Monolithen fit für die Zukunft trimmen

Warum Monolithen entstehen und wie Microservices helfen können, die Biester zu bändigen

Anatole Tresch, Principal Consultant

trivadis
makes IT easier.

BIO

Anatole Tresch

- Principal Consultant, Trivadis AG (Switzerland)
- Star Spec Lead JSR 354
- Technical Architect, Lead Engineer
- PPMC Member Apache Tamaya

- Twitter/Google+: @atsticks
- atsticks@java.net
- anatole.tresch@trivadis.com

trivadis
makes IT easier.

Agenda

- Motivation
- Microservices to our rescue!
- How to get there?
- Deep Dive:
  - Coupling
  - Breaking things up
  - API Design
- Summary

trivadis
makes IT easier.

Motivation:
Why are monoliths a problem?

trivadis
makes IT easier.

Why are monoliths a problem?

- Maintenance Costs
- Time to Market
- Performance
- Stability and Robustness (forget Resilience!)
- Flexibility (e.g. for Integration)
- Vendor Support
- Know How
- Motivation
- ...

trivadis makes IT easier.

Compared with the Software Crisis

- Development time exceeds initial expectations
- Poor software quality
- Project cost exceeds estimated budget
- Hard-to-maintain codebase
- A lack of communication with customers

→ **Structure the Software Development Process**

- Planning
- Implementation, Testing and Documentation
- Deploying, Maintenance

Motivation:
Why Monoliths exist...

trivadis
makes IT easier.

Why monoliths exist then?

- Traditional software design
  - Single threaded
  - Synchronous
  - Relying on the „happy-path“
- Organizational Issues
  - Convey's Law
  - Budgeting Policies
  - Short Time Thinking
  - Skills and Culture
  - Missing or unmatching Objectives and Responsibilities

trivadis makes IT easier.

Why monoliths exist then?

- Misunderstandings
  - Software is done once and never touched again
  - Structuring the software process increases software quality
  - Centralization increases efficiency
- Bad Practices
  - Do things later (tests, APIs, docs, design, architecture)
  - Inefficient infrastructure
  - Not-invented-by-me syndrom
  - Ivory Towered Guidelines, Gardening

trivadis
makes IT easier.

Microservices to our rescue!

trivadis
makes IT easier.

Microservices come to our rescue!

- The single concern principle matches better with our brains capabilities
- Organizing around business capabilities clarifies reponsibilities
- Services provide much better isolation than modules
- More fine grained options for scaling and failover
- Product Orientation follows a long term perspective
- Simplicity enables automation
- Microservices must be designed for failure!
- Microservices enable evolutionary design
- Microservices support technological progress

Wow!

trivadis
makes IT easier.

Microservices to our rescue!
But...

trivadis
makes IT easier.

Microservices are the way to go, but...

It is a long way to go...

- Organizing around business capabilities is not a common approach
- They require automation in Ops
- They require modern know-how in Dev
- Resilient Design is not an easy task
- Runtime behaviour, scaling and failover are not easily predictable
- Governance has to be established

→ **Microservices consciously break with many known practices!**

→ **Are you microservice-ready?**

→ **How to get there?**

trivadis makes IT easier.

How to get there?

trivadis
makes IT easier.

How to go the microservice path?

Start with the preconditions:

- Fast provisioning of infrastructure (IaaS)
- Fast provisioning of runtime platforms (PaaS)
- Orchestration services (e.g. Kubernetes)
- Monitoring
- Automated tool-chain

→ Start with a new greenfield organization

→ Only couple with existing providers, if they support the required SLA

→ Start small to gain experience (not yet with microsevices!)

trivadis makes IT easier.

How to go the microservice path?

Add deploying more applications:

- Add dynamic service discovery and location
- Think on microservice and API management
- Think on cloud capable configuration mechanisms

Add further usage scenarios:

- Integration flows (perfect match, forget ESBs ;-))
- Authentication and Authorization services
- Distributed Message Streams, Log Collectors
- Error and Dead Letter Queue Handling
- ...

trivadis
makes IT easier.

How to go the microservice path?

And not to forget :

- Add systematic quality measurements to your tool chain
- Evangelize microservices as an architectural pattern
- Help people getting better in modularization
- Manage your people's skills!

And finally :

- Look at your monoliths

trivadis
makes IT easier.

How to got microservice path?

...expect you will fail !

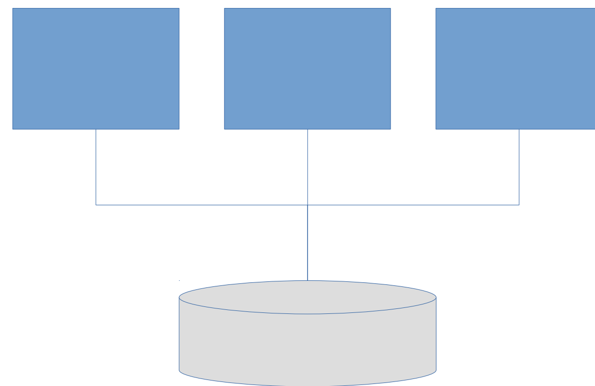
Deep Dive: Coupling

trivadis makes IT easier.

Coupling: Database Integration

Database Integration

- **Database = large shared API**
- Database design impacts the whole system
- Brittle
- Tied to a technology, or even vendor
- Cohesion (especially with stored procedures)

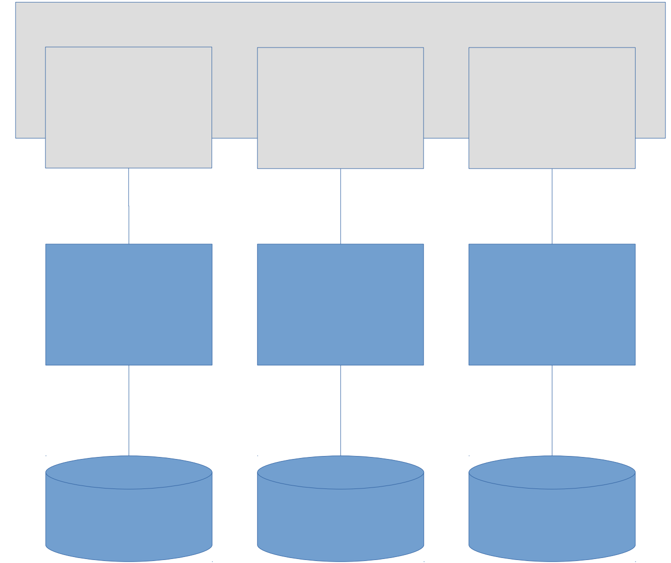


trivadis makes IT easier.

Coupling: UI Integration

UI Integration

- Shared API on UI level
- Brittle
- Tied to a technology, or even vendor
- Multiple, sometimes incompatible frameworks

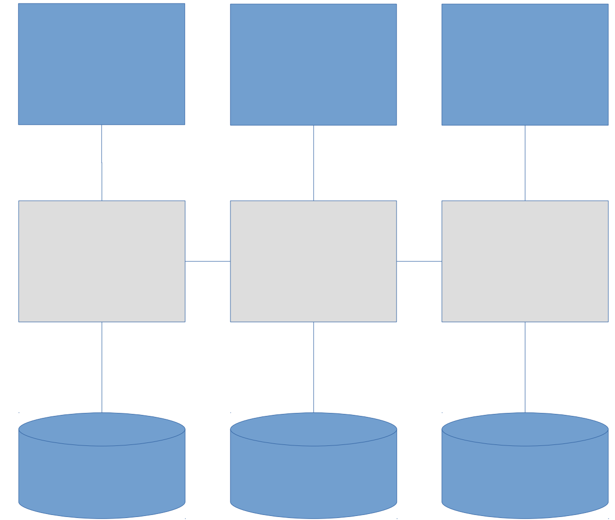


trivadis
makes IT easier.

Coupling: Service Integration

Service Integration

- Shared API on Business Tier level
- Brittle
- Tied to a technology, or even vendor (e.g. RPCs with EJB/RMI)
- Unfortunately commonly used in monolithic systems



trivadis
makes IT easier.

Coupling by Communication Design

Synchonous Communication?

- Long running tasks?
- Failure Handling
- Responsivness (Timeouts, non determinism)
- Request/Response Pattern is also possible with asynch communication

trivadis makes IT easier.

Coupling by Communication Design

Asynchronous Communication

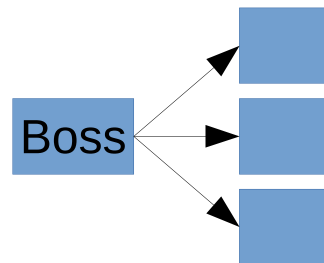
- Event-Based = inversed asynch („IoC“)
- Message Brokers or HTTP events (ATOM)
- Supervision
  - Long running processes?
  - Failures? How to handle bad messages breaking your system?
- Correlation and Monitoring

trivadis makes IT easier.

Supervision Stategies

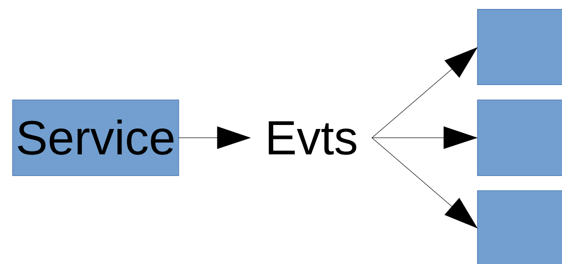
Orchestration

- Central brain to guide and drive



Choreography

- Inform Component to do work and let it do its stuff



trivadis
makes IT easier.

Coupling by Data Formats

XML
- Well defined
- Heavy
- Brittle
- Secure
- Standardized
- **Inherently inflexible**

JSON
- Easy
- Lean
- No link concept, but HAL*
- No automatic client generation
- **Lean and flexible**

* Hypertext app language

trivadis
makes IT easier.

Other Coupling Areas

- Tool Chain and automation infrastructure
- Shared libraries:
  - Hide from clients!
  - Especially dangerous:
    - Communication frameworks
    - Frameworks for bridging a missing model layer
- **Third Party Products**
  - APIs, Lifecycle and Product Quality are out of your control!
  - APIs are normally very badly designed
  - Separate Data Model

trivadis makes IT easier.

How to deal with it?
Breaking things up...

trivadis
makes IT easier.

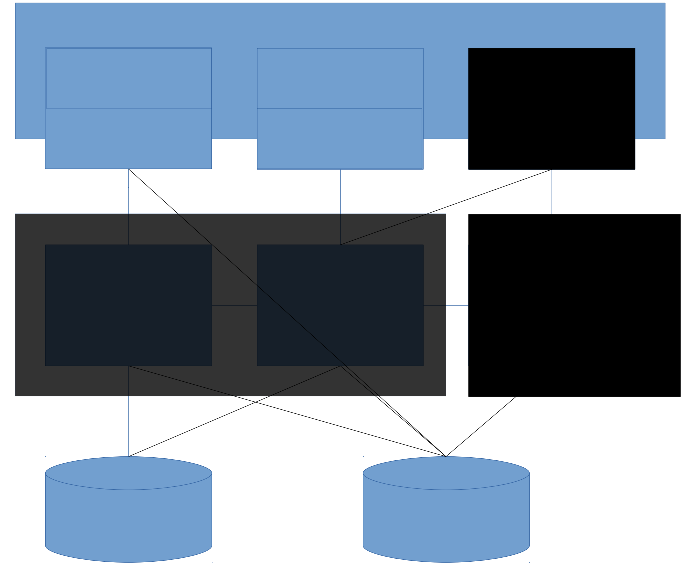
Our Objectives

1. Avoid breaking changes
2. Design APIs technology agnostic and remotable
3. Make services easy for our customers
4. Hide implementation details

trivadis
makes IT easier.

Typical Monolithic Architecture

- Is a monolithic systems a typical combination of all coupling variants we have identified ???
  → No, it is even worse...!
  → You don't know what it is! It has black areas!
  → You start in blind flight mode!



- So where to start?

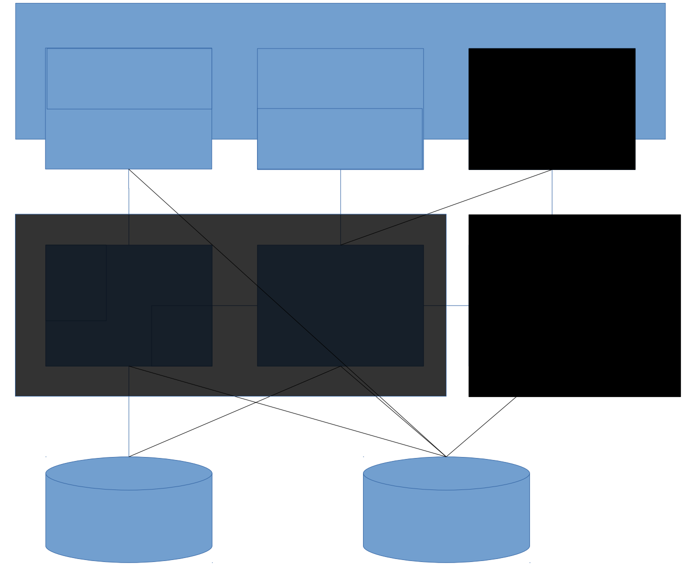
Do small steps and prepare for failure!

trivadis makes IT easier.

Breaking up the Monolith

Where to start?

1. Disentangle the **DB** (mostly not an option at the beginning)
2. Disentangle the **UI** (difficult)
3. Disentangle the **Middle Tier**
   1. Lots of help tool support (IDEs, compiler build tools, tests)
   2. Tests and quality metrics can be automatically evaluated
   3. Fast and automatic feedback

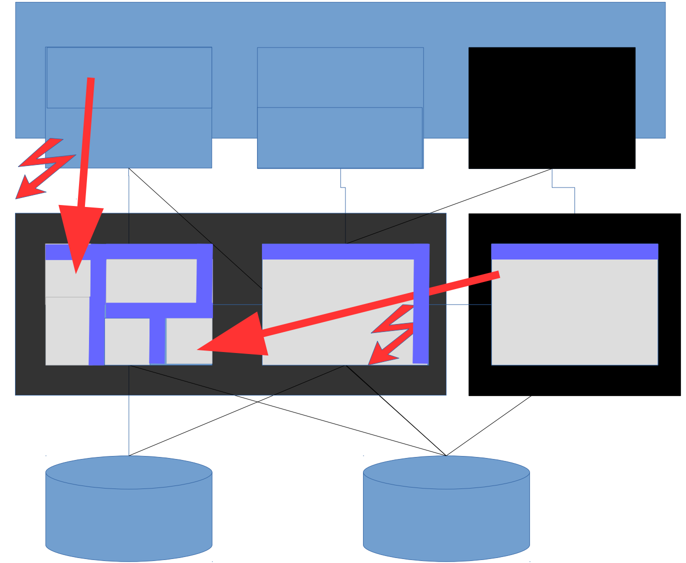


trivadis
makes IT easier.

Breaking up the Monolith (2)

How to add bulkheads in a controlled way

- Analyze your Domains (DDD)
- Identify technical components
- Analyse Dependencies
- Identify Candidates for Isolation
- Define Component Bounderies and APIs
- Hide implementation code
- Discuss impact and issues raised
- Use static code analysis to enforce
- Stabilize
- Do not break up your system (yet)
- Repeat this process!

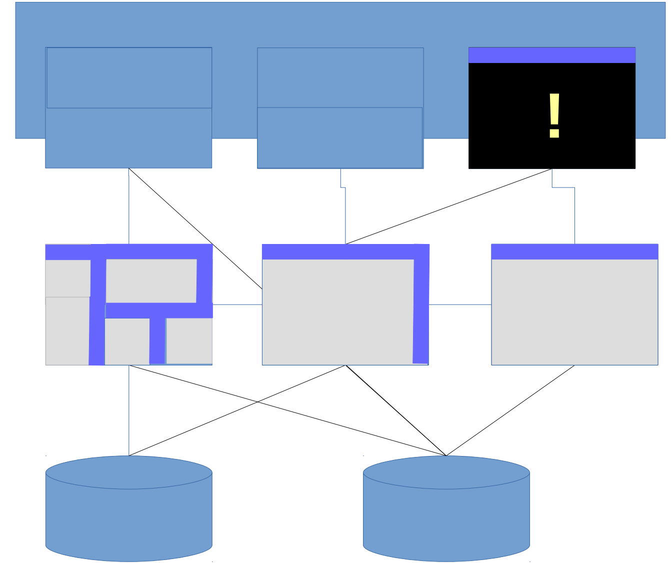


trivadis
makes IT easier.

Breaking up the Monolith (3)

And the *black* components left ?

- Throw away
- Reimplement
- Keep them!
  - But ensure
    clear and minimal APIs!
  - Deprecate them



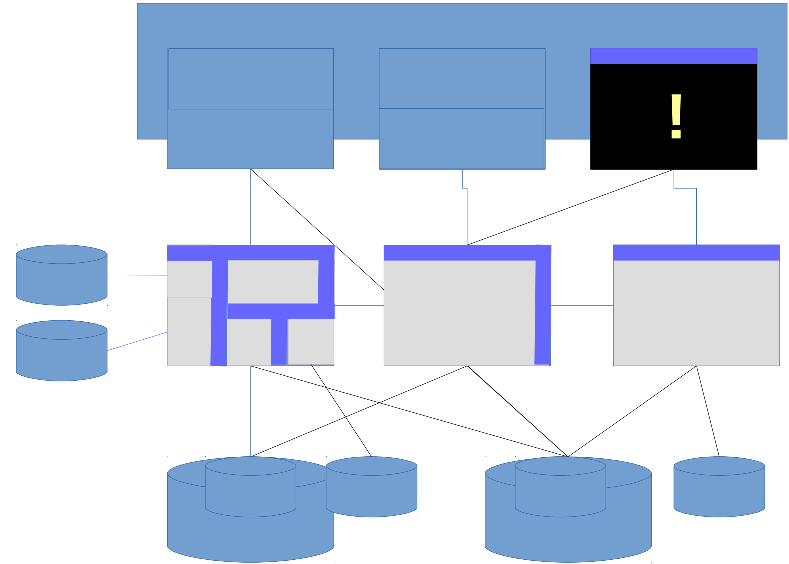
trivadis
makes IT easier.

Breaking up the Monolith (4)

Decouple the database...

- Change Security Credentials and wait for shouts!
- Discuss DB transaction design
- Discuss failure handling
- Separate DBs

→ also here: start small and expect failure!

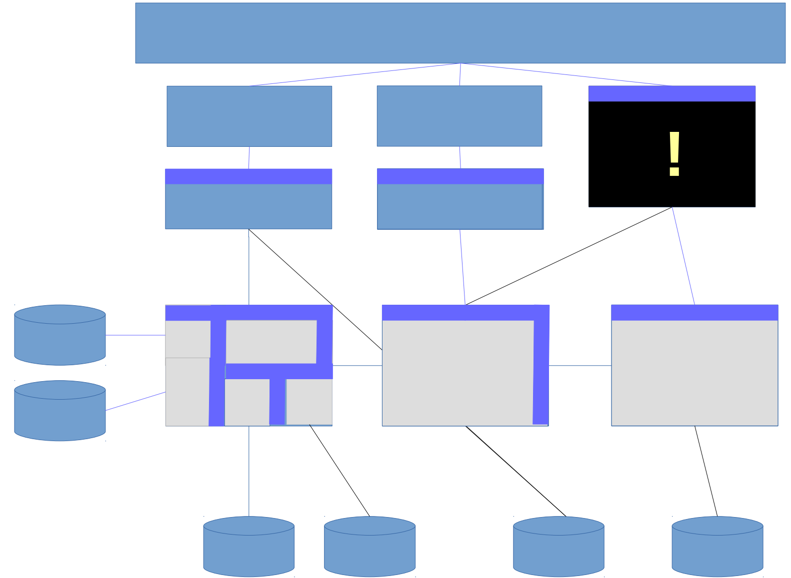


trivadis makes IT easier.

Breaking up the Monolith (5)

Decouple the UI...

- Apply Backend for the Frontend Pattern
- Separate Apps

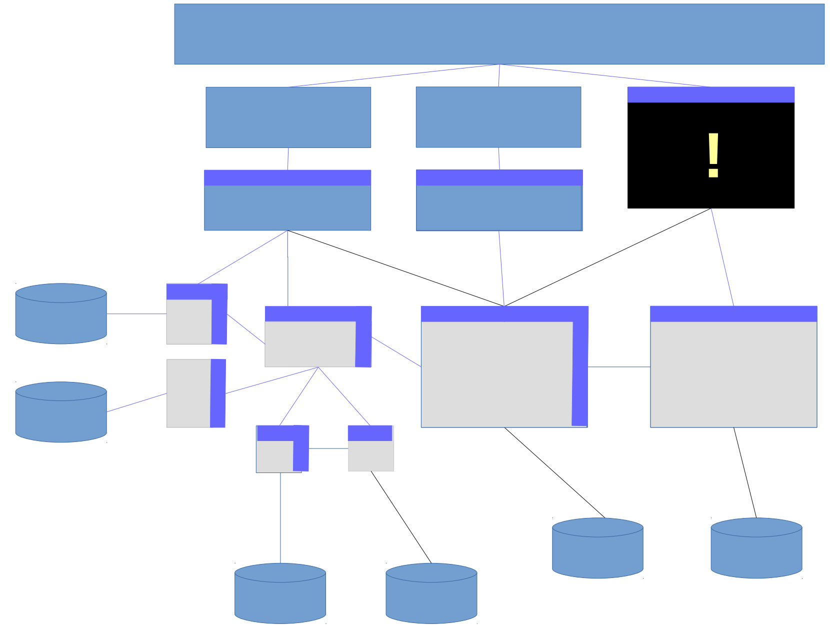


trivadis makes IT easier.

Breaking up the Monolith (6)

And finally you can try to separate things...

→ APIs are crucial that this works!



trivadis makes IT easier.

Aspects not discussed

- Use tool chain to enforce isolation
- Version Management and Coexistence
- Parallel Implementation of Features
- Project Organisation & Governance Aspects
- How to deal with Business Continuity (especially on the beginning)
- ...

trivadis makes IT easier.

API Design

trivadis makes IT easier.

What makes an API a good API ?

- Self-explaining: You don't have to read the documentation.
- Useful - it addresses a need that is not already met (or improves on existing ones)
- Good default behavior - where possible allow extensibility and tweaking
- Excellent documentation
- Sample uses and working sample applications (very important)
- Keep functionality sets distinct and isolated with few if any dependencies.
- Keep it small or segment it so that it is not one huge polluted space.

trivadis
makes IT easier.

But there is more

Good APIs are...

- Minimalistic, technology agnostic with minimal or no dependencies
- Strictly interface based (services)
- Parameters and return types are immutable value types or built-in types
- All parameters are serializable or designed for remoting
- Complete, for completeness an accessor singleton maybe useful, but including a delegation SPI
- Thread-safe

trivadis makes IT easier.

Principles - General Principles

- API Should Do One Thing and Do it Well
- API Should Be As Small As Possible But No Smaller
- Implementation Should Not Impact API
- Minimize Accessibility of Everything
- Names Matter–API is a Little Language
- Documentation Matters, Document Religiously
- Consider Performance Consequences of API Design Decisions
- Effects of API Design Decisions on Performance are Real and Permanent
- API Must Coexist Peacefully with Platform

trivadis
makes IT easier.

Method Design Principles

- Don't Make the Client Do Anything the Module Could Do
- Don't Violate the Principle of Least Astonishment
- Fail Fast - Report Errors as Soon as Possible After They Occur
- Provide Programmatic Access to All Data Available in String Form
- Overload With Care
- Use Appropriate Parameter and Return Types
- Use Consistent Parameter Ordering Across Methods
- Avoid Long Parameter Lists
- Avoid Return Values that Demand Exceptional Processing

trivadis makes IT easier.

Class Design Principles

- Minimize Mutability
- Subclass Only Where it Makes Sense
- Design and Document for Inheritance or Else Prohibit it

trivadis
makes IT easier.

Package Design Principles

- Define a package for each functional module, e.g. `com.mycomp.app.auth`
- Move implementation details into `com.mycomp.app.auth.internal`
- Move spi interfaces into `com.mycomp.app.auth.spi`
- DON'T SEPARATE ARTIFACTS BASED on TYPE, e.g.
  - `com.mycomp.app.auth.beans`
  - `com.mycomp.app.auth.controller`
  - `com.mycomp.app.auth.model`

    → Typically this violates *cohesion* semantics

    → Use corresponding class names instead, e.g. `AuthController`, `AuthModel`

trivadis
makes IT easier.

And even more...

We will look now at examples for the following:

1. Establish strong terms (what is a *Helper, Utility, …???*)
2. Apply symmetry to term combinations
3. Add convenience through overloading
4. Consistent argument ordering
5. Establish return value types
6. Be careful with exceptions
7. Follow the KISS principle
8. Think on type safety
9. Avoid over-engineering
10. Hide internal at all price
11. Only pass Data or functional types
12. Make APIs Remote Capable

trivadis
makes IT easier.

Rule #1: Establish strong terms

- Use same names for similar functionality
- Example JDBC `Statement`, execution of a statement, closing:

```
execute(String)
executeBatch()
executeQuery(String)
executeUpdate(String)
Connection.close()
Statement.close()
ResultSet.close()
```

- `close()` even ended up in `Closeable, AutoCloseable`

trivadis makes IT easier.

Rule #1: Establish strong terms - Violations

- Example of a rule violation in the JDK: `java.util.Observable`
- Normally collection like types declare:

```
size();
remove();
removeAll();
```

- But `Observable` declares:

```
countObservers();
deleteObserver(Observer);
deleteObservers();
```


trivadis makes IT easier.

Rule #1: Establish strong terms - Violations

- Example of rule violation: the Spring framework. Enjoy:

```
AbstractBeanFactoryBasedTargetSourceCreator
AbstractInterceptorDrivenBeanDefinitionDecorator
AbstractRefreshablePortletApplicationContext
AspectJAdviceParameterNameDiscoverer
BeanFactoryTransactionAttributeSourceAdvisor
ClassPathScanningCandidateComponentProvider
J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource
```

→ *Creator* vs. *Factory*, a *Source* vs. *Provider, vs Advisor vs Discoverer ???*
→ Is an *Advisor* related to an AspectJAdvice?
→ Is it a *ScanningCandidate* or a *CandidateComponent*?
→ *TargetSource* vs. *SourceTarget, or even a SourceSource* or a *SourceSourceTargetProviderSource*?

trivadis
makes IT easier.

Rule #2: Apply symmetry to term combinations

- Example:

```
add(E)
addAll(Collection<? extends E>)
remove(Object)
removeAll(Collection<?>)
contains(Object)
containsAll(Collection<?>)
```

- Violation: java.util.Map
  - `keySet()` → `containsKey(Object)`
  - `values()` → `containsValue(Object)`
  - `EntrySet()` → ***missing*** `containsEntry(K, V)`

trivadis makes IT easier.

Rule #3: Add convenience through overloading

- Example:

  `toArray()`, which is a convenient overload of…
  `toArray(T[])`

- Violation: `java.util.TreeSet`
  - `TreeSet(Collection<? extends E>)`
  - `TreeSet(SortedSet<E>)` → adds some convenience to the first in that it extracts a well-known `Comparator` from the argument `SortedSet` to preserve ordering, which results in different behavior:

```
SortedSet<Object> original = // [...]
// Preserves ordering
new TreeSet<Object>(original);
// Resets ordering
new TreeSet<Object>((Collection<Object>) original);
```


trivadis
makes IT easier.

Rule #4: Consistent argument ordering

- Example `java.util.Arrays` (array is always first):

```
copyOf(T[], int), which is an incompatible overload of…
copyOf(boolean[], int)
copyOf(int[], int)
binarySearch(Object[], Object)
copyOfRange(T[], int, int)
fill(Object[], Object)
sort(T[], Comparator<? super T>)
```

- Violation: `java.util.Arrays` by „subtly” putting optional arguments in between other arguments, when overloading methods:

```
fill(Object[], Object)
fill(Object[], int, int, Object)
```

trivadis
makes IT easier.

Rule #5: Establish return value types

- Methods returning a single object should return `null` when no object was found
- Methods returning several objects should return an empty `List, Set, Map, array,` etc. when no object was found (never `null`)
- Methods should only throw exceptions in case of an … well, an exception

- Violation: `java.util.File.list()`:
  Javadocs: *An array of strings naming the files and directories in the directory denoted by this abstract pathname. The array will be empty if the directory is empty.* ***Returns null*** *if this abstract pathname does not denote a directory,* ***or*** *if an I/O error occurs.*

trivadis
makes IT easier.

Rule #6: Be careful with exceptions

- Example: **`java.sql.ResultSet extends java.sql.Wrapper`**:

```
public interface Wrapper {
    <T> T unwrap(java.lang.Class<T> iface)
        throws java.sql.SQLException;
    boolean isWrapperFor(java.lang.Class<?> iface)
        throws java.sql.SQLException;
}
```

→ Exceptions are likely to leak into client code...

→ Use `RuntimeException` if possible,

trivadis
makes IT easier.

Rule #6: Be careful with exceptions

- Be careful with exceptions, client code:

```
public void myMethod(){    throws Exception{
  ResultSet rs = …;
  if(rs.isWrapperFor(X.class){
    …;
  }
}
```

How to handle the exception here?

→ The exception gets an integral part of your API as well!

trivadis makes IT easier.

Rule #7: Follow the KISS principle

Multiple concerns: **`java.util.Preferences`**:

```
public abstract class Preferences {
    private static final PreferencesFactory factory = factory();
    private static PreferencesFactory factory() {...}
    private static PreferencesFactory factory1() {...}

    public static final int MAX_KEY_LENGTH = 80;
    public static final int MAX_VALUE_LENGTH = 8*1024;
    public static final int MAX_NAME_LENGTH = 80;

    public static Preferences userNodeForPackage(Class<?> c) {...}
    public static Preferences systemNodeForPackage(Class<?> c) {---}
    private static String nodeName(Class<?> c);
    private static Permission prefsPerm = new RuntimePermission("preferences");

    public static Preferences userRoot();
    public static Preferences systemRoot();


    protected Preferences() {}

    public abstract void put(String key, String value);
    public abstract String get(String key, String def);
    public abstract void remove(String key);
    [,,,]
```

→ Management of factories and user and system tree

→ Implementation details

→ Mapping artifacts to nodes

→ Singleton access to trees

→ Node base class (not an interface!)

→ Mutability is built in already ;(

trivadis

makes IT easier.

Rule #7: Follow the KISS principle

Multiple concerns: **`java.util.Preferences`**:

```
[…]

protected Preferences() {}

public abstract void put(String key, String value);
public abstract String get(String key, String def);
public abstract void remove(String key);
public abstract void clear() throws BackingStoreException;
public abstract void putInt(String key, int value);
public abstract int getInt(String key, int def);
[...]
public abstract void putByteArray(String key, byte[] value);
public abstract byte[] getByteArray(String key, byte[] def);
public abstract String[] keys() throws BackingStoreException;
public abstract String[] childrenNames()
   throws BackingStoreException;
public abstract Preferences parent();
public abstract Preferences node(String pathName);
public abstract boolean nodeExists(String pathName)
    throws BackingStoreException;
public abstract void removeNode() throws BackingStoreException;
public abstract String name();
public abstract String absolutePath();
public abstract boolean isUserNode();
[…]
```

→ Multi type value support

→ Tree navigation
→ Path/Node translation
→ Tree manipulation
→ Import/Export
→ Utility functions

trivadis
makes IT easier.

Rule #7: Follow the KISS principle

Multiple concerns: **`java.util.Preferences`**:

```
  […]

  public abstract void flush() throws BackingStoreException;
  public abstract void sync() throws BackingStoreException;
  public abstract void addPreferenceChangeListener(
      PreferenceChangeListener pcl);
  public abstract void removePreferenceChangeListener(
      PreferenceChangeListener pcl);
  public abstract void addNodeChangeListener(NodeChangeListener ncl);
  public abstract void removeNodeChangeListener(NodeChangeListener ncl);
  public abstract void exportNode(OutputStream os)
      throws IOException, BackingStoreException;
  public abstract void exportSubtree(OutputStream os)
      throws IOException, BackingStoreException;

  public static void importPreferences(InputStream is)
      throws IOException, InvalidPreferencesFormatException{}
}
```

→ Tree manipulation
→ Observer pattern

→ Import/Export

trivadis
makes IT easier.

Rule #8: Think on type safety

```
private String date;

public void setDate(String date);
public String getDate(String date);
```

- Reason for this design:
  custom formats/host APIs
- Flaws:
  → Unclear format
  → No validation
  → Missing datatype
- Alternatives:
  - Use JDK time API or JodaTime

trivadis makes IT easier.

Rule #9: Avoid over-engineering

```
public class UID{
  private String uid;
  public UID(String uid){
    this.uid = uid;
  }
  public String getUid(){
    return uid;
  }
  public void setUid(String uid){
    this.uid = uid;
  }
}
```

- Reason for this artifact: type safety
- Flaws:
  - → Not thread safe
  - → No validation
  - → No documentation
  - → Adds unnecessary complexity
- Alternatives:
  - → URIs, e.g.

```
new URI(“user:a123456“);
```

trivadis
makes IT easier.

Rule #10: Hide internals at all price

```
public InternalDataSetImpl getDataset(
  String setId, String location,
  String userId, String databaseName,
  String... rules);

public void performAction();

public String manage(Object... args);
```

- Flaws:
  - → Exposes internal data
  - → Requires internal data as input
  - → Parameter types are error prone
  - → Meaningless method name
  - → Unconstraint parameters and return types

trivadis makes IT easier.

Rule #11: Only pass data or functional types

```
public interface MyFancyService{

  public XY evaluateXY(
    Param1 param,
    EvaluationService service);
  …
}
```

- Flaws:
  - → Requires passing a service
  - → not remote capable
- Improvements:
  - → Remove service argument
  - → It is an implementation detail
- Note:
  - → OK for simple functional interfaces
  - → Be aware this is broken with remoting!

trivadis
makes IT easier.

Rule #12: Make APIs Remote Capable

- JSON enabled Datatypes
  - Strings
  - Numbers
  - Boolean
  - Arrays
  - Maps (JSON Object)

  → So everything must be mappable

  to this few things!

trivadis
makes IT easier.

API Design – Summary

- API Design is the key discipline of software engineering
- Start small!
- Be very causcious to add functionality
  - is it the same concern
  - can't it be done with existing functionality
- If the API feels too complicated, it definitively IS!
- Be aware that you have to rework it at least 3 times to get it right
- **Involve colluegues that have experience building public APIs**
- **Eat your own dog food!**

trivadis makes IT easier.

Summary

trivadis
makes IT easier.

Summary

- Microservices, Containers and Resilient Design are the Future.
- Monoliths are inherently broken for building highly scalable, resilient systems.
- Decoupling affects everything: your infrastructure, your organization, your data, your skills.
- It's a long and difficult path.
- You will probably fail.
- But IMO you dont have a choice *
- Act now!

* compare the impact on taxi companies when Uber.com entered the market

trivadis
makes IT easier.

Referenzen

- Sam Newman: Building Microservices. O'Reilly, 2015
- Martin Fowler: Microservices. http://martinfowler.com/articles/microservices.html
- Resilient Software Design:

  https://jaxenter.de/unkaputtbar-einfuehrung-resilient-software-design-15119
- API Design in Java: https://dzone.com/articles/how-design-good-regular-api

trivadis
makes IT easier.

Monolithen für die Zukunft trimmen

Anatole Tresch
Principal Consultant

Tel. +41 58 459 53 93
anatole.tresch@trivadis.com

trivadis
makes IT easier.