Patterns for Modularity II:
Revenge Of the patterns

Zoran Sevarac, Faculty of Organizational Sciences, University of Belgrade

Jaroslav Tulach, NetBeans Team, Oracle, Prague

Anton Epple, Eppleton IT Consulting

Module Systems

- OSGI
- NetBeans Platform
- Jigsaw

Software Design Patterns

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

„patterns are abstraction of experience“

„antipattern is a pattern that may be commonly used but is ineffective and/or counterproductive in practice“

Modularity Patterns

What are commonly occurring problems when creating modular applications?

1) reusability
2) flexibility (change and extensibility with backward compatiblity)
3) dependency

How to evaluate a pattern goodness?

Modularity principles - criteria for evaluating patterns

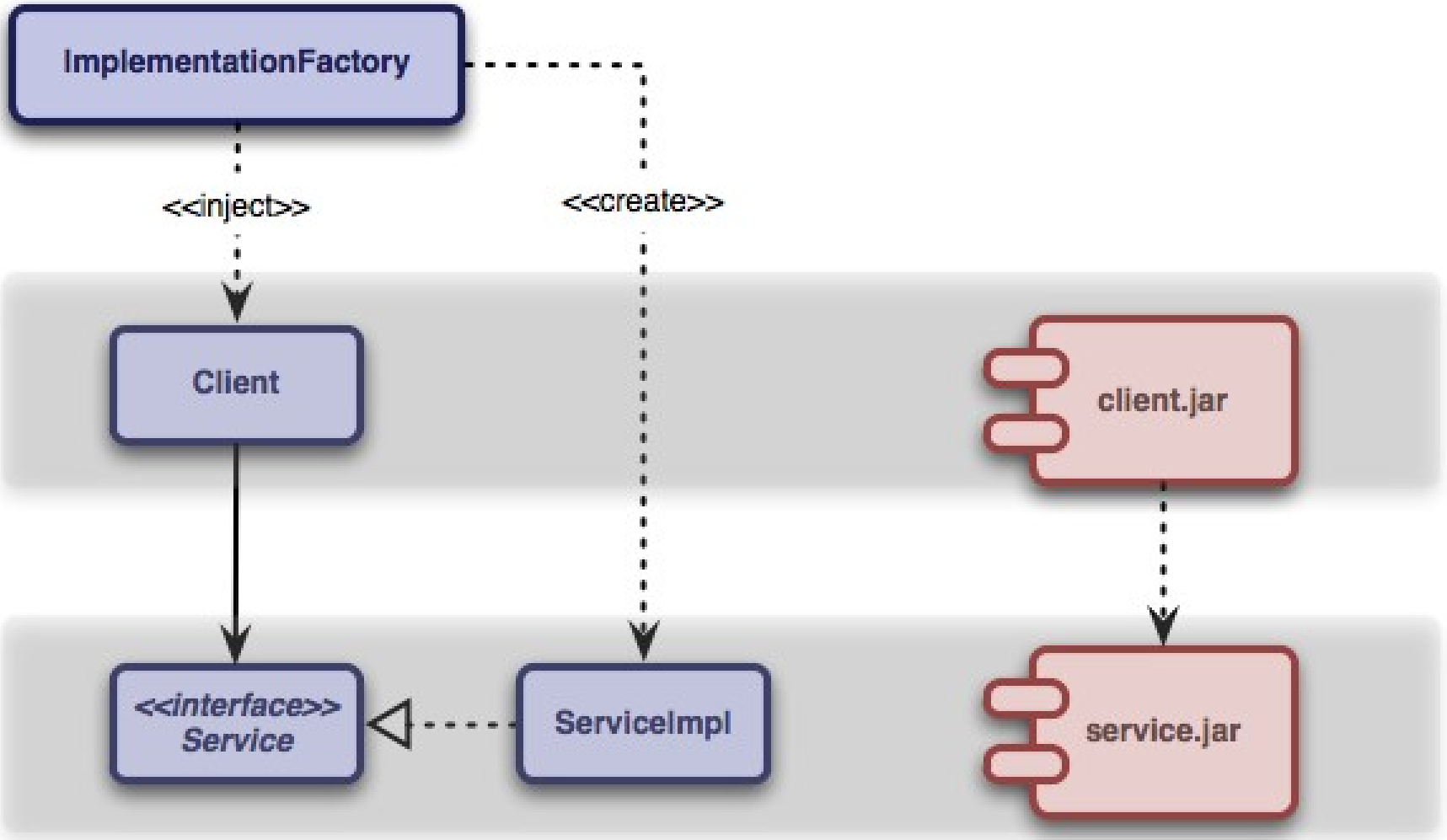
- Maximize reuse
- Minimize coupling
- Deal with change
- Ease Maintenance
- Ease Extensibility
- Save resources

Modularity Patterns

by Kirk Knoernschild

- **ModuleReuse** – Emphasize reusability at the module level.
- **ModuleFacade** – Create a facade serving as a coarse-grained entry point to the modules underlying implementation
- **AbstractModules** – Depend upon the abstract elements of a module.
- **SeparateAbstractions** – Separate abstractions from the classes that realize them.
- **DefaultImplementation** – Provide modules with a default implementation.
- **ImplementationFactory** – Use factories to create a modules implementation classes.

Implementation Factory



Modularity Patterns

by Kirk Knornschild

- **PhysicalLayers** – Module relationships should not violate the conceptual layers.
- **ExternalConfiguration** – Modules should be externally configurable

  (branding, internationalization)
- **ManageRelationships –** Inverting and eliminating dependencies

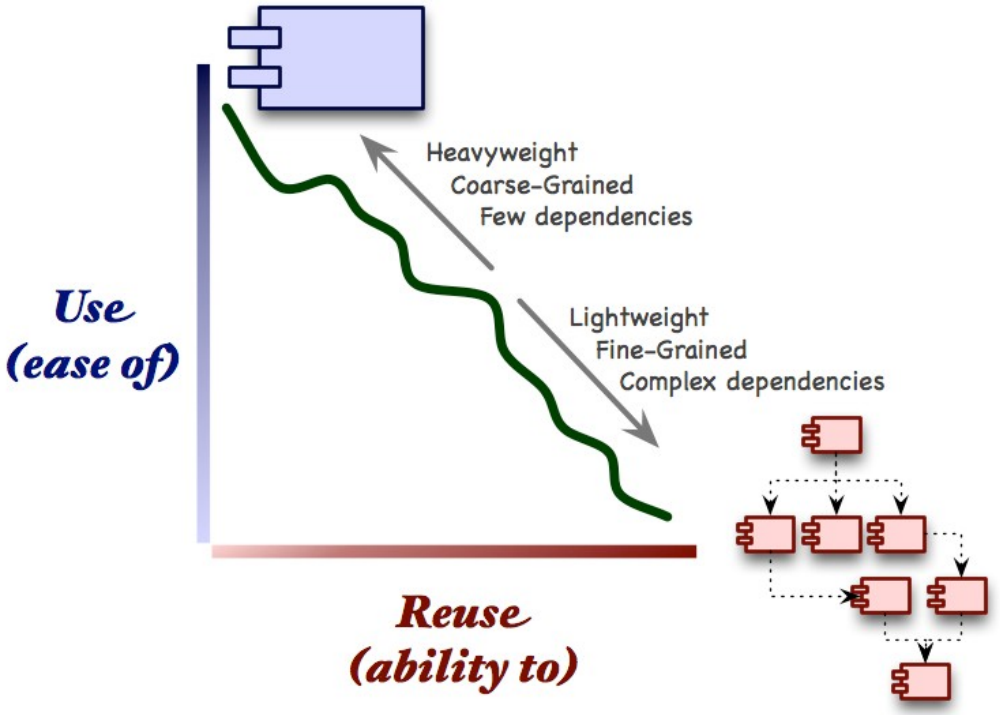
Example

- We want to create reusable software component/module, easy to extend and change
- Starting with few simple classes, then creating abstractions (AbstractModules, SeparateAbstractions, PublishedInterface), breaking it into parts, depend on other modules (ManageRelationships, PhysicalLayers) and add configuration (ExternalConfiguration).
- Since we have few abstractions it will make sense to have ImplementationFactory and DefaultImplementation
- Put ModuleFacade to make it easier to ese by the rest of app
- Use AdapterModule to make it work with existing code

AntiPatterns

- **Over-Generalized Interfaces** - attempt to create systems with infinite flexibility, but succeed only in creating systems that are impossible to maintain
- **FUD Architecture** - The fear of being wrong, or creating an architecture that will change later, results in an architecture that actually solves nothing
- **MeaninglessAbstraction** – Example: RegexPattern extends Pattern
- **NeverReusedReusableModule** - Reusable Module
- **PileOfParts** - Too granular modular architecture
- **BigBallOfMudModule** - Too heavy module, put everything in one module

Reusability



Kirk Knoernschild: *Maximizing reuse complicates use*

Means that increase in reusability, also increases complexity and decreases usability of the software component

Reusability

- **Granularity** - extent to which a system is broken down into parts

  Coarse-grained components are easier to use, but fine-grained components are more reusable.

- **Weight** - extent to which a component depends on other components (dependancies).

  Lightweight components are more reusable, but heavyweight components are easier to use.

Discussion

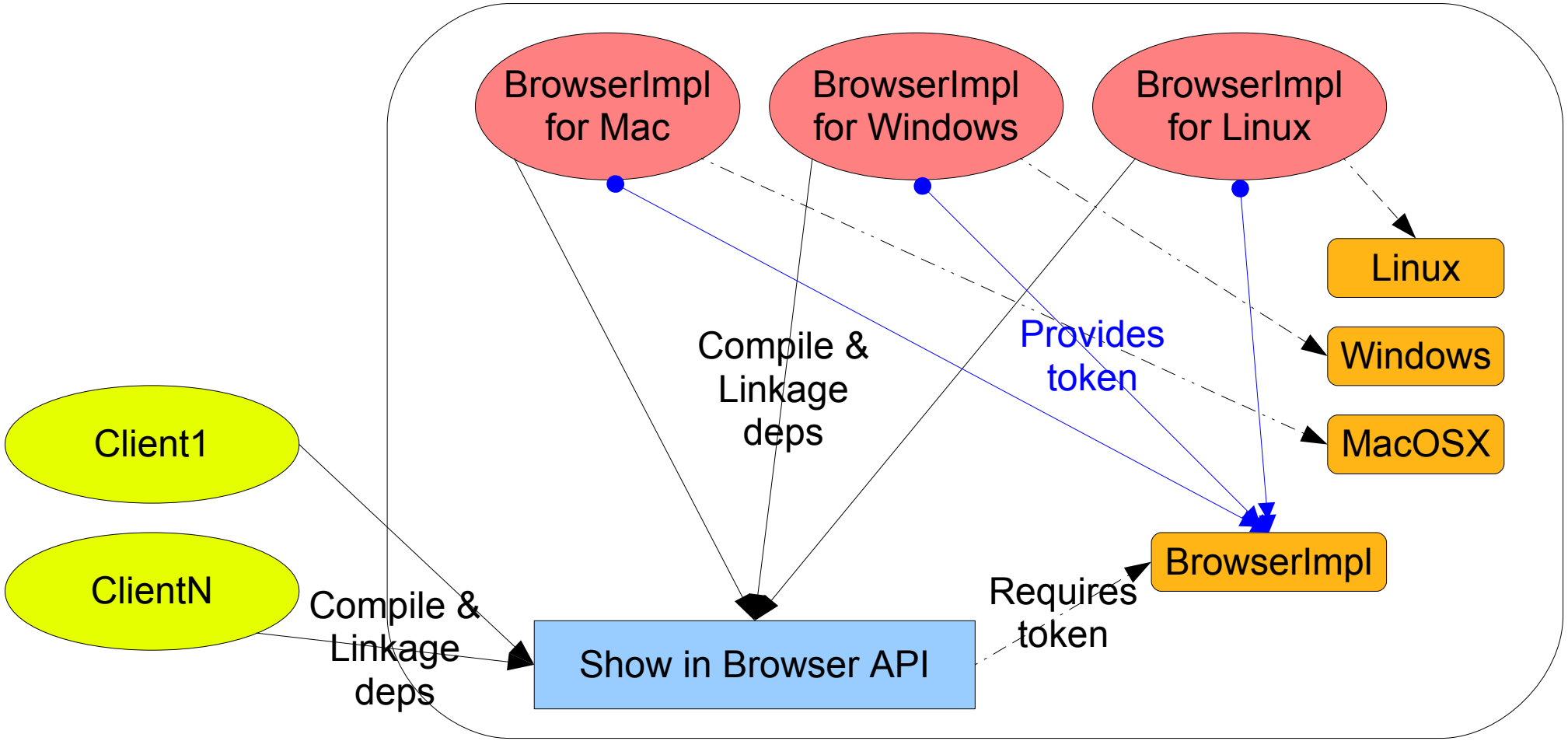
Has anyone managed to create non trivial resuable component, that has actually been reused in other apps in same domain?

Weight/dependencies

- Compile
  - Put the module on classpath during compilation
  - Usually implies the module is needed during runtime too
  - But not in case of annotation processors
- Linkage
  - Classloader needs to see these modules
- Execution
  - Just be present in the running environment
- Good modular system needs to express them!

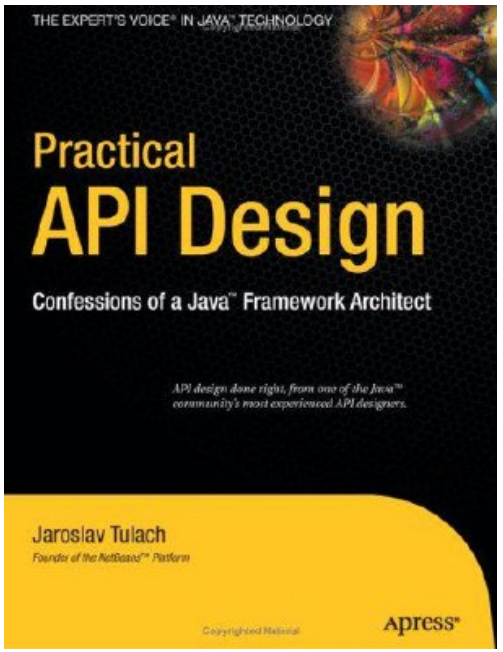
Blackbox Configuration Pattern

Proper combination of dependencies balances reuse and ease of use.



Modularity Without APIs?

- Incremental deployment
  - Old and new versions linked together
  - Backward compatibility of public module interface
- Distributed development
  - Independent schedules
  - Can't organize global change
- API-less world prevents updates
  - MediaWiki close proxymity
- Compatible APIs minimize coupling



Anti: Magical Strings

- OSGi spec identifies modules by URL:

```
public Bundle installBundle(String url);
```

- The JAR is downloaded & copied from the URL
- Can I install a bundle without copying it?
- Spec is silent, but Felix and Equinox support:

```
ctx.installBundle("reference:file:///path/to/the.jar");
```

- Magical strings give you loose coupling
  - Too little coupling!

Links of interest

- http://www.kirkk.com/modularity/chapters/
- http://techdistrict.kirkk.com/2009/07/08/reuse-is-the-dream-dead/
- http://c2.com/cgi/wiki?AntiPatternsCatalog
- Component Software: Beyond Object-Oriented Programming, Clemens Szyperski
- http://wiki.apidesign.org/wiki/APIAntiPatterns