustration. When you are using this with your components, you will typically see many automatically generated actions. You can see what the automatically generated actions do in the 'Component Types' reference guide.

Editors

Buckminster has graphical editors for CSPEC and CQUERY — these are covered in the respective chapters.

Preferences

Buckminster's preference setting are found under *Eclipse* → *Preferences...* → *Buckminster*. The Buckminster preference pane looks like this: *New screenshot needed - the OPML flag is not included - but see Bug 288359 as the preference pane may be refactored.*



Site name	default
Buckminster project folder	
Console logger level:	DEBUG
Eclipse logger level:	WARNING
Ant logger level:	WARNING
[ ] Copy Eclipse log events to Console	
Max number of parallel materializations	4
Connection retry count	3
Connection retry delay (seconds)	1

Order of resolution: rmap

rmap

Resource map URL	
[ ] Override URL in Component Query	
[✓] Perform local resolution	
Maximum number of resolver threads	4

~~Site name~~	Deprecated.

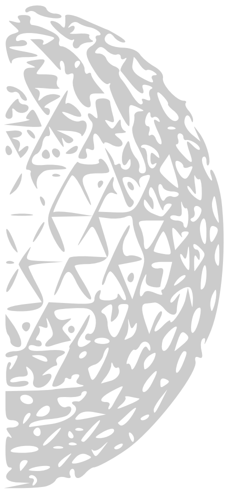
Buckminster project folder	The name of the project where Buckminster keeps workspace related information. Defaults to `.buckminster`.
Console logger level	The logging level for things that log to the console.
Eclipse logger level	The logging level for things that log to the Eclipse log.
Ant logger level	The logging level for things that log to the ANT log.
Copy Eclipse log event to Console	When checked, all events to the Eclipse log are echoed to the console. This means that both Buckminster output and Eclipse Events will go to the console. This makes it possible to view them in the sequence they occur. This is useful when there are problems with a resolution or build, as it is difficult to otherwise correlate general problems with build problems when outputs are separate.
Max number of parallel materializations	This controls how many materializations (i.e. downloads) to run in parallel.
Connection retry count	In case of a failure to connect/download other than “file not found”, Buckminster will repeat the attempt after a delay. This setting tells Buckminster how many times to try.
Connection retry delay (seconds)	This tells Buckminster the amount of time in seconds to wait between connection retries.
Order of resolution	Buckminster supports plugging in different resolution services. The one service that is always available is the RMAP resolution service. You may have other such resolution services in your IDE, in which case, you can select if they are used and in which order they are consulted.
Resource map URL	This is a default RMAP URL. If entered in the preferences, a CQUERY without a RMAP URL will use this default.
Override URL in Component Query	If selected, the default *Resource map URL* will override the URL in any CQUERY.

Note

This is only available when running a CQUERY from the user interface — not when running headless Buckminster. This means that you should only use this as a testing/troubleshooting mechanism when you need to run a query with a different RMAP.

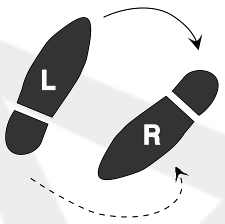
Perform local resolution	If a remote resolution service is in use (See ‘Order of resolution’ above), this setting will disable the use of any remote resolution.
Maximum number of resolver threads	This controls how many parallel threads the resolution process will use. When running into resolution problems it is useful to set this to 1 — as this disables multi-threading. All trace output will then be in sequence for the single executing thread. This makes it much easier to read the output. Increasing the value will have a positive effect on resolution speed when many servers are contacted during the resolution, or when the in-

	volved servers have high bandwidth and allow many connections from the same client.
Enable support for component bookmarks (OPML)	When checked, Buckminster will scan for a `buckminster.opml` file in the component and include the result in the resolution.
Clear URL Cache	This clears Buckminster's cache of downloaded artifacts, thus forcing Buckminster to download them again.
Refresh Meta-data	This refreshes Buckminster's metadata. Buckminster tries to stay in sync with the target platform and the workspace. This is however not perfect as things can change without Buckminster noticing (no events are generated in some cases, esp. when files are changed via means external to Eclipse), or where things change in such an order that Buckminster will get confused.
Restore Defaults	Handy if you managed to configure yourself into trouble...



12 Troubleshooting

In this chapter we have collected some advice regarding troubleshooting. Throughout the chapters there are some bits of advice and warnings, but these are not as easy to find when something is indeed wrong — and it can be hard to tell from the symptoms what is really happening.



THIS CHAPTER IS W.I.P....

Installation Issues

Here is a checklist for common issues when installing:

- Obviously, the first thing to check is that you followed the instructions on the Buckminster Download page.
- Installing the 3.5 version into a 3.4 or vice versa is a really bad idea — make sure you are using the correct update site.
- Installing the headless support into the standard Eclipse IDE will cause all sorts of problems — check that you did not use the wrong URL from the download page by mistake.
- There are two different SVN adapters to choose from — you can not use both at the same time. If you tried to install one and it failed — make sure you do not already have the other installed. Some users have also been confused over which SVN client they are using — if you are uncertain check if you are using Subclipse, or Subversive.
- Sometimes there are issues with Eclipse download site (infrequent), and sometimes there are issues regarding mirroring and not all mirrors being up to date (or simply misbehaving). The mirror selection may be confused over your location and may send you to the least optimal server. You may also encounter network errors.

  In most cases, the issues sort themselves out — just try again a little later. But there are other things you can do. One thing is to download the Buckminster archived site and perform a local installation. This gives you manual control over from where you are downloading the archive.

- If you still have trouble installing — try installing into a fresh Eclipse 3.5 installation. The issues may have nothing to do with Buckminster at all.

Headless issues

Here are some common issues encountered when running the headless buckminster.

- When nothing works, and you just get errors... The standard headless can not do much except a few basic commands. You must install the wanted features into the headless product before it is useful. If you forget this, you will see error messages for services and classes that can not be found. To correct, follow the instructions in Appendix A, *Installation*.
- Under no circumstances is it a good idea to install the Buckminster features intended for use in the Eclipse IDE into the headless product. Make sure you did not use the wrong URL when installing.

Resolution issues

When something is wrong it usually manifests itself as an failure to resolve a query. There are many possible causes for a failed resolution — ranging from the trivial to fix to really hard problems. There is naturally also a difference how to troubleshoot something not resolving for the very first time (i.e. when you are just starting out), and when you get resolution errors for something that once worked.

In general there are two types of issues:

- *Addressing issues* — there is nothing wrong with the actual dependencies but the components are not found when looking them up in the RMAP. This breaks down further into:
  - RMAP issues — it does not route the lookup to the correct place.
  - Repository issues — the expected content is not there.
- *Dependency issues* — there are erroneous/unwanted/conflicting dependencies.

When looking at a “unresolvable” issue, there is no way to know the type of problem. Here is a check list:

- Does the unresolvable request look reasonable (i.e. does it have a reasonable name, correctly spelled, and with a version that makes sense)? If not, then the problem is to be found in a component that has stated a dependency in error. Use ‘materialize to wizard’ and look at the result to see the dependency chain that leads to the unresolvable component.
- Check the RMAP route taken. You need to turn on logging and also disable parallel resolution (or you get events from multiple threads to decipher) — this is done under Preferences the section called “Preferences”. Run the query again, and look at the output — is it using the expected route through the RMAP? If it is difficult to find the information due to requests for many/similarly named components, try a simple query that just asks for the unresolved component.
- Try a wider search — create a query for the unresolved component, but open up the range, and see what it resolves to — maybe the requested version simply does not exists.
- If you see it trying the correct RMAP route, and the data is in the repository, but it still fails, maybe you have some mapping/formatting in the RMAP that it is wrong. Try a hacked RMAP where you use a constant expression for the actual component, and then query for just this unresolved component — if possible to get it when using a constant, then you know you have a faulty mapping entry in the RMAP.
- Maybe you have issues that are specific to a reader type — if possible make a copy of the component in your local file system and set up an entry for it in the RMAP — if this works then you know that you have issues in the original RMAP entry for that repository.

- Maybe you are using stale information — try refreshing the meta data, and clear the Buckminster cache. This is done under Preferences the section called “Preferences”.

Materialization issues

What are typical issues here? (Authorization obviously.)

- If you see errors for unresolved bundles for platforms other than the one you are running on, then you are probably trying to build a RCP application (and where this build is not constrained to your current platform), and you have forgotten to install the Eclipse Delta Pack.

Execution issues

What is typically wrong here? (Did you forget passing properties? Forgetting qualifier replacement? ANT related issues (I recall having some that Thomas helped me sort out)?

- Where did the files go?

- Strange qualifier, or qualifiers say ‘qualifier’

Component issues

If you are having problems with components, here are some things to check:

- Buckminster currently places Eclipse products in the name space used for a component’s attributes. You will run into trouble if you have given attributes the same name as a product it includes. The reverse is also true (you have a product that has the same name as an automatically generated attribute), but this is very unlikely as the automatically generated attribute names are quit technical.

- If you have troubles with bundles (plugins or features), check that you are using the correct meta data format - they should have “Bundle-ManifestVersion: 2” and the meta data should show without errors in the Eclipse manifest editor.

Part III. Examples

In this part we are showing several examples, from the simple Hello World kind, to a full build of a RCP product and p2 repository. As you probably want to run through these examples live, you should follow the instructions in Appendix A, *Installation*, so you can experiment with the examples yourself.

13 *Building a p2 Update Site*

In this example we will build a p2 update site for a plugin and a feature. This example is very easy to set up from scratch so there is no source available.

This example demonstrates:

- Building a site using automatically generated actions
- Defining a category that is used by “Install new Software”
- Defining and using properties to control the Buckminster build
- Installing from a generated p2 repository

Prerequisites. In order to run this example, you need to have JDT, PDE, as well as Buckminster installed.

Creating the content

In order to create an update site, we must naturally have something to publish to this update site. We will create a plugin and a feature that references this plugin. The only role for the feature is to categorize it and thus make it appear in the p2 user interface for installing new software.

Creating the plugin

Create a plugin by using *File* → *New* → *Other...* → *Plugin Project*. In the wizard that appears, give your project the name ‘`org.demo.demoplugin`’, select *Next* in the wizard (leave all the default settings) until you reach the template selection page. Select the template called ‘`Plug-in with a view`’, and click *Next*. Change the *View Name* from ‘`Sample View`’ to differentiate it from other samples (e.g. ‘`Demoplugin View`’). Click *Finish*, and you should get a project in your workspace.

Creating the feature

Create a feature by using *File* → *New* → *Other...* → *Plug-in Development* → *Feature project*. In the wizard that appears, name the project ‘`org.demo.demofeature`’ and click *Next*. A list of available plugins is displayed. Select the ‘`org.demo.demoplugin`’ and click *Finish*.

Creating the site feature

We need an additional feature that describes what we want to include in the update site we are going to build. In this feature we will also categorize the content.

Note

A feature that is published as an update site by Buckminster does not include itself in the update site.

Create the site feature by using *File* → *New* → *Other...* → *Plug-in Development* → *Feature project*. In the wizard that appears, name the project ‘`org.demo.demosite`’ and click *Finish*.

In the feature editor that opens, navigate to the ‘*Included Features*’ tab, select Add and pick `org.demo.demofeature` in the pop up list that appears.

Navigate to the build.properties tab, and add the following three lines:

```
category.id.org.demo.democategory=Cool Features (demo)
category.members.org.demo.democategory=org.demo.demofeature
category.description.org.demo.democategory=Cool stuff build in a Buckminster demo
```

This defines a category called ‘`org.demo.democategory`’ and its label, description, and members are defined.

Note

Don't forget to save the feature.

Building the site

Building the site is easy — everything we need is generated automatically, but we have to provide some properties to Buckminster that defines where the output should go.

These properties can be set in a properties file, or you can set them directly in the IDE under *Eclipse* → *Preferences* → *Run/Debug* → *String substitution*. If you want them in a properties file instead (which is reusable when running headless) then you should create a file called ‘`buckminster.properties`’ in the `org.demo.demosite` project, with the following content:

```
# Where all the output should go
buckminster.output.root=${user.home}/demosite

# Where the temp files should go
buckminster.temp.root=${user.home}/tmp/demosite.tmp

# How .qualifier in versions should be replaced
qualifier.replacement.*=generator:lastRevision
```

These parameters direct where output and temporary-output should go, and it defines how versions marked as ‘qualifier’ should be handled. Here we selected to use the ‘last revision’ scheme even if we do not yet have this because our projects are not stored in a source code repository yet. There are many other parameters that control how the building is done, if signing and packing should take place etc. Please see ‘description of site.p2’.

With the property file in place, we can now build the site. Right click over the `org.demo.demosite` project, and select *Buckminster* → *Invoke Action...*, and (if you decided to use a properties file) in the pop up that appears, browse for the `buckminster.properties` file we just created (you will need to navigate in the file system to find the workspace, project, and then the file since the properties files are typically placed in the file system, and not in the workspace). When the properties file has been found and selected (alternatively set the properties using Run/Debug string substitution), select the action called ‘`site.p2`’ in the list of presented actions, and click ‘`Ok`’. You will now see some trace output, and if everything worked ok, you now have an update site.

Using the update site

The first question is typically “where is the site?”. Remember that we set the output root to be under `${user.home}`, so you need to navigate to your home directory, and then to the directory `demosite/org.demo.demosite_1.0.0-eclipse.feature/` where you will find two directories ‘`site`’, and ‘`site.p2`’ — the first directory is a Eclipse 3.4 update site that was built as a bonus, and the ‘`site.p2`’ is the p2 update site. (You will find the output for every project that was built under the respective project name under `${user.home}/demosite`, and once you are done with this example, you probably want to remove both this and the `${user.home}/tmp/demosite` directory).

You can now take the content under ‘`site.p2`’ and make that available as an update site by using a web server. You can also use it directly in your Eclipse IDE to try it. To do this, go to Help → Install New Software, and enter the URL to your just created repository (either prepend `file://` to the absolute path, or use the “`Local`” button to browse for the location).

You will now see the ‘*Cool Features (demo)*’ category, with the *Demofeature* as its content. Select it, and install to run the view in your IDE. (If you do this, you probably want to uninstall it, and then remove the update site when you are done testing)

14 Building a Legacy Update Site

Buckminster has support for building legacy update sites (i.e. the simpler older format using a site.xml file). It is however recommended that p2 based update sites are used for Eclipse 3.4 and later. You can skip this example if you are not going to produce legacy sites.

To use the support for legacy update sites, you will need to add the actions to the CSPEC used to define the site as the support is not automatically generated by Buckminster (as it is for p2 site generation). This example shows you how to do this step by step.

Creating the update site project

To Buckminster, an update site is not different from any other type of component. We need to keep the definition of the update site in a project, and we need some additional meta data to enable convenient building of the site. The first step is to create the update site project:

1. Right click in the Package Explorer and select *New → Project*
2. In the *New Project* wizard that pops up, open the *Plug-in Development* folder
3. Click on *Update Site Project*
4. Give the project a name. In this example we use `org.test.update`.
5. Click on *Finish*

The project appears in the workspace and it contains one single file, an empty `site.xml`. Buckminster will use this as the update site template. This means that you can add site categories to this file but you should *not* add any features. Buckminster will generate a new `site.xml` where the features are added.

Creating an index.html file

Just create an empty file in the root folder of your new update site project for now and call it `index.html`. You can add content to this file that will be what the user will see if they happen to access your update site with a browser.

Creating the Component Specification

Buckminster describes all components in terms of CSPECs. The update site is no exception. Here are the steps to create such a CSPEC and enter the needed information:

1. Right click on the update site project and select *New → Other*

2. In the *New* wizard that pops up, open the *Buckminster* folder

3. Select *Component Specification file* and click on *Next*

4. Click on *Finish* to accept the default values for *Container* and *File name*

A file named `buckminster.cspec` is created in the project and the Buckminster CSPEC editor opens. Do not change the name of this file.

Main information

- The name of the component is normally the same as the name of the project. This makes it easier to find the component.
- The component type must in our case be set to `buckminster`.
- The version can be any OSGi compliant version such as `1.0.0`

Artifacts

Artifacts denotes files and folders that are present inside of a component. The action that will create the update site needs to know about the `site.xml` template and other files to copy so we need to add that to our specification:

1. Click on the *Artifacts* tab
2. Click on *New* below the Artifacts table
3. Enter a name such as `site.template`
4. Click on the *New* button next to the *Path* table
5. Enter the name `site.xml` in the dialog that pops up and click *OK*
6. Click on *New* below the Artifacts table
7. Enter the name `site.rootFiles`
8. Click on the *New* button next to the *Path* table
9. Enter the name `index.html` in the dialog that pops up and click *OK*

We now have two artifacts, each with one path. The separation is necessary in this particular case since the build action will reference the artifacs separately. An artifact may have several paths and you can add as many files and folders as you wish to the `site.rootFiles` artifact.

Dependencies

We need to define the features that will be included on the update site. Buckminster considers them to be *dependencies*:

1. Click on the *Dependencies* tab
2. For each feature that you want to add, repeat the following:
   a. Click on *New* just next to the Dependencies table
   b. Enter the name of a feature component
   c. Set the *Component Type* to `eclipse.feature`

d. Click *OK*

Groups

The build action expects one prerequisite that lists all the feature jars and one that lists all plugin jars. Conveniently, Buckminster has already generated CSPECs for all features with attributes that will provide just that. A feature will always have the two public attributes:

`feature.jars`	This is the transitive closure of all features (including the feature itself) in jared format.
`bundle.jars`	This is all the plugins that the transitive closure of all features is referencing in jared format.

Since we do not have a feature that describes the update site itself, we need to create two new groups. One to group all the feature.jars, and one to group the bundle.jars.

1. Click on *New* just below the *Groups* table
2. Enter the name of the group. Call it `feature.jars`
3. For each feature that should be included in this group
   a. Click on *New* just next to the prerequisites table
   b. Select a feature from the drop down menu.
   c. Enter the name `feature.jars`
   d. Click *OK*

Repeat these steps for a group that is called `bundle.jars` that references the `bundle.jars` attribute of each feature.

Defining site categories

This step is not required, but typically you want to categorize the contents of the update site. In order to define a *Site Category*, you must first create the category in the `site.xml` file and then add it as a group in the CSPEC.

In order to create a category in the `site.xml` file, do the following:

1. Double click on the `site.xml` file. The Update Site Editor opens
2. On the *Site Map* tab, click *New Category*
3. Give the category a name. In this example we will use `Basic`
4. Add a label such as “`Basic features`”

Repeat these steps and create an additional categogy named `Optional` with label “`Optional features`”.

Back to the CSPEC editor:

1. Click on the *Groups* tab
2. Click on *New* just below the Groups table
3. Enter the name of the group, i.e. *Basic*

4. For each feature that should be included in this group

    a. Click on *New* just next to the prerequisites table

    b. Select a feature from the drop down menu.

    c. Enter the name `feature.jars`

    d. Click *OK*

Repeat these steps for the `Optional` group.

Attributes, Groups, and Actions are all *Attributes* in Buckminster terms. A group contains attributes. Subsequently, a group can include other groups. This allows for a simplification of the `feature.jars` group that we created earlier. Instead of having that group include all features, it could instead include the two category groups, i.e. instead of having:

```
feature.jars
    a[feature.jars]
    b[feature.jars]
    c[feature.jars]
    d[feature.jars]

Basic
    a[feature.jars]
    b[feature.jars]

Optional
    c[feature.jars]
    d[feature.jars]
```

we can simplify and do:

```
feature.jars
    [Basic]
    [Optional]

Basic
    a[feature.jars]
    b[feature.jars]

Optional
    c[feature.jars]
    d[feature.jars]
```

(The example assumes that a, b, c, and d are features and [xxx] denotes attribute xxx).

If you follow the example (simplification or not), you now have four groups, `Basic`, `Optional`, `feature.jars`, and `bundle.jars`.

The action

The final thing to add to the CSPEC is the action that will trigger the actual build of the update site.

1. Click on the *Actions* tab

2. Adding General action information

    a. Click on *New* below the *Actions* table

    b. Enter the name `build.site`

    c. Check the *Public* checkbox

    d. Enter the *Actor Name*: `ant`

3. Adding the `site.template` prerequisite
   a. Click on the *New* button next to the *Prerequisites* table
   b. Leave *Component* blank (this means *current component*)
   c. Select `site.template` from the *Attribute* combobox
   d. Enter the *Alias* name `template`
   e. Click *OK*
4. Adding the `rootFiles` prerequisite
   a. Click on the *New* button next to the *Prerequisites* table
   b. Leave *Component* blank
   c. Select `site.rootFiles` from the *Attribute* combobox
   d. Enter the *Alias* name `rootFiles`
   e. Click *OK*
5. Adding the features
   a. Click on the *New* button next to the *Prerequisites* table
   b. Leave *Component* blank
   c. Select `feature.jars` from the *Attribute* combobox
   d. Enter the *Alias* name `features`
   e. Click *OK*
6. Adding the plugins
   a. Click on the *New* button next to the *Prerequisites* table
   b. Leave *Component* blank
   c. Select `bundle.jars` from the *Attribute* combobox
   d. Enter the *Alias* name `plugins`
7. Adding general properties. These properties control the general behavior.
   a. In the middle pane, click on *Properties*
   b. Click on *New* next to the *General Properties* table
   c. Enter *Key* `site.name` and a value such as `test.archivedsite`
   d. Click *OK*
   e. If you want your site to have some extra suffix such as `_incubation` then:
      i. Click on *New* next to the *General Properties* table
      ii. Enter *Key* `site.extra.suffix` and a value such as `_incubation`
      iii. Click *OK*

8. Adding actor properties. These properties control behavior specific to an actor. We need two of them. One to specify the ant build script that will be used and another to specify what ant target to call in that file.

   a. Click on *New* next to the *Actor Properties* table

   b. Enter *Key* `buildFileId` and the value `buckminster.pdetasks`

   c. Click *OK*

   d. Click on *New* again

   e. Enter *Key* targets and the value `create.legacy.site` (in releases prior to 3.5 this target was called `create.site`)

   f. Click *OK*

9. Finally, we must specify the product of this action and give it an alias that it passes on to ANT.

   a. Click on *Products* in the middle pane.

   b. Enter the *Product Alias* `action.output`

   c. Enter the *Product Base Path* `site/`

      This concludes the CSPEC editing.

      Save it using CMD-S or *File* → *Save*.

Building the site

Right-click on your project, select *Buckminster* → *Invoke action* → *build.site*. The output will end up in `${user.temp}/buckminster` by default. You can change this by setting the property `buckminster.output.root` in a property file that you reference when you execute the action. You can also specify properties using *Eclipse* → *Preferences* → *Run/Debug* → *String substitution*.

15 *Hello XML World*

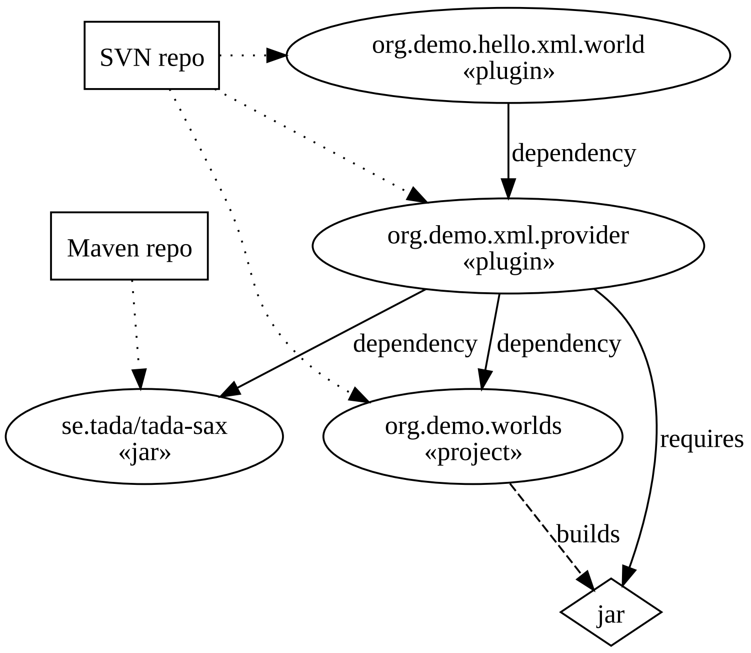
In this examples, we show how Buckminster is used to assemble two Eclipse plugin projects, a regular Eclipse project, and a jar downloaded from a maven repository. These are all handled as components, and one of them produces a new jar file that is required by one of the other components.

This example is called “*Hello XML World*” because the code consists of a ‘world provider’ that reads a configuration of ‘worlds’ in XML and code that uses this to say ‘hello’ to one of the worlds. The example demonstrates:

- Getting components from different repositories
- Using a downloaded jar in a component
- Using Buckminster `prebind` to perform actions before projects are bound to the workspace.
- Integration with ANT, to execute the prebin action, and to build the regular project.
- Using a `buckminster.cspec` to describe a component that is not automatically handled.
- Using a CSPEX extension to add a prebind action to an automatically generated component.

All of the source code for this examples is available in the Buckminster SVN repository and can be viewed with a browser at this location [http://dev.eclipse.org/viewsvn/index.cgi/trunk/org.eclipse.buckminster/demo/?root=Tools_BUCKMINSTER].

Here is a diagram of this example:


SVN repo
org.demo.hello.xml.world «plugin»
dependency
Maven repo
org.demo.xml.provider «plugin»
dependency
dependency
se.tada/tada-sax «jar»
org.demo.worlds «project»
requires
builds
jar

Prerequisites. If you want to run this example in your IDE, you must have Buckminster installed as well as support for Java, and PDE. You also need the Buckminster features for Maven and SVN installed, and for SVN, you also need an actual SVN client installed. Please refer to Appendix A, *Installation* for how to install these.

Without Buckminster

Without using Buckminster, the steps to build a functioning project requires the following:

- The three projects are checked out from Buckminster's SVN repository. This can be done manually, or using a team project set file.
- The three projects does not build because the jar file with the SAX parser needs to be downloaded from a Maven repository. This is easily downloaded, but as the name of the file contains a version number, it is best if it is renamed so that the components using it does not have to have their manifests changed when the version is changed.
- The project still does not build, as one of the projects require a jar file that is built by one of the other projects. This can be done with the Eclipse jar packager, (and a description is available in the project as illustration of the manual alternative) — it however stores absolute paths, and refers to a specific location on the initial developer's machine. All other users must edit this file before they can build.

With Buckminster in use

To set these projects up with Buckminster, the following is needed:

- A RMAP is needed so the resources can be found. The RMAP needs to have entries for the three projects (they are in Buckminster's SVN repo in our example), and the component from a maven repository.
- We need a CQUERY to be able to materialize the entire set up.
- We need some special actions to handle the building and inclusion of the SAX parser jar, and the jar file produced by one of the projects.

Once we have this set up, the configuration is very easy to materialize and build for anyone that wants to work on the software as well as being buildable in headless fashion on a build server.

The RMAP

The RMAP for this demo looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<rmap
    xmlns="http://www.eclipse.org/buckminster/RMap-1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mp="http://www.eclipse.org/buckminster/MavenProvider-1.0"
    xmlns:bc="http://www.eclipse.org/buckminster/Common-1.0">

<searchPath name="default">
    <provider readerType="svn" ❶
        componentTypes="eclipse.feature,osgi.bundle,buckminster" ❷ source="true" ❸>
        <uri format="http://dev.eclipse.org/svnroot/tools/org.eclipse.buckminster↵
           ¬/trunk/org.eclipse.buckminster/demo/{0}"> ❹
            <bc:propertyRef key="buckminster.component" /> ❺
        </uri>
    </provider>
</searchPath>

<locator searchPathRef="default" pattern="^org\.demo\..*" /> ❻
<redirect href="http://www.eclipse.org/buckminster/samples/rmaps/dogfood.rmap"
    pattern=".*" ❼
    />
</rmap>
```

This RMAP looks up all components that begins with ‘org.demo’ in the Buckminster SVN repository, and then redirects to the Buckminster standard RMAP used when building Buckminster (see the section called “The ‘dogfood’ RMAP”). The demo RMAP shown above is available at http://www.eclipse.org/buckminster/samples/rmaps/demo.rmap so you don't have to type it in to run this example.

Here are some details regarding this RMAP:

❶ We want to get projects from Buckminster’s SVN, so we use the svn reader.
❷ We want features, bundles, and components with buckminster metadata.
❸ We want source. (We are not asking for mutable source, as you would need to be a committer on the buckminster project for that to be a meaningful request).
❹ The mapping is trivial, the URI to the Buckminster SVN repository simply needs the name of the component at the end — i.e. the parameter `{0}` which it gets from the nested element...
❺ The `buckminster.component` property is always a reference to the component being looked up. This is the value that goes into the uri in the format string at `{0}`.
❻ This locator uses a pattern that will direct anything that starts with ‘org.demo’ to the search path for the demo components.
❼ If not found in the first locator, we redirect the search to the dogfood RMAP (see the section called “The ‘dogfood’ RMAP”), which (among other things) has entries for standard Eclipse content.

The CQUERY

Now that we have a RMAP that will find the components we are interested in, we can issue queries that materializes them.

As the query is very simple, you can just open the Buckminster query editor and simply enter three things — the name of the component; `org.demo.hello.xml.world`, that is an `osgi.bundle`, and the URL to the RMAP (set up in the previous section), or you can copy/paste in the following into a file with the suffix `.cquery` and then open that file in Eclipse:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<cq:componentQuery
    xmlns:cq="http://www.eclipse.org/buckminster/CQuery-1.0"
    resourceMap="http://www.eclipse.org/buckminster/samples/rmaps/demo.rmap" ❶
    >
    <cq:rootRequest
        name="org.demo.hello.xml.world" ❷
        componentType="osgi.bundle" ❸
        />
</cq:componentQuery>
```

❶ The RMAP URL from the previous section.
❷ The name of the top component
❸ It is an `osgi.bundle`, which we state in case the name is not unique, and possibly speed up the lookup as only osgi.bundles needs to be considered.

Now that we have the query in place, we can materialize it to our workspace (see the section called “Materialization wizard” for more information). The result is that we get three projects in our workspace:

- `org.demo.hello.xml.world`
- `org.demo.xml.provider`
- `org.demo.worlds`

You will also notice that the projects got built, and that there were no errors. If you just checked the projects out manually (without using Buckminster), and tried to build them, you would get errors, as things would be missing. The extra work performed by Buckminster originates from some extra instructions put into the components’ meta data.

Running the example

Since the projects were built without errors after running the query, the sample can also be executed. Simply run the `org.demo.hello.xml.world` as an Eclipse Application. When the self hosted IDE has launched, you will find the view under *Window* → *Show View* → *Other...* → *Sample Category* → *Sample*. When you open that view, it will say “*Hello XML Earth World*”.

How the code is structured

The `org.demo.hello.xml.world` is a standard Eclipse plugin with an Eclipse view that displays text. The viewer uses the classes `WorldMap`, and `TheReader` which are made available by the `org.demo.xml.provider`, so a dependency is declared on this bundle. Apart from this, the `org.demo.hello.xml.world` is quite uninteresting.

The `org.demo.xml.provider` pulls things together. It is a plugin, and it provides `TheReader` class which in turns makes use of the tada XML SAX parser jar downloaded from the Maven repository at Ibiblio. It also provides the `WorldMap` class which is in a jar built by the non-plugin project `org.demo.worlds`.

In the following two sections we explain how the `org.demo.worlds` is built, and how the resulting jar, together with the tada parser jar are handled.

org.demo.worlds

The `org.demo.worlds` project is an Eclipse ‘plain java’ project — so the system does not offer any help with putting everything together, like it does when using OSGi. To manage this project as a component we added a `buckminster.cspec` file where we have declared the required actions. The component does not have any dependencies — but if that were the case, we would have declared those as well. You can explore this CSPEC in the CSPEC editor if you like, but here we show the XML for the CSPEC, to explain how it is constructed.

```
<?xml version="1.0" encoding="UTF-8"?>
<cs:cspec xmlns:cs="http://www.eclipse.org/buckminster/CSpec-1.0" name="org.demo.worlds">
    <cs:artifacts>
        <cs:public name="source" path="src/"/> ❶
    </cs:artifacts>
    <cs:actions>
        <cs:public name="java.binary.archives" actor="ant"> ❷
            <cs:actorProperties>
                <cs:property key="buildFile" value="make/build.xml"/> ❸
            </cs:actorProperties>
            <cs:prerequisites alias="input">
                <cs:attribute name="eclipse.build"/> ❹
            </cs:prerequisites>
            <cs:products alias="output" base="${buckminster.home}/bin/jars/"> ❺
                <cs:path path="worlds.jar"/>
            </cs:products>
        </cs:public>
        <cs:private name="eclipse.build" actor="eclipse.build"> ❻
            <cs:prerequisites>
                <cs:attribute name="source"/> ❼
            </cs:prerequisites>
            <cs:products base="${buckminster.home}/bin/classes/"> ❽
                <cs:path path="."/>
            </cs:products>
        </cs:private>
    </cs:actions>
    <cs:groups>
        <cs:public name="java.binaries"> ❾
            <cs:attribute name="eclipse.build"/>
        </cs:public>
    </cs:groups>
</cs:cspec>
```

❶ We state that this component has an attribute called ‘`source`’ which is a static reference to artifacts with a relative path to where the source is inside this component. We need this reference later wen we are going to build the source.

❷ We state that the component has an attribute called ‘`java.binary.archives`’ (we picked this name as it is also used in automatically generated components for the same purpose) and we want this to attribute to have the value of all the jars in this component. As the jar file (there is only one in this case) is produced by an ANT script, we declare that we want to use an ‘`ant`’ actor.

❸ Here we provide `actorProperties` to control the `ant` actor. We set the `buildFile` to ‘`make/build.xml`’. This ANT script will produce the jar, and it needs to know where the compiled classes are, and where the resulting jar should be produced. We will (as you will see below) declare two aliases called ‘`input`’ and ‘`output`’ to provide this. The script itself is very simple — it looks like this:

```
<?xml version="1.0"?>
<project name="org.demo.worlds"> 31
    <target name="java.binary.archives"> 32
        <dirname property="output.dir" file="${sp:output}"/> 33
        <buckminster.valuefileset id="input.fileset" value="${fs:input}"/> 34
        <mkdir dir="${output.dir}"/> 35
        <jar destfile="${sp:output}"> 36
            <fileset refid="input.fileset"/> 37
        </jar>
    </target>
</project>
```

31 A project root element is required. The name of the projects is stated for historical reasons — in Eclipse 3.4 it was considered an error if not stated. Has no significance in 3.5.

32 The ant target (i.e. the ant action) is named the same as the action in the component — this is how they are linked together. (It is also possible to link the target using a `targets` actor property, but we use the default here).

33 We declare an ant `dirname` variable called ‘`output.dir`’ to have the value of the property ‘`output`’ passed from Buckminster. The ‘`sp:`’ prefix means we want a single path (the ‘`output`’ is declared to have a single path).
34 We declare a `buckminster.valuefileset` variable called ‘`input.fileset`’ (which adapts a Buckminster *path group* to an ant fileset), and we set it to the value of the property ‘`input`’ (all the compiled classes).
35 We make sure the directory where we are going to place the resulting jar file exists by creating it.
36 We execute the ANT `jar` action telling it to produce the jar file named in the buckminster property ‘`output`’.
37 We provide the parameter to the `jar` ANT task that tells it what to include in the jar — this is done with a reference to the earlier created ‘`input.fileset`’ (the compiled classes).

For details see the ‘Buckminster ANT tasks’ reference guide. Now back to the CSPEC...

4 We declare the prerequisites (the input) to the `java.binary.archives` action to be aliased ‘`input`’ and that this input is the value of the `eclipse.build` attribute (which we are declaring further down to compile the classes and return them). As you saw in the ANT script, we used the alias ‘`input`’ to access the compiled classes.
5 Here we declare the `products` (the result/output of the action — which is also the value of the the `java.binary.archives` attribute). We use an alias ‘`output`’ so the ANT script can access the value and actually produce the result where we want it). We only have one product, so it is declared directly in the products element. Its base is relative to the `buckminster.home` property which points to the root of the component — so we get the jar file under ‘`bin/jars`’ inside the project.
6 Here we declare an action that we call ‘`eclipse.build`’ — it uses the `eclipse.build` actor, which is the same as running a build of the project inside the IDE. This will compile the source into class files.
7 We use the ‘`source`’ attribute as a prerequisite (input) to the ‘`eclipse.build`’ action, so it knows what to build.
8 We declare the `products` (output/result) of the action to be located relative to the component’s root (‘`buckminster.home`’) under ‘`bin/classes`’.
9 To be complete, and compatible with automatically generated components, we also declare an attribute called ‘`java.binaries`’ which includes all binary content produced by the component — in this case, the classes produced by the `eclipse.build` action, and nothing more. (This attribute is not actually further used in our example).

For details regarding the CSPEC syntax please refer to the appropriate section in Chapter 6, *Components*.

Summary. The net result of the `buckminster.cspec`, and `make/build.xml` ant script is that we now can get the resulting `worlds.jar` as an attribute called `java.binary.archives` in the component `org.demo.worlds` — the fact that this triggers compilation of the source and production of the jar file is not visible to the user of the component. This is exactly what we wanted. Later we may restructure how this component is built and we can now do so with confidence.

org.demo.xml.provider

The `org.demo.xml.provider` project is a plugin. We chose to make it a plugin as it is then easy to to use from the `org.demo.hello.xml.world` component (and other future enhancements that wants access to worlds). We decided to use XML as the lingua franca between worlds (the messages sent to worlds are in XML), so we need an XML parser. We decided on using a SAX parser available in the Maven repository at Ibiblio. This, to show how to use the Maven integration, and how to make use of a downloaded jar file inside a plugin. We also decided to let the `org.demo.xml.provider` package make the `worlds.jar` file (we built ourselves) available to show how such an integration is made. This also gives the opportunity to demonstrate the use of a CSPEX — extension.

As the `org.demo.xml.provider` is a plugin, it gets an automatically generated CSPEC, but this CSPEC does not contain any of the extras we want (i.e. the tada SAX parser, and the `worlds.jar`). To integrate those, we use a CSPEX. It is quite straight forward as we only need to add things, i.e. there is no need to alter any of the automatically generated values. This is what it looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<cs:cspecExtension
    xmlns:com="http://www.eclipse.org/buckminster/Common-1.0"
    xmlns:cs="http://www.eclipse.org/buckminster/CSpec-1.0">
    <cs:dependencies> ❶
        <cs:dependency name="org.demo.worlds"/>
        <cs:dependency name="se.tada/tada-sax"
                       versionDesignator="1.0.0"
                       versionType="OSGi"/>
    </cs:dependencies>
    <cs:actions>
        <cs:public name="buckminster.prebind" actor="ant"> ❷
            <cs:actorProperties>
                <cs:property key="buildFile" value="make/prebind.xml" /> ❸
            </cs:actorProperties>
            <cs:prerequisites>
                <cs:attribute
                    component="se.tada/tada-sax" alias="tada-sax.jar" ❹
                    name="java.binary.archives"/>
                <cs:attribute
                    component="org.demo.worlds" ❺
                    alias="worlds.jar"
                    name="java.binary.archives"/>
            </cs:prerequisites>
            <cs:products alias="output" base="${buckminster.home}">
                <cs:path path="jars/" /> ❻
            </cs:products>
        </cs:public>
    </cs:actions>
</cs:cspecExtension>
```

❶ We add two dependencies to the component — the `org.demo.worlds` (because we are going to pick the '`worlds.jar`' from it), and `se.tada/tada-sax` (as we want the sax parser jar from it). We can not add these dependencies in the plugin's manifest as it only handles dependencies on other plugins. (We could have done this a different way though — the packages could have been made required, and we could have included the two jars via an additional component, but this extra component would look very similar to what we present here).

❷ We declare an action called `buckminster.prebind` to use an `ant` actor. The `buckminster.prebind` is called automatically by Buckminster (if it exists) as part of binding this project to the workspace. We want this action to run before the content becomes visible to the rest of Eclipse as we need the two extra jars in the correct place before automatic building kicks in (or we would se errors if the extra jar files are not there).

❸ We configure the `ant` actor to use the ANT script '`make/prebind.xml`'

❹ We declare the prerequisites (input) to include the `se.tada/data-sax` component's attribute called `java.binary.archives` (i.e. its jar) — we declare this with an alias of '`tada-sax.jar`' so we can use this alias in the ANT script.

❺ We declare the prerequisites (input) to also include the `worlds.jar` from the `org.demo.world` component's attribute `java.binary.archives`, and we give it a suitable alias ('`worlds.jar`') to be used within the ANT script.

❻ We declare the `products` (the result/output) of the action to be under the component's `jars` directory, and we declare this with the alias '`output`' so the ANT script can use this to determine where the result should go.

The ANT script looks like this:

```
<?xml version="1.0"?>
<project name="project">
    <target name="buckminster.prebind">
        <mkdir dir="${sp:output}"/>
        <copy file="${sp:tada-sax.jar}" tofile="${sp:output}/tada-sax.jar"/>
        <copy file="${sp:worlds.jar}" tofile="${sp:output}/worlds.jar"/>
    </target>
</project>
```

This simply copies the two jar files using the aliases for the jar files and output locations.

Summary. We extended the automatically generated CSPEC with additional dependencies to get the two required jars. We also added actions to pre-bind these jars into the workspace via copy operations performed by an ANT script.

16 Building RCP Products

In this chapter we show an example how Buckminster is used to build a complete RCP application. This example demonstrates:

- How to build a RCP app to a p2 update site (with very little authoring required).
- How to use a CSPEX to extend the build to also run the director to install the generated application and turn it into a zip file.
- How to organize features and .product in a good way to make building and maintenance easy.
- Use of a “releng” project as a sharing mechanism for RMAP and CQUERY artifacts.
- Use of rules in the RMAP to provide routing to different types of builds (nightly, milestone, release, ec.).
- How to perform a platform agnostic build — the resulting repository supports all platforms.

Prerequisites. To run this examples, you must have Buckminster installed with support for JDT and PDE. Since the build is platform agnostic, you must have the *Eclipse Delta Pack* installed in your IDE. You also need support for SVN, and a SVN client installed to get the source code from the Buckminster SVN repository.

Conventions. In this example we have abbreviated the first part of project names — the abbreviated ‘ *`o.e.b`* ’ stands for ‘`org.eclipse.buckminster`’ and ‘ *`o.e.b.t`* ’ for ‘*`o.e.b.`*`tutorial`’. Also abbreviated are ‘`o.e`’ and '`o.e.e`’ for `org.eclipse`, and `org.eclipse.equinox`.

Getting the code

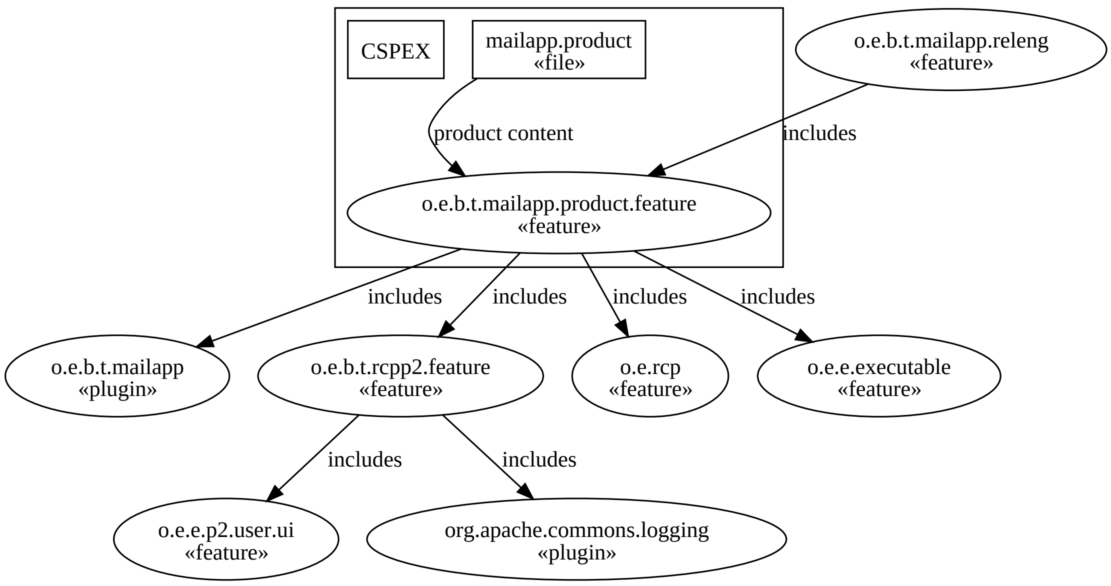
In this example we have used a “release engineering project” to store the Buckminster artifacts used to set up and build the project. Such a project is useful as it serves as a starting point for developers, but can also be used as a starting point for fully automatic builds. In this example we also used the release engineering project to define the actual site we are building, but in a more complex project we may be building different update sites, and would then have separated that out.

To get the code, checkout the project *`o.e.b.t.`*`mailapp.releng` into your workspace from `http://dev.eclipse.org/svnroot/tools/org.eclipse.buckminster.`

Inside the project, there are two files of primary intrest, the `developer.cquery`, and the `eclipse.rmap`. Double click on the `developer.cquery` to open up the CQUERY editor. Note that it is a query for *`o.e.b.t.`*`mailapp.product.feature` which is the feature describing the product we are building. Also, note that the query references the `eclipse.rmap` found in the releng project we just checked out. Simply click on “*Resolve and Materialize*” to get all of the required projects into your workspace.

Structure

Here is a diagram illustrating how the parts fit together:


CSPEX
mailapp.product «file»
o.e.b.t.mailapp.releng «feature»
product content
includes
o.e.b.t.mailapp.product.feature «feature»
includes
includes
includes
includes
o.e.b.t.mailapp «plugin»
o.e.b.t.rcpp2.feature «feature»
o.e.rcp «feature»
o.e.e.executable «feature»
includes
includes
o.e.e.p2.user.ui «feature»
org.apache.commons.logging «plugin»

Things are kicked off from `o.e.b.t.mailapp.releng` — which you already used to get the query that materializes the rest. Here is a brief description of the parts and their role in the overall structure.

`o.e.b.t.mailapp.releng`
: This feature is the “root component” which is used it to get the rest, and it serves as a central location for some files we need when building the rest. The component is also used as the definition of the content of the p2 repository we are building. This is done by the inclusion of the `o.e.b.t.mailapp.product.feature` in the releng component's `feature.xml`. Since Buckminster’s action that builds a p2 repository from a feature does not include the feature itself in the resultingrepository, this gets us exactly what we want as we have no interest in publishing the releng component itself, only what it refers to. If we wanted to we could also have put categorization into the releng component, but we felt it was more natural to do so in the product defining feature.

`o.e.b.t.mailapp.product.feature`
: This feature is used to define what is included in the product. The component contains a `mailapp.product` file that defines the product properties (i.e. branding, icons, splashscreen, etc.), but we let the feature define *what* to include (other features and bundles) in the product instead of the `mailapp.product` file — hence the reference from `mailapp.product` to the feature.

  The feature includes the plugins and features that should be included in the product.

`o.e.b.t.mailapp`
: This is the plugin that contain the actual product code. It was generated by the standard “create product wizard” using the mailapp template for a RCP application. The only difference from the standard setup is that we moved the generated `.product` file to a separate feature which we use to keep track of the product’s content (as opposed to keeping this in the `.product` file). Everything else is default.

`o.e.b.t.rcpp2.feature`
: This feature defines what is needed to make a RCP application include a p2 agent — thus making it self maintained in terms of installing new features into the product, and to handle updates. The integration is the simplest possible — there are many options available regarding how p2 can be used in a RCP application but this is beyond the scope of this book.

`o.e.e.p2.user.ui`
: This p2 feature includes all the things required to use p2 in a RCP application (except the `org.apache.commons.logging` plugin which we included in the `o.e.b.t.rcpp2.feature`).

`o.e.rcp`
: Required for RCP applications.

`o.e.e.executable`
: Required to make the RCP application launchable on multiple platforms.

The RMAP

The RMAP in this example is interesting because it shows how to define rules for picking different components from different repositories for nightly, milestone, and release builds. Although somewhat lengthy, it is quite simple as the different entries follow a pattern.

```
<property key="useBuild" value="RBUILD"/> ❶

<searchPath name="org.eclipse.buckminster"> ❷
    <provider
        readerType="svn"
        componentTypes="osgi.bundle,eclipse.feature,buckminster"
        mutable="true" source="true">
        <uri format="http://dev.eclipse.org/svnroot/tools/org.eclipse.buckminster↵
             ¬/trunk/{0}?moduleAfterTag&amp;moduleAfterBranch">
            <bc:propertyRef key="buckminster.component" />
        </uri>
    </provider>
</searchPath>

<searchPath name="org.eclipse.platform.NBUILD"> ❸
    <provider
        readerType="eclipse.import"
        componentTypes="osgi.bundle,eclipse.feature"
        mutable="false" source="false">
        <uri format="http://download.eclipse.org/eclipse/updates↵
            ¬/3.5-N-builds?importType=binary"/>
    </provider>
</searchPath>

<searchPath name="org.eclipse.platform.IBUILD"> ❹
    <provider
        readerType="eclipse.import"
        componentTypes="osgi.bundle,eclipse.feature"
        mutable="false" source="false">
        <uri format="http://download.eclipse.org/eclipse/updates↵
             ¬/3.5-I-builds?importType=binary"/>
    </provider>
</searchPath>

<searchPath name="org.eclipse.platform.MBUILD"> ❺
    <provider readerType="eclipse.import"
        componentTypes="osgi.bundle,eclipse.feature"
        mutable="false" source="false">
        <uri format="http://download.eclipse.org/eclipse/updates↵
             ¬/3.5milestones?importType=binary"/>
    </provider>
```

```
</searchPath>

<searchPath name="org.eclipse.platform.RBUILD"> ❻
    <provider
        readerType="eclipse.import"
        componentTypes="osgi.bundle,eclipse.feature"
        mutable="false" source="false">
        <uri format="http://download.eclipse.org/eclipse/updates↵
             ¬/3.5?importType=binary"/>
    </provider>
</searchPath>

<searchPath name="org.eclipse.galileo"> ❼
    <provider readerType="eclipse.import"
        componentTypes="osgi.bundle,eclipse.feature"
        mutable="false" source="false">
        <uri format="http://download.eclipse.org/releases↵
             ¬/galileo?importType=binary"/>
    </provider>
</searchPath>

<searchPath name="orbit"> ❽
    <provider
        readerType="eclipse.import"
        componentTypes="osgi.bundle"
        mutable="false" source="false">
        <uri format="http://download.eclipse.org/tools/orbit/downloads/drops↵           ¬/R2009052
    </provider>
</searchPath>

<locator searchPathRef="org.eclipse.buckminster" ❾
         pattern="^org\.eclipse\.buckminster(\..+)?"/>
<locator searchPathRef="org.eclipse.platform.${useBuild}" failOnError="false" /> ❿
<locator searchPathRef="org.eclipse.galileo" failOnError="false" /> ⓫
<locator searchPathRef="orbit" /> ⓬
```

❶ We define a default property called `useBuild` and set it to `RBUILD`. This is the value we use for a release-build. (If this property is not set in a CQUERY or CQUERY advisor node or as a system property, then we will get a release-build).

❷ A search path entry for getting Buckmisnter related material from the Buckminster SVN repository. In this example, we are getting all the tutorial components from this location.

❸ Here we set up a path called `org.eclipse.platform.NBUILD`, where `NBUILD` stands for *nightly build*. As you can see from the `uri`, this search path uses the 3.5 nightly repoitory.

❹ `org.eclipse.platform.IBUILD` goes to 3.5 integration build repository.

❺ `org.eclipse.platform.MBUILD` goes to 3.5 milestone build repository

❻ `org.eclipse.platform.RBUILD` goes to 3.5 release build repository.

❼ `org.eclipse.galileo` goes to the named final 3.5 Galileo release repository

❽ `orbit` search path picks things from the Eclipse orbit repository. (Note that the Orbit RMAP entry needs to be updated from time to time as the repository gets updated and given a new timestamp).

❾ The first locator matches anything starting with `org.eclipse.buckminster`

❿ This location is what makes everything work. It uses the property `useBuild` as part of the search path name when routingthe search, so depending on its value (one of `NBUILD`, `IBUILD`, `MBUILD`, or `RBUILD`) we get a different search path. We also use `failOnError=false` to make the search continue if not found in the repository.

⓫ If we did not find the component in the repository designated by `useBuild`, we continue the search in the galileo repository. Again, we use `failOnError=false` to continue the search if the component is not found in the galileo repository.

⓬ Finally, we try to find the component in the Eclipse orbit repository.

The functionality in this RMAP was very important to us as we developed this example for EclipseCon 09. Eclipse 3.5 was not yet released, and we needed to be able to use various components from various repositories to make things work. Now, with 3.5 being released there is no actual need to pick anything but the released, but we thought that the RMAP also is a good example on its own.

Using ‘useBuild’

If you want to pick certain components in the resolution from a particular repository you can do so by setting the `useBuild` property (to one of the values shown in the RMAP section above). To set the value for some components, you add an advisor node in your CQUERY with a pattern that matches the components, and then you add the `useBuild` property with the value you want.

In the example project, the `developer.cquery` has a sample entry for a fictous set of components matched with the pattern `org.eclipse.buddyproj.*` — the advisor node for this pattern sets `useBuild` to NBUILD to get nightly builds. (Nothing will actually happen as there are no such components, and they are never requested — the entry is only there as an illustration).

Building the update site

Building the update site for the product is no different than building any p2 update site with Buckminster — as shown in Chapter 13, *Building a p2 Update Site*. We simply need to invoke the action `site.p2` on the `o.e.b.t.mailapp.releng` feature using the properties file `buckminster.properties` (in the releng component) to control the build (where to place output, if packing and signing should take place etc.).

The difference from the example in ‘Building a p2 Updte Site’ are minor:

- We use a different set of categories (obviously) — the categories are defined in `o.e.b.t.mailapp.product.feaure`.
- The output is found under `${user.home}/bmtutorial`.

After you built the p2 repository (invoked site.p2 on the releng component), you will findthe output at:

```
${user.home}/bmturorial/↵
    ¬org.eclipse.buckminster.tutorial.mailapp.product.feature_1.0.0-eclipse.feature↵
    ¬/site.p2
```

Installing the product

As you may remember from the introcution part of this book, there are several ways you can install a product. In this example we will show you two ways; using the p2 installer to install from the p2 repository, and how to create a ready-to-run zip file.

Installation using the p2 installer

Although the p2 installer was created as an example how to install the Eclipse SDK itself, it is a useful utility for installing other small applications where you are now willing to invest the time and effort in creating a fancier installer. Since it was created to install the Eclips SDK, we do need to make some small modifications to the installer’s configuration before it can install our mailapp.

If you want to run this part yourself, you must start by downloading the p2 installer. You will find the installer by:

- Go to the equinox download page [http://download.eclipse.org/equinox/], and select the release you want (if you have not other requirements, pick the 3.5 release).
- On the page that appears, scroll down to the section called ‘provisioning’ and select a p2 installer that is suitable for your platform.
- Unzip the installer to a location of your choice — we will refer to this location as *the p2 installer location* below.

Now that you have the installer, you can use it to install the Eclipse SDK itself from the Eclipse 3.5 release repository. But we are going to modify it so it installs our mailapp instead. To do this you need to do the following:

- We need to define a set of properties that refers to our mailapp
- We need to tell the p2 installer to use these properties instead of the default (that came with the download).

Installer properties

Create a file called `mailapp_installer.properties` and enter the following:

```
eclipse.p2.metadata=«repoLocation»
eclipse.p2.artifacts=«repoLocation»
eclipse.p2.flavor=tooling
eclipse.p2.profileName=MailappProduct
eclipse.p2.launcherName=eclipse
eclipse.p2.rootId=org.eclipse.buckminster.tutorial.mailapp.product
eclipse.p2.autoStart=false
```

You should replace «`repoLocation`» with the actual location of where the p2 repository you build is. If your home directory is `/Users/mary`, then use:

```
file:///Users/mary/bmturorial/↵
    ¬org.eclipse.buckminster.tutorial.mailapp.product.feature_1.0.0-eclipse.feature↵
    ¬/site.p2/
```

As you can probably guess, what we are doing here is simply telling the p2 installer to install the installable unit (IU) called `org.eclipse.buckminster.tutorial.mailapp.product`, and to get both meta data and artifacts from the p2 repository we just built. The profileName is the name of the configuration, you may need it later to be able to install into the same configuration again — but we are not using it further in this example. We also set the autoStart to false (as there have been issues with the p2 installers ability to actuall start under some circumstances, but you can alter this to true, as it may work — the idea is to be able to launch the application after it has been installed).

We are now done with the properties file.

Using the properties

To use the properties, we must alter how the p2 installer is launched as an additional command line parameter is required. We do this by editing the `p2installer.ini` file. The location of this file is platform dependent. If you are on a Mac, you need to use the Finder command *Show Package Content* on the `p2Installer.app`, and then navigate to `Content/MacOS`. On other platforms, the `p2installer.ini` should be in the p2 installer location directly.

You need to modify the `p2installer.ini` to contain the following setting:

```
-vmargs
-Dorg.eclipse.equinox.p2.installDescription=«properties location uri»
```

The «*`properties location uri`*» is naturally an URI refering to the `mailapp_installer.properties` we created earlier. Depending on your platform, you may have to use a variation on the `file://` URI, e.g. if you placed the properties file in your home directory `/Users/mary`, you may need to use `file://localhost/Users/mary`... instead of just `file:///Users/mary`... You will know if you got it right when you launch the installer.

Running the installer

To run the customized installer, simply invoke it. You are prompted for the location where you want to intall the application. You are also prompted if you want to make a stand alone or a shared installation.

Pick “stand-alone” as the sharing will setup sharing between everything installed with the p2 installer, and you probably do not want this when running this example).

Once you have installed, you should now have the invokable mailapp in the location you specified.

Creating an installable zip

The standard way of creaing an installable zip file is to run the p2 director app to do an installation, and then zip up the result. You can do so with the embedded director app available in every Eclipse SDK, or you can use the stand alone directory available from Buckminster (as described in the the section called “Installing the Headless Product”).

It is also possible to do the same using Buckminster to orchestrate the actions. We have included a CSPEX in the `o.e.b.t.mailapp.product.feature` that is capable of doing both the install, and creating the final zip. The CSPEX adds two actions; `create.product`, and `create.product.zip`.

Note

To use `create.product`, and `create.product.zip`, you must supply a set of properties for the platform you want to create the install for — i.e. the `target.*` properties. The properties we used for the site.p2 build itself can not be used as they specify all values as ‘*’ (any). This can not be used for a install — the install is always for a particular platform.

You must also first run the `site.p2` action to create the repository or the repository will only contain content for the platform you are running on and it will not be possible to generate zips for other platforms.

To create the requird property file, simply copy the buckmisnter.properites file in the releng component, and modify the three last lines by either removing them (which gives you an install for what you are currently running), or set them explicetly.

The CSPEX

The CSPEX in `o.e.b.t.mailapp.product` feature adds two actions `create.product`, and `create.product.zip`. It is included to enable creating a ready to run product in zip form. It does this by invoking ANT scripts. One of the tasks — to create the zip file is already available in Buckminster so the action an simply refer to this with a suitable set of parameters, but running the director to perform the installation is not available as a standard task so this is supplied in the `build/product.ant` file.

There is nothing special in how these ant tasks are invoked from Buckminster — look at the CSPEX, and consult the information in the Chapter 6, *Components* if there is something you do not understand. Then look at the build/product.ant file and see that there is a create.product task there that gets invoked from the CSPEX action with the same name. The rest of the product.ant file is basically a very long list of parameters to the director app.

17 *POJO Projects*

In this chapter we show examples how Buckminster can be used with Plain Old Java Objects (POJO) project — i.e. projects that are in Java, but not in the shape of bundles, plugins, features, fragments, or Eclipse products.

18 *Non Java Projects*

In this chapter we show examples how Buckminster can be used with projects that are written in other languages than Java.

19 *RMAP Examples*

This chapter contains RMAP examples. You find all Buckminster examples RMAPs at http://www.eclipse.org/buckminster/samples/rmaps [http://www.eclipse.org/buckminster/samples/rmaps/].

The ‘dogfood’ RMAP

The so called ‘dogfood’ RMAP is the resource map that is used when building Buckminster itself. It is available at http://www.eclipse.org/buckminster/samples/rmaps/dogfood.rmap.

Warning

Since this file is an integral part of the Buckminster release engineering, it changes from time to time without warning. It is kept up to date for Buckminster, demo and samples, but changes being made may not suit your needs.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- See copyright in original file -->

<rmap
  xmlns="http://www.eclipse.org/buckminster/RMap-1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mp="http://www.eclipse.org/buckminster/MavenProvider-1.0"
  xmlns:pmp="http://www.eclipse.org/buckminster/PDEMapProvider-1.0"
  xmlns:bc="http://www.eclipse.org/buckminster/Common-1.0">

<searchPath name="org.eclipse.buckminster"> ❶
  <provider readerType="svn"
    componentTypes="osgi.bundle,eclipse.feature,buckminster"
    mutable="true"
    source="true">
   <uri format="http://dev.eclipse.org/svnroot/tools/org.eclipse.buckminster/↵
       ¬trunk/{0}?moduleAfterTag&moduleAfterBranch">
      <bc:propertyRef key="buckminster.component" />
   </uri>
  </provider>
 </searchPath>

<searchPath name="org.eclipse.ecf"> ❷
  <provider readerType="eclipse.import"
     componentTypes="osgi.bundle,eclipse.feature"
     mutable="false" source="false">
   <uri format="http://download.eclipse.org/rt/ecf/2.0↵
       ¬/updateSite?importType=binary"/>
  </provider>
  <provider xsi:type="pmp:PDEMapProvider" readerType="cvs"
   componentTypes="osgi.bundle,eclipse.feature" mutable="false" source="true">
   <uri format=":pserver:anonymous@dev.eclipse.org:/cvsroot/rt,org.eclipse.ecf↵
```

```
        ¬/releng/org.eclipse.ecf.releng.maps"/>
    </provider>
  </searchPath>

   <searchPath name="org.eclipse.spaces"> ❸
    <provider readerType="svn" componentTypes="osgi.bundle,eclipse.feature,buckminster"
  mutable="true"
     source="true">
     <uri
      format="http://dev.eclipse.org/svnroot/technology/org.eclipse.spaces↵
        ¬/trunk/{0}?moduleAfterTag&moduleAfterBranch">
      <bc:propertyRef key="buckminster.component" />
     </uri>
    </provider>
   </searchPath>

   <searchPath name="org.eclipse.platform"> ❹
    <provider readerType="eclipse.import" componentTypes="osgi.bundle,eclipse.feature"
     mutable="false" source="false">
     <uri format="http://download.eclipse.org/eclipse/updates/3.5?importType=binary"/>
    </provider>
   </searchPath>

   <searchPath name="galileo"> ❺
    <provider readerType="eclipse.import" componentTypes="osgi.bundle,eclipse.feature"
      mutable="false" source="false">
     <uri format="http://download.eclipse.org/releases/galileo?importType=binary"/>
    </provider>
   </searchPath>

   <searchPath name="buckminster.test"> ❻
    <provider readerType="cvs" componentTypes="osgi.bundle,eclipse.feature,buckminster"
     mutable="true"
     source="true">
     <uri format=":pserver:anonymous@dev.eclipse.org:/cvsroot↵
        ¬/technology,org.eclipse.buckminster/test/{0}">
      <bc:replace pattern="^buckminster\.test\.(.+)" replacement="$1" quotePattern="false">
       <bc:propertyRef key="buckminster.component" />
      </bc:replace>
     </uri>
    </provider>
   </searchPath>

   <searchPath name="org.eclipse.dash"> ❼
    <provider readerType="cvs" componentTypes="osgi.bundle,eclipse.feature,buckminster"
     mutable="true"
     source="true">
     <uri format=":pserver:anonymous@dev.eclipse.org:/cvsroot/technology,org.eclipse.dash/{0}">
      <bc:replace pattern="^buckminster\.test\.(.+)" replacement="$1" quotePattern="false">
       <bc:propertyRef key="buckminster.component" />
      </bc:replace>
     </uri>
    </provider>
   </searchPath>

   <searchPath name="subclipse"> ❽
    <provider readerType="eclipse.import" componentTypes="osgi.bundle,eclipse.feature"
     mutable="false"
     source="true">
     <uri format="http://subclipse.tigris.org/update_1.6.x?importType=binary" />
    </provider>
   </searchPath>

   <searchPath name="svnkit"> ❾
    <provider readerType="eclipse.import" componentTypes="osgi.bundle,eclipse.feature"
     mutable="false"
     source="true">
     <uri format="http://eclipse.svnkit.com/1.3.x?importType=binary" />
    </provider>
   </searchPath>
```

```
 <searchPath name="polarion"> ❿
  <provider readerType="eclipse.import" componentTypes="osgi.bundle,eclipse.feature"
   mutable="false"
   source="true">
   <uri format="http://www.polarion.org/projects/subversive/download↵
       ¬/eclipse/2.0/update-site?importType=binary" />
  </provider>
 </searchPath>

 <searchPath name="org.eclipse.team.svn"> ⓫
  <provider readerType="eclipse.import"
   componentTypes="osgi.bundle,eclipse.feature"
   mutable="false"
   source="true">
   <uri format="http://download.eclipse.org/technology/subversive/0.7↵
       ¬/update-site?importType=binary" />
  </provider>
 </searchPath>

 <searchPath name="orbit"> ⓬
  <provider readerType="eclipse.import"
   componentTypes="osgi.bundle" mutable="false" source="false">
   <uri format="http://download.eclipse.org/tools/orbit/downloads↵
       ¬/drops/R20090529135407/updateSite?importType=binary"/>
  </provider>
 </searchPath>

 <searchPath name="maven"> ⓭
  <provider xsi:type="mp:MavenProvider" readerType="maven2" componentTypes="maven"
   mutable="false"
   source="false">
   <uri format="http://repo1.maven.org/maven2" />
  </provider>
 </searchPath>

 <locator searchPathRef="org.eclipse.buckminster"
   pattern="^org\.eclipse\.buckminster(\..+)?" />
 <locator searchPathRef="org.eclipse.buckminster" pattern="^org\.slf4j\.extendable$" />
 <locator searchPathRef="org.eclipse.buckminster"
   pattern="^org\.eclipse\.equinox\.p2\.director\.product$" />
 <locator searchPathRef="org.eclipse.buckminster"
   pattern="^org\.eclipse\.equinox\.p2\.director\.feature$" />
 <locator searchPathRef="org.eclipse.ecf" pattern="^org\.eclipse\.ecf(\..+)?" />
 <locator searchPathRef="org.eclipse.ecf" pattern="^org\.jivesoftware\.smack$" />
 <locator searchPathRef="org.eclipse.ecf" pattern="^org\.eclipse\.bittorrent$" />
 <locator searchPathRef="org.eclipse.spaces" pattern="^org\.eclipse\.spaces(\..+)?" />
 <locator searchPathRef="org.eclipse.dash" pattern="^org\.eclipse\.dash(\..+)?" />
 <locator searchPathRef="buckminster.test" pattern="^buckminster\.test(\..+)?" />
 <locator searchPathRef="svnkit" pattern="^org\.tmatesoft\.svnkit(\..+)?" />
 <locator searchPathRef="subclipse" pattern="^org\.tigris\.subversion(\..+)?" />
 <locator searchPathRef="polarion" pattern="^org\.polarion\.team(\..+)?" />
 <locator searchPathRef="polarion" pattern="^org\.polarion\.eclipse\.team(\..+)?" />
 <locator searchPathRef="org.eclipse.team.svn"
   pattern="^org\.eclipse\.team\.svn(\..+)?$" />
 <locator searchPathRef="orbit" failOnError="false"/>
 <locator searchPathRef="galileo" failOnError="false"/>
 <locator searchPathRef="maven"/>

</rmap>
```

❶ This entry references Buckminster's SVN.

❷ This entry gets binary ECF components from an ECF update site, and if not available there, it looks up ECF source via a releng source map found in ECF's CVS repository.

❸ This entry references the spaces project's SVN.

❹ This entry goes to the the Eclipse 3-5 stream. It is now obsolete as the content is included in the Galileo composite repository. The entry is not used anywhere.

❺ This entry looks up binary components from the Galileo release.

❻ `buckminster.test` — goes to technology CVS where the Buckminster project keeps some components that are used to test the CVS integration. Only used for testing.

❼ project dash — included in this RMAP because it contains the site-assembler used to assemble the update site for the Ganymede release (also called Ganymatic). This is the predecessor to the Galileo-builder used to assemble the Galileo update site.

❽ This entry picks Subclipse SVN integration from Tigris.

❾ This entry picks the ‘SVN Kit’ protocol for use with Subclipse from `eclipse.svnkit.com`.

❿ This entry pick Subversive SVN integration from Polarion.

⓫ This entry picks the Eclipse Team SVN support from the subversive technology project at Eclipse.

⓬ This entry picks bundles from Eclipse orbit repository.

⓭ This entry picks things from the Maven repository at Ibiblio.

Part IV. Reference

This part consists of all the gory details.

Component Types

This reference guide contains detailed information about the component types supported by Buckminster in its standard configuration. For each type all automatically generated attributes/actions are documented.

Conventions used

The following conventions are used in this reference guide:

Common attributes	Attributes common to all types are listed in this category.
Inherited attributes	Attributes common to several types have been broken out into separate reference entries.
Abstract type	A component type that can not be used on its own. Its purpose is only to describe attributes that are shared by other component types.
dottified	Refers to the transformation of a path using slash '/' as separator to a string using period/dot '.' as separator.
source only, binary only	Attributes marked with *source only* are only available when the component is generated from source. Attributes marked with (binary only) are only available when the component is generated from binary data.
«replaceable»	A replaceable part of a name is written within guillemots « »
aggregation	Denotes a physical aggregation on disk of a group (i.e. the result of copying a group to a common location).

Common Attributes

All components have the attribute `buckminster.component.self`, which refers to the entire component.

The value of the `buckminster.component.self` can be in one of two forms depending on if the component is a *directory* or a *file*:

- If the component is a *directory*, then the base of the *path group* is equal to the location of the component and the array of paths is empty.
- If the component is a *file*, then the base appoints the directory that contains this file and the path array has one path which is the file relative to that base.

All components that can be materialized into a workspace can declare an action called `buckminster.prebind`. This action will be called as part of the workspace materialization before the component is bound to the workspace. Note that this action is only invoked by the workspace materializer.

buckminster

buckminster — component type for components that have buckminster meta data.

Synopsis

Common Attributes
`buckminster.component.self`
Attributes
-

Attributes

A buckminster based component has the attributes defined in the file `buckminster.cspec` found in the component's root.

Dependencies

A buckminster based component has the dependencies defined in the file `buckminster.cspec` found in the component's root.

Description

The buckminster component type is fully covered in Chapter 6, *Components*.

eclipse.feature

eclipse.feature — component type for Eclipse features.

Synopsis

Common Attributes
`buckminster.component.self`
Inherited from PDE component type

public
 	
`buckminster.clean`
`build.properties (source only)`
`bundle.jars`
`manifest`
`product.configuration.exports`
 	
private
 	
`product.configurations`
`«product name»`
`jar.contents (source only)`
`raw.manifest`

Public Attributes

`feature.exports`
`feature.jars`
`feature.references`
`site.feature.exports`
`site.p2`
`site.p2.zip`
`site.packed`
`site.signed`
`source.bundle.jars`
`source.feature.jars`
`source.feature.references`

Private Attributes

`copy.features`
`copy.plugins`
`copy.subfeatures`
`feature.jar`
`site.repacked`
`source.feature.jar`

Attributes

Public Attributes

`feature.exports`
: A *group* consisting of the actions `copy.features` and `copy.plugins`

`feature.jars`
: A *group* consisting of the attributes `feature.jar` and `feature.references`

`feature.references`
: A *group* that aggregates the `feature.references` of all included features (but not self).

`site.feature.exports`
: A *group* consisting of the actions `copy.plugins` and `copy.subfeatures`. This is different from the `feature.exports` in that the feature itself is not included. That’s because the feature that defines the site is not included in the site.

`site.p2`
: An *action* that creates the final p2 site. It uses the internal actor `p2SiteGenerator` which in turn uses the *p2 publisher*. The input to this action comes from the attributes `manifest`, `product.configuration.exports`, `site.feature.exports`, `site.packed`, and `site.signed`. Only one of the three `site.`*«xxx»* inputs will be used. Which one depends on the settings of the properties `site.pack200` and `site.signing` such that:

  - if both are `false`, then `site.feature.exports` is used
  - if `site.pack200` is set, then `site.packed` is used regardless of setting of `site.signing`
  - if `site.signing` is set and `site.pack200` is not set then `site.signed` is used

  See Description Section below for more details how to control the `site.p2` action regarding signing, packing, and category definition.

`site.p2.zip`
: An *action* that zips the result of the `site.p2` action into a zip with a file name that indicates its version.

`site.packed`
: An *action* that runs `pack200` on all artifacts that it finds on input. The input is either `site.feature.exports` or `site.signed`. The latter is chosen if the property `site.signing` is set.

`site.signed`
: An *action* that performs *jar signing* on all jars from its input. The input is `site.feature.exports` or `site.repacked`. The latter is chosen if the property `site.pack200` is set.

`source.bundle.jars`
: A grouping of the `bundle.and.fragments.source` attribute of all bundle dependencies.

`source.feature.jars`
: A *group* that contains the `source.feature.jar` and `source.feature.references` i.e. all source for the feature, including self.

`source.feature.references`
: A *group* that aggregates the `source.feature.jars` attribute of all included features.

Private Attributes

`copy.features`
: An *aggregation* (physical, on disk) of `feature.jars` and `source.feature.jars`.

`copy.plugins`
: An *aggregation* (physical, on disk) of `bundle.jars` and `source.bundle.jars`.

`copy.subfeatures`
: An *aggregation* (physical, on disk) of `feature.jar`, `feature.references`, `source.feature.jar`, and `source.feature.references`.

`feature.jar`
: An *action* that builds a `feature.jar` based on its input `jar.contents` and `manifest`.

`site.repacked`
: An *action* that reconditions its input by running a `pack200` followed by `unpack200`. The input is `site.feature.exports`.

`source.feature.jar`
: An *action* that builds a source feature jar based on its input `jar.contents` and `source.manifest`.

Dependencies

In case the CSPEC is created from a `feature.xml`, the feature's dependencies are generated from the *included* features and plugins. The feature *requirements* (things that are not included but needed in order to install the feature) are not subject to interpretation, and no dependencies are generated for these.

When the CSPEC is created from a p2 IU, all *required capabilities* referencing capabilities in the `org.eclipse.equinox.p2.iu` or `osgi.bundle` namespaces will cause a corresponding CSPEC dependency to be generated. All other required capabilities are ignored.

Description

The `eclipse.feature` component type is automatically generated for all Eclipse features. It can generate a CSPEC from source or binary data. When generating the CSPEC, the various meta data files in the source, or available in the binary representation are used to create the attributes.

If the CSPEC is generated from a p2 IU, then the IU is the single source of information.

If not generated from a p2 IU, the content of the `feature.xml` and `build.properties` files are used. The `build.properties` is the source for what is included in the jar (both *source* and *binary*) and the *site category definitions*. The rest of the information is picked from the *feature.xml*

site.p2

The `site.p2` action builds a p2 update site, and it can be controlled using properties.

Categorization. The site.p2 action is (in addition to handling categories in the standard `category.xml` file) also capable of creating categories by interpreting entries in the feature's `build.properties` file. A category, its label, content, and description are defined by entering:

`category.id.«category-id»=«category-label»`
`category.description.«category-id»=«short description»`
`category.members.«category-id»= «feature-id» [ , «feature-id»...]`

The «*category-id*» should reflect your organization to separate it from other defined categories. The «*feature-id*» is the identity of a feature to include in the category. (A feature can be part of many categories).

Default category. It is possible to use a default category which will be applied to all features that are not part of any other specified category. The default category is defined by entering:

`category.default=«category-id»`
.

Translation. All category strings are subject to translation via the `feature.properties` (i.e. the same way other feature strings are translated). As an example:

```
// in build.properties
category.description.mycategory=%MyCategoryDesc
```

the translation is placed in `feature.properties`:

```
MyCategoryDesc=This is the core functionality of my cool feature.
```

Content from feature.xml. The content from `feature.xml` is used as follows:

Label/Name
: Defines the name of the repository.

Included Features
: The features listed in Include Features become root IUs (i.e. top level installable things).

Discover Sites
: Sites listed under Discovery Sites are added as site references in the created repository.

Mirrors Site
: Is used as the Mirrors Site of the created repository.

Note

The feature itself is not included in the site.

Properties controlling the build. There are several properties controlling the build and site generation. These are:

`buckminster.output.root`
: Absolute file system path where output should be generated.

`buckminster.temp.root`
: Absolute file system path where temporary output should be placed when building.

`eclipse.committer.name`
: The login-name of an eclipse.org committer. Is only used with the special eclipse.org signing, and is only available to eclipse committers that have the right to run signing at eclipse.

`eclipse.committer.password`
: The password for the `eclipse.committer.name`. Is only used with the special eclipse signing.

`eclipse.committer.keyfile`
: If this property is set to the full path of a private key file, the special eclipse signing will use key authentication and the `eclipse.committer.password` is not needed.

`eclipse.committer.keyfile.passphrase`
: Optional. Only needed if the keyfile (specified with `eclipse.comitter.keyfile`) was created in such a way that a passphrase is needed in order to access it.

`eclipse.staging.area`
: Required when using eclipse signing. Each project has a staging area for builds at eclipse.org. This area is used when performing the signing. As an example, the Buckmisnter project uses `/home/data/httpd/download-staging.priv/tools/buckminster`. (This property was earlier called just `staging.area`. Use of this old property is still supported, but its use is deprecated in favor of `eclipse.staging.area` as it only affects signing at eclipse.org).

`local.keystore.path`
: An absolute file system path that refers to your personal certificate. Only used when performing local signing. Example: `/home/mary/certificates/personal.certificate`

`local.keystore.alias`
: An alias used by the local keystore. Example ‘`mary`’.

`local.keystore.password`
: Password for the local keystore.

`site.pack200`
: A boolean value controlling if pack200 should be performed. If combined with signing normalization is also performed.

`site.signing`
: A boolean value controlling if signing should be performed.

`signing.type`
: The type of signing to use. Can be either ‘`eclipse.remote`’ (requires Eclipse committer credentials as well as signing privileges) or `local` (requires local certificate). See more information below.

`cbi.include.source`
: Controls generation of source features and bundles. When set to true, source bundles are generated and included in the update site.

Warning

Source features and bundles are generated and included in the update site unless you set this property to false. For open source projects this is typically what is wanted, but it may not be suitable for your project.

Remote eclipse signing. If eclipse.remote signing is used, the build will package all relevant jars in a zip file and send it to `eclipse.org` using `scp`. It will request signed by adding the transfered material to the queue for the Eclipse signer and then await the result. Once the signing is complete, it will be picked up and the build will continue. Although sometimes a bit slow (more then 20 minutes is rare), the process is fully automatic and does not require any manual intervention.

Local signing. Local signing can be used by anyone interested in signing the jars that are included in the generated site. To use this, you must create a personal certificate. This is done with the `keytool` executable. You will find it in the bin catalog of your `JDK`. It is used as follows:

```
keytool -genkeypair -keystore «path to keystore file» -alias «your alias»
```

You will be asked for some information about name, location, and password. Enter some sane values. Finally you will be asked to confirm the information and you are then asked if you want the same password as for the keystore — simply hit return, you do not want an additional password.

jar

jar — component type for a *Plain Old Java Object* (POJO) jar file.

Synopsis

Common Attributes
`buckminster.component.self`
Inherited Attributes from POJO

public
`java.binary.archives`
`java.binary.folder`
`java.binaries`

Attributes
-

Attributes

Public Attributes

`java.binary.archives`
: Is a group that contains the `buckminster.component.self` attribute.

`java.binary.folder`
: The attribute `java.binaries` will point to this instead of the `java.binary.archives` when the jar is in source form. The `java.binary.folder` will typically appoint the `bin` directory (the output of the Java compiler).

`java.binaries`
: Is a group that contains the `java.binary.archives` attribute.

Dependencies

The jar component type does not support dependencies.

Description

The jar type is used for jar based components that lack additional meta data.

Note

If you have a jar component, and want additional meta data, you should turn it into a Buckminster component by adding a `buckminster.cspec` to it. When doing this, you probably also want to add the attributes that are common to all POJO components — they are not added automatically.

maven, maven2

maven, maven2 — component types for Maven-1 and Maven-2 based jar files.

Synopsis

Common Attributes
`buckminster.component.self`
Inherited Attributes from POJO

public
`java.binary.archives`
`java.binary.folder`
`java.binaries`

Attributes
-

Attributes

Public Attributes

`java.binary.archives`
: Is a group that contains the `buckminster.component.self` attribute.

`java.binary.folder`
: The attribute `java.binaries` will point to this instead of the `java.binary.archives` when the jar is in source form. The `java.binary.folder` will typically appoint the `bin` directory (the output of the Java compiler).

`java.binaries`
: Is a group that contains the `java.binary.archives` attribute.

Dependencies

Dependencies in the Maven POM file(s) are transformed into dependencies in the resulting CSPEC.

Description

The maven(1), and maven2 types are used for jar based components that have maven (version 1 or 2) meta data embedded in the jar file.

Buckminster supports the resolution/materialization aspects of maven binary repositories. (In theory, Buckminster should be able to generate a correct CSPEC from source as well since the Maven POM is present there, but this is untested).

osgi.bundle

osgi.bundle — component type for Eclipse plugins and OSGi bundles.

Synopsis

Common Attributes
`buckminster.component.self`
Inherited from PDE component type
public
`buckminster.clean`
`build.properties (source only)`
`bundle.jars`
`manifest`
`product.configuration.exports`
private
`product.configurations`
`«product name»`
`jar.contents (source only)`
`raw.manifest`
Public Attributes
`bundle.and.fragments`
`bundle.and.fragments.source`
`bundle.jar`
`eclipse.build.source (source only)`
`java.binaries`
`source.bundle.jar (source only)`
Private Attributes
`source.manifest (source only)`
`bin.includes (source only)`
`bundle.classpath (binary only)`
`create.«jar name» (source only)`
`eclipse.build.requirements (source only)`
`eclipse.build, eclipse.build.output.«dottified output directory» (source only)`

Attributes

Public Attributes

`bundle.and.fragments`
: A *group* consisting of the `bundle.jar` and `target.fragments`.

`bundle.and.fragments.source`
: A *group* consisting of the `source.bundle.jar` and `target.fragments.source`.

`bundle.jar`
: Represents a jar containing an OSGi-bundle. For bundles in binary form, this is typically just an *artifact* with a path referencing the bundle. For bundles in source form, this points to an *action* that builds the jar using the predefined `create.bundle.jar` ANT-task with the two attributes `jar.contents` and `manifest` as input.

`eclipse.build.source` (*source only*)
: An *artifact* deduced from the `.classpath`-file that denotes where the Java source for the `eclipse.build` action is to be found. In case there is more then one source folder, the artifacts will be named `eclipse.build.source_0`, `eclipse.build.source_1` etc.

Note

The `eclipse.build.source` attribute(s) is declared to be public, but should really be private.

`java.binaries`
: For a *binary bundle,* this is a *group* containing the `bundle.classpath`. For a *source bundle*, this is a *group* that includes the `eclipse.build` *action* and the `java.binaries` attribute of all bundles listed as dependencies.

`source.bundle.jar` (*source only*)
: An *action* that builds the source bundle using the predefined `create.bundle.jar` ANT-task with the two attributes `src.includes` and `source.manifest` as input.

Private Attributes

`source.manifest` (*source only*)
: An *action* that produces a manifest for a source bundle based on its input attributes `manifest` and `build.properties`.

`bin.includes` (*source only*)
: An *artifact* with multiple paths. Each path appoints a file or a directory denoted in the `bin.includes` property of the `build.properties` file.

`bundle.classpath` (*binary only*)
: Appoints the jar file that represents the bundle.

`create.`*«jar name»* (*source only*)
: This *action* only applies to *nested bundles*. It describes how one of the nested jars of the bundle is build.

`eclipse.build.requirements` (*source only*)
: This is a group of all `java.binaries` from all dependent components. Used as prerequisite in the `eclipse.build` action.

`eclipse.build` , `eclipse.build.output.`*«dottified output directory»* (*source only*)
: The actions of the internal *Eclipse Builder* are represented in the CSPEC by the `eclipse.build` action. This action uses the *Eclipse Builder* to build the project. The action has two prerequisite attributes — `eclipse.build.requirements` and `eclipse.build.source`.

  The `.classpath` file of the project is consulted to find the output folder(s) used by the *Eclipse Java Compiler*. Each such folder will result in a *product* of the action `eclipse.build`.

  As an example, if the `.classpath` file contains this entry:

```
<classpathentry kind="output" path="bin/classes"/>
```

  then `eclipse.build.output.bin.classes` will be the name of one *product* produced by the `eclipse.build` action.

Dependencies

Dependencies are supported. In cases where the CSPEC is generated from a p2 IU, all *required capabilities* appointing an `osgi.bundle` will be transformed to a CSPEC dependency. All other required capabilities are ignored.

When the CSPEC is not based on a p2 IU, the `META-INF/MANIFEST.MF` file is consulted and each bundle listed in the the OSGi property `Require-Bundle` is transformed into a CSPEC dependency.

Description

The `osgi.bundle` component type is automatically generated for all OSGi bundles, and Eclipse plugins. It can generate a CSPEC from source or binary data. When generating the CSPEC, the various meta data files in the source, or available in the binary representation are used to create the attributes.

The `META-INF/MANIFEST.MF` file is the source of the generated dependencies. The `.classpath` gives the `eclipse.build.requirements` and the outputs of the `eclipse.build`. The `build.properties` file tells Buckminster which jars to build in case of nested components and what to include in the built binary and source jars.

PDE (abstract)

PDE (abstract) — abstract component having attributes shared by all PDE based component types.

Synopsis

Public Attributes

`buckminster.clean`
`build.properties (source only)`
`bundle.jars`
`manifest`
`product.configuration.exports`

Private Attributes

`product.configurations`
«product name»
`jar.contents (source only)`
`raw.manifest`

Attributes

Public Attributes

`buckminster.clean`
: Cleans out any result from a previous build. Might be a null operation since some artifacts just provide themselves ‘as is’, and there is never anything to clean.

`build.properties` (*source only*)
: An *artifact* that represents the `build.properties` file of the feature or plug-in. It is optional since the file from which it is generated is optional.

`bundle.jars`
: This is an aggregation of the transitive scope of all `bundle.jars`. For a bundle, this is a *group* that also contains the `bundle.and.fragments` attribute (which means that the result is actually a listing of all bundles and fragments).

`manifest`
: An action that updates the version qualifiers of a `manifest.mf` or `feature.xml` file according to the version qualifiers that are in effect for the build (see the ‘buckminster.versionQualifier ANT-task’ for information how the Buckminster version qualifier mechanism works). The input to the `manifest` action are the `build.properties` and `raw.manifest` attributes.

`product.configuration.exports`
: This is an aggregation of the transitive scope of all `product.configuration.exports`. It might also contain a reference to a `product.configurations` attribute.

Private Attributes

`product.configurations`
: References private *«product name»* artifacts.

«product name»
: One attribute is generated per identified *Eclipse product*. (A component can have multiple `.product` definitions, and they are identified by a ‘product name’ stored in the `.product` file).

  These artifacts contain one single path each. Each path represents a `.product` file found in the source.

Warning

The name is currently not prefixed by Buckminster, and a product name may thus conflict with another CSPEC attribute. As a consequence — do not name attributes the same as your Eclipse based products!

`jar.contents` (*source only*)
Describes the artifacts (minus the `manifest.mf`) that should go into the jar that represents the component. For a feature, this is generated as an artifact with multiple paths. Each path appoints a file or a directory denoted in the `bin.includes` property of the `build.properties` file (i.e. similar to a `bin.includes` in a bundle). For a bundle, it gets more complicated. There are two types of bundles:

- **Simple bundle.** Basically a normal java jar file with a special OSGi manifest. For this type, the `jar.contents` is a group consisting of the `bin.includes` and the `eclipse.build.output.`*«dottified output directory»* attributes.
- **Nested bundle.** A java jar file with a special OSGi manifest that contains a class path that in turn points to embedded nested jar files. All `.class` files reside in the nested jars. Here, the `jar.contents` is a group consisting of the `bin.includes` attribute and then one `create.`*«jar name»* attribute for each of the nested jar files.

`raw.manifest`
An artifact representing the raw manifest file (the `manifest.mf` or `feature.xml` file) that contains versions that has not yet been qualified.

Dependencies

Dependency generation for PDE based components is different for the subtypes. See the respective subtype for more information.

Description

The PDE type is abstract and should never be directly used in any Buckminster artifacts. Its only purpose is to describe the attributes that are common to all PDE component types.

To work correctly, the meta data must use “Bundle-ManifestVersion: 2”, and be free of errors.

POJO (abstract)

POJO (abstract) — abstract component having attributes shared by all *Plain Old Java* (POJO) based component types.

Synopsis

Public Attributes

`java.binary.archives`
`java.binary.folder`
`java.binaries`

Private Attributes
-

Attributes

Public Attributes

`java.binary.archives`
: Denotes the component's export of jar files.

`java.binary.folder`
: Denotes the component's export of folders containing java binary artifacts (class files or resources).

`java.binaries`
: A *group* containing either the `java.binary.archives` or `java.binary.folder`. Typically used when assembling class paths used as input for a compiler. Can be thought of as a format agnostic (i.e. directory or jar) form of referencing compiled java code.

Dependencies

The POJO component type is abstract and does not specify any dependencies. Specialized types may add support for dependencies.

Description

The POJO type is abstract and should never be directly used in any Buckminster artifacts. Its only purpose is to describe the attributes that are common to all POJO component types.

Note

If you have a POJO component, and want additional meta data, you should turn it into a Buckminster component by adding a `buckminster.cspec`. When doing this, you probably also want to add the attributes that are common to all POJO components — they are not added automatically.

Actors

This reference guide contains detailed information about the actors supported by Buckminster in its standard configuration. Actors are used in CSPEC actions. See Chapter 6, *Components* for more information where actors are used.

Conventions used

The following conventions are used in this reference guide:

Actor properties	A group of properties in an action called "actor properties".
General Properties	A group of properties in an action called "general properties".
Prerequisites alias	Alias assigned to prerequisites in an action. Used by actions to reference files.
Buckminster *f* Feature	When displayed under the actor name then the *f* feature must be installed for the actor to be available.
Action product	An action's output is called *product*. The product is specified as a path group.
«replaceable»	A replaceable part of a name is written within guillemots « »

ant actor

`ant` actor — an actor capable of executing ANT-scripts.

Synopsis

Actor Properties
`targets` `[buildFile	buildFileId]`
General Properties
Declared properties are passed to the ANT-script
Action Prerequisites & Action Products
All aliased action prerequisites and action product(s) are passed to the ANT-script

Actor Properties

`targets`
: A comma separated list of ANT targets to call. By default, a target with the same name as the action attribute is called.

Note

The order in which the targets are called is determined by the ant runtime and may differ from the order in which they are declared in case the target has inter-dependencies.

`buildFile`
: The build file to use as input. The value should be an absolute file system path. One of `buildFile` or `buildFileId` should be used.

`buildFileId`
: The ID of a build file that is registered by an extension point. The *Buckminster PDE bundle* registers the build file `buckminster.pdetasks`.

General Properties

Any general properties set in the action invoking the `ant` actor are available in the ant-script as properties. See the section called “Access to properties”.

Description

The `ant` actor is capable of invoking targets in an ANT-script. Such a script is referred to as a *build-file*, and a reference to the build-file to use must be passed either by using the actor property `buildFile` which is an absolute file system path to an ANT *build file*, or via the actor property `buildFileId`, which is the identity of a pre-registered build-file. The *Buckminster* PDE *Feature* adds the identity `buckminster.pdetasks`, which is used by the automatically generated CSPECs for PDE based component types. (See the ‘Component Types’ reference guide for information about the PDE component types and the available actions). Although the `buckminster.pdetasks` can be used from actions you construct, it is not considered to have a stable API, so use the automatically generated actions if you can.

Advanced — Reusable Build Scripts Extensions

If you are an advanced users it may be good to know that the Buckminster extension point `org.eclipse.buckminster.ant.buildScripts` can be used to register additional build identifiers in order to create reusable ANT-targets. Explaining how this is done is not within scope for this reference guide.

Ant Runner

Buckminster uses the standard Eclipse *Ant Runner* to run ANT-scripts. This has several advantages:

- The process can be canceled from a normal progress monitor.
- No additional JVM is started which is faster and saves resources.
- The default `javac` compiler will be the one provided by Eclipse JDT.
- All ANT-tasks provided by the `org.eclipse.ant.core.antTasks` extension point becomes available (There are many. Search the *Eclipse Help* for ‘*Ant task*’ to see an index).
- Properties provided by the `org.eclipse.ant.core.antProperties` extension point is automatically available (see ‘*Ant Properties*’ in *Eclipse Help*). By default, this adds at least three properties:

`eclipse.home`	The Eclipse installation directory (or, in a headless scenario, the Buckminster installation directory since Buckminster *is* Eclipse in this context).
`eclipse.running`	Will always be set to `true` when executing an ANT-action.
`buckminster.pdetasks`	The location of the build script provided by the *Buckminster PDE Feature*. Useful if you want to reference its targets from another script.

- Types provided by the `org.eclipse.ant.core.antTypes` extension point is automatically available. The platform does not currently provide any new types but Buckminster does (as explained later).

Access to properties

An action has two kinds of properties. The *actor properties* that controls the actor, and the *general properties*. The `ant` actor configures itself using the *actor properties*. They tell the actor where to find the build script and what targets to execute. The *actor properties* are not available from within the build script. The *general properties* however, are provided as normal user defined properties, just as if you execute ANT from the command line and pass the properties using `-D«xxx»=«yyy»` settings. You reach them in your ANT-script by using standard `${«name»}` syntax to expand the property `«name»` to its value.

Access to prerequisites and product locations

You should already be familiar with the concept that all attributes in Buckminster can be thought of as one or several *path groups* where each *path group* has a *base path* and a list of zero or more paths. You also know that when used as a *prerequisite* in an action, the prerequisite can be given a name (the prerequisite alias). The same is true about the action’s product(s).

Buckminster will pass prerequisites and products as properties to ANT using the prefix ‘`fs:`’ before the alias. The value of such a property is formatted as follows:

```
?«base»[;«path»[;«path» «...» ]][?«base»[;«path»[;«path» «...» ]] «...»]
```

As a convenience, for the common case where the property only represents one single path, (i.e. there is only one single *path group* and this *path group* has zero or one path), a second property is provided for the alias with the prefix ‘sp:’ containing the single path as its value.

Usage in ANT-script

The properties passed by Buckminster can be used in ANT-scripts by using ANT-types designed for this particular purpose. See “filesetgroup support” in the “Buckminster ANT tasks” reference guide for how this is done.

See Also

jdt.ant actor

copyTargetAction actor

`copyTargetAction` actor — an internal actor that copies fragments.
Buckminster PDE Feature

Synopsis

Actor Properties
ignored
General Properties
ignored
Action Prerequisites & Action Products
The output/product must be provided and must appoint a single directory.

Description

An actor that finds all fragments for the bundle associated with the CSPEC invoking the actor, and copies them to the output specified by the containing action. Fragments ending with ‘`.compatibility`’, ‘`.test`’, or ‘`.dummy`’ are excluded from the copy.

Note

This actor is intended for internal use in the PDE build process.

eclipse.build actor

`eclipse.build` actor — an actor that builds by invoking the *Eclipse Build System*.

Synopsis

Actor Properties
`kind`
General Properties
ignored
Action Prerequisites & Action Products
ignored

Actor Properties

`kind`

The type of build to invoke can be set to `eclipse.incremental` or `eclipse.full`. The default if kind is not specified is `eclipse.incremental`.

Deprecated kinds. The following values for `kind` are deprecated: ~~`eclipse.auto`~~ (which is the same as `eclipse.incremental`), ~~`eclipse.clean`~~ (replaced by the `eclipse.clean` actor), and ~~`eclipse.build`~~ (which is the same thing as `eclipse.full`).

Description

Requests a build from the *Eclipse Build System*. Essentially the same as doing a '*Project* → *Build Project*' in the Eclipse IDE. This actor looks at the actor property `kind` which can be set to one of `eclipse.incremental` or `eclipse.full`. The default is `eclipse.incremental`.

See Also

`eclipse.clean` actor

eclipse.clean actor

`eclipse.clean` actor — an actor performing the same as the Eclipse IDE “clean”.

Synopsis

Actor Properties
ignored
General Properties
ignored
Action Prerequisites & Action Products
ignored

Description

Requests a clean from the Eclipse Build system. Essentially the same as doing a ‘*Project* → *Clean...*’ in the Eclipse IDE for the project associated with the CSPEC invoking this actor.

If you want to clean the entire workspace, you can do so with the command ‘*Project* → *Clean...*’ command (and select ‘*Clean all projects*’ in the Eclipse IDE user interface, or if you are running headless by using the command `buckminster clean` on the workspace you want to clean.

executor actor

`executor` actor — an actor that executes a system command using ‘exec’

Synopsis

Actor Properties

`env`
`newenvironment`
`[exec | shell]`
`execDir`
`failOnError`

General Properties
ignored
Action Prerequisites & Action Products
ignored

Actor Properties

`env`
: A semicolon separated string of environment variable settings. The variables are subject to property expansion.

`newenvironment`
: A boolean value that if set, will make `env` the only environment that is provided i.e. the current environment is not inherited.

`exec`
: The command to execute. One of `shell` or `exec` must be specified, but not both.

`shell`
: A shell command to execute. Will be prepended by ‘`cmd.exe /C`’ on Windows or ‘`sh -c`’ on other platforms. One of `shell` or `exec` must be specified, but not both.

`execDir`
: The directory to use as current for the execution. Defaults to the component’s location and if relative, becomes relative to the component’s location.

`failOnError`
: If set (which is the default), the actor will fail unless the `exec` or `shell` command returns a zero exit status.

Description

This actor executes an external command (through Java `Runtime.getRuntime().exec()`). The execution is controlled by the actor properties.

This makes it possible to run system commands and scripts.

Note

You can currently only pass information to the executed command/script via the `env` string.

Warning

There is no special handling of escapes, special characters and quotes. You will need to experiment to get the correct values into the env string, as well as the command line arguments to suit the operating system you are running on. Do not just blindly issue dangerous commands without first making absolutely sure you are passing arguments to the command/script the way you expect.

fetcher actor

`fetcher` actor — an actor that fetches additional artifacts from an URL

Synopsis

Actor Properties

`url`
`dir`
`options`
`login`
`pass`

General Properties
ignored
Action Prerequisites & Action Products
ignored

Actor Properties

`url`
: The URL to fetch.

`dir`
: The directory where the fetched result should be stored. Defaults to the component's location. If relative, becomes relative to the component's location.

`options`
: Semicolon separated list of options controlling how to deal with the fetched archive. Can contain `uncompress`, `flatten`, and multiple `include`. Example:

```
uncompress;flatten;include=*.cpp;include=*.c;
```

`uncompress`
: Uncompresses a fetched archive. Supports archives in the following formats: `.zip`, `.tar.gz`, `.tgz`, `.tbz2` and `.tar.bz2`.

`flatten`
: Flattens an uncompressed archive (i.e. all files are uncompressed into the same location). Has no effect if `uncompress` is not also stated.

`include=[-]`*«pattern»*
: Allows specification of which files to include or exclude when uncompressing. An `include` has no effect if `uncompress` is not also stated. The *«pattern»* is a simplified regular expression describing which files to include. If preceded by a '`-`' the files matched by the pattern are excluded. Several include options can be used in the same options argument.

```
include=binaries/*;include=-*.html
```

The simplified regular expression handles `*` to mean zero or more characters, and `?` to mean one or more characters. There is no need to quote '`.`' (it normally mens 'any character' in a regexp). The pattern is also rewritten so that path separators ('/' or '\' are changed into '/').

Note

This has the effect that simple patterns using path separators, `*` and `?` works well, but it is not possible to use the pattern as a full regular expression. If the name you are trying to match with a pattern that needs to include characters that are special in a regular expression, they can not be quoted if you are running on windows as a '\' is treated as a path separator.

`login`
: The login to use (optional). Subject to property expansion.

`pass`
: The password to use (optional). Subject to property expansion.

Warning

The password is stored in plain text. Proper care must be taken to protect a CSPEC that contains a password string. It is recommended to pass the login/password using system properties. When entering the login/password directly in the CSPEC, you also need to protect any BOM files that you save as they contain copies of the CSPECs. If you use properties however, only provider related properties are expanded when the BOM is created (i.e. expansion of login/password properties are not saved in the BOM).

Description

An actor capable of fetching things during a build.

Warning

Use of the `fetcher` actor should not be your first choice since it overlaps with provisioning, and there is no support for dependency management. In some cases however, the build itself concludes what to fetch in mysterious ways, and the fetcher becomes the only option.

The `fetcher` is typically used to fetch an archive, but can be used to fetch any single file.

When fetching an archive, it can be both uncompressed, and flattened as controlled by the `options` actor property.

jarprocessor actor

`jarprocessor` actor — an actor capable of performing packing operation on a jar file
Buckminster PDE Feature

Synopsis

Actor Properties
ignored
General Properties
`command`
Action Prerequisites Aliases
`jar.folder`
Action Product Requirements
The action must have a single path product with a path to a directory for output.

Actor Properties

Note

Values should be passed using General Properties.

General Properties

`command`	Should be set to `repack`, `pack`, or `unpack`.

Component Attributes

`project.classpath`
A *path group* containing the class path from a JDT based project.

Description

This actor is a much improved version of the *p2 jarprocessor*. It does not rely on external processing and creates a minimum amount of temporary files on disk during processing (the data is streamed between the different stages). The processor is also intelligent in that it automatically excludes jars that do not include Java binaries (`.class` files) from `pack200` processing and it does not `gzip` the container of gzipped files. One important improvement over the p2 predecessor is that the `jarprocessor` actor does not silently ignore errors.

The actor uses the property `command` which can be set to one of `repack`, `pack`, or `unpack`. The actor also expect to find a *prerequisite* with the *alias* `jar.folder` and that the *action product* appoints one single path of a directory where it can put its result.

About Pack200

Pack200 is not a lossless compression. Packing and unpacking will produce a jar that is semantically the same as the original, but class-file structures will be rearranged; the resulting jar will not be identical to the original. However, this reordering is idempotent so a second pack-unpack will not further change the jar.

Pack200 reduces the size of a JAR file by:

1. Merging and sorting the constant-pool data in the class files and co-locating them in the archive.
2. Removing redundant class attributes.
3. Storing internal data structures.
4. Using delta and variable length encoding.
5. Choosing optimum coding types for secondary compression.

Signing a jar hashes the contents and stores the hash codes in the manifest. Since packing and unpacking a jar will modify the contents, the jar must be normalized prior to signing. Normalizing the jar will also be referred to as *repacking* or conditioning the jar.

From http://wiki.eclipse.org/Pack200

See Also

http://wiki.eclipse.org/Pack200

jdt.ant actor

`jdt.ant` actor — an actor capable of executing ANT-scripts with support for JDT project classpath.
Buckminster PDE Feature

Synopsis

Actor Properties

`targets`
`buildFile`
`buildFileId`

General Properties
Declared properties are passed to ANT-script
Action Prerequisites & Action Products
`project.classpath`

Actor Properties

See `ant` actor.

General Properties

See `ant` actor.

Component Attributes

In addition to the component attributes made available to the `ant` actor , the `jdt.ant` actor also makes the following component attribute available to the ANT-script.

`project.classpath`
: A *path group* containing the class path from a JDT based project.

Description

See `ant` actor for a description. The `jdt.ant` actor just adds access to the `project.classpath` component attribute.

See Also

`ant` actor

null actor

`null` actor — an actor that (for testing purposes) does nothing.

Synopsis

Actor Properties
ignored
General Properties
ignored
Action Prerequisites & Action Products
ignored

Description

The `null` actor does nothing. It can be used when testing and some actor should be temporarily disabled in XML. The real actor can be replaced with a `null` actor to trigger all the prerequisites without invoking any real actor.

See Also

`simulation` actor

p2SiteGenerator actor

`p2SiteGenerator` actor — an actor generating a p2 (update) site
Buckminster PDE Feature

Synopsis

Actor Properties
ignored
General Properties
ignored
Action Prerequisites Aliases

`site`
`site.definer`
`product.configs`

Action Product Requirements
The action must have a single path product with a path to a directory for output.

Description

This actor will produce a p2 update site with co-located meta-data and artifacts. This actor expects the the following aliased *prerequisites*:

`site`	A pre built directory that contains a features/ and plugins/ sub-directory which in turn contains the feature and bundle jar files.
`site.definer`	A reference to a `feature.xml` file, or a *«xxx»*`.product` file that defines the site.
`product.configs` (optional)	A prerequisite that lists all additional product configurations to include on the site.

This actor also requires that the *action product* appoints one single path of a directory where it can put its result.

See Also

Chapter 2, *p2*.

simulation actor

`simulation` actor — an actor (for testing purposes), that does nothing except report progress.

Synopsis

Actor Properties
`ticks`
General Properties
ignored
Action Prerequisites & Action Products
ignored

Actor Properties

`ticks`
: A delay value in milliseconds between 0 and 60000. If not specified, the default is 3000.

Description

An actor that imposes a delay by ticking on a progress monitor and logs when its finished. For testing purposes. This actor looks at the actor property `ticks` which is a millisecond value that has to be between 0 and 60000. The default is 3000.

See Also

`null` actor

Buckminster ANT tasks

This reference guide describes the Buckminster features available when writing ANT scripts.

filesetgroup support

filesetgroup support — Adds fileset-group support to ANT, and handles Buckminster path groups.

Synopsis

```
buckminster.filesetgroup
buckminster.valuefileset
buckminster.valuepath

buckminster.apply
buckminster.copy
buckminster.delete
buckminter.jar
buckminster.zip
```

Description

The properties passed by Buckminster can be used in ANT-scripts by using ANT-types designed for this particular purpose. The Buckminster ANT-types are:

`buckminster.filesetgroup`	As the name suggests, this is a group of ANT *filesets*. (Prior to Ant 1.7, it was not possible to define such groups in ANT). This type can be initialized from a `fs:`*«xxx»* property value.
`buckminster.valuefileset`	An ANT *fileset* that can be initialized from a `fs:`*«xxx»* property. Only the first *path group* in the property will be considered in case it contains several.
`buckminster.valuepath`	An ANT-*path* (which actually is several paths separated by a path separator (i.e. more similar to a classpath then just a path). When initialized from a `fs:`*«xxx»* property it will contain all paths (expanded) that the property contained.

The `buckminster.valuefileset` and `buckminster.valuepath` can be used as slot-in replacements wherever ANT expects a *fileset* or a *path* respectively. The `buckminster.filesetgroup` however, cannot be used this way since ANT does not know what it is. ANT has several tasks that can take a *list of filesets* as input and Buckminster provides extended versions of these tasks that also accepts `buckminster.filesetgroups`. These tasks are:

`buckminster.apply`	Extends the ANT `apply` task.
`buckminster.copy`	Extends the ANT `copy` task.
`buckminster.delete`	Extends the ANT `delete` task.
`buckminster.jar`	Extends the ANT `apply` task.
`buckminster.zip`	Extends the ANT `zip` task.

This current construction handling *fileset-group* is subject to future improvement since the handling of filesets in ANT has undergone a major overhaul in ANT 1.7 and now have resources that can be used to group *filesets*.

Examples

This ANT-script example assumes that the action product (output) is aliased `action.output` and the prerequisites (input) aliased `action.requirements`. It will copy everything found in the input (relative to the path groups' bases) into the action output.

```
<target name="copy.group">
    <mkdir dir="${sp:action.output}"/>
    <buckminster.copy todir="${sp:action.output}" overwrite="true">
        <buckminster.filesetgroup ❶ value="${fs:action.requirements}" ❷ />
    </buckminster.copy>
</target>
```

❶ The `buckminster.filesetgroup` is used to adapt the input to an ANT filesetgroup.

❷ The `action.requirements` is the input (a *path group*) which needs adaption to be used as an ANT *filesetgroup*.

buckminster.importResource

buckminster.importResource — like ants import command but for a resource provided with an optional classpath

Synopsis

```
<buckminster.importResource
resource=«resource-ref»
[ classpath=«antpath» ]
[ optional=«flag» ]
/>
```

Description

Works similar to ANT's import command but instead of providing a filename, a resource is provided together with an optional classpath (defaults to the system classpath). The resource is resolved using the Java Class Loader.

`resource=«resource-ref»`
: A string denoting the resource.

`classpath=«antpath»`
: An ANT path.

`optional=«flag»`
: A boolean indicating that a missing resource is OK

`optional=«flag»`
: If set, the output is less verbose.

buckminster.lastTimestamp

buckminster.lastTimestamp — obtain the last timestamp from a repository location.

Synopsis

```
<buckminster.lastTimestamp
property=«name»
readerType=«name»
repositoryLocation=«uri»
[ versionSelector=«branch--or-tag» ]
[ dateFormat=«fmtstring» ]
[ timezone=«tz-name» ]
/>
```

Description

Uses a specific buckminster reader type to obtain the last timestamp from a repository location. The timestamp is stored in a property.

`property=«name»`
: The name of the property that receives the timestamp.

`readerType=«name»`
: The name of the Buckminster reader type.

`repositoryLocation=«uri»`
: The reader type specific URI that appoints the remote source.

`versionSelector=«branch-or-tag»`
: A branch or tag. Default is main (i.e. HEAD/trunk, etc.). Branches are entered by simply stating their name, and tags are entered with a leading slash ‘`/`’ character. The special keyword ‘`main`’ is used to refer to the repositories notion of *main branch* (e.g. ‘trunk’ for SVN, ‘head’ for CVS, etc.).

`dateFormat=«fmt-string»`
: The `SimpleDateFormat` format to use. Defaults to ISO standard “`yyyy-MM-dd'T'HH:mm:ss`”.

`timezone=«tz-name»`
: The timezone to use. Defaults to “`UTC`”.

buckminster.lastRevision

buckminster.lastRevision — obtain the last revision from a repository location.

Synopsis

```
<buckminster.lastRevision
property=«name»
readerType=«name»
{ repositoryLocation=«uri» | workingCopy=«fmtstring» }
[ versionSelector=«branch--or-tag» ]
[ timezone=«tz-name» ]
/>
```

Description

Uses a specific buckminster reader type to obtain the last revision from a repository location. The revision is stored in a property.

`property=«name»`
: The name of the property that receives the timestamp.

`readerType=«name»`
: The name of the Buckminster reader type.

`repositoryLocation=«uri»`
: The reader type specific URI that appoints the remote source. Only one of `repositoryLocation` and `workingCopy` can be used.

`workingCopy=«file»`
: A file denoting a local working copy of the source. Only one of `workingCopy` and `repositoryLocation` can be used.

`versionSelector=«branch-or-tag»`
: A branch or tag. Default is main (i.e. HEAD/trunk, etc.).

buckminster.substitute

buckminster.substitute — performs regular expression substitution on a property.

Synopsis

```
<buckminster.substitute
property=«name»
pattern=«regexp»
[ quotePattern=«flag» ]
[ replacement=«replacement-string» ]
[ value=«val» ]
/>
```

Description

Performs regular expression substitution on a property.

`property=«name»`
: The name of the property that receives the result.

`pattern=«regexp»`
: The regular expression pattern.

`quotePattern=«flag»`
: A boolean denoting that the pattern should be quoted. Optional and defaults to `false`.

`replacement=«replacement-string»`
: A replacement string for the expression.

`value=«val»`
: The input to the substitution. (Typically a reference to a property e.g. `$someProperty`).

buckminster.versionQualifier

buckminster.versionQualifier — substitutes the ‘qualifier’ part of the fourth segment of an OSGi version.

Synopsis

```
<buckminster.versionQualifier
componentName=«name»
componentType=«name»
property=«name»
[ propertiesFile=«file» ]
[ qualifier=«value» ]
[ version=«version-string» ]
/>
```

Description

Substitutes the ‘qualifier’ part of the fourth segment of an OSGi version according to environment properties. There are currently three types of qualifier replacement strategies; `lastRevision` (replacing the qualifier with the last revision), `lastModified` (replacing the qualifier with a timestamp of the latest changed resource), and `buildTimestamp` (replacing the qualifier with a specified timestamp (or the current time)). The `lastRevision` uses a qualifier generator that works like the `buckminster.lastRevision` task (it executes the same code underneath). It will always use the working copy. The `lastModified` works like the `buckminster.lastTimestamp` task.

Properties

The task performs substitution as directed by properties. One set of properties control strategy selection given a component name, and one set controls the revision and timestamp formats.

Strategy selection is done by specifying properties on the format:

```
qualifier.replacement.«pattern» =
{ generator:lastModified | generator:lastRevsision | generator:buildTimestamp }
```

The *«pattern»* specifies with which components the specified version qualifier strategy applies. The pattern is not a regular expression, instead it has a simplified syntax where ? means any character and * means zero or more characters.

Controlling the version qualifier formats are done with the properties:

`generator.lastRevision.format=«numberFormat»`
: Should be a Java `MessageFormat` for a number such as “`r{0,number,00000}`”.

`generator.lastModified.format=«dateTimeFormat»`
: Should be a Java `SimpleDateFormat` such as “`'v'yyyyMMdd-HHmm`”.

`generator.buildTimestamp.format=«dateTimeFormat»`
: Should be a Java `SimpleDateFormat` such as “`'v'yyyyMMdd-HHmm`”.

`buckminster.build.timestamp`
: Used by the `buildTimestamp` generator. It should be a timestamp in ISO-8601 format, i.e. “yyyy-MM-dd'T'HH:mm:ss.SSSZ” and reflect a timestamp in UTC. If the `buildTimestamp` generator is used and this property is not set, the time of the perform command invokation is used as the timestamp.

Task Attributes

The attributes are used as follows:

`componentName=«name»`	The name of the designated component.
`componentType=«name»`	The name of the component type of the designated component.
`propertiesFile=«file»`	The properties file to use. Will be superimposed on top of system properties if provided.
`property=«name»`	The name of the property that will receive the result.
`qualifier=«value»`	Explicit qualifier to use for the replacement (normally not used).
`version=«version-string»`	The version in omni version format.
~~`versionType=«version-type»`~~	Deprecated, and has no effect. Must be empty or set to `OSGi`.

buckminster.signatureCleaner

buckminster.signatureCleaner — removes signing from jar files so that the file can be re-packed

Synopsis

```
<buckminster.signatureCleaner >
[ «file-set-element» ]
</buckminster.signatureCleaner >
```

Description

Removes signing from jar files so that the file can be re-packed and re-signed. This task has no attributes. Instead it uses a nested fileset as input.

buckminster.perform

buckminster.perform — (advanced) call on Buckminster from within ANT

Synopsis

```
<buckminster.perform
attribute=«name»
component=«name»
[ inWorkspace=«flag» ]
[ quiet=«flag» ]
/>
```

Description

Warning

This is an advanced action that if used the wrong way will cause confusing results, and can potentially deadlock. Do not use this action unless you know exactly how Buckminster and ANT works together, and you know exactly why you want this mechanism instead of some other solution.

This special operation allows nested calls to Buckminster from within ANT. This is normally discouraged since it becomes unclear who is responsible for the orchestration, Buckminster or ANT, but may be needed in very advanced situations. Proper care needs to be taken to not create deadlocks.

`attribute=`*«name»*
: The CSPEC attribute to perform (typically an action)

`component=`*«name»*
: The component containing the attribute

`inWorkspace=`*«flag»*
: Advanced usage. Used for avoiding deadlocks when running workspace jobs.

`quiet=`*«flag»*
: If set, the output is less verbose.

Filters

This reference guide describes the filter mechanism used by Buckminster and Eclipse. It also contains a reference for the system properties most commonly used in Buckminster filter expressions.

How the filters work

Buckminster uses filters to control inclusion of components (in resolutions), dependencies (in components), and component attributes (in attribute prerequisites).

A filter is specified in an *item* and is used to determine if the item should be included or not in a "search result". The terminology is in reverse from what you may think — i.e. the filter is specified on the item that is perceived to be "filtered out". Further confusion arises as the filter is used to specify the inclusion of the item (i.e. the item is included if the filter evaluates to true). An empty filter always evaluates to true.

As an example, a component may specify a filter like `(target.os=macosx)` which means that it will only accept inclusion in a search result where the context has a value for `target.os` that is equal to '`macosx`'.

Filter variables

The filter operates on properties that are made available to the filter logic. In an Eclipse environment the filter logic is given access to the system properties. Buckminster mimics the variables defined by OSGi, but since Buckminster is system agnostic, it uses variables called "target" instead of "osgi" (e.g. OSGi filters use `osgi.os`, and Buckminster filters use `target.os`). When Buckminster reads PDE artifacts the filter names are translated from the `osgi` form to the `target` form.

Tip

It is possible to use any system property should you have some very special needs.

Warning

The filtering mechanism simply attempts to obtain the value for the named variable and then use it to filter. This means that if you mistype a property name, you will not get an error e.g. if you mistyped `(target.os=macosx)` and instead entered `(tagret.os=macosx)` then the filter will always return false as there probably is no `tagret.os` system property with a value of '`macosx`'.

In some places in the Eclipse UI, the system property names have been abbreviated (the prefix is dropped). In some cases a user interface may also hide the fact that the entire filter expression must be enclosed in parentheses. This makes the UI somewhat less cluttered, but has unfortunately tricked users into entering filters without parentheses and using the abbreviated system property names in the underlying LDAP filter strings.

Note

You should always use the full system property name; i.e. `target.os`, `target.arch` in LDAP filter strings, and always enclose the filter expression in parentheses.

Wildcards

Buckminster handles wildcard as *input*. If you define a property to have the value '`*`' any comparison against this value in a filter expression will yield true. As an example, if you set `target.os=*` and a filter specifies `(target.os=macosx)` the result is true).

Warning

Note that filters that use reverse logic `(!(target.os=macosx))` will return false and not be included. Such filters are however typically not used. You will get a similar effect if wildcards are used in the filter — `(target.os=mac*)`, will also return false on wildcard input.

Properties, types and matching (implementation)

When the filter value is compared, the comparison does not automatically default to string comparison unless the available property is a string. Instead, a check is made if the class of the property has a public constructor that takes a single string argument.

Consider the following:

```
(max.ratio=0.50)
```

and a property that is set like this:

```
props.put("max.ratio", "0.5");
```

Here, the comparison will be between two strings and it will yield false `("0.50" != "0.5")`. If however, the property is set like this:

```
props.put("max.ratio", new Double(0.5));
```

then, the filter mechanism will detect that the class of the provided value has a constructor that takes a string argument.

Consequently, the filter will coerce its own value prior to the comparison i.e. something equivalent to this:

```
if(new Double("0.50").equals(props.get("max.ratio")))
```

and that comparison will yield true.

Eclipse (and Buckminster) uses this by replacing all occurrences of the ‘*’ string with a *MatchAll* value in the property set prior to evaluating a filter. The *MatchAll* has a string constructor and it will answer true to all calls to equal. This means that if you specify that your `target.os` is ‘*’ in your CQUERY or in properties provided during a call to `perform`, then a subsequent evaluation of a filter like `target.os=x86` (or any other value) will yield true. The coercion mechanism does not apply when the declared filter value contains ‘*’ characters (a.k.a. as a *substring filter*). A substring filter will yield false when compared to anything but a string. No coercion takes place. A side effect of this is that any substring filter will yield false when compared to the ‘*’ input since the *MatchAll* is not a string.

LDAP Filters

The syntax of a filter string is the string representation of LDAP search filters as defined in RFC 1960: *A String Representation of LDAP Search Filters* (available at http://www.ietf.org/rfc/rfc1960.txt). It should be noted that RFC 2254: *A String Representation of LDAP Search Filters* (available at http://www.ietf.org/rfc/rfc2254.txt) supersedes RFC 1960 but only adds extensible matching and is not applicable for the implementation of filter used in the OSGi Framework API.

Here is an the syntax described in (a variant of) BNF:

```
filter ::= '(' filtercomp ')'
filtercomp ::= and | or | not | item
and ::= '&' filterlist
or ::= '|' filterlist
not ::= '!' filter
```

```
filterlist ::= filter | filter filterlist
item ::= simple | present | substring
simple ::= attr filtertype value
filtertype ::= equal | approx | greater | less
equal ::= '='
approx ::= '~='
greater ::= '>='
less ::= '<='
present ::= attr '=*'
substring ::= attr '=' initial any final
initial ::= NULL | value
any ::= '*' starval
starval ::= NULL | value '*' starval
final ::= NULL | value
```

The `attr` is the name of the value — i.e. it is a string representing an attribute, or key, in the properties objects of the services registered in the OSGi Framework. Attribute names are not case sensitive; that is, cn and CN both refer to the same attribute. The `attr` should contain no spaces though white space is allowed between the initial parenthesis ‘(’ and the start of the key, and between the end of the key and the equal sign ‘=’. The `value` is a string representing the value, or part of one, of a key in the properties objects of the registered services. If a `value` must contain one of the characters ‘`*`’ or ‘`(`’ or ‘`)`’, these characters should be escaped by preceding them with the backslash ‘`\`’ character. Spaces are significant in `value`. Space characters are defined by `java.lang.Character.isWhiteSpace()`. Note that although both the `substring` and `present` productions can produce the ‘`attr=*`’ construct; this construct is used only to denote a presence filter (i.e. that the attr is set to some value).

Consult the javadoc for the `org.osgi.framework.Filter` Interface for more information.

Note

There is no ‘not equal’ operator. To express *a != x*, you have to write `(!(a=x))`

target.arch

target.arch — filter on CPU architecture.

Synopsis

```
x86
PA_RISC
ppc
sparc
x86_64
ia64
ia64_32
win32
```

Description

The `target.arch` property is used to specify the CPU architecture.

The values listed in the Synopsis are the values for CPU architecture defined and used in Eclipse. If you are using Eclipse/Buckminster to build for other architectures you can use other values.

target.os

target.os — filter on operating system.

Synopsis

```
win32
linux
aix
solaris
hpux
qnx
macosx
epoc32
os/400
os/390
z/os
unknown
```

Description

The `target.os` property is used to specify the operating system.

The values listed in the Synopsis are the values known to Eclipse. It should be quite clear what these values stand for. If you are building software for other operating systems you can use other values.

target.nl

target.nl — filter on natural language.

Synopsis

The value for `target.nl` property is expressed as an *ISO 639 Language Code*. It is sometimes followed by (an underscore separated) *ISO 3166 Country Code* to denote the language specific to a particular county - e.g. `en_US`, `en_UK`.

The number of codes are too many to include in this documentation, and they are available from multiple online sources for instance http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt, and http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html.

Description

The `target.nl` property is used to specify the natural language.

When matching on language code you may want to use a wildcard as it is quite common to see combination of en_XX where there is no special form of English for the country 'XX'(e.g. in this author's case the locale is en_SE). At the same time, if there is both en_US, and en_UK material available you probably do not want both — so more complex logic is needed to state "Use en_US or en_UK if specified, otherwise if some unknown en* is used, use en_US". To accomplish this the filters for the resources could look like this:

```
// for the en_US resource
//
(|(target.nl=en_US)(&(target.nl=en*)(!(target.nl=en_UK))))

// for the en_UK resource
//
(target.nl=en_UK)
```

target.ws

target.ws — filter on windowing system.

Synopsis

```
wpf
motif
gtk
photon
carbon
cocoa
s60
unknown
```

Description

The `target.ws` property is used to specify the Windowing system. The values shown under Synopsis are defined in Eclipse. If you are building for some other windowing system, you can invent your own value for use with your software.

Two values need a special mention; `wpf` stands for Windows Presentation Foundation, which is more known by the name Vista, and `s60` stands for the Nokia S60 device.

Headless Commands

This reference guide contains detailed information about the headless Buckminster commands. All of the commands are given as option to the command `buckminster`.

Conventions used

The following conventions are used in this reference guide:

«replaceable»	All values that should be replaced with some value are shown withing guillemots « »
line breaks	Command synopsis for command may be broken up on several lines. When typing commands enter all option on the same line.
command names in page titles	The command names are always entered without space, but spaces have been inserted in the page titles for increased readability. See the Synopsis of each command for how the command should be entered. Some commands have short form aliases that can be used instead of the longer name — the alias is shown in the title, but the synopsis always shows the full/long form.

Common options

There are options common to all commands — see the entry for “buckminster command”.

buckminster

buckminster — invokes the headless buckminster.

Synopsis

```
buckminster [ { -? | --help } ]
[ -data  «workspace» ]
[ { -L | --loglevel } «level» ]
[ -consoleLog ]
[ --displaystacktrace ]
{ { -S | --scriptfile } «filename» |  «commandname»  [  «options»  ] }
```

Description

The headless buckminster application is used to run Buckminster commands. The command flags listed for `buckminster` are available for all commands.

Workspace

As Buckminster at its core uses the Eclipse framework, this exhibits itself in a couple of ways. One of these ways is that an ingrained concept is to work with a 'workspace', just like the Eclipse IDE does. As it happens, the location of this workspace must be set very early in the life cycle of an Eclipse application, and this is thus the interface to it. The flag is '-data' and should point to a directory (automatically created if not existing). By default, a workspace called `workspace` will be used (created when first referenced) in the user's home directory.

Logging

Buckminster internally has a logging system. This is clearly visible when in the IDE (from the preferences pages). From the command line there are a number of flags to control logging behavior. A basic idea is that Buckminster goes to some lengths to trap writes to system out/err as well as the Eclipse internal log, and to do so in a synchronous fashion with specific log writes. A full log from a run which mysteriously fails will help a great deal in isolating the problem in an 'after-the-fact' fashion.

Preferences

The following preferences affect the behavior of the headless Buckminster. If you have the need to set additional preferences (there are *many* preferences in Eclipse) you will need to use a workspace where these preferences have been set, and then use this workspace as a template workspace (which is done with an option to the headless commands that creates a workspace).

`org.eclipse.buckminster.core.buckminsterProjectPath`
: The location of the `.buckminster` project. This project is the default location for non-osgi binary artifacts when using the workspace materializer — i.e if you use the `jar`, `maven`/`maven2`, components, the downloaded material will end up in the `.buckminster` project.

`org.eclipse.buckminster.core.maxParallelMaterializations`
: The maximum number of parallel jobs used for materialisation.

`org.eclipse.buckminster.core.maxParallelResolutions`
: The number of threads to run in parallel for RMAP resolution.

`org.eclipse.buckminster.download.connectionRetryCount`
: The number of times an IO request is retried in case of an exception (other than `FileNotFound`).

`org.eclipse.buckminster.download.connectionRetryDelay`
: The number of seconds to wait between retries in case of IO exceptions. Default 1.

`org.eclipse.buckminster.jdt.complianceLevel`
: Sets the java compiler compliance level. The default is ‘1.4’ but is normally overridden by the Eclipse JDT component based on the active JRE (one of the “Installed JRE’s”). So from Buckminster's perspective, the default is whatever the Eclipse JDT decides.

`org.eclipse.buckminster.pde.targetArch`
: Sets the CPU architecture on the currently active target platform.

`org.eclipse.buckminster.pde.targetDefinition`
: Sets the active Target Platform Definition. A target platform must have a name (see ‘*Eclipse* → *Preferences* → *Plug-in Development* → *Target Platform*’ in the IDE). The value of this preference must be equal to the name of one of the available target platforms.

`org.eclipse.buckminster.pde.targetNL`
: Sets the “natural language” (i.e. `target.nl`/`osgi.nl`) property in the currently active target platform. Buckminster uses this value as the default for `target.nl`. See the ‘Filters reference guide’

`org.eclipse.buckminster.pde.targetOS`
: Sets the operating system on the currently active target platform.

`org.eclipse.buckminster.pde.targetPlatformPath`
: Set the active Target Platform to a definition that appoints a directory. (In order to maintain backward compatibility, in Eclipse 3.5, this preference creates a Target Definition with one Directory entry, named the same as the directory). See sidebar.

A note about Target Platform

A Target Definition defines a set of locations. Each location can be one of: *Directory* — a directory in the local file system. *Installation* — an installation (such as an Eclipse SDK) in the local file system. *Features* — one or more features from an installation. *Software Site* — downloads plug-ins from a p2 repository. Management of Target Definitions are new in Eclipse 3.5. Prior to this, a target platform was just a directory in the local file system.

The preferred way of handling target platforms in 3.5 is to create one with the IDE. This target definition is then saved to a file. The Buckminster command `importtargetdefinition` is then used in the headless Buckminster to use this definition.

Options

`-?`, `--help`
: Shows help for the command.

`-data «workspace»`
: This sets the workspace on which the Buckminster command operates. The argument *«workspace»* is a file path to either an existing workspace, or is a name to a workspace that will be created. If *«workspace»* is not stated, the default is to use a workspace called `workspace` located in the users home directory.

`-consoleLog`
: Enables the OSGi console logger and is very useful for debugging.

`-L «loglevel»`, `--loglevel «loglevel»`
: Sets the logging level for the command. The *«loglevel»* can be set to one of `DEBUG`, `INFO`, `WARNING`, or `ERROR`.

`--displaystacktrace`
: This option can be given if the user desires a full stack trace printout in case of actual code problems — normally it just prints a less daunting problem report if that happens. Regular ‘user errors’ (e.g. bad usage of flags or similar are still reported in a more human friendly manner).

`-S` *«filename»*`, --scriptfile` *«filename»*
: Informs Buckminster that commands should be read from a file. The file should contain one command per line, and not include the initial `buckminster` used on the command line.

«commandname» `[`*«options»*`]`
: The *«commandname»* is the name of a Buckminster command to be executed — see the other entries in this reference guide. The *«options»* are the options valid for the *«commandname»*.

listcommands (lscmds)

listcommands (lscmds) — lists the commands installed in the currently running instance.

Synopsis

```
listcommands [ { -? | --help } ]
[ --style  «name» ]
```

Description

Lists all the commands in the currently running instance. This reference guide shows the commands that are typically installed, but other configuration of Buckminster may have a different set of commands available.

Options

`-?`, `--help`
: Shows help for this command.

`--style` *«name»*
: Select the style of the printout; where *«name»* is one of `normal` (all names are shown), `short` (only the full/basic name of a command is shown), or `long` (a verbose listing detailing all aliases). The default is `normal`. (There is no short form for this option).

build (make)

build (make) — runs workspace build.

Synopsis

```
build [ { -? | --help } ]
[ { -c | --clean } ]
[ { -t | --thorough } ]
```

Description

Runs a workspace build. If `-c` is specified, a clean is performed before the build. If `-t` is specified a full build is performed if an incremental build fails.

Options

`-?` , `--help`
: Shows help for this command.

`-c` , `--clean`
: Performs a clean before the build.

`-t` , `--thorough`
: Performs a second full build instead of just adding incremental builds when the initial build is not successful.

clean

clean — cleans the workspace

Synopsis

`clean` [{ `-?` | `--help` }]

Description

This command calls on the internal eclipse builder to do a workspace clean. It is the same clean that takes place when you do a “`build --clean`” (although here it is not followed by a workspace build).

Options

`-?`, `--help`
: Shows help for `clean`

get preference (getpref)

get preference (getpref) — display the value of a preference.

Synopsis

```
getpreference [ { -? | --help } ]
[ { -d | --default } «default value» ]
[ --onlyvalue ]
«name»
```

Description

Display the value of a preference, and optionally use a default value if the selected preference does not have a value.

Options

`-?` , `--help`
: Shows help for this command.

`-d «default value»` , `--default «default value»`
: If preference has no value, the value *«default value»* becomes the result.

`--onlyvalue`
: Displays only the value.

«name»
: The name of the preference for which the value should be displayed.

import (resolve)

import (resolve) — runs a workspace import

Synopsis

```
import [ { -? | --help } ]
[ { -B | --bomfile } «filename» ]
[ -C | --continueonerror ]
[ { -D | --define } «key» [ =«value» ] ]
[ { -P | --properties } «url or path» ]
[ { -t | --template } «template workspace» ]
«url or path»
```

Description

Populates the workspace from a CQUERY, MSPEC, or BOM file. The `-N` allow this command to stop after construction of a BOM, and the `-B` writes the resulting BOM to a file.

Options

`-?` , `--help`
Shows help for this command.

`-B «filename»` , `--bomfile «filename»`
Stores the resulting BOM file in «*filename*»

`-C` , `--continueonerror`
Continue even if not all components can be imported.

`-D «key»[=«value»]` , `--define «key»[=«value»]`
Defines a property as a key=value pair. The value may include ANT-style expansion constructs that will be expanded using both System properties and other properties that has been set.

`-N` , `--noimport`
Stop after the BOM file has been created i.e. do not populate the workspace.

`-P «URL or path»` , `--properties «URL or path»`
The URL or file system path of a properties file. The values in the file may include ANT-style expansion constructs that will be expanded using both system properties and other properties that has been set.

`--t «template workspace»` , `--template «template workspace»`
Initialize the workspace from a template workspace prior to import.

`«url or path»`
An url/path to a CQUERY, MSPEC, or BOM file.

list preferences (lsprefs)

list preferences (lsprefs) — lists information about preferences

Synopsis

`listpreferences [ { -? | --help } ] [ «namepattern» ]`

Description

List information about preferences. If no *«namepattern»* has been stated, all preferences are listed.

Options

`-?` , `--help`
: Shows help for the command.

«namepattern»
: A regular expression pattern to select the preferences to list. If not provided, lists all preferences.

perform

perform — performs one or several component actions

Synopsis

`perform` [{ `-?` | `--help` }]
[{ `-D` | `--define` } *«key»* [=*«value»*]]
[{ `-F` | `--forced` }]
[{ `-P` | `--properties` } *«url or path»*]
[{ `-Q` | `--quiet` }]
[{ `-W` | `--maxwarnings` } *«n»*]
«component»#«action» ...

Description

Performs one or several component actions.

Options

`-?` , `--help`
: Shows help for the command.

`-D «key»[=«value»]` , `--define «key»[=«value»]`
: Defines a property as a key=value pair. The value may include ANT-style expansion constructs that will be expanded using both system properties and other properties that has been set.

`-F` , `--forced`
: Force all actions to be performed regardless of timestamps.

`-P` *«URL or path»* , `--properties` *«URL or path»*
: The URL or file system path of a properties file. The values in the file may include ANT-style expansion constructs that will be expanded using both System properties and other properties that has been set.

`-Q` , `--quiet`
: Don't print errors and warnings. Just exit with a non zero exit code on failure.

`-W` *«n»* , `--maxwarnings` *«n»*
: Give the number of warnings acceptable. If the number of warnings are higher, treat as error and exit with 1. Default is infinite warnings.

«namepattern»
: A regular expression pattern to select the preferences to list. If not provided, lists all preferences.

«component»#«action»
: The action to perform identified by the *«component»* name, and *«action»* name. Can be given multiple times in order to perform many actions in the most optimized way.

set preference (setpref)

set preference (setpref) — Sets one or several preferences.

Synopsis

`setpreference` [{ `-?` | `--help` }] *«name»=«value»* ...

Description

Sets one or several preferences.

Options

`-?`, `--help`
: Shows help for the command.

«name»
: The name of the preference to set.

«value»
: The wanted value of the preference.

unset preference (unsetpref)

unset preference (unsetpref) — unsets one or several preferences.

Synopsis

`unsetpreference` [{ `-?` | `--help` }] *«name»* ...

Description

Unsets one or several preferences.

Options

`-?`, `--help`
: Shows help for the command.

«name»
: The name of the preference to unset.

import target definition (importtarget)

import target definition (importtarget) — imports target definitions

Synopsis

```
importtargetdefinition [ { -? | --help } ]
[ { -A | --active } ]
«url or path»
```

Description

Imports a target definition into the workspace, and optionally makes it active.

Options

`-?, --help`
: Shows help for the command.

`-A, --active`
: Load the target definition (and make it the active definition) after it has been imported.

`«url or path»`
: The location of the target definition.

list target definitions (lstargets)

list target definitions (lstargets) — lists target definitions known in the workspace.

Synopsis

```
listtargetdefinitions [ { -? | --help } ]
```

Description

Lists target definitions known in the workspace.

Options

`-?, --help`
: Shows help for the command.

export preferences (exportprefs)

export preferences (exportprefs) — exports preferences settings.

Synopsis

`exportpreferences` [{ `-?` | `--help` }]
[{ `-F` | `--filename` } *«filename»*]
[{ `-S` | `--scope` } *«scope»*]
[*«rootKey»* [`#`*«subKey»* [`,`*«subKey»* ...]] ...]

Description

Exports preferences settings to a file or standard out.

Options

`-?` , `--help`
: Shows help for the command.

`-F` *«filename»* , `--filename` *«filename»*
: The file to write preferences settings to.

`-S` *«scope»* , `--scope` *«scope»*
: Determines which set of preferences to export. Valid values for *«scope»* are `configuration` (preferences set in the IDE) and `instance` (preferences set in the workspace).

«rootKey»`[#`*«subKey»*`[,`*«subKey»* `...]`
: Export only matching preferences. The match can be for a root only or a root qualified with one or several sub keys.

import preferences (importprefs)

import preferences (importprefs) — Imports preferences settings.

Synopsis

```
importpreferences [ { -? | --help } ]
[ { -F | --filename } «filename» ]
[ { -S | --scope } «scope» ]
[ «rootKey» [ #«subKey» [ ,«subKey» ...] ] ...]
```

Description

Imports preferences settings from a file or standard input.

Options

`-?` , `--help`
: Shows help for the command.

`-F «filename»` , `--filename «filename»`
: The file to read preferences from.

`-S «scope»` , `--scope «scope»`
: Determines which set of preferences to import. Valid values for *«scope»* are `configuration` (preferences set in the Eclipse IDE) and `instance` (preferences set in the workspace).

`«rootKey»[# «subKey»[,«subKey» ...]]...`
: Import only matching preferences. The match can be for a root only or a root qualified with one or several sub keys.

install

install — installs a feature.

Synopsis

`install` [{ `-?` | `--help` }] *«site»* *«feature»* [*«version»*]

Description

Installs a feature into the running headless Buckminster.

Options

`-?` , `--help`
: Shows help for the command.

«site»
: URL or path to site.

«feature»
: The id of the feature to install.

«version»
: The version to install.

list site

list site — lists features available at a site.

Synopsis

```
listsite [ { -? | --help } ] «url or path»
```

Description

Lists the features found in a local or remote repository.

Options

`-?, --help`
Shows help for the command.

`«url or path»`
The location of the repository.

uninstall

uninstall — uninstalls a feature.

Synopsis

`uninstall` [{ `-?` | `--help` }] *«feature»* [*«version»*]

Description

Uninstalls a feature.

Options

`-?` , `--help`
: Shows help for the command.

«feature»
: The id of the feature to uninstall.

«version»
: The version to uninstall. If not stated, and there are several versions installed, an error is generated.

Buckminster XML Schemas

This Reference Guide contains information about the Buckminster XML schemas, and the use of XML name spaces.

The schemas are also made available in an XML catalog at http://www.eclipse.org/buckminster/schemas/buckminster.xmlcatalog, here is the current content of this catalog:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
    <uri name="http://www.eclipse.org/buckminster/Common-1.0"
         uri="http://www.eclipse.org/buckminster/schemas/common-1.0.xsd" />
    <uri name="http://www.eclipse.org/buckminster/CQuery-1.0"
         uri="http://www.eclipse.org/buckminster/schemas/cquery-1.0.xsd" />
    <uri name="http://www.eclipse.org/buckminster/CSpec-1.0"
         uri="http://www.eclipse.org/buckminster/schemas/cspec-1.0.xsd" />
    <uri name="http://www.eclipse.org/buckminster/RMap-1.0"
         uri="http://www.eclipse.org/buckminster/schemas/rmap-1.0.xsd" />
    <uri name="http://www.eclipse.org/buckminster/MavenProvider-1.0"
         uri="http://www.eclipse.org/buckminster/schemas/maven-provider-1.0.xsd" />
    <uri name="http://www.eclipse.org/buckminster/PDEMapProvider-1.0"
         uri="http://www.eclipse.org/buckminster/schemas/map-provider-1.0.xsd" />
    <uri name="http://opml.org/spec2"
         uri="http://www.eclipse.org/buckminster/schemas/opml-2.0.xsd" />
    <uri name="http://www.w3.org/XML/1998/namespace"
         uri="http://www.eclipse.org/buckminster/schemas/xml-1998.xsd" />
    <uri name="http://www.w3.org/1999/xhtml"
         uri="http://www.eclipse.org/buckminster/schemas/xhtml-1999.xsd" />
    <uri name="http://www.w3.org/2001/XMLSchema"
         uri="http://www.eclipse.org/buckminster/schemas/XMLSchema-2001.xsd" />
</catalog>
```

bc (Common-1.0)

bc (Common-1.0) — Buckminster common XML schema.

Synopsis

Namespace Declaration	
`xmlns:bc="http://www.eclipse.org/buckminster/Common-1.0"`	
Namespace	`http://www.eclipse.org/buckminster/Common-1.0`
Prefix	**`bc`**
Schema Location	http://www.eclipse.org/buckminster/schemas/common-1.0.xsd

Description

Contains the Resource Map (RMAP) XML Schema definition.

cs (CSpec-1.0)

cs (CSpec-1.0) — Buckminster Component Specification (CSPEC), and Component Specification Extension (CSPEX) XML schemas.

Synopsis

Namespace Declaration	
`xmlns:cs="http://www.eclipse.org/buckminster/CSpec-1.0"`	
Namespace	`http://www.eclipse.org/buckminster/CSpec-1.0`
Prefix	**`cs`**
Schema Location	http://www.eclipse.org/buckminster/schemas/cspec-1.0.xsd

Description

Contains the Buckminster Component Specification (CSPEC), and Component Specification Extension (CSPEX) XML Schema definition.

cq (CQuery-1.0)

cq (CQuery-1.0) — Buckminster Component Query (CQUERY) XML schema.

Synopsis

Namespace Declaration	
`xmlns:cq="http://www.eclipse.org/buckminster/CQuery-1.0"`	
Namespace	`http://www.eclipse.org/buckminster/CQuery-1.0`
Prefix	**`cq`**
Schema Location	http://www.eclipse.org/buckminster/schemas/cquery-1.0.xsd

Description

Contains the Buckminster Component Query (CQUERY) XML Schema definition.

md (MetaData-1.0)

md (MetaData-1.0) — Buckminster XML schema for Bill of Materials (BOM), and Materialization Spec (MSPEC).

Synopsis

Namespace Declaration	
`xmlns:md="http://www.eclipse.org/buckminster/MetaData-1.0"`	
Namespace	`http://www.eclipse.org/buckminster/MetaData-1.0`
Prefix	**`md`**
Schema Location	http://www.eclipse.org/buckminster/schemas/metadata-1.0.xsd

Description

Contains the Buckminster Bill of Materials (BOM), and Materialization Specification (MSPEC) XML Schema definitions.

mp (MavenProvider-1.0)

mp (MavenProvider-1.0) — Buckminster XML schema for the Maven Provider extension to the Resource Map (RMAP).

Synopsis

Namespace Declaration	
`xmlns:mp="http://www.eclipse.org/buckminster/MavenProvider-1.0"`	
Namespace	`http://www.eclipse.org/buckminster/MavenProvider-1.0`
Prefix	
Schema Location	http://www.eclipse.org/buckminster/schemas/maven-provider-1.0.xsd

Description

Contains the Buckminster Maven Provider extension to the Resource Map (RMAP). It adds the ability to provide Maven specific information for the maven reader type in a `rm:provider` element.

opml (OPML-2)

opml (OPML-2) — The Outline Processor Markup Language Namespace

Synopsis

Namespace Declaration	
`xmlns:opml="http://www.opml.org/spec2"`	
Namespace	`http://www.opml.org/spec2`
Prefix	**`rm`**
Schema Information	http://www.opml.org/spec2

Description

The OPML-.2 (draft) specification is used to describe bookmarks, and RSS feed links.

pmp (PDEMapProvider-1.0)

pmp (PDEMapProvider-1.0) — RMAP Provider Extension XML schema for PDE extension.

Synopsis

Namespace Declaration	
`xmlns:rm="http://www.eclipse.org/buckminster/PDEMapProvider-1.0"`	
Namespace	`http://www.eclipse.org/buckminster/PDEMapProvider-1.0`
Prefix	**`pmp`**
Schema Location	http://www.eclipse.org/buckminster/schemas/map-provider-1.0.xsd

Description

Contains an extension for `PDEMapProvider` to the `Provider` element in the RMap-1.0 Schema.

rm (RMap-1.0)

rm (RMap-1.0) — Buckminster RMAPXML schema.

Synopsis

Namespace Declaration	
`xmlns:rm="http://www.eclipse.org/buckminster/RMap-1.0"`	
Namespace	`http://www.eclipse.org/buckminster/RMap-1.0`
Prefix	**`rm`**
Schema Location	http://www.eclipse.org/buckminster/schemas/rmap-1.0.xsd

Description

Contains the Resource Map (RMAP) XML Schema definition.

xh (xhtml)

xh (xhtml) — The XHTML schema is included to allow XHTML in documentation elements.

Synopsis

Namespace Declaration	
`xmlns:xh="http://www.w3.org/1999/xhtml"`	
Namespace	`http://www.w3.org/1999/xhtml`
Prefix	**`xh`**
Schema Information	http://www.w3.org/1999/xhtml/

Description

Contains the XHTML namespace. The namespace is imported in the Buckminster schemas where documentation elements are defined. This means that it is possible to directly use the XHTML tags without prefixing them with `xh:`.

xi (XMLSchema-instance)

xi (XMLSchema-instance) — Common definitions used in XML schemas such as type information.

Synopsis

Namespace Declaration	
`xmlns:xi="http://www.w3.org/2001/XMLSchema-instance"`	
Namespace	`http://www.w3.org/2001/XMLSchema-instance`
Prefix	**`xi`**
Schema Location	-

Description

Defined in the W3C XML Schema specification (Part 1 & 2)

See XML Schema Part 1: Structures Second Edition [http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/], and XML Schema Part 2: Datatypes Second Edition [http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/].

Defines, among other things, the common type specifications used in the Buckminster XML schemas such as `xi:string`.

Part V. Appendixes

Table of Contents

Appendix A. Installation

This Appendix describes how to install Buckminster for Eclipse versions 3.4 and 3.5. Note that this book describes the current version of Buckminster, so if you for some reason are using the 3.4 version, the described features, references, and examples will not always work on 3.4.

Tip

It is possible to develop for Eclipse 3.4 as a target even if you are using 3.5 and the latest Buckminster.

Buckminster comes in two different packagings — for use in the Eclipse SDK (the IDE), and for headless use.

Warning

Do NOT install the headless features into your Eclipse SDK! (There is absolutely no reason to do this, and it will cause instabilities).

Note

The latest download instructions are always found at **http://www.eclipse.org/buckminster/downloads.html**. The information in this appendix describes the instructions for the versions current in July of 2009.

Installing for Eclipse SDK

1. **Check if Buckminster is already available for install.** Buckminster is part of the Eclipse release trains and if you are using a packaged Eclipse downloaded from eclipse.org then chances are that you can install directly from the repository for the release train. Check under *Help* → *Install New Software...* for the Buckminster category. If you can't find the Buckminster category, you need to add a repository location. (You may also want to do this to get the latest updates as the release train sites are updated only infrequently).

2. **Add repository location (if needed).** Installing into the Eclipse SDK is done by adding the repository you want to install from. This is done in Eclipse 3.5 by adding the repository location either under '*Help* → *Install New Software...*' or under '*Eclipse* → *Preferences...* → *Install/Update* → *Available Software Sites*' and then selecting the wanted features under '*Help* → *Install New Software...*'.

   Please consult the Buckminster download page for an up to date list of available repositories, and alternatives such as downloading a full copy of a repository to facilitate a later local install.

   For convenience, here are the current locations — please note that these links are for use with Eclipse p2 installer, and *not* for use in a web browser:

   - **The Buckminster update site for Eclipse 3.4.x.** `http://download.eclipse.org/tools/buckminster/updates-3.4`.
   - **Buckminster update site for Eclipse 3.5 (the 'latest fixes').** http://download.eclipse.org/tools/buckminster/updates-3.5.
   - **Select features.** There are several features available. They are categorized into *core* and *optional*. Please note that you are expected to *make a choice* of what optional categories you need. Do *not* select all of them.

Warning

Buckminster's support for *Subversive* and *Subclipse* are **mutually exclusive**. Do NOT install both.

3. **Verify.** Buckminster is not highly visible in the Eclipse UI, so you may wonder if your installation was successful. You can naturally try to run one of the examples, but a quick check is to look for the menu entry *File* → *Open a Component Query...*

Installing the Headless Product

The *Headless Product* application is based on the Eclipse Runtime. This product is intended to be used when Buckminster's functionality is wanted, but without using a graphical user interface — e.g. from the command line, in automated scripting, etc. The headless application contains only the bare minimum to get a working headless command line utility. To make it useful, you must install the features *you* need into it, and the result can then be shared as necessary.

The packaging and installation is different between Eclipse 3.4 and 3.5. For instructions for Eclipse 3.4 (and earlier), please consult the Buckminster download page.

Note

The following instructions are for Eclipse 3.5 *only*.

1. **Download the director.** The (headless) director is a command line packaging of the p2 director — an installer that is a general purpose installer for software available in p2 repositories. Consult the Buckminster download page for the current address.

2. **Unpack the zip.** Unpack the zip file to a location where you want the director. Note that the director application is also used in many headless use cases — it is not just for installing the headless Buckminster, so select a location that is reachable from your current `PATH`, or update the `PATH` to include the location. (You don't have to set the path if you are just installing the headless Buckminster as you can do this from the directory where you unzipped the director).

3. **Perform the install.** You perform the installation by running the director with the following command (type everything as a single line of input):

   ```
   director -r «repo-location»
   -d «install-folder»
   -p Buckminster
   -i org.eclipse.buckminster.cmdline.product
   ```

   Where the command line option have the following meaning:

   `-r «repo-location»`
   : Replace *«repo-location»* with the URL to the headless Buckminster repository. The location is currently `http://download.eclipse.org/tools/buckminster/headless-3.5/`, but you should check on the Buckminster download page for the latest information. Alternatively, download the entire archived repository as instructed on the download page, and the use the local URI to the location where you unpacked the repository.

   `-d «install-folder»`
   : Replace *install-folder* with the folder/directory where you want the headless Buckminster installed.

   `-p Buckminster`
   : Type `-p Buckminster` literally, this is the name of the p2 profile.

`-i org.eclipse.buckminster.cmdline.product`
: Type `-i` and the entire identity literally, this is a reference to the installable unit you are installing.

4. **Install additional features (at least one is required).** The installation in the previous step installed the basic Buckminster bootstrap and command line shell, the only useful thing it can perform is to install additional features. You will probably want support for Java and PDE development, and some connectors to source repositories. You can use the director as shown in the previous step to install these features or use the just installed Buckminster (which has a simpler syntax):

   `buckminster` *`install «repository-url» «feature-id»`* [*`«version»`*]

   Where *`«repository-url»`* is the same as in the previous step, and *`«feature-id»`* is one of the features listed below. Optionally, a specific version can be installed. Here are the features you can install:

   `org.eclipse.buckminster.core.headless.feature`
   : The Core functionality — this feature is required if you want to do anything with Buckminster except installing additional features.

   `org.eclipse.buckminster.maven.feature`
   : Maven support. (In case you noticed, there is no special headless needed for maven, this is the same feature that is used with the user interface).

   `org.eclipse.buckminster.cvs.headless.feature`
   : Headless CVS support.

   `org.eclipse.buckminster.pde.headless.feature`
   : Headless PDE and JDT support. Required if you are working with Java based components.

   If you use the director to install, use '`-i` *`«feature-id»`*`.feature.group`' as the p2 IU identities for features have a '`feature.group`' suffix appended to the feature identity.

5. **Install SVN support (if required).** If you require support for Subversion (SVN), you must install this in a separate step as the required plugins have a license that is not compatible with Eclipse EPL, and they can therefore not be distributed directly from the eclipse.org repositories. Instead, Cloudsmith Inc. has made them available in a repository located at `http://download.cloudsmith.com/buckminster/external`.

   You install either support for subversive or subclipse by issuing the following command (type everything as a single line of input):

   ```
   director -r http://download.cloudsmith.com/buckminster/external
   -d «install-folder»
   -p Buckminster
   -i «svn-adapter-id»
   ```

   Where the command line option have the following meaning:

   `-r http://download...`
   : Use the literal location `http://download.eclipse.org/tools/buckminster/headless-3.5/`, but you should check on the Buckminster download page for the latest information.

   `-d` *`«install-folder»`*
   : Replace *`«install-folder»`* with the folder/directory where you have installed the headless Buckminster.

   `-p Buckminster`
   : Type `-p Buckminster` literally, this is the name of the p2 profile.

`-i` *«svn-adapter-id»*
: Type `-i` and then the identity of either the subclipse or the subversive integration feature. You should use `org.eclipse.buckminster.subclipse` for subclipse, and `org.eclipse.buckminster.subversive` for subversive.

Tip

You can prepare a file with the Buckminster install commands you want to perform, and tell the initial Buckminster to execute this file. This saves you work if you are installing the headless Buckminster on different machines. See "buckminster command" for more information about using a script.

Connectors

Buckminster can be extended to support many different types of connectors. Here are notes regarding installation for those that require more than just installing the connector.

Subversion (SVN)

There are different ways to connect to a SVN — the Buckminster connector distributed from Eclipse is not enough. Unfortunately, the various SVN clients all contain code with licenses that are not allowed for redistribution from eclipse.org. Cloudsmith Inc. provides these bundles from a special repository, and you can also get these bundles directly from the the respective publishers.

Depending on which combination of Eclipse plugins and protocols you select, and which platform you are running on, the instructions are quite different. On Windows it is particularly complicated to set up access over svn+ssh with use of certificates as windows does not have any support for this out of the box (whereas Un*x systems do).

There are currently two connectors for SVN — and you have to make a choice between *Subversion* and *Subclipse*.

Warning

Do NOT install support for both Subversive and Subclipse in the same environment!

Perforce (P4)

The Perforce (P4) connector is available directly in the repositories at eclipse.org. It can be installed without having perforce installed, but will not function unless perforce is also installed on the system. You need to consult perforce documentation regarding the installation of perforce on your system.

Note

If you have experience with P4 and have information that you think should be included in this book, please help improve this section.

Configuring Eclipse for XML Editing

Some of the Buckminster artifacts do not have specific graphical editors, and you edit the XML directly. To make this easier, you can configure Eclipse to include an XML editor and make it understand the Buckminster XML schemas. This way, you will get validation, content assist, and code completion while editing. See information in Buckminster XML Schemas regarding where to find the schemas.

Appendix B. Extending Buckminster

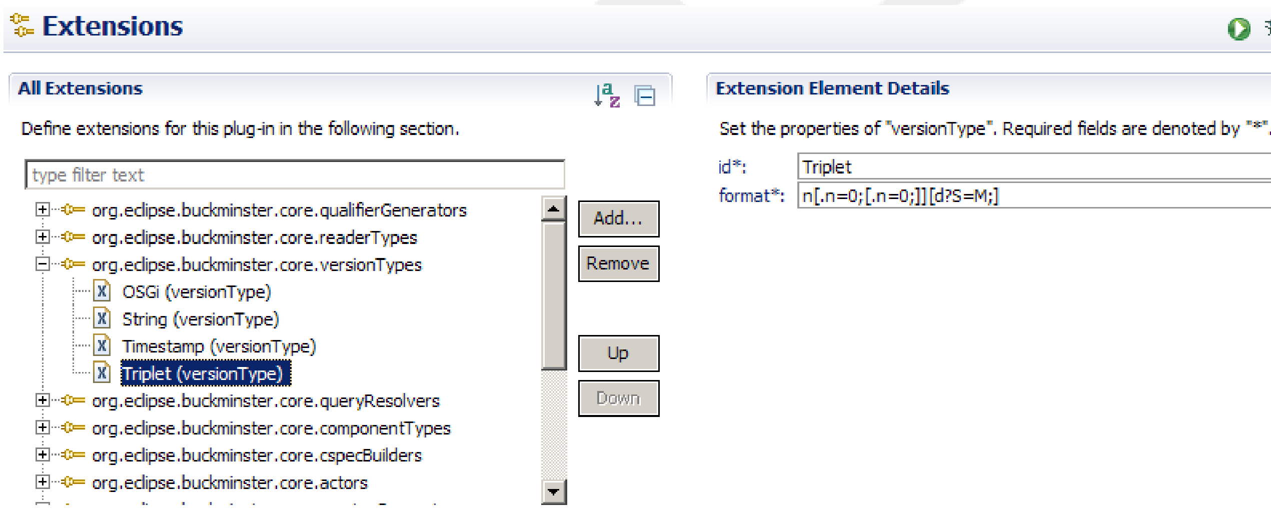
This appendix contains information how to extend Buckminster.*THIS APPENDIX IS VERY MUCH W.I.P... THE IDEA IS NOT TO SHOW HOW TO WRITE THE EXTENSIONS, BUT RATHER SHOW ALL THE POSSIBLE EXTENSIONS.*

Core extension

Version type

A version type is a named Omni Version format, as described in Appendix C, *Omni Version Details*. The named version formats are called a version type in Buckminster.

New formats can easily be included by extending org.eclipse.buckminster.core.



Define extensions for this plug-in in the following section.

type filter text

- org.eclipse.buckminster.core.qualifierGenerators
- org.eclipse.buckminster.core.readerTypes
- org.eclipse.buckminster.core.versionTypes
  - OSGi (versionType)
  - String (versionType)
  - Timestamp (versionType)
  - Triplet (versionType)
- org.eclipse.buckminster.core.queryResolvers
- org.eclipse.buckminster.core.componentTypes
- org.eclipse.buckminster.core.cspecBuilders
- org.eclipse.buckminster.core.actors

Set the properties of "versionType". Required fields are denoted by "*".

id*:	Triplet
format*:	n[.n=0;[.n=0;]][d?S=M;]

RMAP extensions

This section describes extensions that relate to mapping components.

Extending Reader Type

A reader type is the connector to a particular repository technology. The extension point is called `org.eclipse.buckminster.core.readerTypes`.

TBD.

Extending Component Type

A component type translates between meta data in some native/external form to the form used in Buckminster.

TBD.

Extending Version Converter

A version converter translates bi-directionally from internal versions to repository names (such as a branch or tag name). The mechanism can be extended to cater for more advanced mappings, or if a new types of repository connector is being added, there may be other mechanisms than branch/tag to consider.

TBD.

CQUERY Extensions

This section describes extension to the CQUERY and resolution process.

Custom resolver

Custom resolvers are added through the `org.eclipse.buckminster.core.queryResolvers` extension point. Buckminster provides reference implementations for two resolvers: ‘local’ and ‘rmap’.

Appendix C. Omni Version Details

Introduction

This appendix describes the *Omni Version* implementation handling instances of version and version ranges. The omni version implementation resides in equinox p2, and is also used in Buckminster. The omni version was created because of the need to have a version format capable of describing versions using another versioning scheme than OSGi (which was the only versioning scheme supported by p2 prior to Eclipse 3.5 and omni version).

Buckminster has always been capable of handling different versioning schemes, but did so (prior to Eclipse 3.5) using the Eclipse extension mechanism which in practice meant that it was only meaningful to make extensions in the Buckminster code base itself. This because it would not be possible for someone to parse a version if the implementation of the versioning scheme was not present.

With the omni version contribution to p2 — which fully describes a format, a canonical version comparable against versions with different formats, as well as containing the original version string, Buckminster can now use p2 for provisioning also for non OSGi based components.

Background

There are other versioning schemes in wide use that are not compatible with OSGi version and version ranges. The problem is both syntactic and semantic.

Many open source projects do their versioning in a fashion similar to OSGi but with one very significant difference. For two versions that are otherwise equal, a lack of qualifier signifies a higher version then when a qualifier is present — i.e.

```
1.0.0.alpha
1.0.0.beta
1.0.0.rc1
1.0.0
```

The 1.0.0 is the final release. The qualifier happens to be in alphabetical order here but that's not always true.

Mozilla Toolkit versioning has many rules and each segment has 4 (optional) slots; *`numeric`*, *`string`*, *`numeric`*, and *`string`* where each slot has a default value set to 0 or *max string* respectively for the numeric and string slots if a particular slot is missing).

```
1.2a3b.  // yes, a trailing . is allowed and means .0
1.a2
```

Mozilla also allows bumping the version (using an older Mozilla scheme)

```
1.0+
```

This means `1.1pre` in Mozilla.

Example of syntax issue

Here are some examples of versions used in Red Had Fedora distributions.

```
KDE Admin version 7:4.0.3-3.fc9
Compat libstdc version 33-3.2.3-63
Automake 1.4p6-15.fc7
```

And here are some Mozilla toolkit versions:

```
1.*.1
1.0+
1.-1  // negative integer version numbers are allowed, the '-' is not a delimiter
```

```
1.2a3b.a
```

These are not syntactically compatible with OSGi versions.

Implementation

The current implementation in p2 uses the omni versions throughout. This means that p2 can create a plan including units that have non OSGi versioning scheme.

One implementation

Equinox p2 has one implementation of `Version` and one of `VersionRange` that are capable of capturing the semantics of various version formats. The advantages over previous proposed implementations (like the implementation in Buckminster prior to Eclipse 3.5) are that there is no need to dynamically plugin new implementations, and new formats can be more easily be introduced.

One canonical format

The omni version and omni version range are “universal” — all instances of version should be comparable against each other with a fully defined (non ambiguous) ordering. The API is (as today) based on a single string fully describing a version or version range.

The canonical string format is called “raw” and it is explained in more detail below. To ensure backward compatibility, as well as providing ease of use in an OSGi environment, version strings that are not prefixed with the omni version keyword `raw` have the same format and semantics as the current OSGi version format.

Ad an example the following two version strings are both valid input, and express exactly the same version:

```
1.0.0.r1234
raw:1.0.0.'r1234'
```

Version

The omni version implementation uses an vector to store version-segments in order of descending significance. A segment is an instance of `Integer`, `String`, `Comparable[]`, `MaxInteger`, `MaxString`, or `Min`.

Comparison

Comparison is done by iterating over segments from 0 to n.

- If segments are of different type the rule `MaxInteger` > `Integer` > `Comparable[]` > `MaxString` > `String` is used — the comparison is done and the version with the greater segment type is reported as greater.
- If segments are of equal type — they are compared — if one is greater the comparison is done and the version with the greater segment is reported as greater.
- All versions are by default padded with `-M` (absolute min segment) to “infinity”. A version may have an explicit pad element which is used instead of the default.
- A shorter version is compared to a longer by comparing the extra segments in the longer version against the shorter version’s pad segment.
- If all segments are equal up to end of the longest segment array, the pad segments are compared, and the version with the greater pad segment is reported as greater.

- If pad segments are also equal the two versions are reported as equal.
- As a consequence of not including delimiters in the canonical format; two versions are equal if they only differ on delimiters.

As an example — here is a comparison of versions (expressed in the raw format introduced further on in the text — '`p`' means that a pad element follows, and '`-M`' the absolute min segment):

```
1p-M < 1.0.0 < 1.0.0p0 == 1p0 < 1.1 < 1.1.1 < 1p1 == 1.1p1 < 1pM
```

Raw and Original Version String

The original version can be kept when the raw version format is used, but it is not an absolute requirement as simple raw based forms such as raw:1.2.3.4.5 could certainly be used directly by humans. Someone (who for some reason does not want to use OSGi or some other known version scheme), could elect to use the raw format as their native format.

A version string with raw and original is written on the form:

```
'raw' ':' raw-format-string '/' format(...):original-format-string
```

The p2 Engine completely ignores the original part — only the raw part is used, and the original format is only used for human consumption.

Example using a Mozilla version string (as it has the most complex format encountered to date)[1].

```
raw:<1.m.0.m>.<20.'a'.3.'b'>p<0.m.0.m>
/format((<n=0;?s=m;?n=0;?s=m;?>(.<n=0;?s=m;?n=0;?s=m;?>)*)=p<0.m.0.m>;)
:1.20a3b.a
```

An original version string can be included with unknown format:

```
raw:<1.m.0.m>.<20.'a'.3.'b'>p<0.m.0.m>/:1.20a3b.a
```

See below for full explanation of the raw format.

Omni Version Range

The version range holds two version instances (lower and upper bound). A version range string uses the delimiters `[]`, `()` and `,`. If these characters are used in the lower or upper bound version strings, these occurrences must be escaped with `\` and occurrences of `\` must also be escaped.

The version range is either an OSGi version range (if raw prefix is not used), or a raw range. The format of the raw range is:

```
'raw' ':' ( '[' | '(' ) raw-format-string ',' raw-format-string ( ']' | ')' )
```

The raw-range can be followed by the original range:

```
raw-range '/' 'format' '(' format-string ')'
':' ( '[' | '(' ) original-format-string ','
original-format-string ( ']' | ')' )
```

An original version range can be included with unknown format:

```
raw: [<1.m.0.m>.<20.m.0.m>p<0.m.0.m>,
<1.m.0.m>.<20.'a'.3.'b'>p<0.m.0.m>]
/:[1.20,1.20a3b.a]
```

The p2 Engine completely ignores the original part — only the raw part is used, and the original format is only used for human consumption.

See below for full explanation of the raw format.

[1]line breaks are inserted for readability

Other range formats

Note that some version schemes have range concepts where the notion of inclusive or exclusive does not exist, and instead use symbolic markers such as "next larger", "next smaller", or use wild-cards to define ranges. In these cases, the translator of the original version string must use discrete versions and the inclusive/exclusive notation to define the same range.

Some range specifications allows the specification of union, or exclusion of certain versions. This is *not* currently supported by p2. If introduced it could be expressed as a series of ranges where a `^` before a range negates it. Example `[0,1][3,10]^[3.1,3.7)` which would be equivalent to `[0,10]^(1,3)^[3.1,3.7)`

Format Specification

There are two basic formats *default OSGi string format*, and *raw canonical string format*. There are also two corresponding range formats OSGi-version-range, and raw-version-range.

The raw format is a string representation of the internally used format — it consists of the keyword "raw", followed by a list of entries separated by period. An entry can be numerical, quoted alphanumerical, or a sub canonical list on the same format. A canonical version (and sub canonical version arrays) can be padded to infinity with a special padding element. Special entries express the notion of 'max integer' and 'max string'.

The OSGi string format is the well known format in current use.

The raw format in BNF:

```
   digit: [0-9];
   letter: [a-zA-Z];
   numeric: digit+;
   alpha: letter+;
   alpha-numeric: [0-9a-zA-Z]+;
   delimiter: [^0-9a-zA-Z];
   character: .;
   characters .+;

// A sequence of charactes quoted with " or ', where ' can
// be used in a " quoted string and vice versa
   quoted-string: ("[^"]*")|('[^']*');

// a sequence of any characters but
// with ',' ']', ')' and '\' escaped with '\'
   range-safe-string:  TBD;

   sq: ['];
   dq: ["];

   version:
      | osgi-version
      | raw-version
      ;
   osgi-version:
      | numeric
      | numeric '.' numeric
      | numeric '.' numeric '.' numeric
      | numeric '.' numeric '.' numeric '.' .+
      ;
   raw-version:
      | 'raw' ':' raw-segments optional-original-version
      ;
   optional-original-version:
      |
      | '/' original-version
      ;
   version-range:
      | osgi-version-range
```

```
            | raw-version-range
            ;
         rs: ('[' | '(');
         re: (']' | ')');

         osgi-version-range:
            | rs osgi-version ',' osgi-version re
            ;
         raw-version-range:
            | 'raw' ':' rs raw-segments ',' raw-segments re
              optional-original-range
            ;
      optional-original-range:
          |
          | '/' original-range
          ;
      raw-segments:
          | raw-elements optional-pad-element
          ;
      raw-elements:
          | raw-elements '.' raw-element
          | raw-element
          ;
      raw-element:
          | numeric
          | quoted-strings  // strings are concatenated
          | '<' raw-elements optional-pad-element '>'
                // subvector of elements
          | 'm'   // symbolic 'maxs' == max string
          | 'M'   // symbolic 'absolute max'
                // i.e. max > MAX_INT > maxs
          | '-M   // symbolic 'absolute min'
                // i.e. -M <  empty string < array <  int
          ;
      optional-pad-element:
          |
          | pad-element
          ;
      quoted-strings:
          | quoted-strings quoted-string
          | quoted-string
          ;
      pad-element:
          | 'p' raw-element
          ;
      original-version:
          | optional-format-definition ':' .*
          ;
      original-range:
          | optional-format-definition ':' rs range-safe-string
            ',' range-safe-string re
          ;
      optional-format-definition:
          |
          | format-definition
          ;
      format-definition:
          | 'format' '(' pattern ')'
          ;

      // Definition of parsing patterns
      //
      pattern:
          | pattern pattern-element
          | pattern-element
          ;
      pattern-element:
          | pelem optional-processing-rules optional-pattern-range
          | '[' pattern ']' processing-rules
          ;
      optional-processing-rules:
          | optional- processing-rules '=' processing-rule ';'
```

```
    | '=' processing-rule ';'
    |
    ;
optional-pattern-range:
    | repeat-range
    |
    ;

pelem
    | 'r' | 'd' | 'p' | 'a' | 's' | 'S' |  'n' | 'N' | 'q'
    | '(' pattern ')'
    | '<' pattern '>'
    | delimiter
    ;
repeat-range:
    | '?' | '*' | '+'
    | '{' exact '}'
    | '{' at-least ',' '}'
    | '{' at-least ',' at-most '}'
    ;

exact: at-least: at-most: numeric;

processing-rule:
    | raw-element
    | pad-element
    | '!'
    | '[' char-list ']'
    | '[' '^' char-list ']'
    | '{' exact '}'   // for character count
    | '{' at-least ',' '}'
    | '{' at-least ',' at-most '}'
    ;
char-list: TBD; // Sequence of any character but
                // with '^', ']' and '\' escaped with '\'
delimiter:
    | [!#$%&/=^,.;:-_ ] // Any non-alpha-num that
                        // has no special meaning
    | quoted-string
    | '\' .  // any escaped character
    ;
```

Examples:

- OSGi `1.0.0.r1234` is expressed as `raw:1.0.0.'r1234'`
- apache/triplet style `1.2.3` is expressed as `raw:1.2.3.m`
- Mozilla style `1a.2a3c.` can be expressed as

  `raw:<1.'a'.0.m>.<2.'a'.3.'c'>p<0.m.0.m>`

  Mozilla's format is complex — see external links at the end of this appendix, for more information.

Format Pattern Explanation

Here are explanations for the rules in format(pattern).

rule	description
`r`	*raw* — matches one *raw-element* as specified by the `raw` format. The `r` rule does not match a pad element — use `p` for this.
`'characters'`	*quoted delimiter* — matches one or several characters — the matched result is not included in the resulting canonical vector (i.e. it is not a segment). A `\\` is needed to include a single `\`. The sequence of chars acts as one delimiter.
`non-alphanum character`	*literal delimiter* — matches any non alpha-numerical character (including space) — the matched result is not included in the canonical vector (i.e. it is not

rule	description
	a segment). A non alphanumerical character acts as a delimiter. Special characters must be escaped when wanted as delimiters.
`a`	*auto* — a sequence of digits creates a numeric segment, a sequence of alphabetical characters creates a string segment. Segments are delimited by any character not having the same character class as the first character in the sequence, or by the following delimiter. A numerical sequence ignores leading zeros.
`d`	*delimiter* — matches any non alpha-numeric character. The matched result is not included in the resulting canonical vector (i.e. it is not a segment).
`s`	*letter-string* — a string group matching only alpha characters (i.e. “letters”). Use processing rules `=[];` or `=[^]` to define the set of allowed characters. It is possible to allow inclusion of delimiter chars, but not inclusion of digits.
`S`	*string* — a string group matching any group of characters. Use processing rules `=[];` or `=[^]` to define the set of allowed characters. Care must be taken to specify exclusion of a delimiter if elements are to follow the `S`.
`n`	a *numeric* (integer) group with value >= 0. Leading zeros are ignored.
`N`	a possibly *negative value numeric* (integer) group. Leading zeros are ignored.
`p`	parses an explicit *pad-element* in the input string as defined by the raw format. To define an implicit pad as part of the pattern use the processing instruction `=p...;`. A pad element can only be last in the overall version string, or last in a sub array.
`q`	*smart quoted string* — matches a quoted alphanumeric string where the quote is determined by the first character of the string segment. The quote must be a non alphanumeric character, and the string must be delimited by the same character except brackets and parenthesises (i.e. `()`, `{}`, `[]`, `<>`) which are handled as pairs, thus `q` matches `<andrea-doria>` and produces a single string segment with the text `andrea-doria`. A non-quoted sequence of characters are not matched by `q`.
`()`	indicates a group
`< >`	*array* — indicates a group, where the resulting elements of the group is placed in an array, and the array is one resulting element in the enclosing result
`?`	*zero to one* occurrence of the preceding rule
`*`	*zero to many* occurrences of the preceding rule
`+`	*one to many* occurrences of the preceding rule
`{` *n* `}`	*exactly n* occurrences of the preceding rule
`{` *n* `,}`	*at least n* occurrences of the preceding rule
`{` *n* `,` *m* `}`	*at least n* occurrences of the preceding rule, but not more than m times
`[ ]`	short hand notation for an *optional group*. Is equivalent to `()?`
`=` *processing* `;`	an additional *processing rule* is applied to the preceding rule. The *processing* part can be: • a *raw-element* - use this *raw-element* (as defined by the raw format) as the default value if input is missing. The default value does not have to be of the same type (e.g. `s=123;?` produces an integer segment of value `123` if the optional `s` is not matched. • `!` — if input is present do not turn it into a segment (i.e. ignore what was matched) • `[`*list of chars*`]` — when applied to a `d` defines the set of delimiters. The characters `]`, `^`, and `\` must be escaped with `\` to be used in the list of chars.

rule	description
	and Example `d=[+-/];` One or several ranges of characters such as `a-z` can also be used. Example `d=[a-zA-Z0-9_-];` • `[^`*`list of chars`*`]` — when applied to a `d` defines the set of delimiters to be all non alpha numeric except the listed characters. The characters `]`, `^`, and `\` must be escaped with `\` to be used in the list of chars. One or several ranges of characters such as `a-z` can also be used. Example `d=[^$];` • `p`*`raw-element`* — defines “padding to infinity with specified raw-element” when applied to an array, or a group enclosing the entire format. Example `format((n.s)=pM;)` The pad processing rule is only applied to a parsed array, not to a default value for an array. If padding is wanted in the default array value, it can be expressed explicitly in the default value. • `{`*`n`*`} {`*`n`*`,} {`*`n,m`*`}` character ranges — with the same meaning as the rules with the same syntax, but limits the range in characters matched in the preceding `s`, `S`, `n`, `N`, `q`, or `a` rules. For `q` the quotes does not count.
`\`	*escape* removes the special meaning of a character and must be used if a special character is wanted as a delimiter. A `\\` is needed to include a single `\`. Escaping a non special character is superfluous but allowed.

Additional rules:

- if a rule produces a null segment, it is not placed in the result vector

  e.g. `format(ndddn):10-/-12` → `raw:10.12`

- Processing (i.e. default values) applied to a group has higher precedence than individual processing inside the group if the entire group was not successfully matched.
- Parsing is greedy — `format(n(.n)*(.s)*)` will interpret `1.2.3.hello` as `raw:1.2.3.'hello'` (as opposed to being reluctant which would produce `raw:1.'2'.'3'.'hello'`)
- When combining `N` with `={...};` and the input has a negative number, the ‘`-`’ character is not included in the count — `format(N{3}N{2}):-1234` results in `raw:-123.4`
- When combining `n` or `N` with `={...}` and input has leading zeros — these are included in the character count.
- An empty version strings is always considered to be an error.
- A format that produces no segments is always considered to be an error.

Note about white space in the raw format:

- white space is accepted inside quoted strings — i.e. `1.'a string'` is allowed, but not `1. 2`
- white space is accepted between version range delimiters and version strings

  i.e. `[ 1.0, 2.0 ]` is allowed.

Note about timestamps Versions based on a timestamp should use `s` or `n` and ensure comparability by using a fixed number of characters when choosing `s` format.

Examples of Version Formats

Here are examples of various version formats expressed as using the format pattern notation.

type name	pattern	comment
osgi	n[.n=0;[.n=0;[.S=[a-zA-Z0-9_-];]]]	Example: the following are equivalent: • format(n[.n=0;[.n=0;[.S=[a-zA-Z0-9_-];]]]):1.0.0.r1234 • raw:1.0.0.'r1234' • osgi:1.0.0.r1234 • 1.0.0.r1234
triplet	n[.n=0;[.n=0;[.S=m;]]]	A variation on OSGi, with the same syntax, but where the a lack of qualifier > any qualifier, and the qualifier may contain any character. The following are all equivalent: • format(n[.n=0;[.n=0;[.S=m;]]]):1.0.0 • raw:1.0.0.M • triplet:1.0.0
jsr277	n(.n=0;){0,3}[-S=m;]	As defined by JSR 277 — but is provisional and subject to change as it is expected that compatibility with OSGi will be solved (they are now incompatible because of the fourth numeric field with default value 0). The jsr277 format is similar to triplet, but with 4 numeric segments and a '-' separating the qualifier to allow input of "1-qualifier" to mean "1.0.0.0-qualifier". As in triplet the a lack of qualifier > any qualifier. The following are all equivalent: • format(n(.n=0;){1,3}[-S=m;]):1.0.0 • raw:1.0.0.0.M • jsr277:1.0.0
tripletSnapshot	n[.n=0;[.n=0;[-n=M;.S=m;]]]	Format used when maven transforms versions like 1.2.3-SNAPSHOT into 1.2.3-<buildnumber>.<timestamp> ensuring that it is compatible with triplet format if missing <buildnumber>.<timestamp> at the end (format produces max, max-string if they are missing). Example: the following are equivalent: • format(n[.n=0;[.n=0;[-n=M;.S=m;]]]):1.2.3-45.20081213:1233 • raw:1.2.3.45.'20081213:1233'

type name	pattern	comment
		• tripletSnapshot:1.2.3-45.20081213:1233
rpm	<[n:]a(d?a)*>[-n[dS=!;]]	RPM format matches [EPOCH:]VERSION-STRING[-PACKAGE-VERSION], where epoch is optional and numeric, version-string is auto matched to arbitrary depth >= 1, followed by a package-version, which consists of a build number separated by any separator from trailing platform specification, or the string 'src' to indicate that the package is a source package. This format allows the platform and src part to be included in the version string, but if present it is not used in the comparisons. The platform type vs source is expected to be encoded elsewhere in such an IU. Everything except the build-number is placed in an array as build number is only compared if there is a tie. An example of equivalent expressions: • format(<[n:]a(d?a)*>[-n[dS=!;]]):33:1.2.3a-23/i386 • raw:<33.1.2.3.'a'>.23
mozilla	(<n=0;?s=m;?n=0;?s=m;?>(.<n=0;?s=m;?n=0;?s=m;?>)*)=p<0.m.0.m>;	Mozilla versions are somewhat complicated, it consists of 1 or more parts separated by period. Each part consists of 4 optional 'fragments' (numeric, string, numeric,string), where numeric fragments are 0 if missing, and string fragments are MAX-STRING if missing. The versions use padding so that 1 == 1.0 == 1.0.0 == 1.0.0.0 etc.
string	S	a single string
auto	a(d?a)*	serves like a "catch all".

Tooling Support

The omni version implementation is not designed to be extended. An earlier idea was that it should be possible to define named aliases for common formats and that these formats should be parseable by the omni version parser. The reasons for introducing alias was to make it possible for users to enter something like `triplet:1.0.0` instead of entering the more complicated format. This did however raise a lot of questions: Who can define an alias, what if the definition of the alias is changed, where are the alias definitions found. Is it possible to work at all with a version that is using only an alias — what if I want to modify a range and do not have access to the alias?

Instead, the alias handling is a tooling concern. Tooling should keep a registry of known formats. When a version is to be presented, the format string is "reverse looked up" in the registry — and the alias name can be presented instead of the actual format. This way, the version is always self describing. There is still the need to get "well known formats" and make them available in order to make it easier to use non OSGi versions in publishing tools — but there is no absolute requirement to support this in all publishing tools (some may even operate in a domain where version format is implied by the domain) — and there is no “breakage” because an alias is missing.

Tooling support can be as simple as just having preferences where formats are associated with names — the user can enter new formats and aliases. Some import mechanism is probably also nice to have. Further ideas could be that aliases can be published as IU’s and installed (i.e install a preference).

Existing Tooling should naturally use the new omni version implementation to parse strings — thus enabling a user to enter a version in raw or format() form. An implementation can choose to present the full version string (i.e. `Version.toString()`), or only the original version.

More examples using ‘format’

A version range with format equivalent to OSGi

```
format(n[.n=0;[.n=0;[.S=[a-zA-Z0-9_-];]]])
:[1.0.0.r12345, 2.0.0]
```

At least one string, and max 5 strings

```
format(S=[^.][.S=[^.];[.S=[^.][.S=[^.][.S=[^.]]]]]])
:vivaldi.opus.spring.bar5
```

```
format(S=[^.](.S=[^.]){0,4}):vivaldi.opus.spring.bar5
=> 'vivaldi'.'opus'.'spring'.'bar5'
```

At least one alpha or numerical with auto format and delimiter

```
format(a(d?a)*):vivaldi:opus23-spring.bar5
=> 'vivaldi'.'opus'.23.'spring'.'bar'.5
```

The texts ‘opus’ and ‘bar’ should not be included:

```
format(s[.'opus'n[.'bar'n]]):vivaldi.opus23.bar8
=> 'vivaldi'.23.8
```

The first string segment should be ignored — it is a marketing name:

```
format(s=!;.n(.n)*):vivaldi.1.5.3
```

Classic SCCS/RCS style:

```
format(n(.n)*):1.1.1.1.1.1.1.4.5.6.7.8
```

Max depth 8 of numerical segments (limited classic SCCS/RCS type versions):

```
format(n(.n){0,7}):1.1.1.1.1.1.1.4
```

Numeric to optional depth 8, where missing input is set to 0, followed by optional string where ‘empty > any’

```
format(n(d?n=0;){0,7}[a=M;]):1.1.1.4:beta
=> 1.1.1.4.0.0.0.0.'beta'
```

```
format(n(d?n=0;){0,7}[a=M;]):1.1.1.4
=> 1.1.1.4.0.0.0.0.M
```

Single string range

```
format(S):[andrea doria,titanic]
```

Range examples

Examples:

- `raw:[1.2.3.'r1234',2.0.0]`

- `[1.2.3.r1234,2.0.0]`
- `format(a+):[monkey.fred.ate.5.bananas,monkey.fred.ate.10.oranges]`
- `[1.0.0,2.0.0]` equal to `osgi:[1.0.0,2.0.0]`
- `format(S):[andrea doria,titanic]`
- `rpm:[7:4.0.3-3.fc9,8:1]` - an example KDE Admin version `7:4.0.3-3.fc9` to `8:1`
- triplet:[1.0.0.RC1,1.0.0]

FAQ

Is internationalization supported? Alphanumerical segments use vanilla string comparison as internationalization (lexical ordering/collation) would produce different results for different users.

Are users just using Eclipse and OSGi bundles affected? No, users that only deal within the OSGi domain can continue to use version strings like before, there is no need to specify version formats.

How does a user of something know which version type to use? This seems very complicated... To use some non-OSGi component with p2, that component must have been made available in a p2 repository. When it was made available, the publisher must have made it available with a specified version format. The publisher must understand the component‘s version semantics. A consumer that only wants to install the component does not really need to understand the format, and the original version string is probably sufficient. In scenarios where the consumer needs to know more — what to present is domain specific — some tool could show all non OSGi version strings as “non-OSGi” or “formatted” with drill down into the actual pattern (or if there is an alias registry available, it could reverse lookup the format).

Will open (OSGi) ranges produce lots of false positives? Very unlikely. One decision to minimize the risk was to specify that integer segments are considered to be later than array and string segments. (We also felt that version segments specified with integers are more precise). Note that to be included in the range, the required capability would still need to be in a matching name space, and have a matching name. To introduce a false positive, the publisher of the false positive would need to a) publish something already known to others (namespace and name) b) misinterpret how its versioning scheme works, and publishing it with a format of n.n.n.n (or n.n.n.s.<something>), c) having first learned how to actually specify such a format and how to publish it to a p2 repository and d) then persuaded users to use the repository.

What happens when a capability is available with several versioning schemes? A typical case would be some java package that is versioned at the source using triplet notation, and the same package is also made available using OSGi notation (which btw. is a mistake).

As an example, the following capabilities are found:

- org.demo.ships triplet:2.0.0
- org.demo.ships triplet:2.0.0.RC1
- org.demo.ships osgi:2.0.0
- org.demo.ships osgi:2.0.0.RC1

(Reminder: in triplet notation 2.0.0.RC1 is *older* than 2.0.0).

The raw versions will then look like this:

- `2.0.0.m`

- `2.0.0.'RC1'`
- `2.0.0`
- `2.0.0.'RC1'`

And the newest is 2.0.0.m (which is correct for both OSGi, and triplet). When specifying a range, the outcome may depend on if the range is specified with osgi or triplet notation.

- osgi:[1.0.0,2.0.0] == raw:[1.0.0, 2.0.0] => matches the osgi:2.0.0 version only
- triplet:[1.0.0,2.0.0] == raw:[1.0.0.m,2.0.0.m] => matches all the versions, and picks 2.0.0.m as it is the latest.

i.e. result is correct (assuming the bits are identical as different artifacts would be picked)

Now look at the lower boundary, and assume that the following versions are the (only) available:

- org.demo.ships triplet: 1.0.0 == raw: 1.0.0.m
- org.demo.ships triplet: 1.0.0.RC1 == raw:1.0.0.'RC1'
- org.demo.ships osgi: 1.0.0 == raw:1.0.0
- org.demo.ships osgi:1.0.0.RC1 == raw:1.0.0.'RC1'

When specifying ranges:

- osgi:[1.0.0,2.0.0] == raw:[1.0.0, 2.0.0] => matches all the version, and picks 1.0.0.maxs as this is the newest
- triplet:[1.0.0,2.0.0] == raw:[1.0.0.m,2.0.0.m] results in 1.0.0.m as it is the only available version that matches.

i.e. the result is correct and here the exact same version is picked.

The “worst OSGi/triplet crime” that can be committed is publishing an unqualified triplet version as an OSGi version (if the same version is not also available as a triplet) as this would make that version older than what it is even when queried using a triplet range.

What if the publisher of a component changes versioning scheme — what happens to ranges? The order among the versions will be correct as long as the versions are published using the correct notation. The only implication is that users must understand that a query for triplet:1.2.3 means raw:1.2.3.m — e.g. osgi:[1.0.0,2.0.0] != triplet:[1.0.0,2.0.0] (OSGi upper range of 2.0.0 would not match triplet published 2.0.0, and triplet lower range of 1.0.0 would not match OSGi published 1.0.0).

Why not use regexp instead of the special pattern format? This was first considered, and would certainly work if the pattern notation was augmented with processing instructions, or if the regexp is specified as a substitution that produces the raw format. Such specifications would typically be much longer and more difficult for humans to read than the proposed format, except possibly for regexp experts :). Another immediate problem is that regexp breaks the current API requirement. It is not included in execution environment `CDC-1.1/Foundation-1.1` required by p2.

Pattern parsing looks like it could have performance implications — what are the expectations here?A mechanism similar to regular expressions is used — when a format is first seen it is compiled to an internal structure. The compiled structure is cached and reused for all subsequent occurrences of the same format. Once parsed, all comparisons are made using the raw vector, which is comparable in speed to the current implementation (in many cases it is faster).

Also note that the Engine does not have to parse and apply the format to the original string unless code explicitly asks for it, and this is not the normal case during provisioning.

Why not just let the publisher deal with transforming the version into canonical form? The proposal allows this — the publisher is not required to make the format available. We think this is reasonable in domains where humans are not involved in the authoring (or the consumption).

There are several reasons why it is a good idea to include the original version string as well as the format:

- the original version strings needs to be kept as users would probably not understand the canonical representation in many cases.
- if the transformation pattern is not available a user would not be able to create a request without hand coding the canonical form
- making the transformation logic used by one publisher available to others would mean that all publishers must have extensions that allow plugging in such logic, and the plugins must be made available

Would it be possible to use the OSGi implementation of version as the canonical form? The long answer is: To be general, the encoding would need to be made in the qualifier string part of the OSGi version. An upper length for segments must be imposed, numerical sections must be left padded with "0" to that length, and string segments must be right padded with space (else string segment parts may overlap integer segments parts). The selected segment length would need to be big enough to allow the longest anticipated string segment. A fixed length string representation of MAX must be invented. A different implementation would still be needed to be able to keep the original version strings. The short answer is: no (and this is the reason for implementing the omni version in the first place).

Why not use an escape in string segments to be able to have strings with a mix of quotes? There are several reasons:

- this would mean that the version string would need to be preprocessed as it would not have \ embedded from the start
- all version strings that use \ as a delimiter would need to be pre-processed to escape the \
- to date, we [...the authors of this proposal] have not seen a version format that requires a mix of quotes
- In the unlikely event that such strings are present it is possible to concatenate several strings in the raw format.
- parsing performance is affected

Which format should I use? If you have the opportunity to select a versioning scheme — stick with OSGi.

Resources

- mozilla toolkit version format [https://developer.mozilla.org/En/Toolkit_version_format]
- rpm version comparison [http://linux.duke.edu/~mstenner/docs/rpm-version-cmp]
- sun spec version format [http://java.sun.com/j2se/1.5.0/docs/guide/deployment/deployment-guide/version-format.html]

Appendix D. Bookmarks and OPML

Bookmarks

A note about bookmarks

Buckminster supports including bookmarks containing information about web pages and RSS feeds in the component meta data since Eclipse 3.4. The mechanism is based on placing a special `buckminster.opml` file inside a component. Although still supported, our current recommendation is to only use this mechanism in components devised for building and publishing purposes.

Background. There is usually a lot of additional “project” information around a component, and some early attempts were made to create a well defined model. We soon realized that “additional interesting information” was difficult to capture with full semantics — what first looked as a simple exercise, «Let’s see, there is usually a wiki, and a bug issue tracking system, and a home page, and release information, and news, and..., and... » — the list just got longer and longer, then got turned around to «Suppose we did create a model with full semantics — what is Buckminster actually supposed to do with it? It is after all intended for human consumption.»

We still wanted to make it possible to share content that relates to a component, such as a links to wiki, Bugzilla, and documentation, to RSS feeds like a feed with information about new versions, open and closed issues in Bugzilla, examples feed, a “checkin feed”, or a feed with the latest available plugins, etc., but we wanted a more relaxed model, and dropped the semantic requirement, leaving the semantics of the information to the human users.

We selected the OPML 2.0 XML definition for *Outline Processing* — essentially describing a bookmark structure of links and feeds — because of its simplicity and that it is directly supported in some RSS readers.

This means that a component type extension automatically can generate meta data from components that have extra information for a community of users, and make this available in the form of OPML. Buckminster recognizes that any component can have an OPML file embedded in the component (by default it is called `buckminster.opml` and placed in the root of the component).

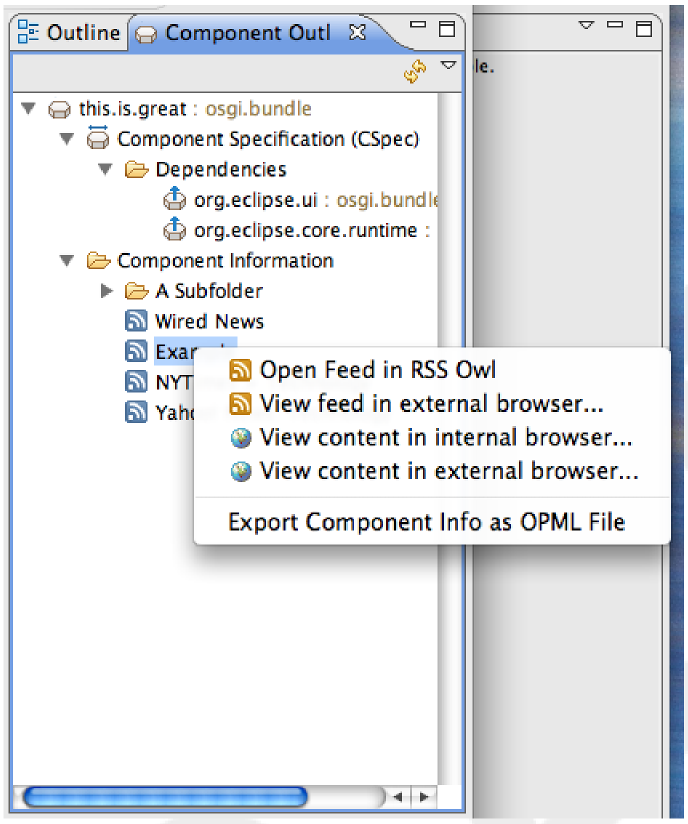
Warning — Do not use bookmarks in regular components

After having created this solution and started using it, we have found issues with its use, and now recommend that bookmarks should not be used inside regular components. The reasons for this are that it is hard to maintain; if there is a need to update the bookmarks for a published component, you need to go back to the source and create a new version, and bookmarks are typically authored late, and are typically not authored by the same people that developed the components. Some organizations also have problems with external links as strict security policies may prevent users within an organization to visit unauthorized sites — the embedding of feeds and bookmarks becomes a problem as these have to be filtered out. There is absolutely no harm however in including bookmarks in components used specifically for building (a component where you keep RMAPs, CQUERYs etc.) as an information sharing mechanism.

Future. The Buckminster project’s intention is to create a more flexible mechanism where anyone can associate bookmarks with components but using a less intrusive mechanism.

How bookmarks are presented. Buckminster includes two views that makes use of the OPML feature; the *Component Explorer*that shows all components known to Buckminster, and a *Component Outline* that follows the current selection showing the related component information. Both views present the OPML bookmark information and provides navigation to the links and feeds. There is also a Buckminster extension point that allows Buckminster to be integrated with a RSS reader.

Here is a screenshot of the Component Outline, (which also shows an integration between Buckminster and the RSS Owl Reader).


Outline
Component Outl
this.is.great : osgi.bundle
Component Specification (CSpec)
Dependencies
org.eclipse.ui : osgi.bundle
org.eclipse.core.runtime :
Component Information
A Subfolder
Wired News
Exar
NYT
Yahc
Open Feed in RSS Owl
View feed in external browser...
View content in internal browser...
View content in external browser...
Export Component Info as OPML File

Authoring OPML

Warning

Before adding bookmarks to a regular component, please see the previous section regarding recommendations.

It is easy to include RSS feeds and links in your components — all you have to do is to place a `buckminster.opml` file in the root of your component.

The OPML file itself has a very simple XML syntax. You start of with the standard declaration that the file is an XML file, and then declaring that you are using OPML 2.0.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<opml version="2.0">
```

This is followed by a *head declaration* where some information about the content of the OPML file is kept.

```
<head>
    <title>Component information for org.demo.exampleComponent</title>
    <dateCreated>Mon, 14 Apr 2008 14:18:51 GMT</dateCreated>
    <ownerName>Your name, or name of project</ownerName>
    <ownerEmail>contact.us@somewhere.com</ownerEmail>
</head>
```

And after the *head* comes the *body* part that consists of a sequence of outline elements — they can be nested if you want to use “subfolders”.

```
<body>
    <!-- outline elements --!>
</body>
```

A *regular link* (non feed) is expressed in an outline element like this:

```
<outline text="Cloudsmith"
    description="Cloudsmith's site"
    url="http://www.cloudsmith.com"
    language="unknown"
    title="Cloudsmith"
    type="link"
/>
```

And a *feed* is just as simple:

```
<outline
    text="My Feed"
    description="This is my feed"
    htmlUrl="http://www.somewhere.org"
    language="unknown"
    title="My Example Feed"
    type="rss"
    version="RSS2"
    xmlUrl="http://feeds.somewhere.org/examplefeed"
/>
```

In both of the above examples — the '`text`' attribute is the label typically used in the RSS reader's bookmarks.

Note

Some OPML viewers use the `title` instead, and the reader may or may not show the `description`.

Regular links (`type="link"`) should use the `url` attribute for the link, and a *feed* (`type="rss"`) should use the `xmlUrl` attribute for the feed, but also add a link to a human readable web page in the `htmlUrl` attribute, and this is often a link to the page where it is possible to subscribe to the feed, or read its content online.

In the feed example above, the *feed type* is set to '`rss`' — and the *version* is set to '`RSS2`'. A feed should always have the type set to `rss` (including *atom feeds*, but for atom feeds, the OPML specification is vague. In practice, a RSS reader will figure things out on it's own, and a version of '`atom`' works just fine. The *feed type* and *version* are mainly indication for a *processor* of the OPML itself, a *feed reader* will look at the actual feed to determine its type anyway.

You can read more about the OPML 2.0 standard at http://www.opml.org/spec2.

And then finally, a *subfolder* is very simple to create:

```
<outline text="A Subfolder">
    <outline .... />
    <outline .... />
</outline>
```

Even if the creation of a component's OPML is done via manual XML editing, we hope the examples above show that it is really quite easy.

Colophon

How to print this book. This book was produced by using the following specifications and tools:

- DocBook 4.5 schema
- Serna 4.1 free — for editing
- Apache FOP 0.95 — for producing PDF output
- Doc Book XSLT style sheets 1.75.1

Parameter settings are required to set the font size for monospaced verbatim areas as code examples otherwise would be truncated. A size of 8pt is required.

Parameters also needs to be set to produce PDF “bookmarks” (i.e. a PDF TOC). This is done on the command line as a directive to `xsltproc`.

Tools used. This book was authored by using the following tools:

- Serna 4.1 free — for DocBook editing
- InkScape 0.46 — for vector graphics
- LineForm 1.5 — for vector graphics
- graphviz 2.24 — for graph generation