Shipping a 100% Pure Java IDE
-
Inside the IBM VisualAge MicroEdition IDE

Erich Gamma

erich_gamma@oti.com

Object Technology International Inc.

Contents

1. Issues with the Java platform
2. A Simple Widget Kit that enables Java Applications with a native look and feel and performance
3. Designing a lite UI framework
4. Evolving the UI framework
5. Making change your friend

Java well established on the server (servlets, EJB etc.) but…

How is Java doing on the desktop?

I have been using the Forte/Netbeans IDEs. They both have major performance issues. I have seen similar problems with other large swing apps having huge memory footprints and being dog-slow at everything. Where are the bottlenecks? Is the JVM just a big memory eater, or could it be swing that is at fault?"

--B. Madigan, member, programming theory & practice
`http://www.javaworld.Com/`

"… And from the user perspective, Java programs often turned out to be annoyingly slow and unstable."

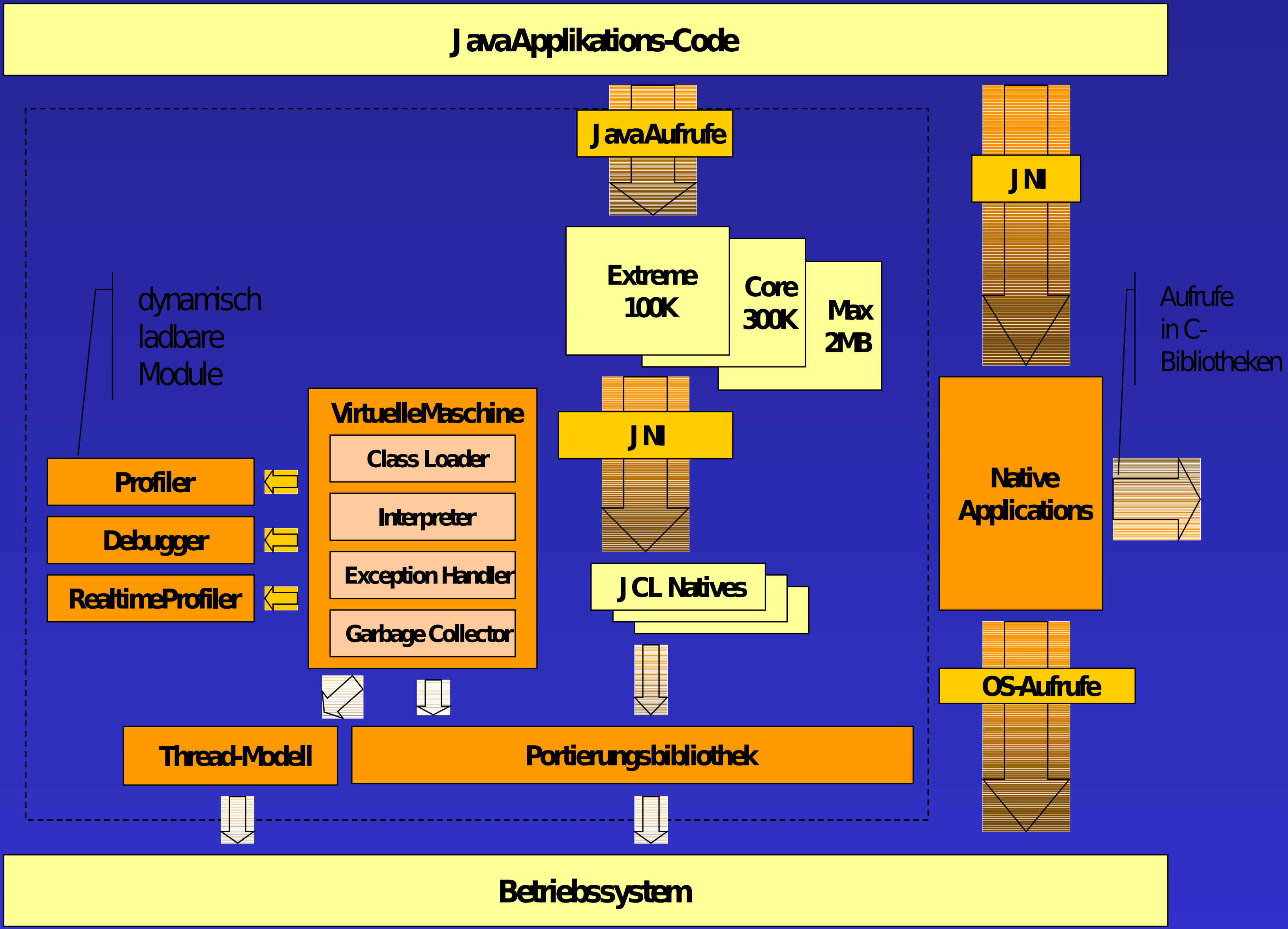
"With so many problems to contend with, Java has made little headway on the desktop. That trend is clearest – and, to me, most disappointing – in the area of standard productivity software"

-- Henry Norr, SF Gate Business & Finance

Background: VisualAge MicroEdition IDE

- IDE developed with focus on embedded systems development
- Classical features
  - browsing, cross referencing
  - incremental development
  - tightly integrated team support
    - “team streams”
- Embedded features
  - remote on target debugging
    - JPDA based
  - smart linking/application compression
  - performance tools
    - MicroAnalyzer

VAME Runtime



Java Applikations-Code

Java Aufrufe

Extreme 100K

Core 300K

Max 2MB

JNI

JCL Natives

dynamisch ladbare Module

Virtuelle Maschine

- Class Loader
- Interpreter
- Exception Handler
- Garbage Collector

Profiler

Debugger

Realtime Profiler

Thread-Modell

Portierungsbibliothek

JNI

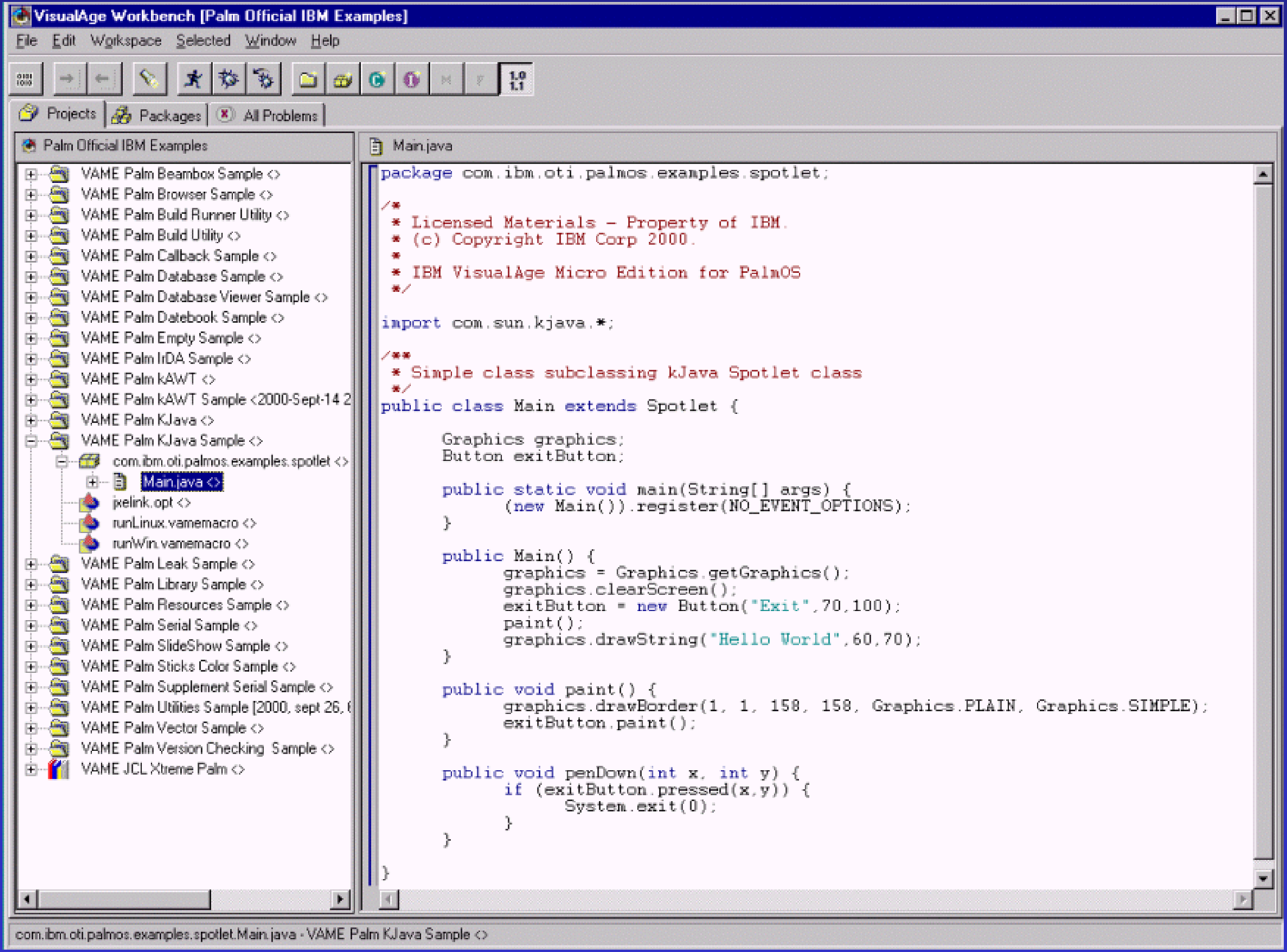
Native Applications

Aufrufe in C-Bibliotheken

OS-Aufrufe

Betriebssystem

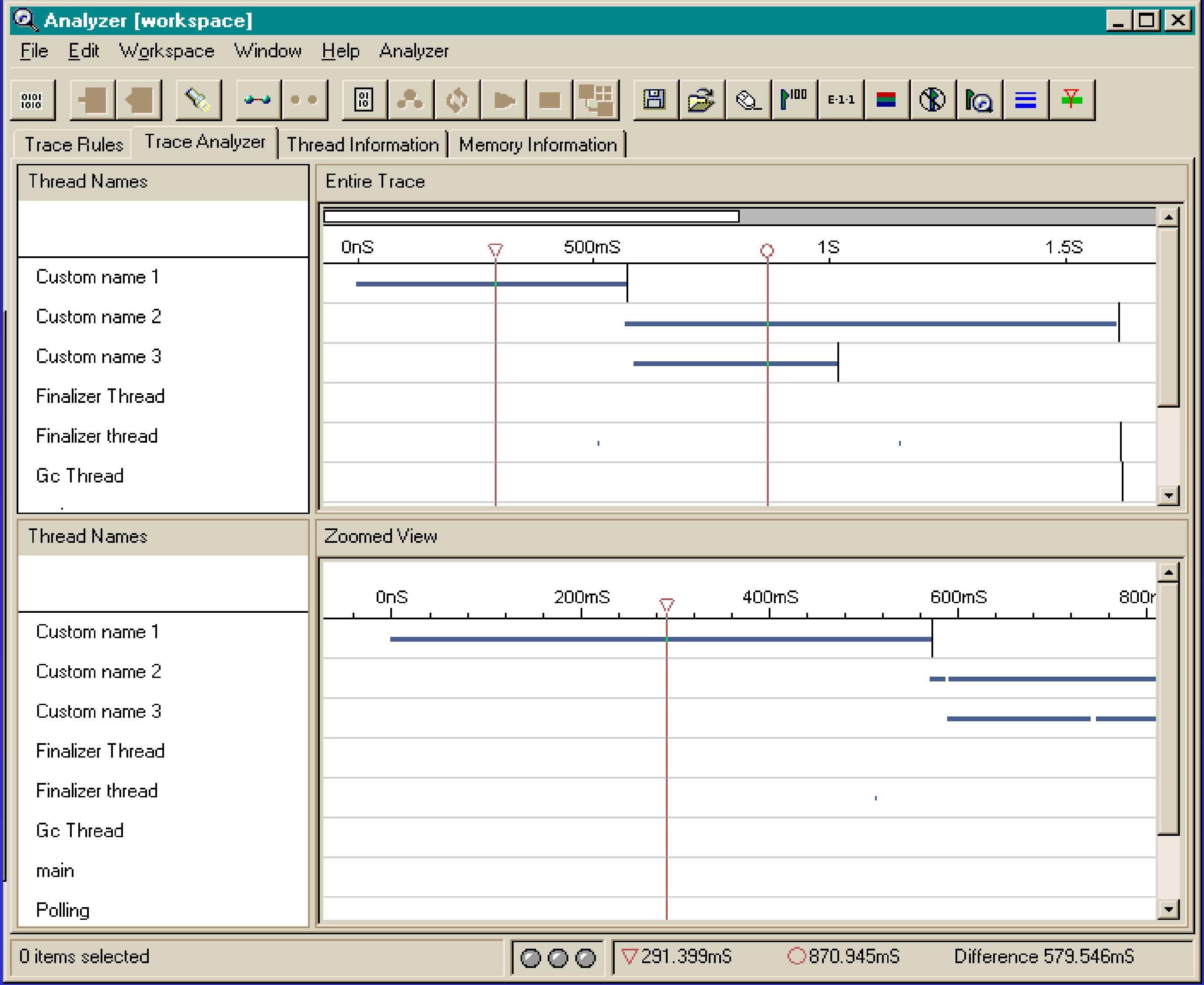
VAME IDE

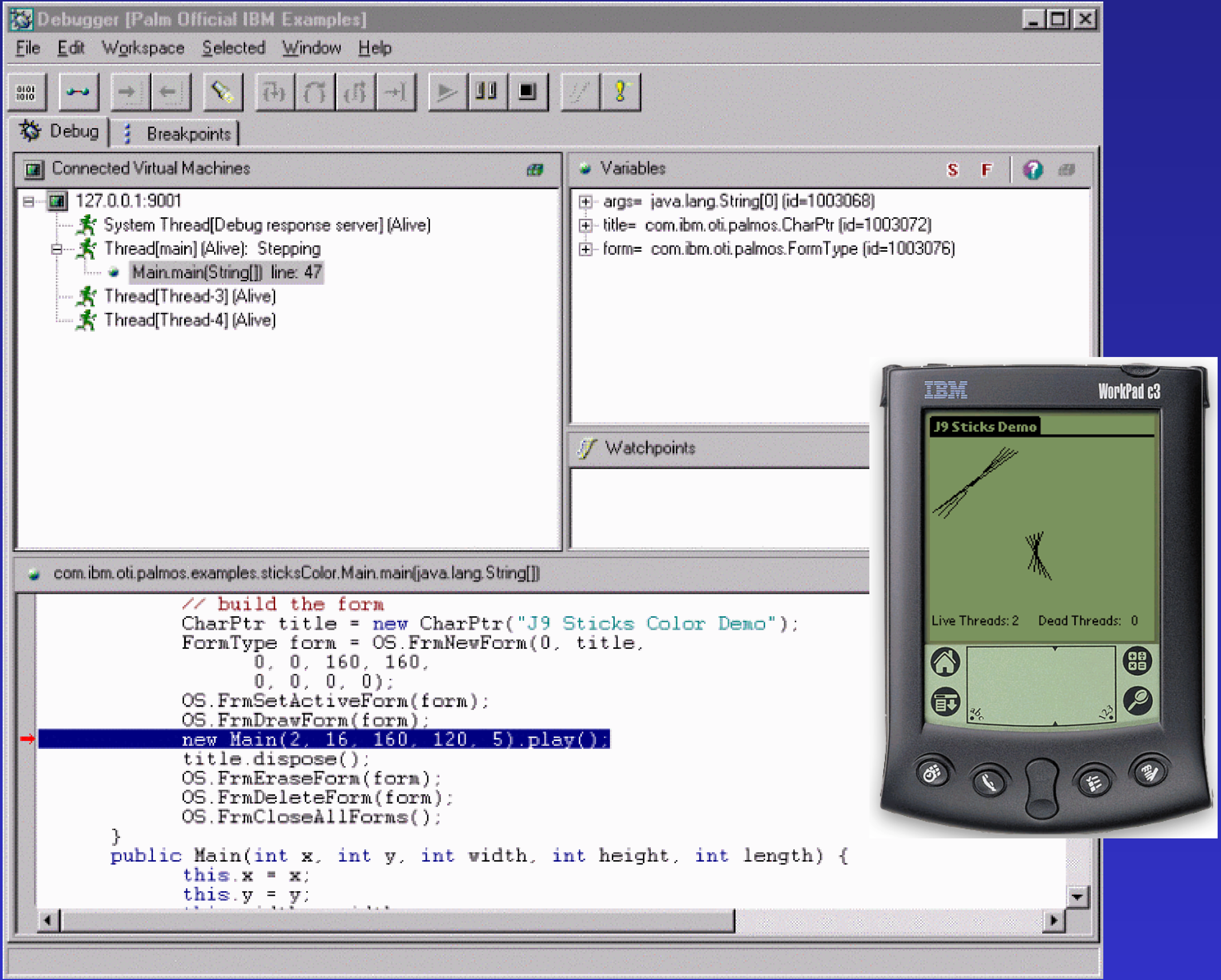

VisualAge Workbench [Palm Official IBM Examples]
File
Edit
Workspace
Selected
Window
Help
1.0 1.1
Projects
Packages
All Problems
Palm Official IBM Examples
VAME Palm Beambox Sample <>
VAME Palm Browser Sample <>
VAME Palm Build Runner Utility <>
VAME Palm Build Utility <>
VAME Palm Callback Sample <>
VAME Palm Database Sample <>
VAME Palm Database Viewer Sample <>
VAME Palm Datebook Sample <>
VAME Palm Empty Sample <>
VAME Palm IrDA Sample <>
VAME Palm kAWT <>
VAME Palm kAWT Sample <2000-Sept-14 2
VAME Palm KJava <>
VAME Palm KJava Sample <>
com.ibm.oti.palmos.examples.spotlet <>
Main.java <>
jxelink.opt <>
runLinux.vamemacro <>
runWin.vamemacro <>
VAME Palm Leak Sample <>
VAME Palm Library Sample <>
VAME Palm Resources Sample <>
VAME Palm Serial Sample <>
VAME Palm SlideShow Sample <>
VAME Palm Sticks Color Sample <>
VAME Palm Supplement Serial Sample <>
VAME Palm Utilities Sample [2000, sept 26, (
VAME Palm Vector Sample <>
VAME Palm Version Checking Sample <>
VAME JCL Xtreme Palm <>
Main.java

```
package com.ibm.oti.palmos.examples.spotlet;

/*
 * Licensed Materials - Property of IBM.
 * (c) Copyright IBM Corp 2000.
 *
 * IBM VisualAge Micro Edition for PalmOS
 */

import com.sun.kjava.*;

/**
 * Simple class subclassing kJava Spotlet class
 */
public class Main extends Spotlet {

      Graphics graphics;
      Button exitButton;

      public static void main(String[] args) {
            (new Main()).register(NO_EVENT_OPTIONS);
      }

      public Main() {
            graphics = Graphics.getGraphics();
            graphics.clearScreen();
            exitButton = new Button("Exit",70,100);
            paint();
            graphics.drawString("Hello World",60,70);
      }

      public void paint() {
            graphics.drawBorder(1, 1, 158, 158, Graphics.PLAIN, Graphics.SIMPLE);
            exitButton.paint();
      }

      public void penDown(int x, int y) {
            if (exitButton.pressed(x,y)) {
                  System.exit(0);
            }
      }

}
```

com.ibm.oti.palmos.examples.spotlet.Main.java - VAME Palm KJava Sample <>

VAME IDE - Analyzer





```
127.0.0.1:9001
    System Thread[Debug response server] (Alive)
    Thread[main] (Alive): Stepping
        Main.main(String[]) line: 47
    Thread[Thread-3] (Alive)
    Thread[Thread-4] (Alive)
```

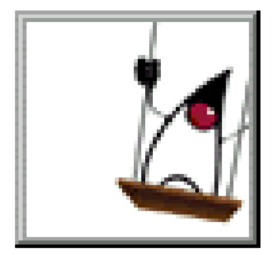
```
args= java.lang.String[0] (id=1003068)
title= com.ibm.oti.palmos.CharPtr (id=1003072)
form= com.ibm.oti.palmos.FormType (id=1003076)
```

com.ibm.oti.palmos.examples.sticksColor.Main.main(java.lang.String[])

```
                // build the form
                CharPtr title = new CharPtr("J9 Sticks Color Demo");
                FormType form = OS.FrmNewForm(0, title,
                        0, 0, 160, 160,
                        0, 0, 0, 0);
                OS.FrmSetActiveForm(form);
                OS.FrmDrawForm(form);
                new Main(2, 16, 160, 120, 5).play();
                title.dispose();
                OS.FrmEraseForm(form);
                OS.FrmDeleteForm(form);
                OS.FrmCloseAllForms();
        }
        public Main(int x, int y, int width, int height, int length) {
                this.x = x;
                this.y = y;
```

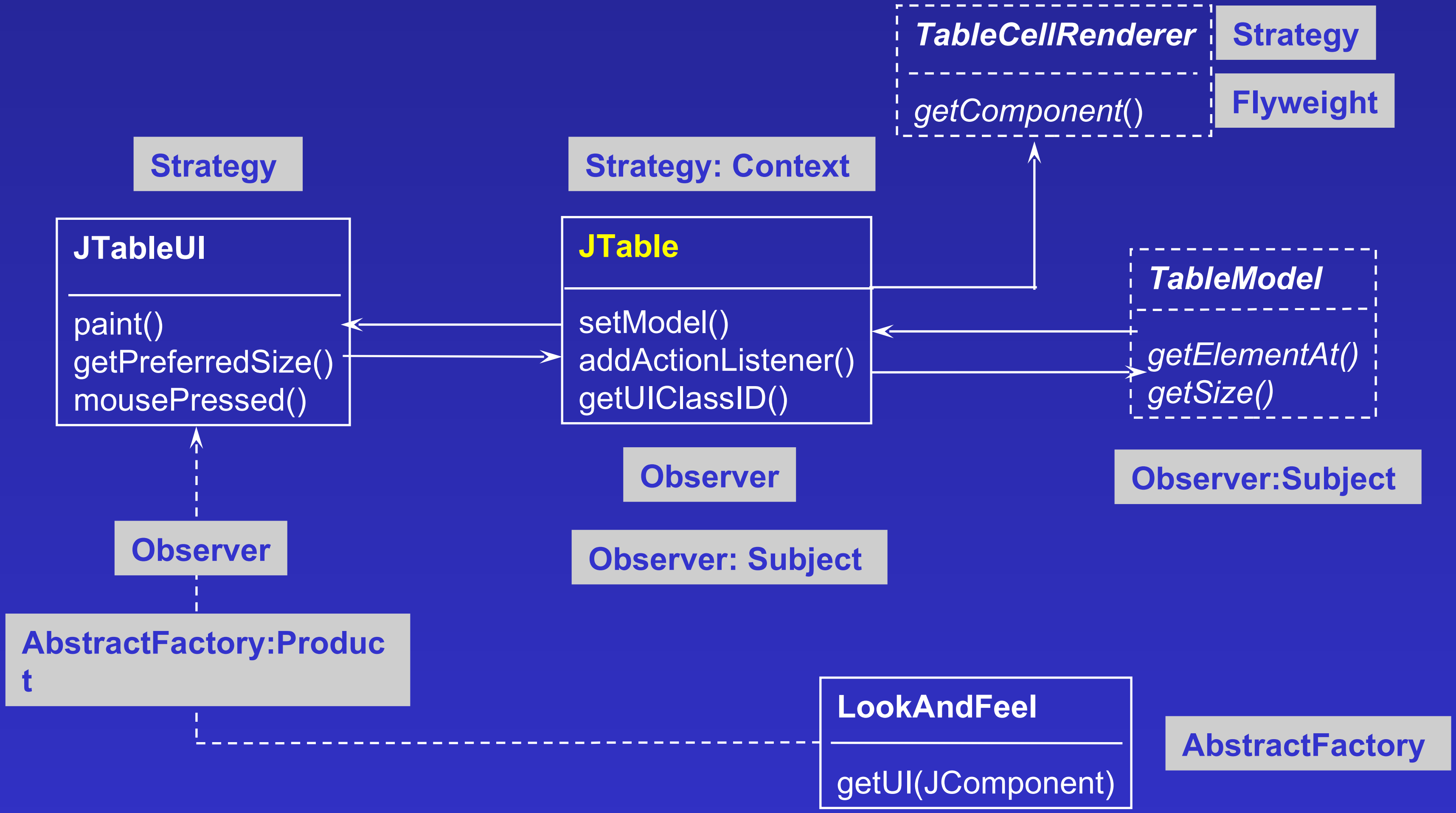
```
J9 Sticks Demo
Live Threads: 2    Dead Threads: 0
```

Swinging on the Bleeding Edge



- It all started with Swing 0.2...
- Swing is cool!
  - model based widgets
    - no gratuitous copying from model data structure into widget
    - Adapter binds model data to widget
    - lazy models
  - renderers
    - plugabble cell rendering Strategies
    - “rubber stamping”
  - pluggable look and feel
  - lots of pattern uses...

Swinging - Patterns at Work



TableCellRenderer

getComponent()

Strategy

Flyweight

Strategy

Strategy: Context

JTableUI

paint()
getPreferredSize()
mousePressed()

JTable

setModel()
addActionListener()
getUIClassID()

TableModel

getElementAt()
getSize()

Observer

Observer:Subject

Observer

Observer: Subject

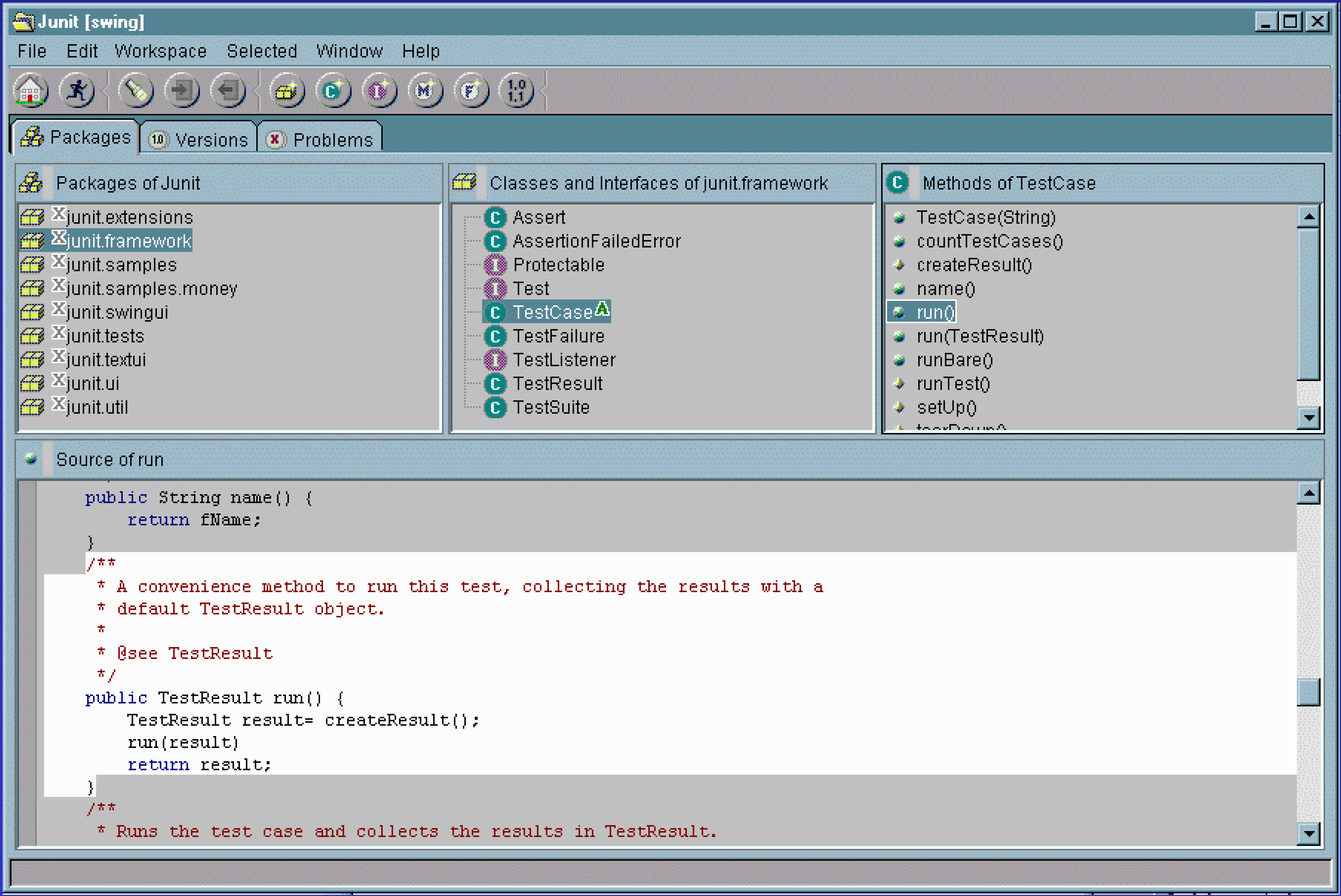
AbstractFactory:Produc
t

LookAndFeel

getUI(JComponent)

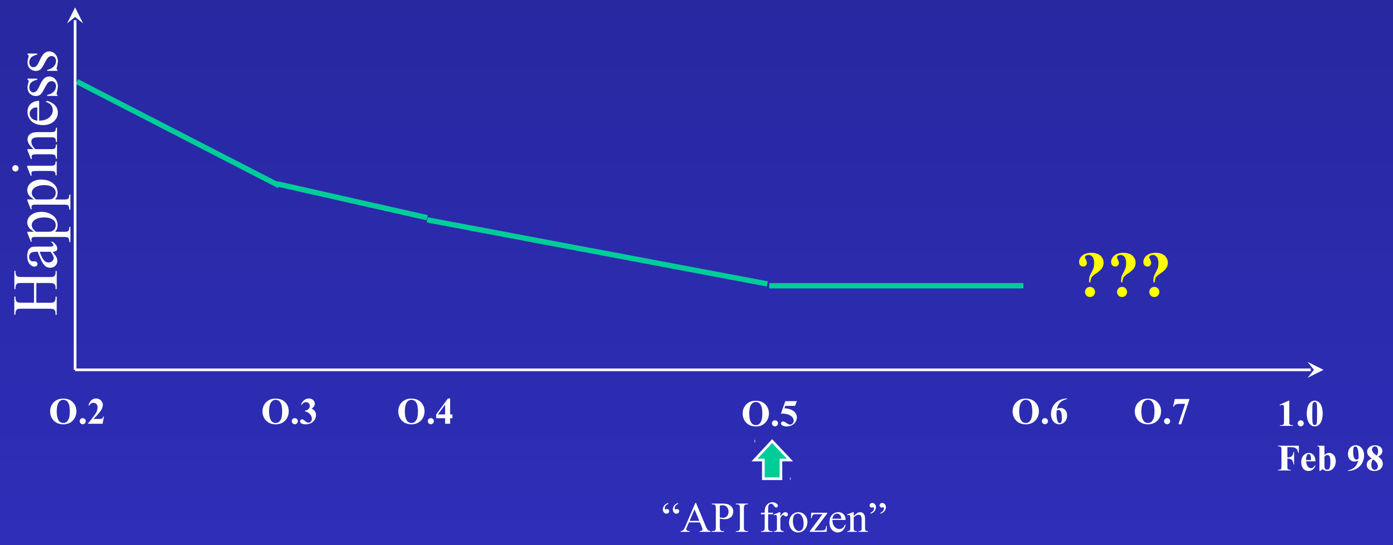
AbstractFactory

Ready to Ship



```
    public String name() {
        return fName;
    }
    /**
     * A convenience method to run this test, collecting the results with a
     * default TestResult object.
     *
     * @see TestResult
     */
    public TestResult run() {
        TestResult result= createResult();
        run(result)
        return result;
    }
    /**
     * Runs the test case and collects the results in TestResult.
```

Swing Happiness



Swing - Unhappiness

- Swing is loosing some coolness
- performance problems
    - start-up
        - lots of classes to load
    - GC pauses
        - lots of temporary objects
  - quality problems
    - lots of code
    - memory leaks
        - easy to create and difficult to track down
        - had to restart once per day

First Internal Feedback

- “sluggish”
  - not as clean and fast as native Windows GUIs
- “eats memory”
  - restart once a day
- “looks and behaves funny”
  - not the real Windows look and feel

    “Swing native look'n'feels are kind of like pod people UIs. They look like the real thing, they act like the real thing, but somehow they just aren't quite right. Dogs bark at them, and children aren't fooled at all.” --John Brewer, AutoDesk
  - e.g. file chooser...

Native Look and Feel

- **Swing lags behing the Native Look and Feel**

  Coming Swing API Changes for Java ™ 2 SDK, Standard Edition, v. 1.4

  > The ideal Swing application running under the Windows look and feel would be **indistinguishable from its native running counterparts**, however due to both changes in the native Windows look and feel (Windows 98 , Windows 2000, etc.) and atrophy of our existing Windows look and feel implementation, **this is not the current reality**. Our goal for this release is to provide an updated Windows look and feel which integrates seamlessly into the Windows desktop.

Lessons Learned

- Windows users are accustomed to things working in a specific way
  - native integration with platform is critical
  - leverage new platform features as they arrive

  ⇒you can build a Swing application for Windows, but you can't build a Windows application using Swing!
- Emulated widgets are hard, e.g. a simple Label
  - multi-line? alignment and NLS issues?
- Native GUI code has been fine-tuned over a long period of time you want to leverage wherever possible

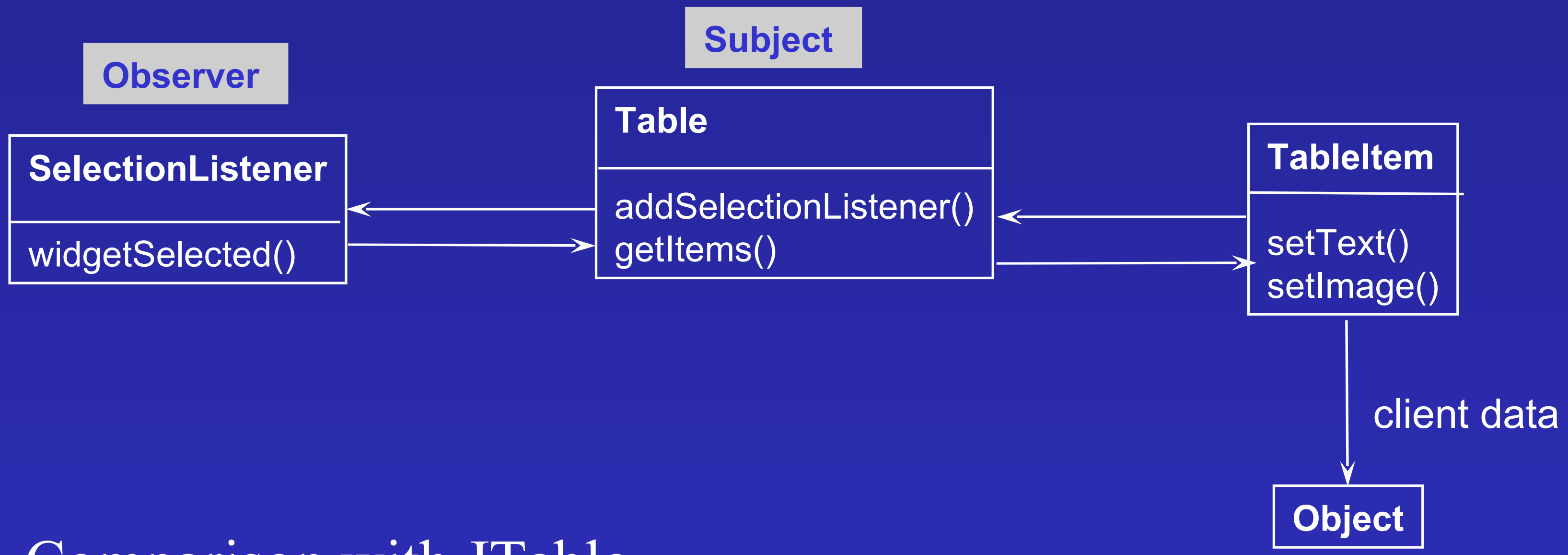
Options

- Proceed as is - technology will improve
  - we can’t fix all the problems ourselves
- Use AWT only
  - limited widget set
    - widget set can’t easily be extended with additional native widgets
  - architectural problems
    - async handling of native events
      - no application code executes in UI, this defeats platform optimizations
    - lazy native widget/peer creation
      - addNotify()
- use Windows specific APIs (e.g. WFC)
  - non-portable

SWT -A Simple Widget Toolkit

- Portable API to a standard set of widgets
  - implemented on Windows and Linux in Java
- Performance! Simplicity, robustness
  - just say no to unneeded generality
- Platform integration
  - native implementation (Win32, X/Motif, others)
  - embrace the native capabilities…
  - ...and accept their limitations
- Don’t sacrifice native integration on Win32
- OLE/ActiveX integration on Win32

Widget "Liposuction" - a Simple Table



- Comparison with JTable
  - No client controlled rendering of items (owner draw)
  - No lazy population of widget based by fetching data from a model
  - **But…less classes, less Java code, less bugs, better performance**

Comparison

	Swing/AWT	SWT
widget creation	widgets are created lazily by peers (addNotify)	no peers! Widgets are created immediately. Constructors require you to specify the parent - there is no addWidget!
event handling	listeners with typed events	listeners with typed events
layout	layout managers: GridBagLayout, GridLayout, BorderLayout, …	layout managers: GridLayout, RowLayout, FillLayout, …
threading model	AWT: free threaded Swing: only event thread is allowed to talk to widgets (not checked)	only thread that created widget is allowed to talk to widgets (checked!)
OS resources	resources are finalized by the GC	client has to destroy OS resources
rendering	renderers, owner draw	no owner draw
data access	Model interfaces, e.g. TreeModel	Data is pushed into widgets directly

Comparison Cont’d

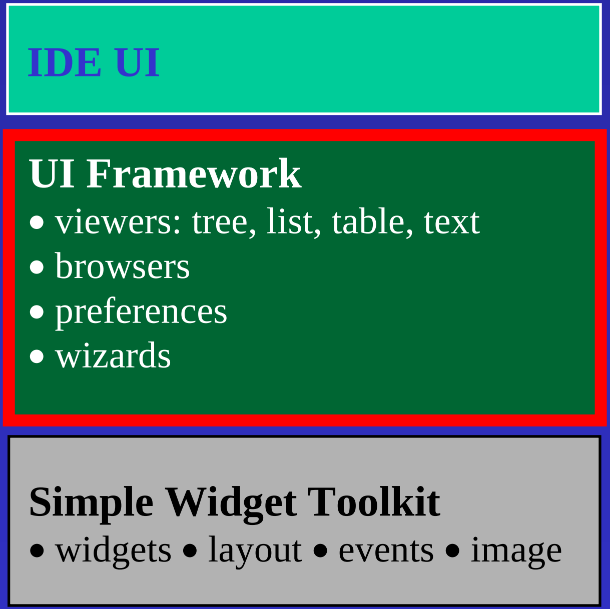
	Swing/AWT	**SWT**
Native code in dll	960 KB (JDK 1.3)	180 KB simple bindings to platform functions
Number of classes	> **1000**	< **200**

Hello World

```
Display display= new Display();
Shell shell= new Shell(display);
shell.setLayout(new FillLayout());
Button b= new Button(shell, SWT.PUSH);
b.setText(”Click Me");

b.addSelectionListener(
  new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
       System.out.println(“Hello World”);}
    }
);
shell.setSize(200, 200);
shell.open();

while(!shell.isDisposed())
  if(!display.readAndDispatch())
    display.sleep();
```

The UI Framework - JFace

- Basic widgets are not enough for application development
- Additional support is required to:
  - populate widgets with domain objects
  - keep widget in synch when domain objects change
- Therefore: build a thin UI framework on top basic widgets



IDE UI

UI Framework

- viewers: tree, list, table, text
- browsers
- preferences
- wizards

Simple Widget Toolkit

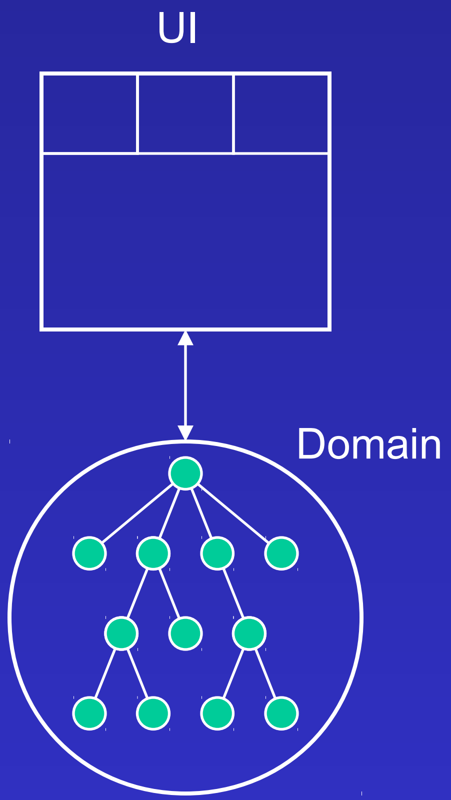
• widgets • layout • events • image

Why a Thin Framework

- Problems with *fad* frameworks
  - difficult to learn
  - difficult to evolve and maintain
    - Frameworks are more “white box” thank toolkits
  - Powerful frameworks have to make some constraining assumptions
    - “Frameworkitis”
    - too smart and can get in the way
    - provide generic behaviour that isn’t needed and get’s in the way

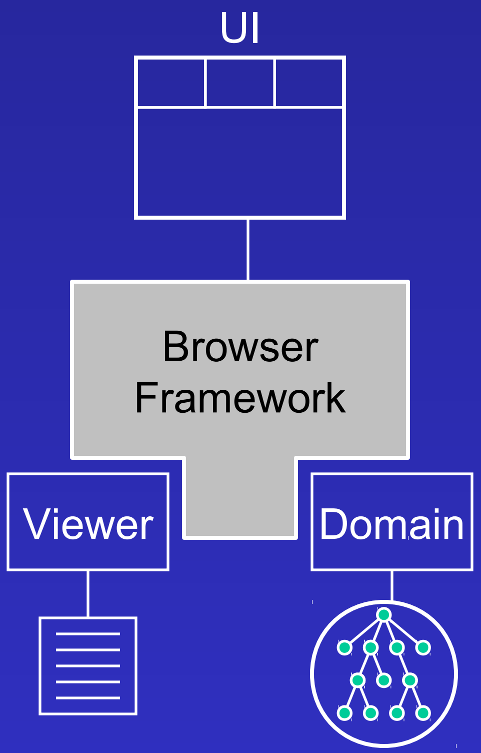
The UI Framework: Iteration 1

- Problem: exploring and manipulating a hierarchically structured *Domains*
- presenting domain in UI
  - keep the UI in sync as model changes
- navigating relationships
  - viewing/editing of a node’s contents



The Goal

- A **thin** framework that…
  - defines the browsing metaphor
    - implements the “complex stuff”
  - allows clients to focus on
    - domain definition
    - node content editors/viewers
  - is simple!
    - small number of concepts



Domain Access

- ***Elements***
  - browseable entities
  - data nodes in the domain
  - examples: a file, a mailbox
- Elements have ***Properties***
  - *aspects* of the browsable entities
- Elements provide a *dynamic data access* API

```
Object getProperty(String)
```

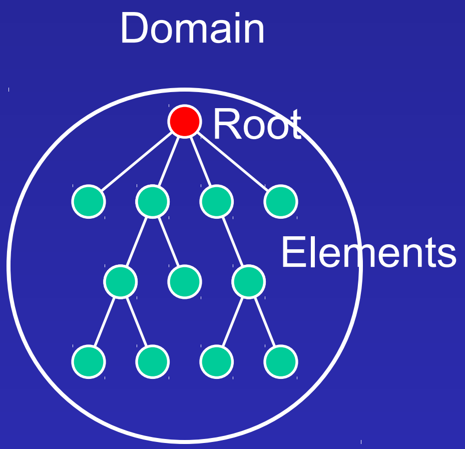
- Property kinds
  - simple: Object, Boolean, String, Element
  - indexed: ordered set of Elements

Type

methods
code
basetypes
versions

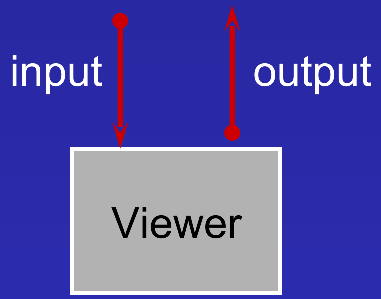
Domain Model

- knows a root element
  - the “portal” into a domain
- is the model in the Model/View architecture
  - notifier for domain changes
  - elements fire domain changes via model
    - ⇒ elements know their domain model
  - notification specifies changed property
  - observers register with domain model



Viewer

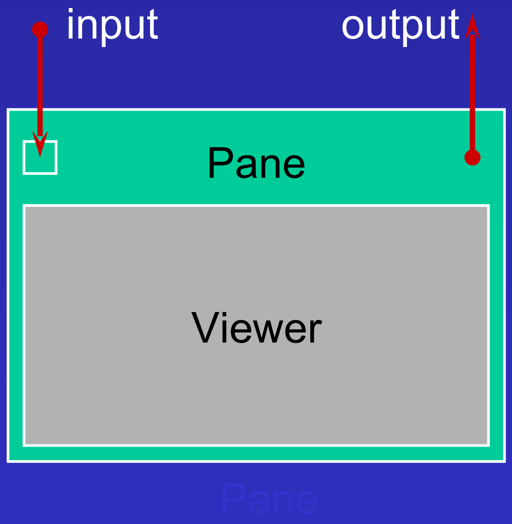
- A Viewer …
  - is fed with input element
  - presents properties of its input element
  - observes domain model for changes
  - handles user interactions
  - sends out selection change events



- Standard Viewers exists
  - Structure oriented Viewers
    - Tree, List, Table
  - Content oriented Viewer
    - Text

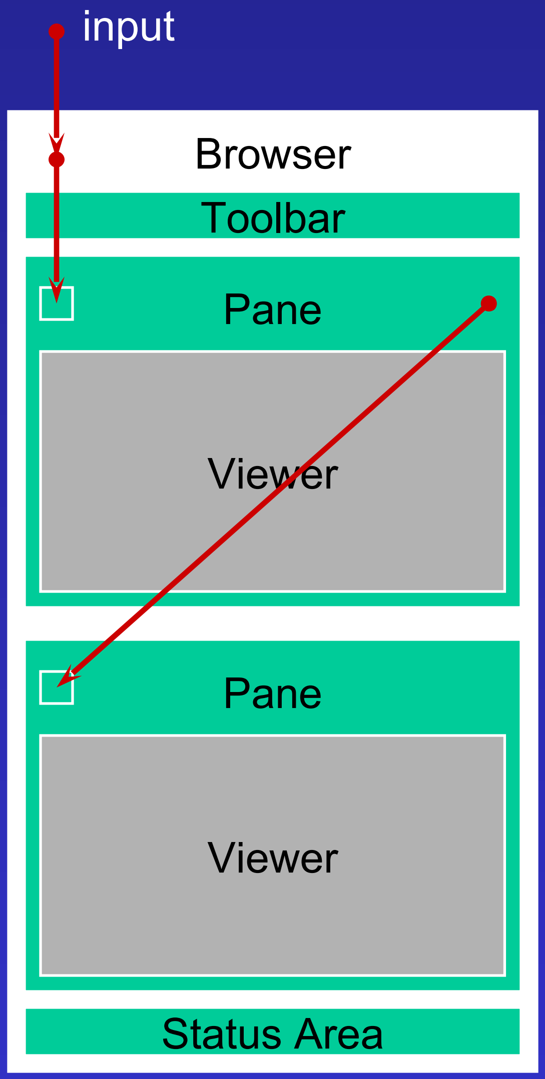
Pane - a Viewer’s Container

- installs Viewer dynamically based on its input
- optionally provides UI to pick other viewers for the viewed property

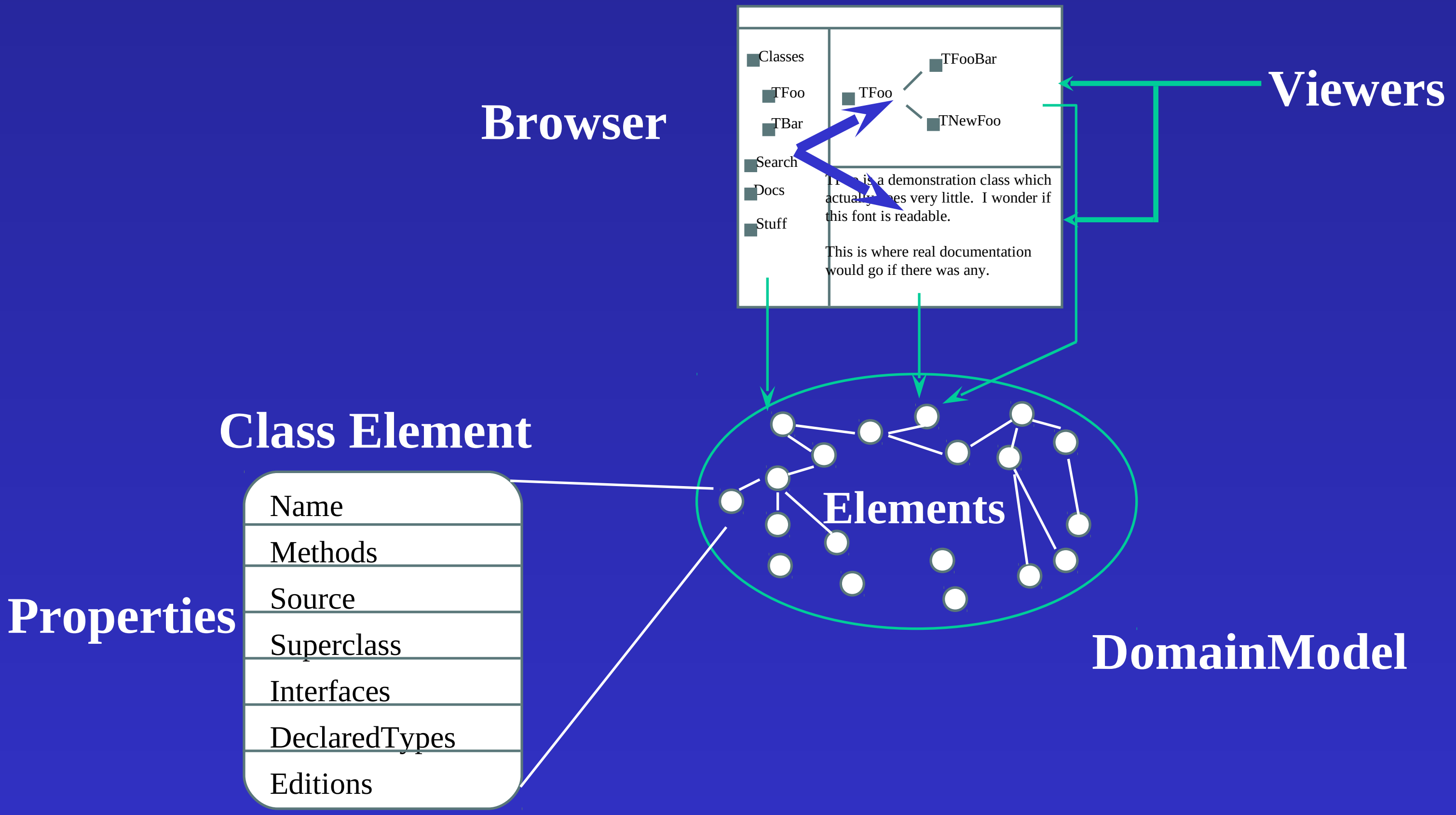


Browser - Pane's Container

- implements browsing metaphor
- is fed with an Element
- manages panes
- defines wiring between panes
- defines layout between panes



Summary



Browser

Viewers

Classes

TFoo

TBar

Search

Docs

Stuff

TFoo

TFooBar

TNewFoo

TFoo is a demonstration class which actually does very little. I wonder if this font is readable.

This is where real documentation would go if there was any.

Elements

DomainModel

Class Element

Properties

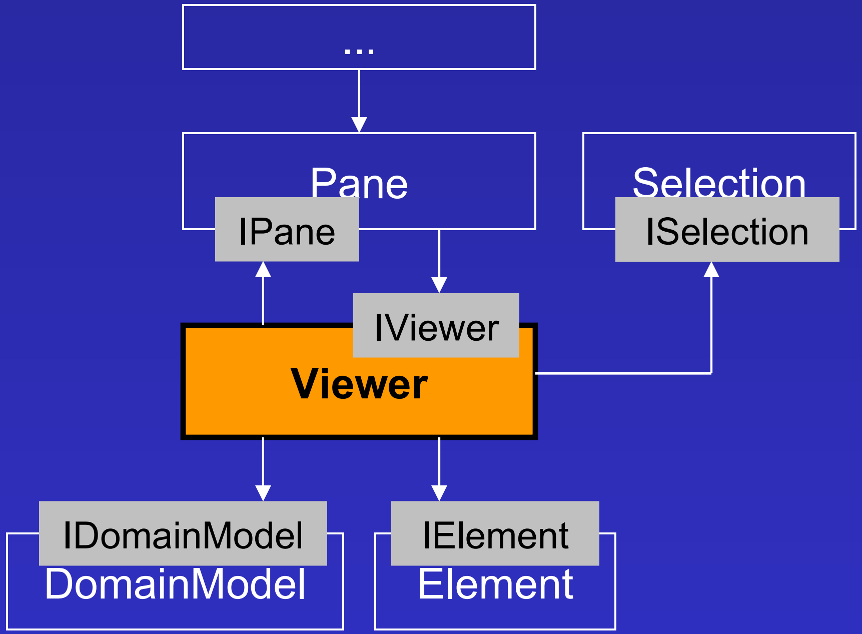
- Name
- Methods
- Source
- Superclass
- Interfaces
- DeclaredTypes
- Editions

Defining the Framework

- Separation of design from code
  - define the “design” as Java interfaces in one package
  - move “implementation details” into a separate package
- Motivation
  - encapsulate volatile implementation details behind stable interfaces
    - make the difference explicit for clients
  - capture the object interactions in interfaces
  - clients shouldn’t be forced into implementation inheritance
    - less flexible

Discovering the Viewer Interface

- An interface defines a role an object plays
  - Captures the collaborations between objects



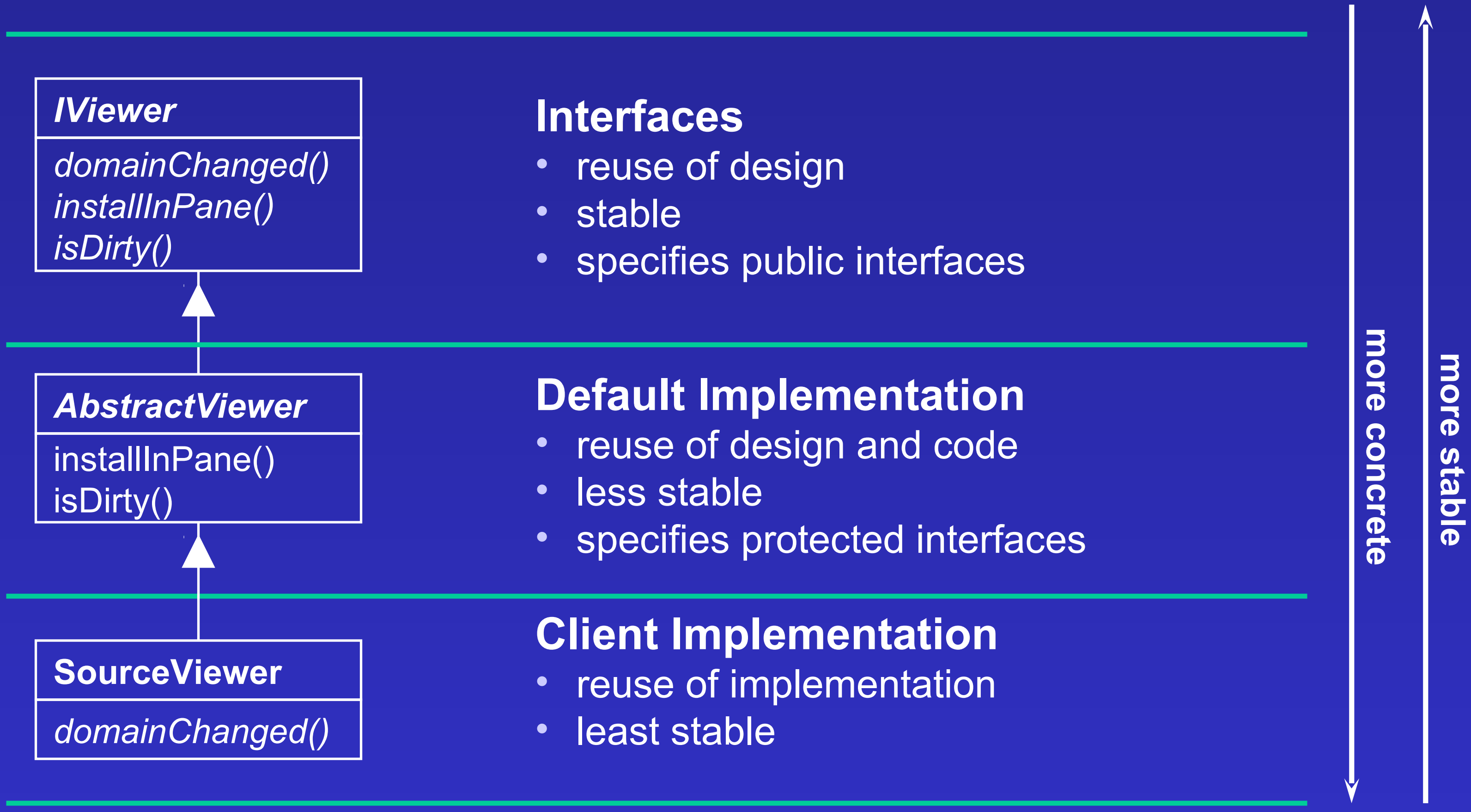
Problems with Interfaces

- Interfaces cannot have default implementation
  - cumbersome for clients to implement
  - every interface change is a breaking change!

⇒ Provide default implementations in a separate layer

- difference between design (interfaces) and implementation remains explicit!

Example: Layering



IViewer

domainChanged()
installInPane()
isDirty()

AbstractViewer

installInPane()
isDirty()

SourceViewer

domainChanged()

Interfaces

- reuse of design
- stable
- specifies public interfaces

Default Implementation

- reuse of design and code
- less stable
- specifies protected interfaces

Client Implementation

- reuse of implementation
- least stable

more concrete

more stable

⇒ Pattern Interface-Implementation Pair

Iteration 2: From White-Box to Black-Box

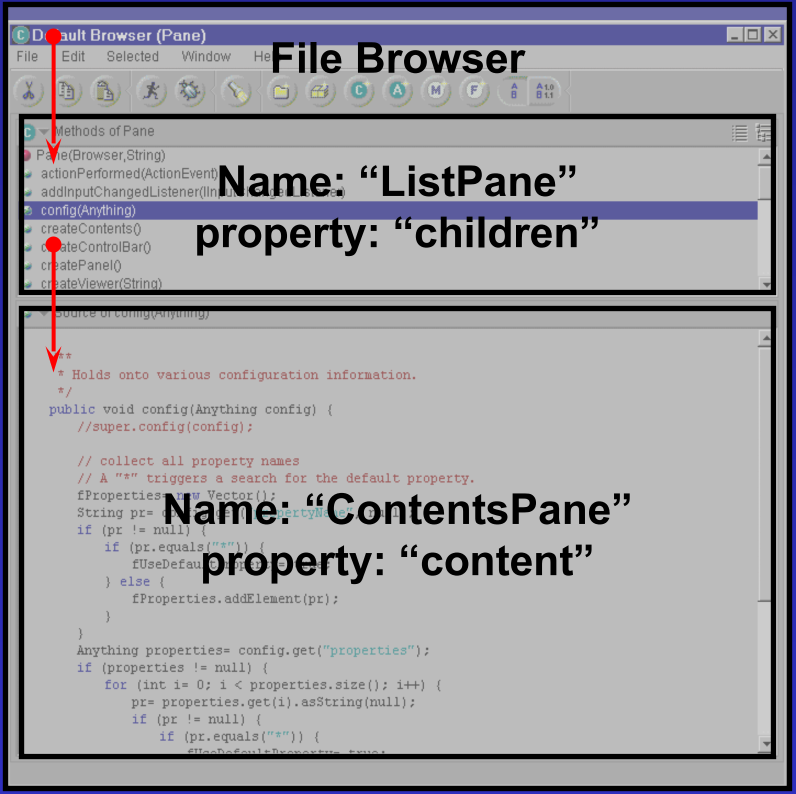
- Clients still have to subclass several framework classes:
  - various factory methods
  - Browser: layout, wiring
  - Pane: property selection

⇒ Introducing composition/configuration instead of subclassing

- white-box frameworks
  - promote flexibility
  - based on inheritance, dynamic binding
- black-box frameworks
  - promote ease of use
  - based on composition, configuration

Configuration with XML
Example: Browser Definition

```
<browser outputs="ListPane">
 <layout>
   <vsplit>
      <pane name="ListPane"
         properties="children"
         outputs="ContentsPane">
      </pane>
      <pane name="ContentsPane"
         properties="contents">
      </pane>
   </vsplit>
 </layout>
</browser>
```



File Browser

Pane(Browser,String)
actionPerformed(ActionEvent)
addInputChangedListener(IInputChangedListener)
config(Anything)
createContents()
createControlBar()
createPanel()
createViewer(String)

Name: “ListPane”
property: “children”

```
/**
 * Holds onto various configuration information.
 */
public void config(Anything config) {
   //super.config(config);

   // collect all property names
   // A "*" triggers a search for the default property.
   fProperties= new Vector();
   String pr= config.get("propertyName", null);
   if (pr != null) {
      if (pr.equals("*")) {
         fUseDefaultProperty= true;
      } else {
         fProperties.addElement(pr);
      }
   }
   Anything properties= config.get("properties");
   if (properties != null) {
      for (int i= 0; i < properties.size(); i++) {
         pr= properties.get(i).asString(null);
         if (pr != null) {
            if (pr.equals("*")) {
               fUseDefaultProperty= true;
```

Name: “ContentsPane”
property: “content”

“Componentizing” Viewers

- End of 1st iteration: many custom viewers
- Consolidation revealed:
  - clients typically changed only a few aspects of viewers:
    - sorting and filtering
    - rendering (how properties of a single element are drawn)
    - action to execute for specific user-interaction
- Refactoring for composability
  - introducing *Strategies*: Sorter, Filter, LabelProvider

    ⇒Fine-grain componentizing

⇒ Configurable viewers without subclassing

Example: Custom TreeViewer

- A single viewer can be customized to different uses without subclassing
  - heterogeneous traversal - enumerating children
    - children property
  - sorter
    - sorting order
  - rendering
    - label property
    - icon property
  - actions

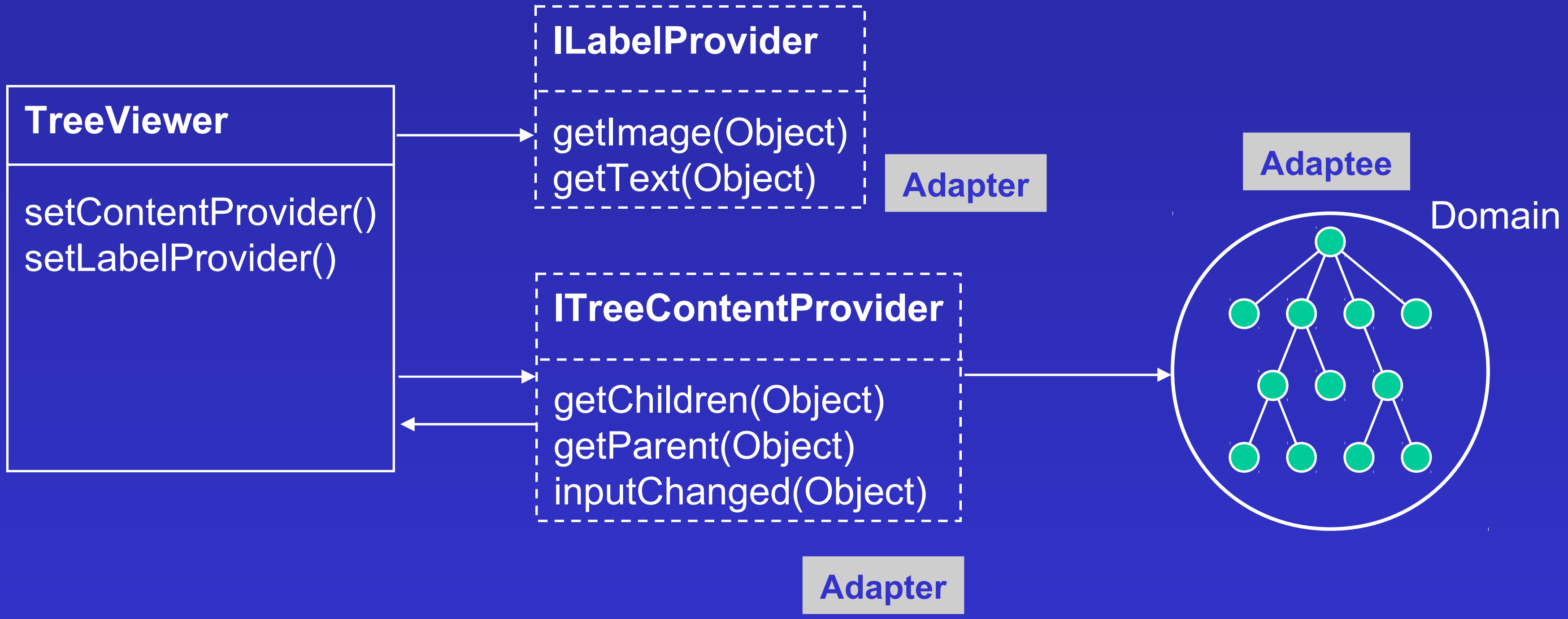
```
/MyTreeViewer {
    /class "com.x.TreeViewer"
   /childrenProperty "variables"
    /sorter { } # no sorter
    /renderer {/class "com.x.VariableRenderer"}
    /actions {
        /DoubleClick { /class "com.x.MyAction" }
    }
}
```

Iteration 3: An even Thinner Framework

- Refactoring
  - Don’t require XML to use the framework
    - configuration with code is more direct
  - Make less constraining assumptions about the domain model
    - domain model doesn’t have to be accessible with Elements and Properties
    - don’t require a standard notification scheme
    - **`Object`** as the common currency, there should be no additional type requirements on the domain model

Domain Access with Adapters

- Domain access implemented as plugins of a Viewer
  - Idea: define (pluggable) Adapters for accessing a domain
- Dimensions
  - Accessing the structure of a domain and tracking changes
  - Rendering a domain object



TreeViewer

setContentProvider()
setLabelProvider()

ILabelProvider

getImage(Object)
getText(Object)

Adapter

ITreeContentProvider

getChildren(Object)
getParent(Object)
inputChanged(Object)

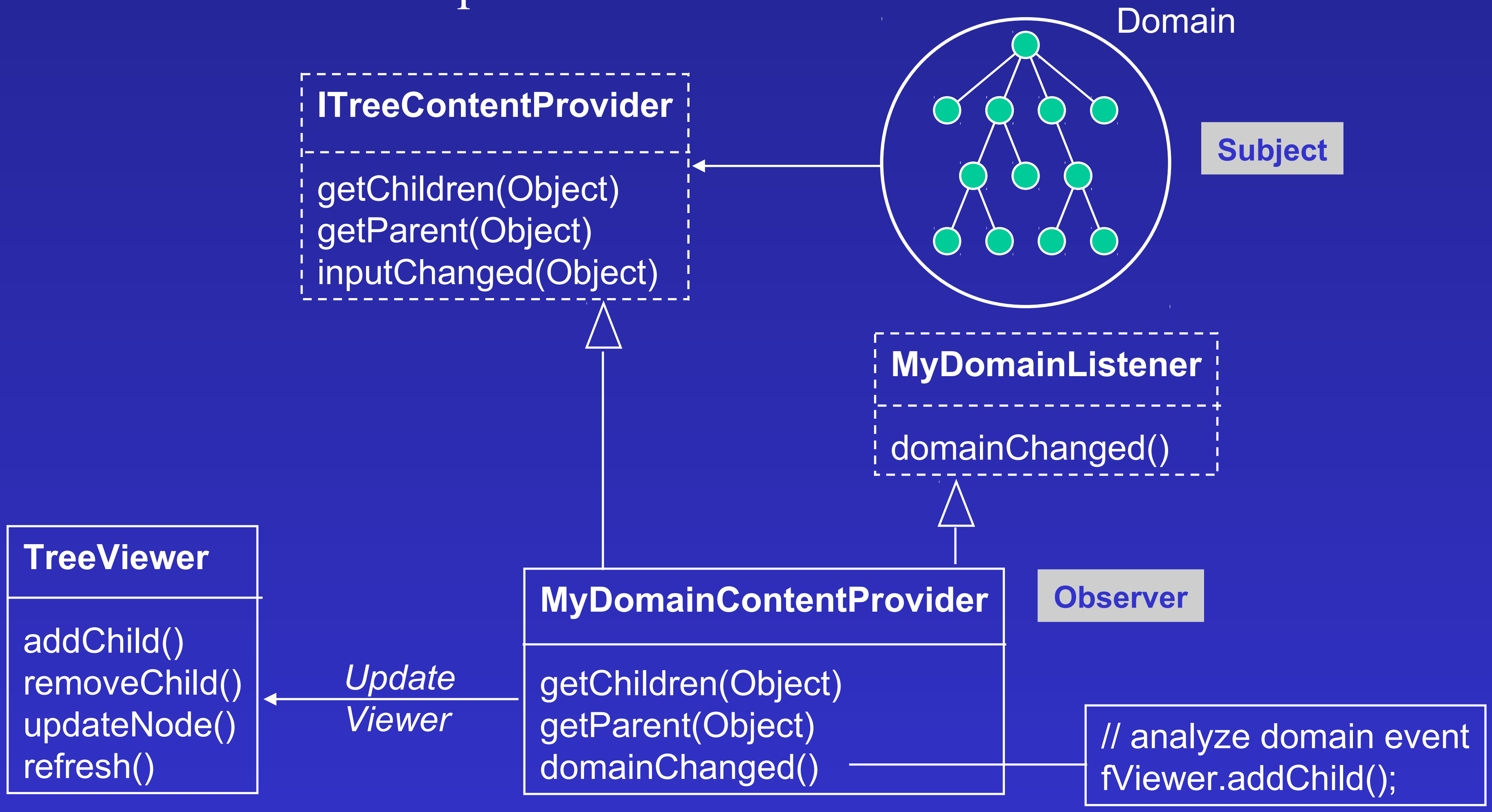
Adapter

Adaptee

Domain

Tracking Domain Changes

- ContentProvider is responsible to translate domain events into Viewer updates



Domain

Subject

ITreeContentProvider

getChildren(Object)
getParent(Object)
inputChanged(Object)

MyDomainListener

domainChanged()

TreeViewer

addChild()
removeChild()
updateNode()
refresh()

Update Viewer

MyDomainContentProvider

getChildren(Object)
getParent(Object)
domainChanged()

Observer

// analyze domain event
fViewer.addChild();

SWT+JFace vs. Swing

- Focus on native widgets
- Clear layering between basic widgets and application functionality
  - Basic widgets are not model based
    - "Pay as you go" – when you need a simple widget you can have one
- More consistent and orthogonal API
- Smaller and simpler
  - SWT (200 classes) +JFace (170 classes) < Swing (1000 classes)
  - JFace provides additional features
    - Wizards
    - Preferences
    - Operations

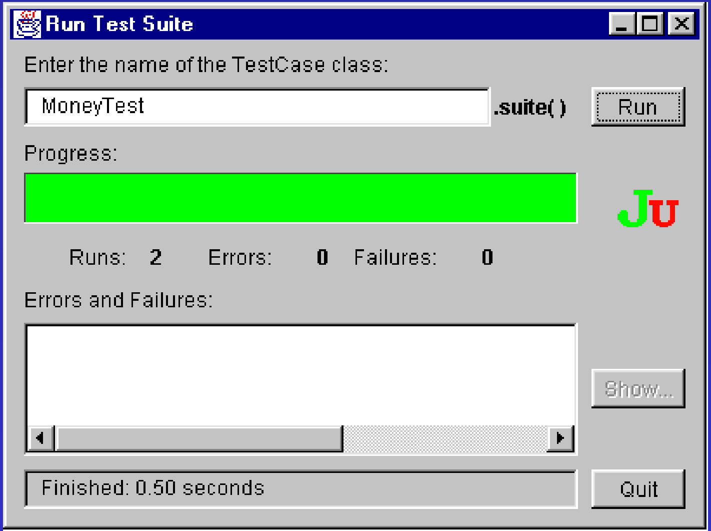
Making Change Your Friend

- Problems:
  - evolving framework while there are existing clients
  - getting confidence in changed framework
- Solutions:
  - backward compatibility
    - use deprecation
    - but deprecation doesn’t work for hook methods when their signature changes
      - client overrides methods that are no longer called
      - declare such methods as **final**
      - compiler warns client about obsolete override
  - unit tests …

Inset: JUnit

- An open source framework for implementing unit tests (www.sourceforge.net/projects/junit)
- Implementing unit tests:
  - define *fixture* to capture common set-up code
  - stimulate the fixture with test cases
  - verify the results
  - Aggregate tests into suites

```
public class MoneyTest extends TestCase {
  public void testMoneyEquals() {
       assert(!f12CHF.equals(null));
       assert(f12CHF.equals(f12CHF));
       assert(!f12CHF.equals(f14CHF));
  }
  public void testAdd() {
    …
  }
}
```



Run Test Suite

Enter the name of the TestCase class: MoneyTest .suite()

Progress:

Runs: 2 Errors: 0 Failures: 0

Errors and Failures:

Finished: 0.50 seconds

Unit Tests for JFace

- Unit tests are required to ensure that refactorings preserve the desired behavior
- Viewer update is an area of breakage
  - focus tests on model-viewer consistency
- There is a class hierarchy of Viewers
  - leverage inheritance to reduce the number of tests that need to be implemented

Unit Test Example

- StructuredViewerTest.setup()

```
public void setUp() {
    fViewer= createViewer();
    fBrowser= createBrowser(fViewer);
    fRootElement= TestElement.createModel(3, 10);  //
create test domain model
    fBrowser.open ();
}
```

- StructuredViewerTest.testDeleteChild

```
public void testDeleteChild() {
    TestElement first= fRootElement.getFirstChild();
    TestElement first2= first.getFirstChild();
    first.deleteChild(first2);                 //
change domain model
    assertNull(fViewer.FindItem(first2));      //
verify
}
```

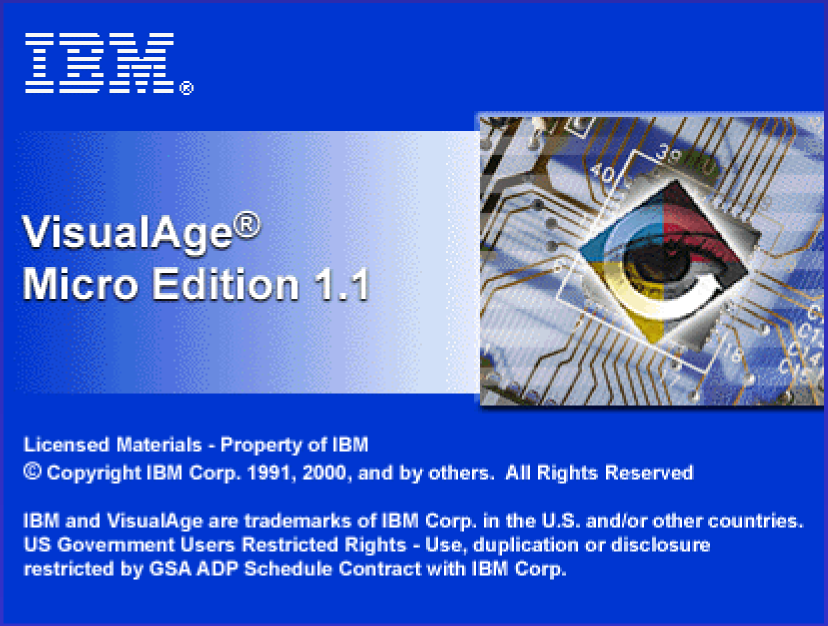
Unit Tests (Cont'd)

- Good design simplifies unit testing
  - "lazy testing"
- Viewers are factored into a class hierarchy
  - StructuredViewer
    - AbstractTreeViewer
      - TreeViewer
      - TableTreeViewer
    - TableViewer
    - ListViewer

  - StructuredViewerTest
    - AbstractTreeViewerTest
      - TreeViewerTest
      - TableTreeViewerTest
    - TableViewerTest
    - ListViewerTest
- Tests against StructuredViewer can be *reused* for subclasses
  - StructuredViewerTest provides Factory Method that subclassers implement to return a concrete Viewer object for the tests

The Happy End

- Shipped (March 2000)
- self hosting
  - “we are eating our own dog food”
- IBM VisualAge Micro Edition Home Page,
  - http://www.ibm.com/software/ad/embedded
  - Free download of VAME IDE with Palm runtime



VisualAge®
Micro Edition 1.1

Licensed Materials - Property of IBM
© Copyright IBM Corp. 1991, 2000, and by others. All Rights Reserved

IBM and VisualAge are trademarks of IBM Corp. in the U.S. and/or other countries.
US Government Users Restricted Rights - Use, duplication or disclosure
restricted by GSA ADP Schedule Contract with IBM Corp.

Conclusions

⇒A minimalist UI toolkit was the key to shipping a client-side Java application that

  - is indistinguishable from a native application
  - performs like a native application

- UI framework had to evolve over multiple iterations
- Unit tests were critical for evolving and refactoring the UI Framework