Patterns for Modularity

Anton Epple, Eppleton IT Consulting
Jaroslav Tulach, NetBeans Team, Oracle, Prague
Zoran Sevarac, Faculty of Organizational Sciences, University of Belgrade

„a general reusable solution to a commonly occurring problem in software design“

When creating modular Applications there are commonly occurring problems

But where are the Design Patterns to solve them?

OSGi, Spring, NetBeans & others suggest different solutions for some of these problems

Let's discuss them!

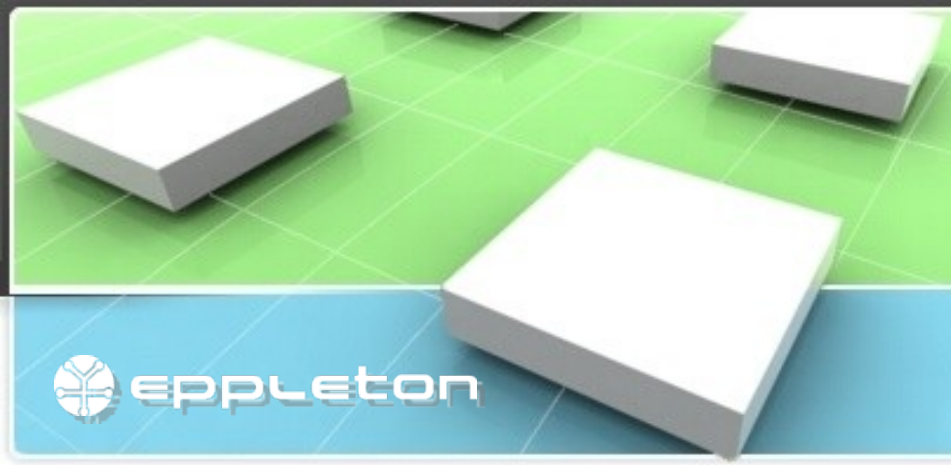
Criteria for Modularity patterns:

- Maximize reuse
- Minimize coupling
- Deal with change
- Ease Maintenance
- Ease Extensibility
- Save resources

Logical design versus physical design

Problem Domain 1: Relationships

Managing Dependencies

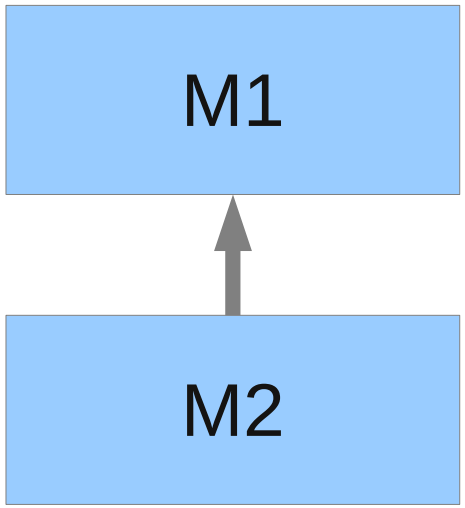


Managing Dependencies

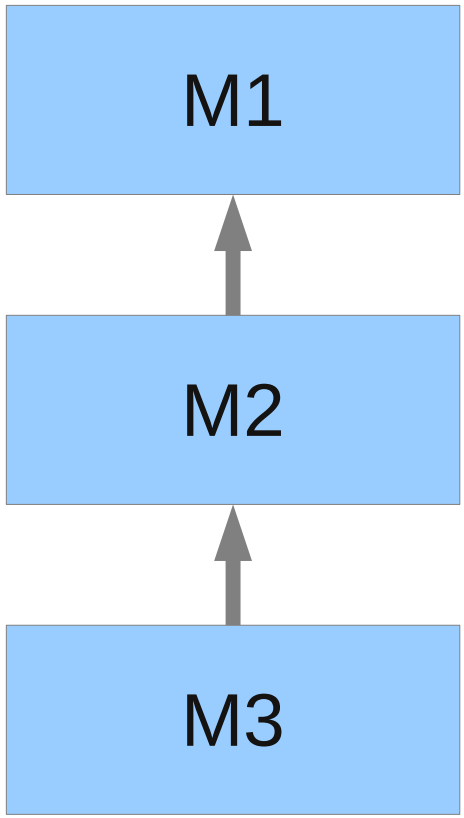
The Basics: Managing Dependencies

- Direct dependencies
- Indirect dependencies
- Cycles
- Incoming versus Outgoing dependencies
- Classical Design Patterns and Modules

Types of Dependencies

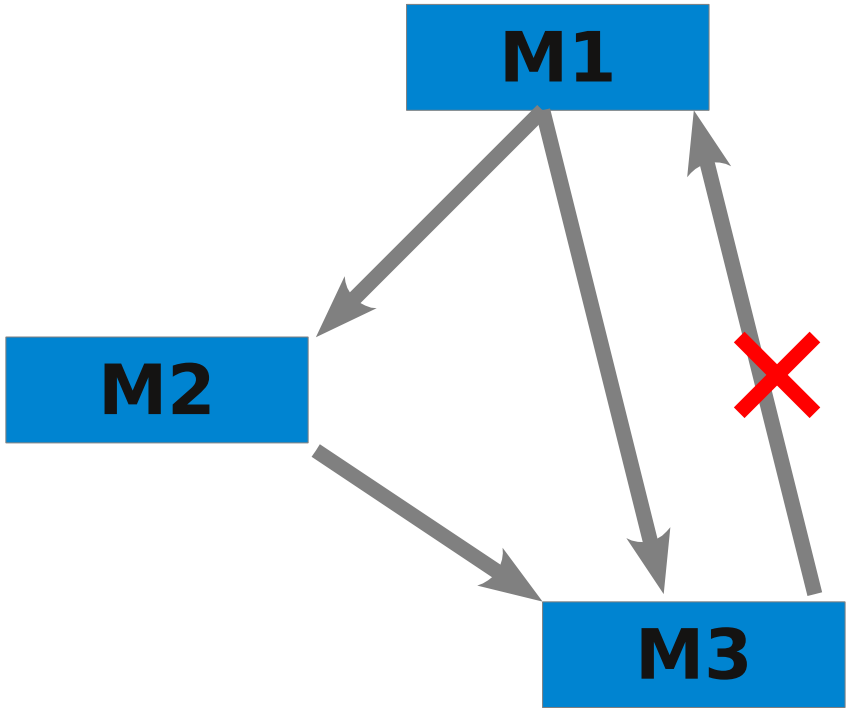


Direct Dependency



Indirect Dependency

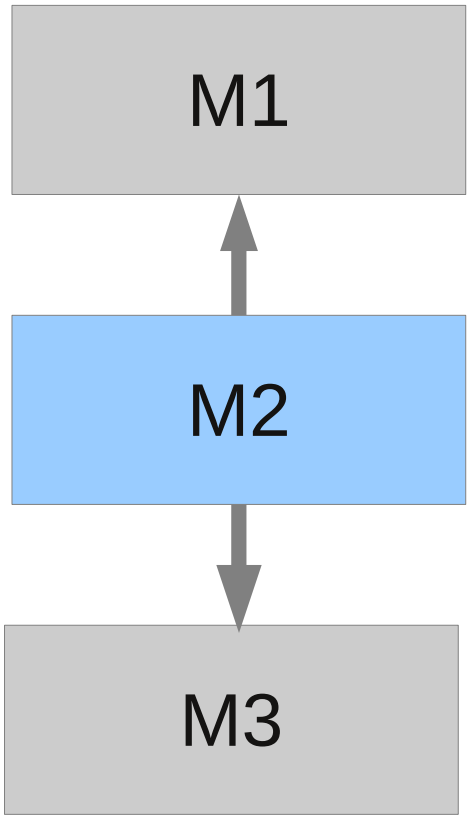
Module relationships should be uni-directional



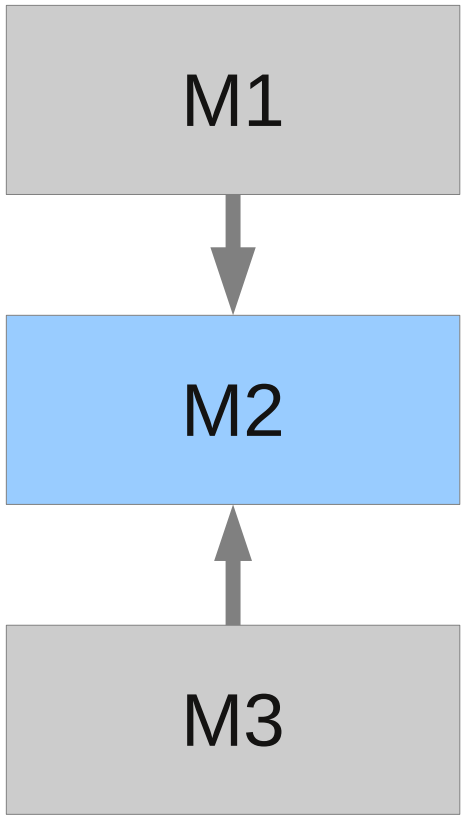
No cycles

eppleton

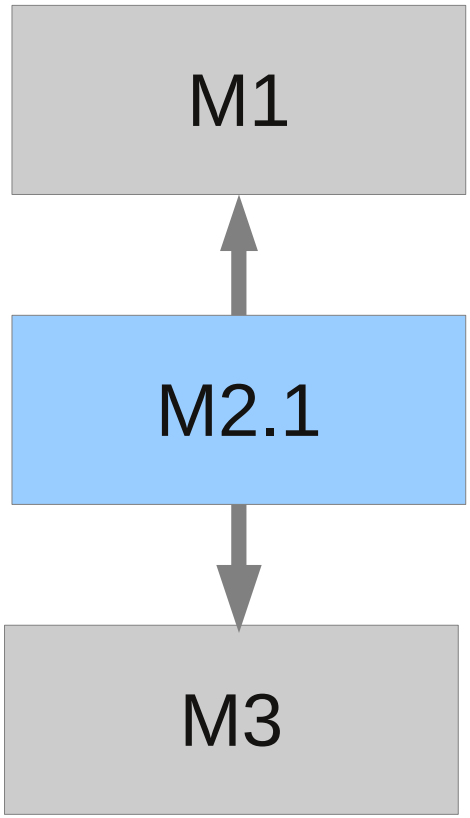
Types of Dependencies



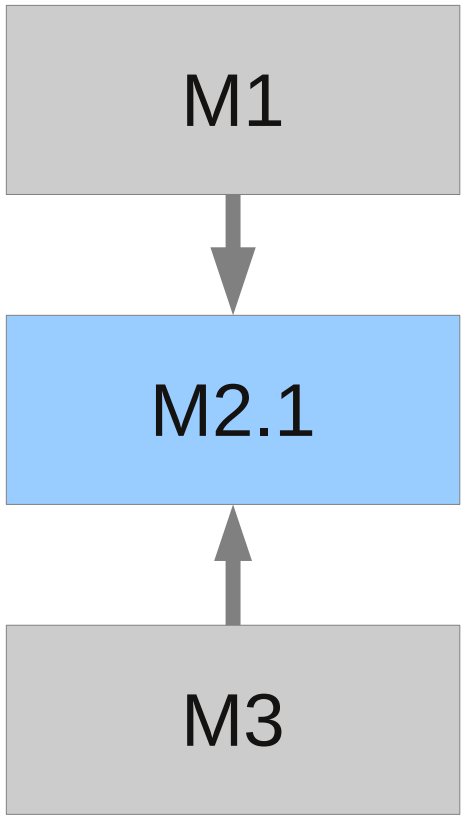
Outgoing Dependency



Incoming Dependency



Easy to Change



Hard to change

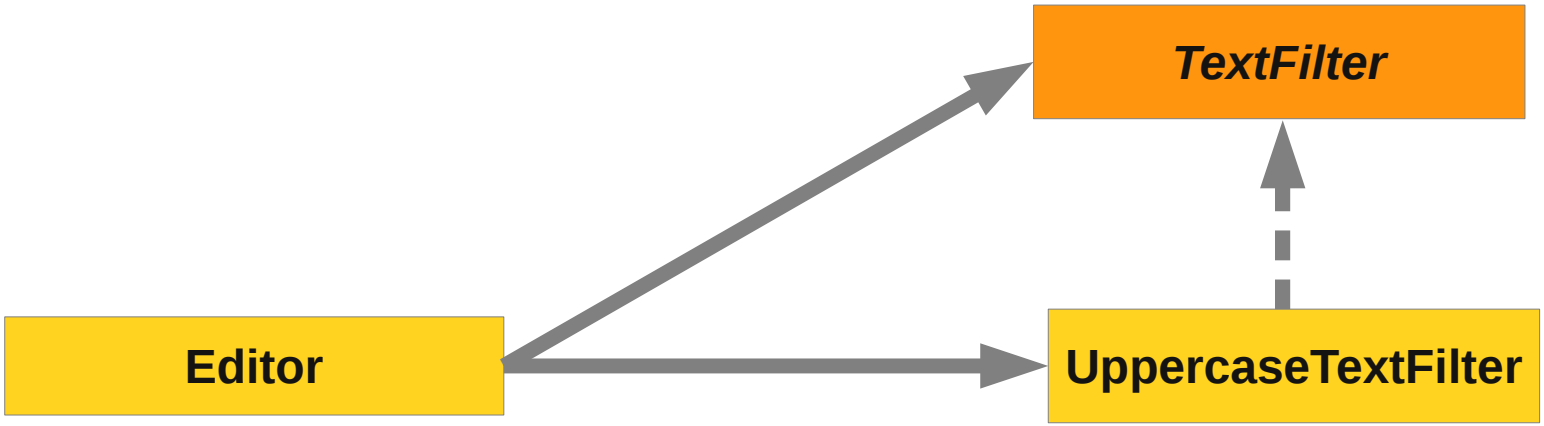
Applying practice from software design patterns

- Adapter – adapt interface between two modules
- Mediator – holds interaction between two or more modules = bridge
- Facade – provides front interface for the set of modules

Problem Domain 2: Communication

Service Providers and Consumers

Reducing Dependencies:



Service Provider Interface, Provider & Registry

Features of Service Infrastructures

- Registering Services
- Retrieving Services
- Disabling Services
- Replacing Services
- Ordering Services
- Declarativeness: Metainformation for Services
- Declarativeness:Codeless Services
- Availability of required Services

Solutions

- JDK
- OSGi Service Registry
- Declarative Services
- Lookups
- General Dependency Injection
- (Spring Dynamic Modules, iPojo, Peaberry, Blueprint Services)

JDK Solution: ServiceLoader

- Declarative Registration in META-INF/services

```
ServiceLoader<TextFilter> serviceLoader =
ServiceLoader.load(TextFilter.class);

for (TextFilter filter : serviceLoader) {

     String s = filter.process("test");

}
```

Problems:

- ServiceLoader isn't dynamic:
  - What if user installs plugin with new service?
  - What if user uninstalls plugin with service?
- ServiceLoader loads all services at startup:
  - What about startup time?
  - What about memory usage?
- No Configuration
    - Standard Constructor
    - No support for Factory Methods or Factory
- No ranking

NetBeans Solution: Lookups and XML-files

- Declarative Registration & Configuration
  - Dynamic (LookupListener)
  - Ordering (position attribute)
  - Lazy Loading
  - Factories and Factory Methods
  - Configuration via Declaration
  - Compatible with ServiceLoader (META-INF/services)
  - Codeless Extensions

OSGi Solution: ServiceRegistry

- Registered with code:

```
Long i = new Long(20);

Hashtable props = new Hashtable();

props.put("description", "This an long value");

bundleContext.registerService(Long.class.getName(),
i, props);
```

Benefits:

- Dynamic (ServiceTracker)
- Factories supported
- Filters
- Configuration via code

Problems:

- Registration introduces dependency to framework
- Eager creation: increase complexity, memory footprint
- Not Typesafe, casting required

OSGi Evolution: Declarative Services

- Declarative Registration

```
<component name="samplerunnable">
<implementation class="org.example.ds.SampleRunnable"/>
<service>
<provide interface="java.lang.Runnable"/>
</service>
</component>
```

OSGi Evolution: Declarative Services

- Declarative Registration

```
<component name="samplerunnable">
<implementation class="org.example.ds.SampleRunnable"/>
<service>
<provide interface="java.lang.Runnable"/>
</service>
</component>
```

- No support for codeless extensions

Dependency Injection (Spring):

```
public class Editor {
private TextFilter filter;

@Autowired
public void setTextFilter(Filter filter) {
    this.filter = filter;
}
…
```

beans.xml file to register Implementation, Injection by framework.

Finding Implementations of Interfaces

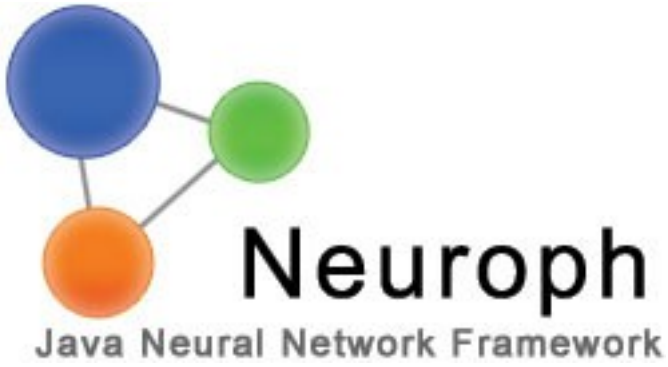
- Dependency Injection (Spring):
  - Static environment
  - Importance of imports
- ServiceLoader/Lookup
  - Dynamic
  - Queries hidden in code
- Are singletons bad?
  - Application context only
  - Injectable singletons

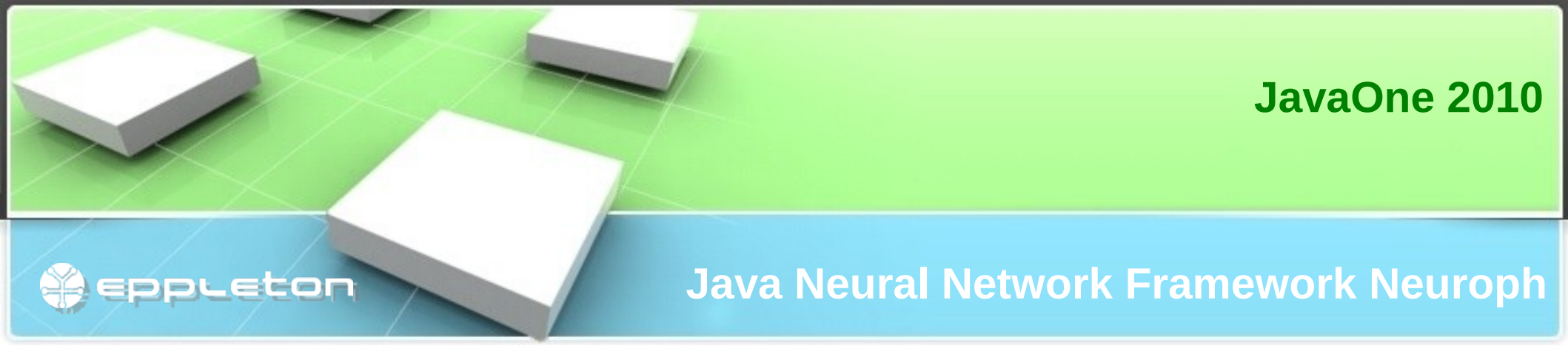
Declarativeness and Speed

- Modular applications are large
- Start time is important on desktop
- Running 3rd party code is dangerous
  - Optimize infrastructure
- Declarative registrations
  - Pull but dont push
  - Importance of imports

Case Study: Neuroph, java neural network

- Porting to NetBeans Module System





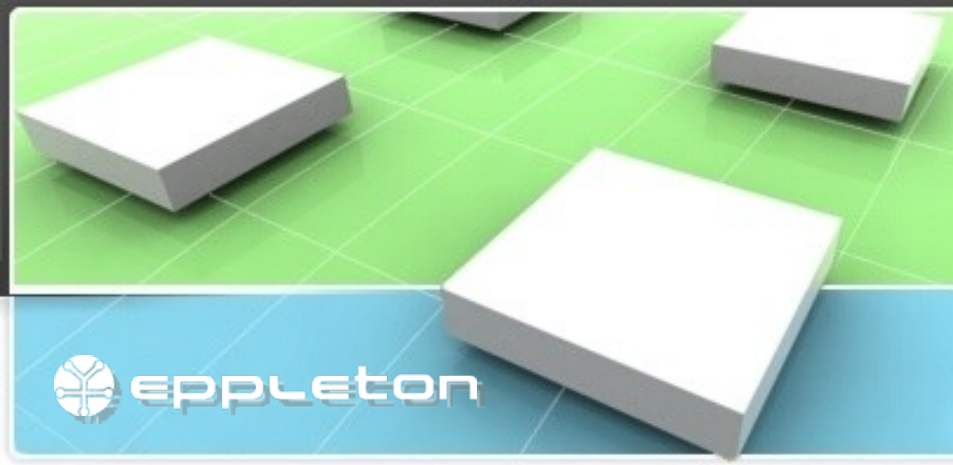
Java Neural Network Framework Neuroph

1. What is Neuroph?
Java Framework for creating neural networks, that can be easily used in Java apps.

2. Key features
- GUI for creating NN
- Java NN library
- Easy to use
- Easy to extend and customize
- Basic tools for: image recognition, OCR, stock prediction

3. Usage:education, research and real world problems (problems like classification, recognition, prediction)

4. Collaboration with Encog and other open source projects



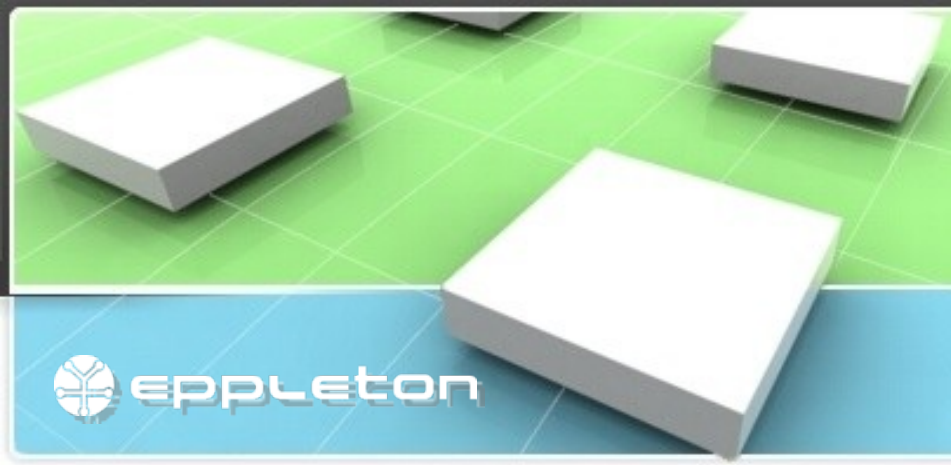
Why Porting to NB platform?

1. At first we wanted nice, professional looking GUI, an IDE for neural networks

2. Later we realized that there is much more to gain from porting:

- Reuse lots of stuff available on NB Platform- like Gephi (for visualization)
- Integration with other apps on NB Platform - like Maltego (NB as the integration platform)s
- Many other usefull features like update, improved design easier to extend and maintain

3. Improved overall quality, competitive advantage and ensured future development



Refactoring to Modules

To Do:

Move existing code to NB Platform/modules

Goal:

Get usable, working app as soon as possible

Main question:

Which modules do we need, how to identify/define modules?

Our approach:

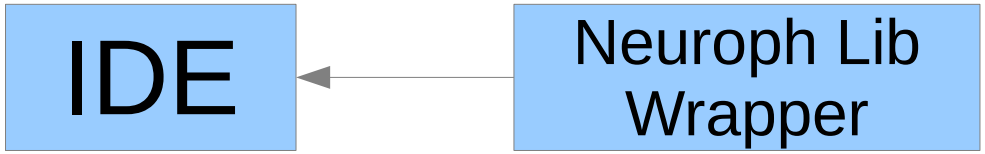
Create few modules to cover basic/core features
Copy/paste/adapt existing code into modules
Create new modules when needed
Do it in an iterative process.

Toni's 5 principles:

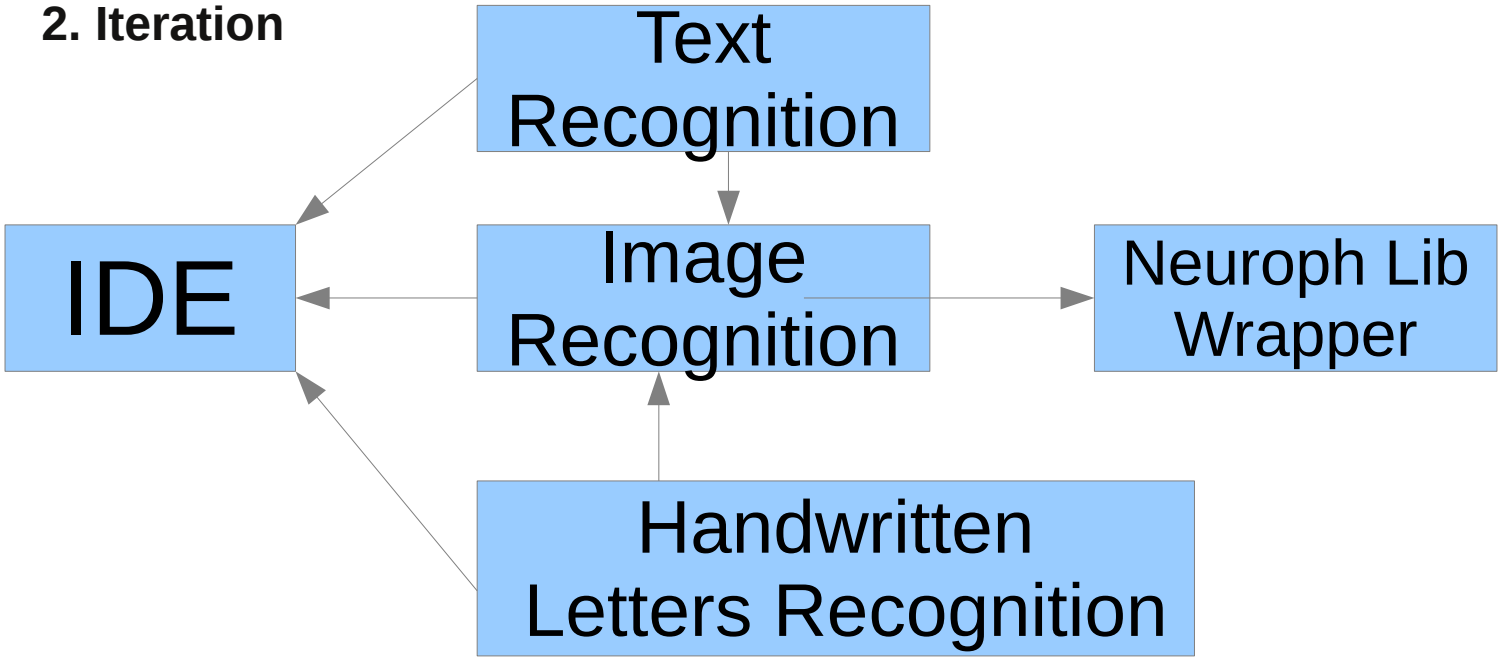
Maximize reuse, minimize coupling, deal with change, ease maintainance, ease extensibility

Module diagram

1. Iteration

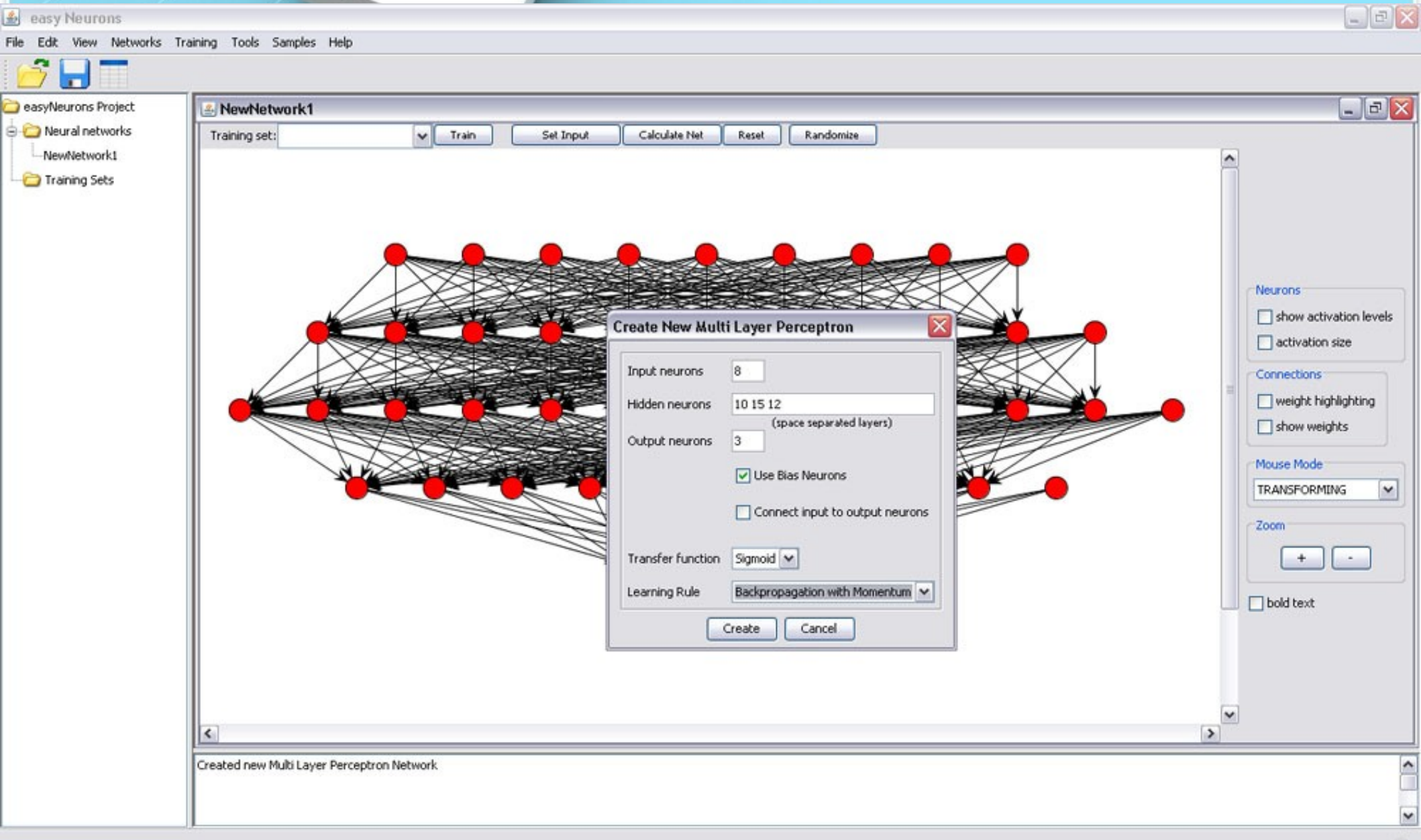


2. Iteration



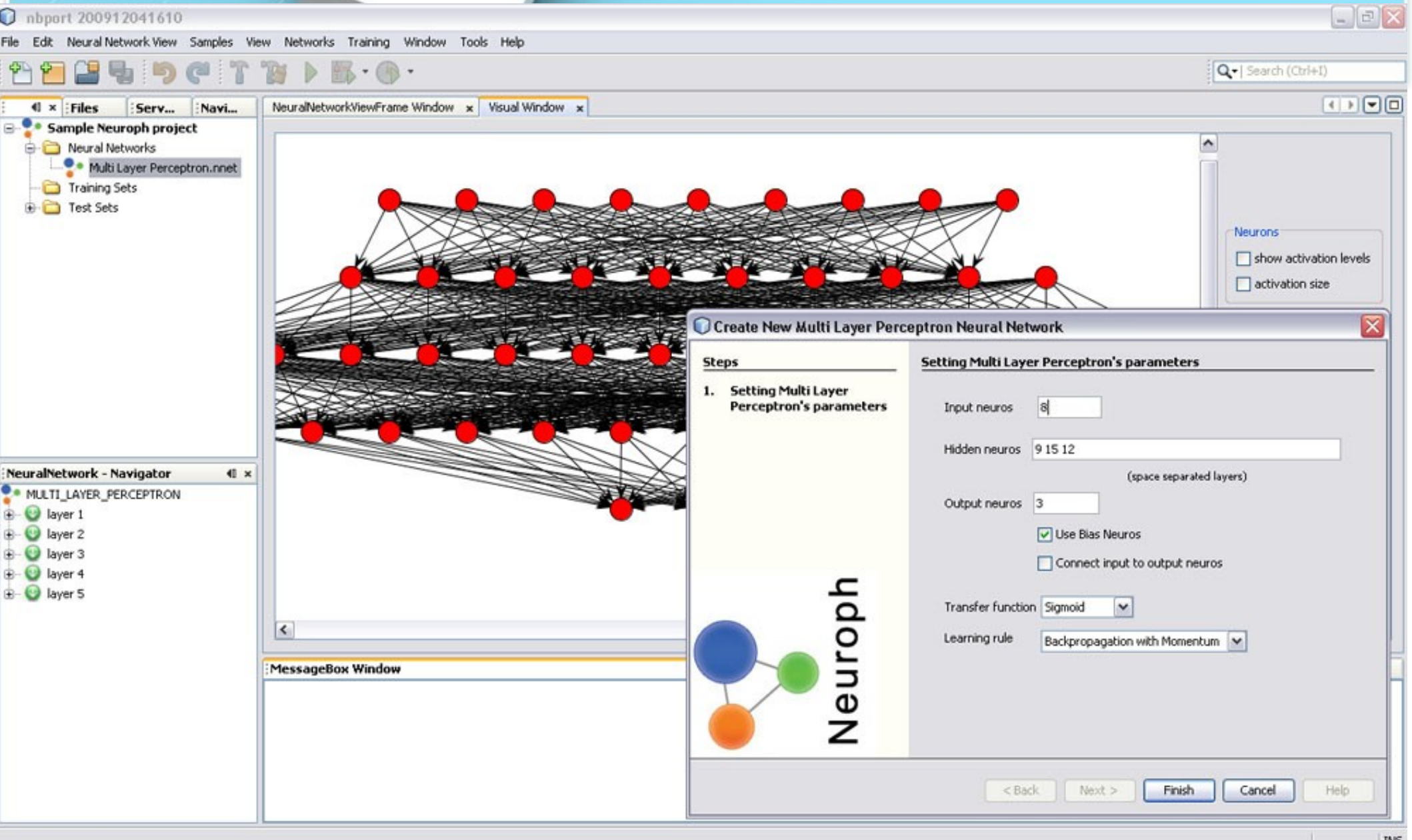
JavaOne 2010

before porting to NetBeans Platform



JavaOne 2010

after porting

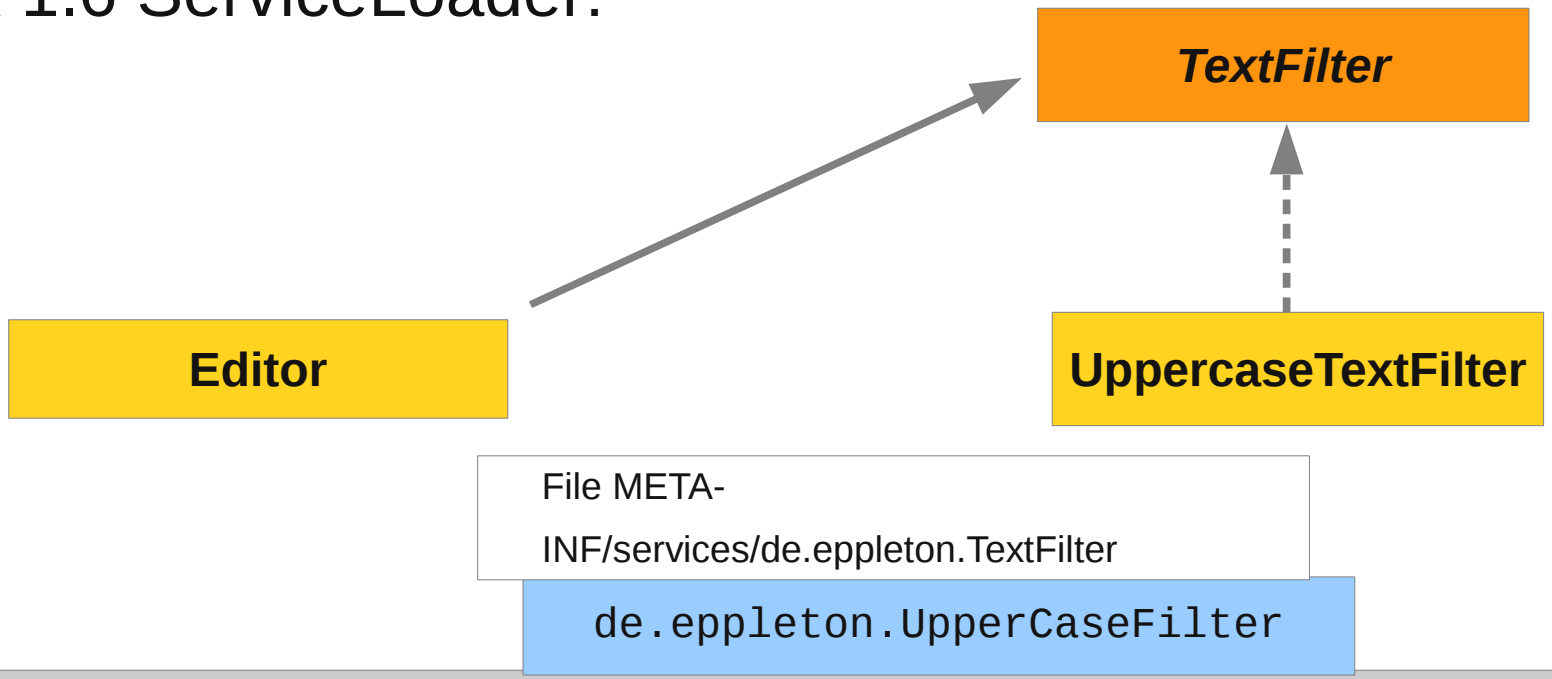


- Dependency Management
- Service Infrastructures

- Many more topics: general API design, compatibility issues...

Q&A

JDK 1.6 ServiceLoader:




```
ServiceLoader<Device> serviceLoader =
ServiceLoader.load(TextFilter.class);
for (TextFilter filter : serviceLoader) {
   String s = filter.process("test");
}
```

Service Infrastructure

- NetBeans, OSGi & Eclipse Services:

	NetBeans	Declarative Services	Extension Points
1:n Service <-> Extension Point	+	+	-
codeless Extensions	+	-	+
Documentation	-/+ ApiDoc	+	+

Designing for backward Compatibility

- Abstract class versus Interface
- Composition versus Inheritance

Composition versus Inheritance

Composition versus Inheritance

- BirdInterface:

void fly();

void quack();

void swim();

void talk();

void run();

Composition versus Inheritance

Composition versus Inheritance

- Empty implementations
- Not easy to enhance:

void crow(); // breaks compatibility

Composition versus Inheritance

Composition versus Inheritance

- Solution split Interface
- Not dynamic:

Composition versus Inheritance

Composition versus Inheritance

- Object „has a“ Capability (e.g. SaveCapability: Editor can be saved)
- Add/remove SaveCapability to Lookup:

```
public interface SaveCapability{
     public void save();
}
```

  - Add SaveCapability on Editor change
  - Save Button listens for Capability
  - Remove capability after save is performed

Dealing with incompatible changes

- Avoid them as shown before
- Source, Binary and Functional Compatibility
- Versioning
- Parallel use of different Versions of a Module

Control: What is part of your API?

- A lot of stuff you didn't think about, behaviour, accessible classes, Basically everything
  - Latest Example: Eclipse & JDK Vendor name
- Package-Private
- PublishedInterface
- Information Hiding
  - Friends & Buddies