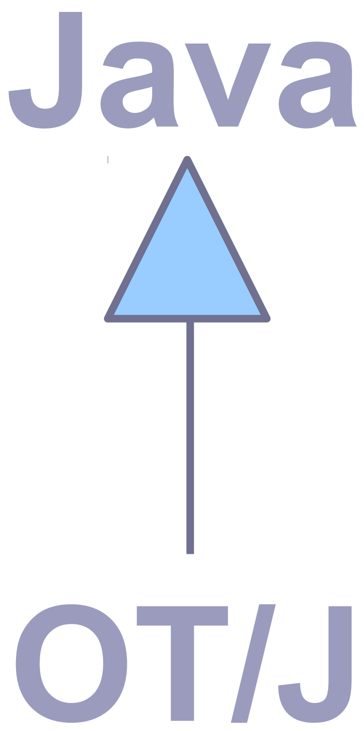
Redefining Modularity, Re-use in Variants and all that with Object Teams

Stephan Herrmann, GK Software AG

Eclipse Day Kraków

September 13, 2012


Java
OT/J

A Little History of Spaghetti

- In the beginning the world consisted of statements:
  - read, store, arithmetics, jump
  - jumps where found to be dangerous because:

Through undisciplined jumps each statement could relate to any other statement

Ⓒ Tim 'Avatar' Bartel

A Little History of Spaghetti

- **Solution**
  - combine statements to sub-routines / procedures
- **But: what about data?**
  - data sharing through global variables
  - each procedure may relate to any global variable

Data Spaghetti

This is not modular

Tim 'Avatar' Bartel

A Little History of Spaghetti

- **Solution**
  - combine procedures and variables to classes
- **But: what about size?**
  - systems made from 1000s of classes
  - each class may relate to any other class

Class Spaghetti

Ⓒ Tim 'Avatar' Bartel

Attempts for Addressing Scale

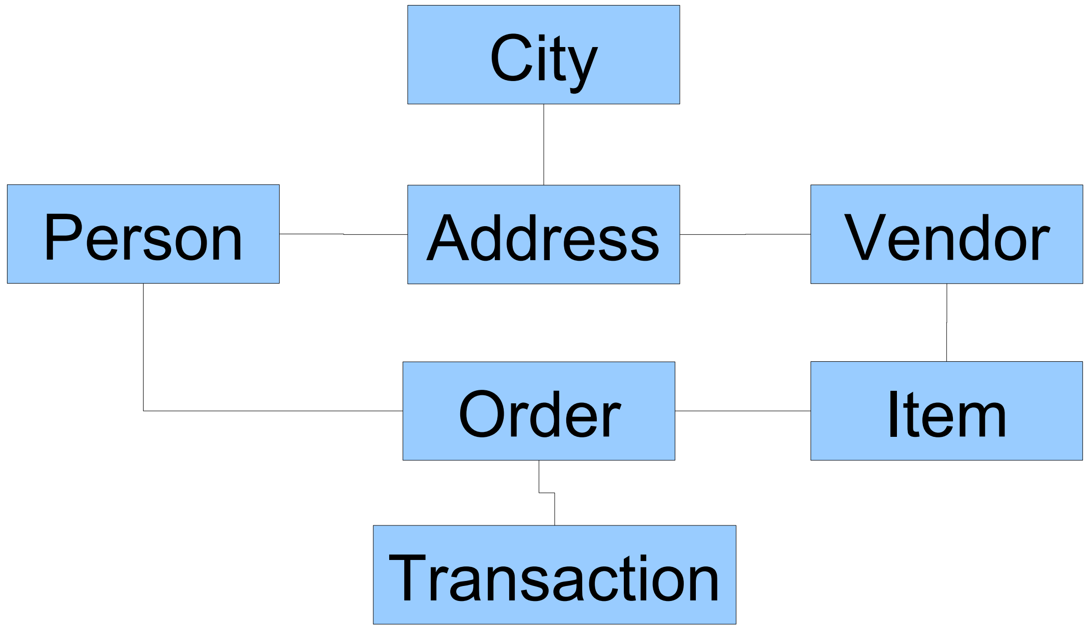
Creating modules	
- everything you write should be a module	
Statement	1 LOC
Procedure	
- module of 20 statements	20 LOC
Class	
- module of 20 procedures ("methods")	400 LOC
Package	
- module of 20 classes	8000 LOC
Bundle	
- module of 20 packages	160000 LOC
Beans, Components, Super Packages, Modules, Jars ...	

Attempts for Addressing Scale

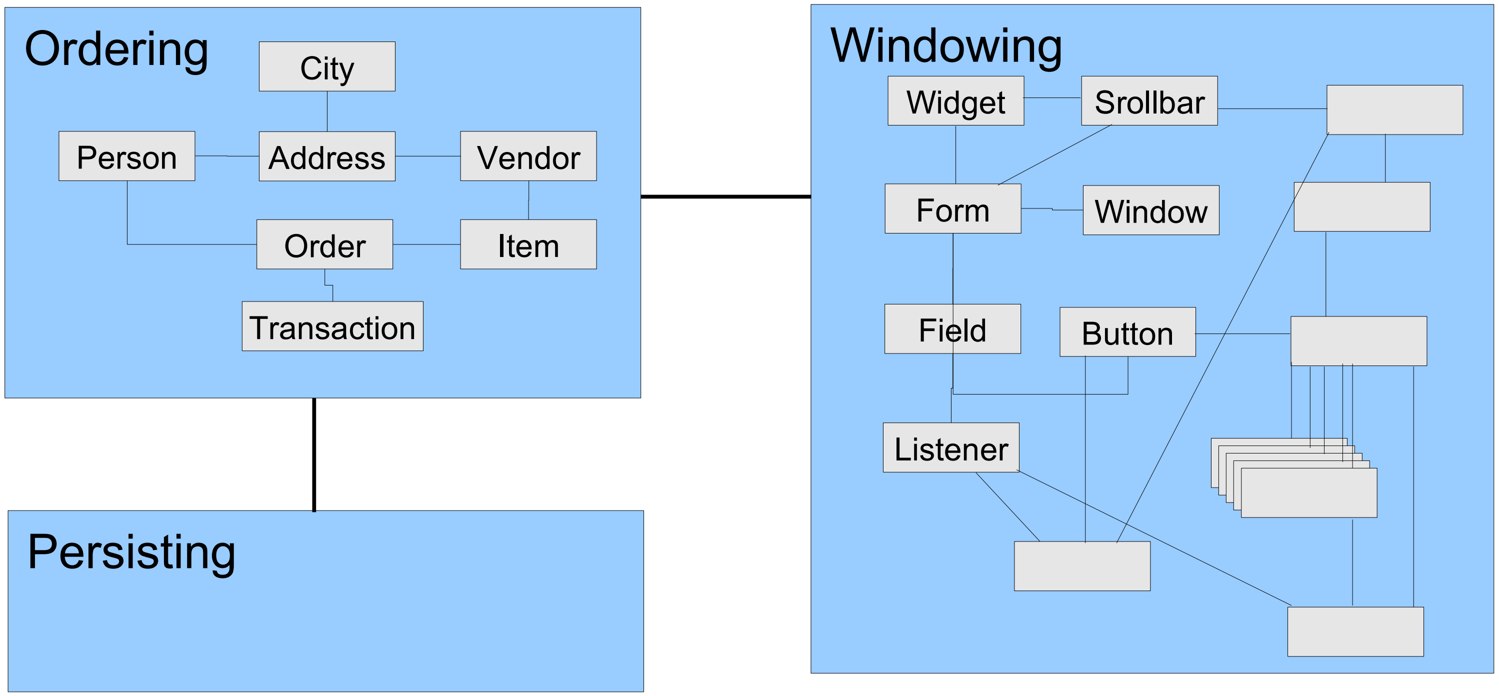
- Creating modules
  - everything you write should be a module
- Statement 1 LOC
- Procedure
  - module of 20 statements 20 LOC
- Clas
  -
- Pac
  - module of 20 classes 8000 LOC
- Bundle
  - module of 20 packages 160000 LOC
- Beans, Components, Super Packages, Modules, Jars ...

1 new concept for each level of scale?
Not an economic solution!

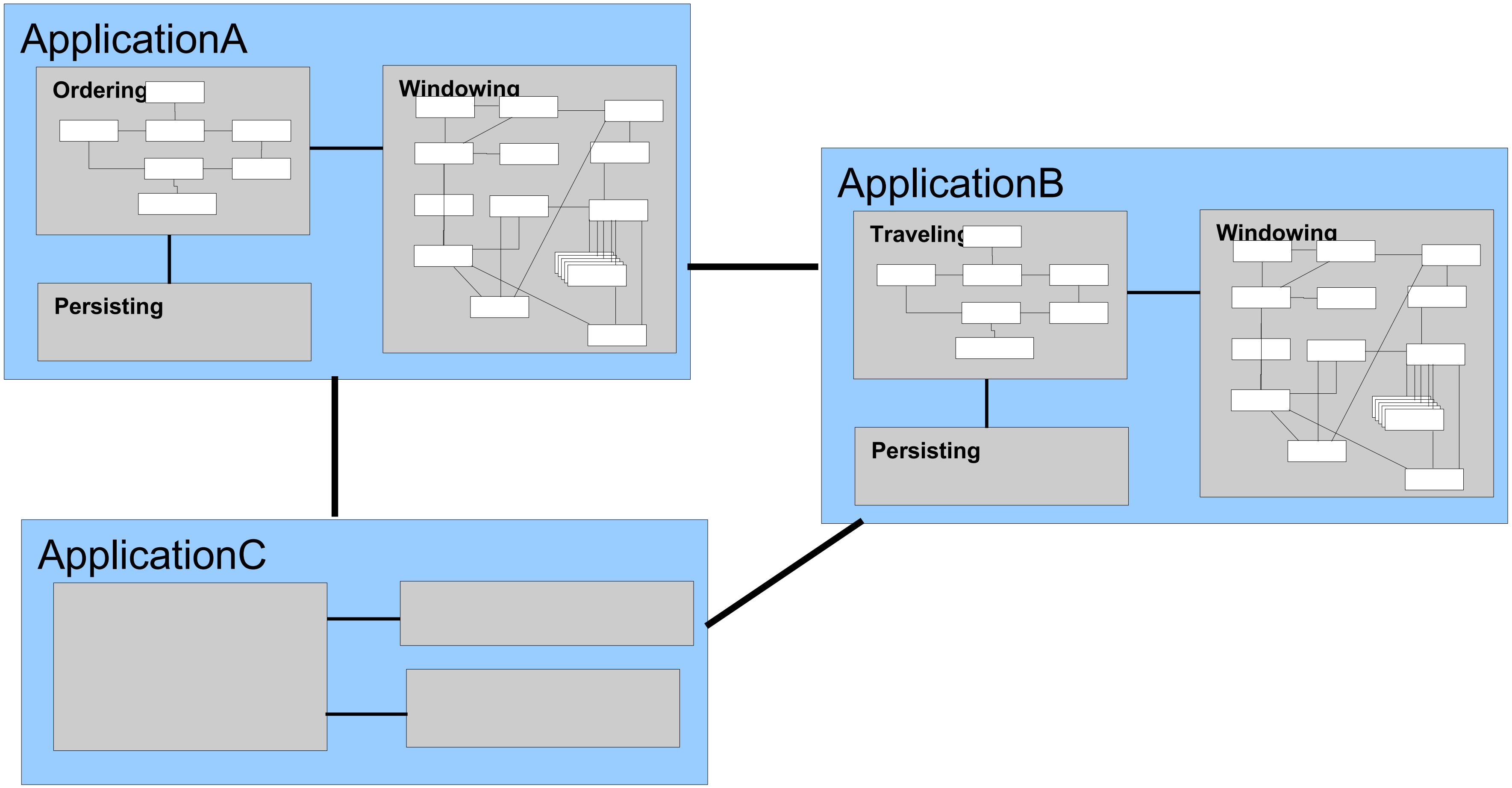
System made from Classes



System made from Nested Classes



System made from Nested Classes



- Cool!
  - but... classes with 100's of inner classes are not manageable

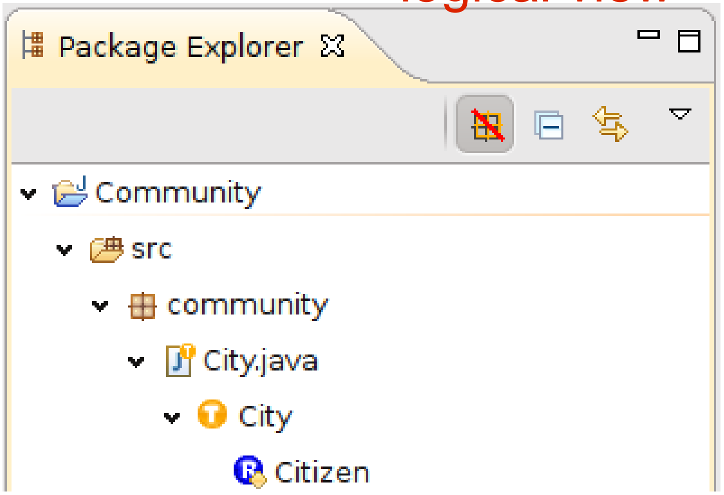
Classes & Packages

- Package
  - hierarchical organization: folders & files
- Class
  - define boundary: signature ↔ implementation
  - support nesting
- Choose one ! ?

Classes & Packages

- **Package**
  - hierarchical organization: folders & files
- **Class**
  - define boundary: signature ↔ implementation
  - support nesting
- **Solution: team = class & package**

logical view

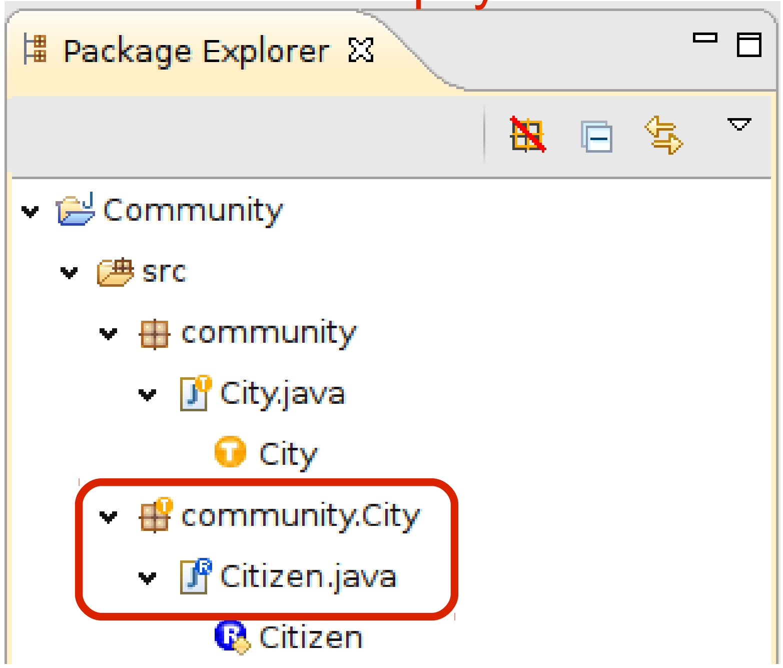


Package Explorer

Community
src
community
City.java
City
Citizen

=

physical view



Package Explorer

Community
src
community
City.java
City
community.City
Citizen.java
Citizen

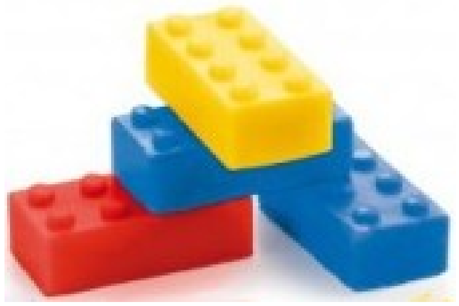
Classes & Packages

- Package
  - hierarchical organization: folders & files
- Class
  - define boundary: signature ↔ implementation
  - support nesting
- Solution: **team** = class & package

- **Teams**
  - unify class and package
  - make nesting feasible
  - modules at any level of scale

Composition: Dream vs. Reality

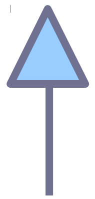
- System construction, ideally:
  - build lots of small building blocks
  - compose small blocks to larger blocks
  - top-level block is your system



- Remaining challenges
  - complexity makes hierarchical breakdown extremely difficult
  - software re-use is more demanding than lego playing
- **Essence of re-use**
  - handle near miss!
  - transform “near miss” into “perfect match”

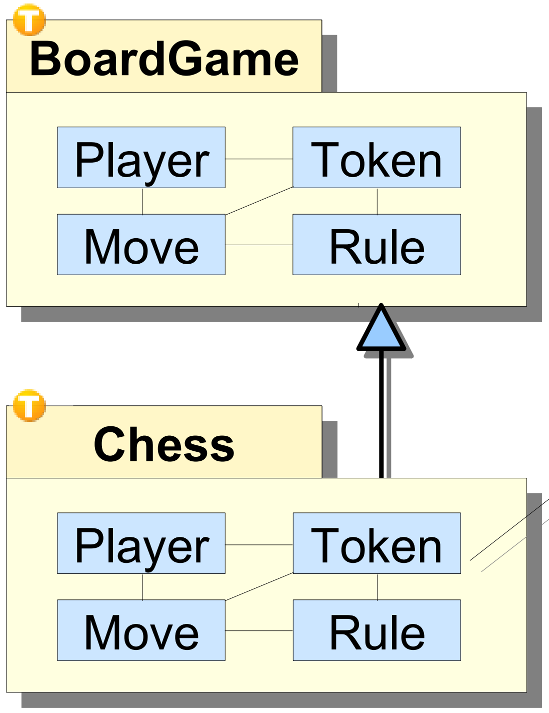
Unanticipated Adaptation

- Transform “near miss” into “perfect match”
  - need a tool for adapting an existing module
  - (anticipated adaptation: parameters)
  - unanticipated adaptation?
- O-O tool for adaptation: inheritance
  - acquire all from parent
  - adapt those parts that don't fit



- Inheritance is “broken” for inner classes in Java!
  - inherited methods can be overridden
  - inherited classes cannot be overridden!

Example: Board Games

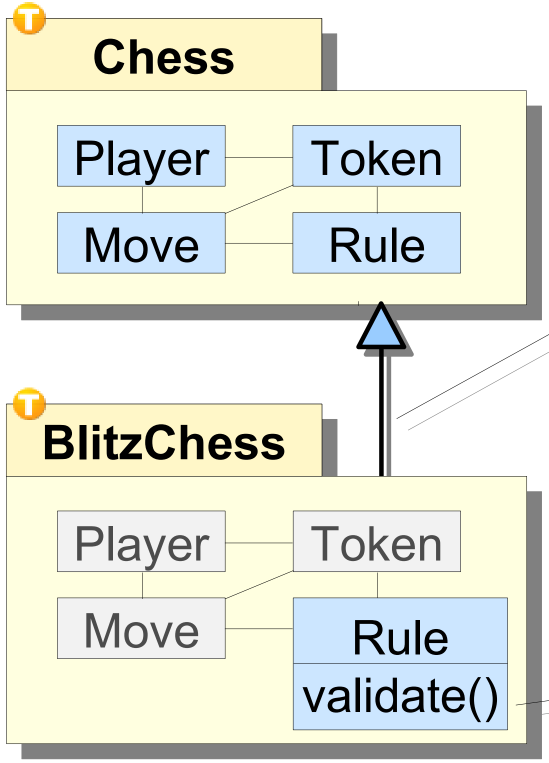


impossible to mix elements from different games

mature, established concept

- Team inheritance, members are **virtual classes**
  - consistent refinement of all members

Example: Board Games



single choice:

new BlitzChess()

- version of Rule
- version of validate()

selectively override at any nesting level

#2

- **Team inheritance, members are virtual classes**
  - consistent refinement of all members
  - deep overriding
  - flexible & modular

Essence of Re-Use

- **Commonalities**
  - interfaces, (abstract) super-classes
  - team classes
- **Variations**
  - sub-classes
  - sub-teams

} decompose

- **Assemble selected variations to a system**
  - what's the top-level?
    - `App app = new ApplicationA();`
    - `app.run();`

} compose

Dominance of the Instantiator

- **Capturing variations with inheritance**
  - type of a variable describes a range of behaviors
  - instantiation selects one class / variant / behavior
  - each instance is locked to one behavior

new

- **Who has the power to create?**
  - every occurrence of **new** decreases re-usability
  - “best practice” to avoid **new** in favor of factories (or DI)
    - for all classes / objects??
    - only those classes that are relevant for sub-classing
    - pre-planning vs. unanticipated adaptation
  - who instantiates the factory?
- **This power creates conflicts**
  - there can only be one winner
  - re-use is limited to one step

Dominance of the Instantiator

- **Capturing variations with inheritance**
  - type of a variable describes a range of behaviors
  - instantiation selects one class / variant / behavior
  - each instance is locked to one behavior

new

- **Who has the power to create?**
  - every occurrence of **new** decreases re-usability
  - “best practice” to avoid **new** in favor of factories (or DI)
  - who instantiates the factory?

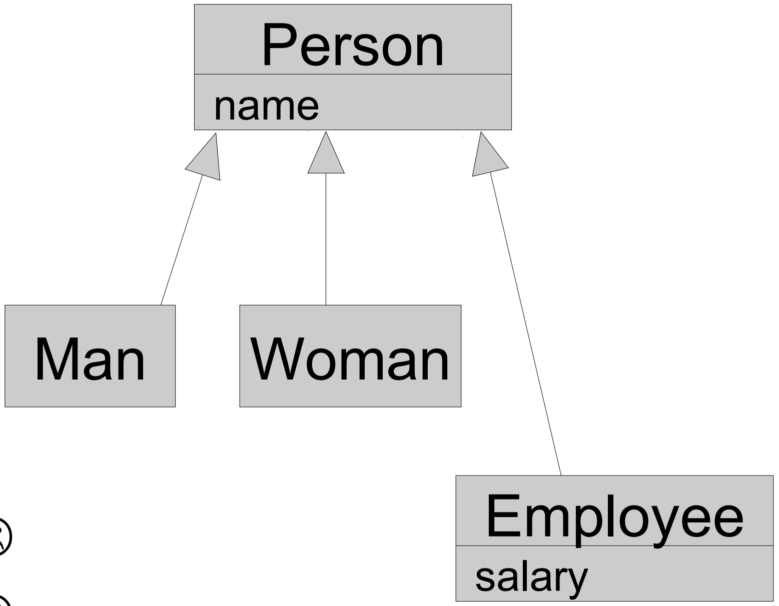
These conflicts are a result from limitations of inheritance.

- **This power creates conflicts**
  - there can only be one winner
  - re-use is limited to one step

Limitations of Inheritance

- **Inheritance is great, but ...**

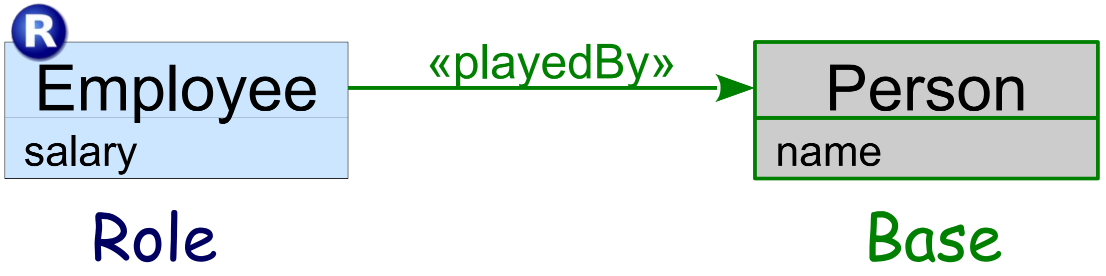
  <u>A text book example:</u>
  - A man/woman **is a** person, OK
  - An employee **is a** person, OK?
    - Born as an employee?
    - Dying when loosing the job?
    - Several jobs, yet only one salary?



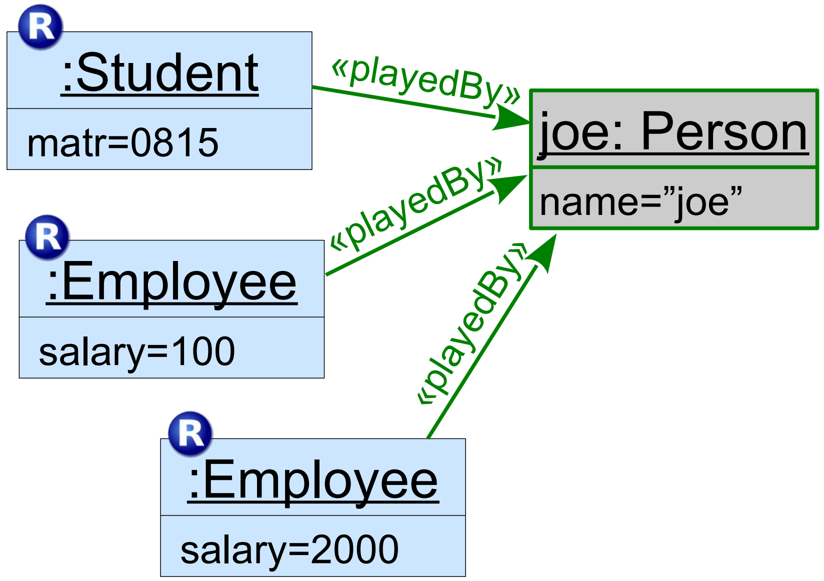
- **Whats wrong with inheritance?**
  - Missing “become”, “quit” ☹
  - Can't duplicate fields ☹
- **Can we do better?**
  - Yes:
  - Employee is a Role played by a Person

Role playing

- playedBy relationship



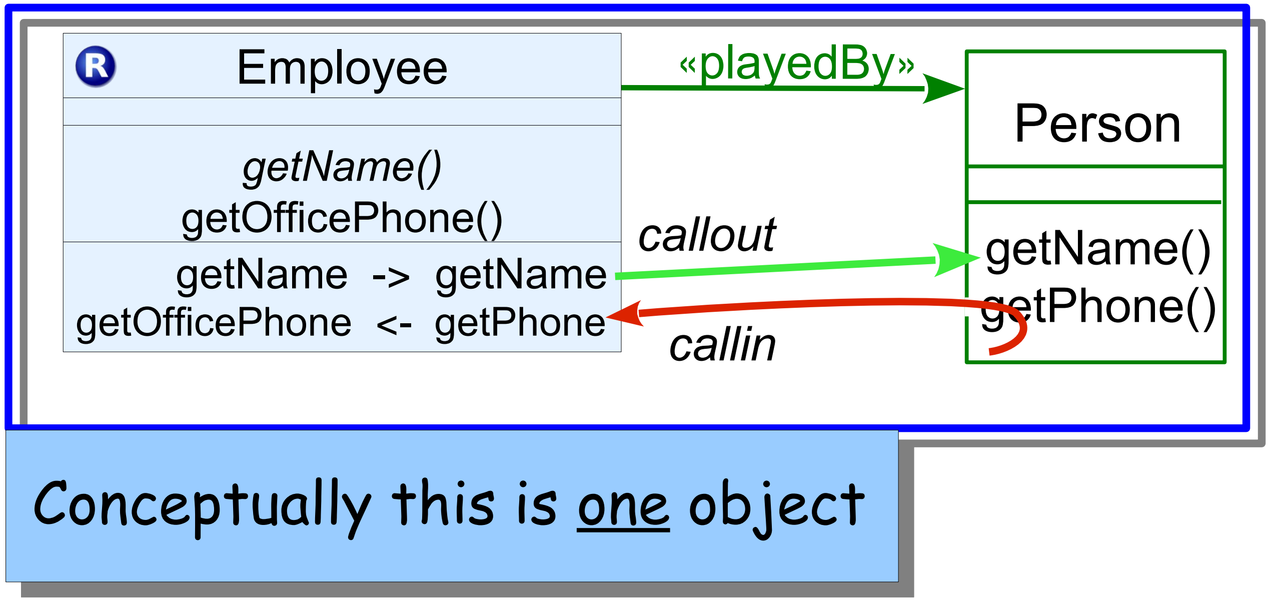
- Advantages:
  - **Dynamism**:
    roles can come and go
    (same base object)
  - **Multiplicities:**
    one base can play several roles
    (different/same role types)



Roles in OOPLs have been studied for approx. 20 years

Capabilities of Roles

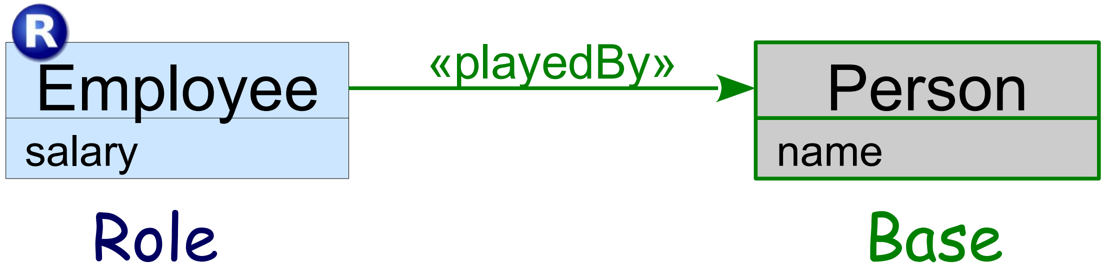
- **playedBy**
  connect role to base
- **callout**
  forward to base
- **callin**
  intercept base method



Composition Redefined

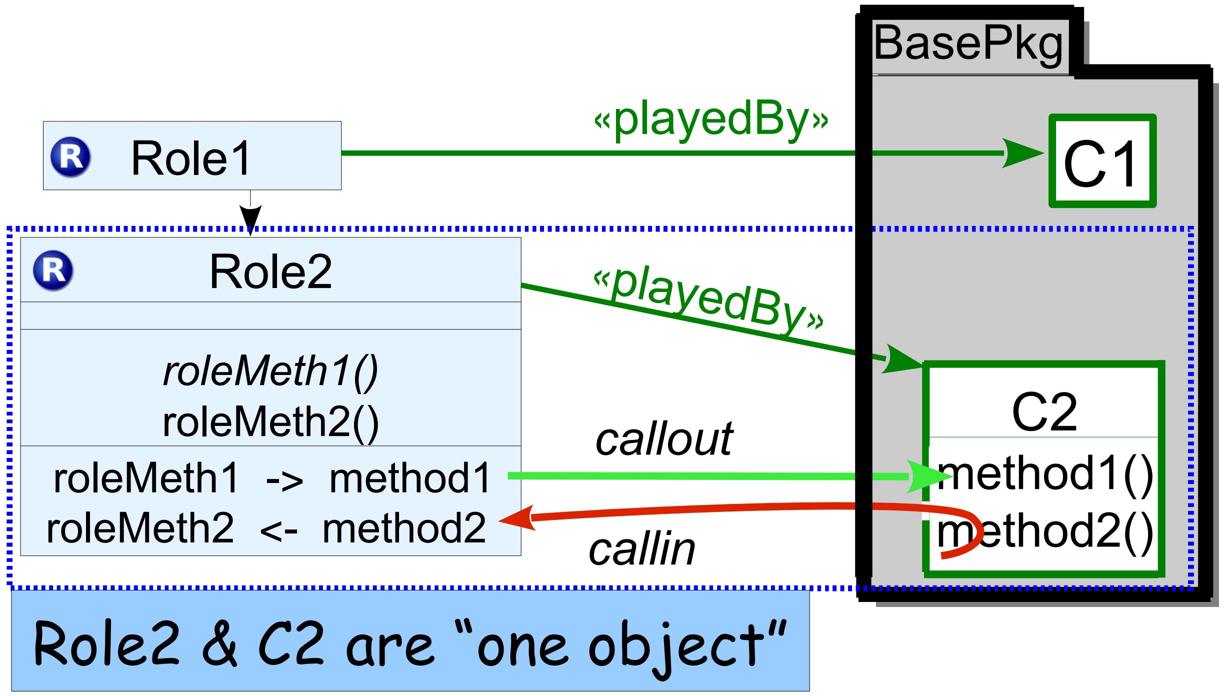
- Class-based inheritance is rigid
  - re-use requires flexibility
  - flexibility is achieved by complex design patterns
  - those are work-arounds

- **Composing instances**
  - one instance can accumulate multiple behaviors
- **Composing at runtime**
  - an instance may change its behavior during its life time



Roles vs. Modules

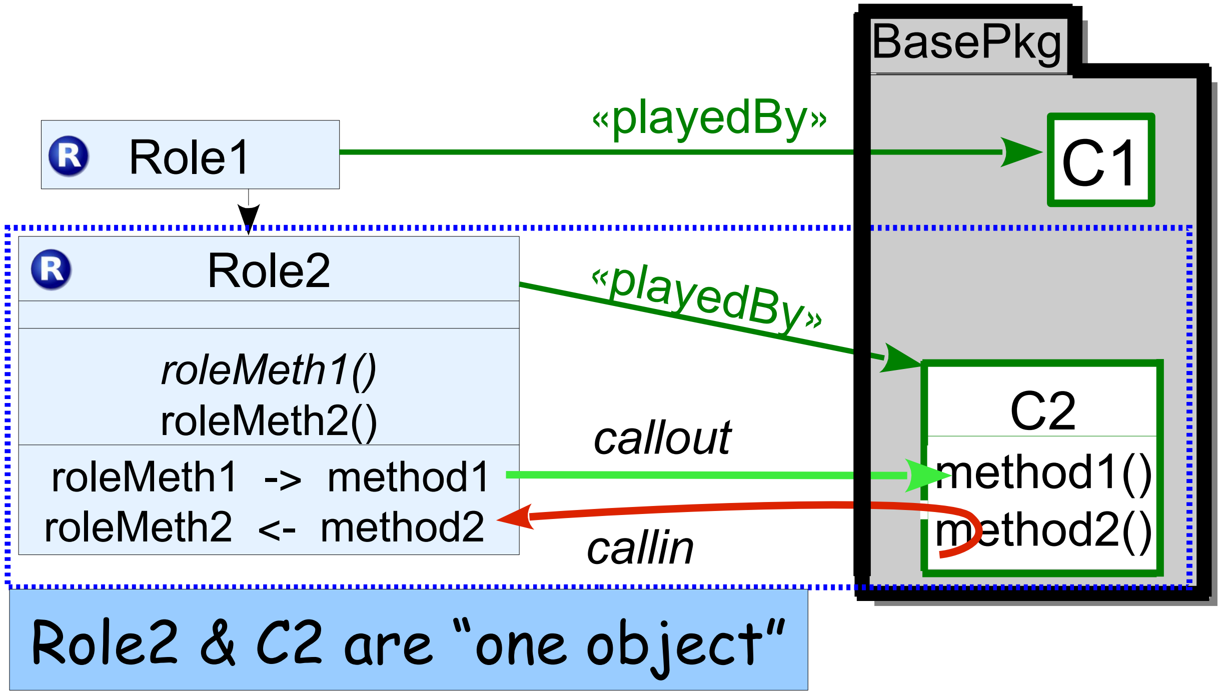
- Avoid **role spaghetti**
- **Roles & base each live in their own context / module**
- **Bases may be encapsulated inside a module**
  - not all bases will be visible to our roles



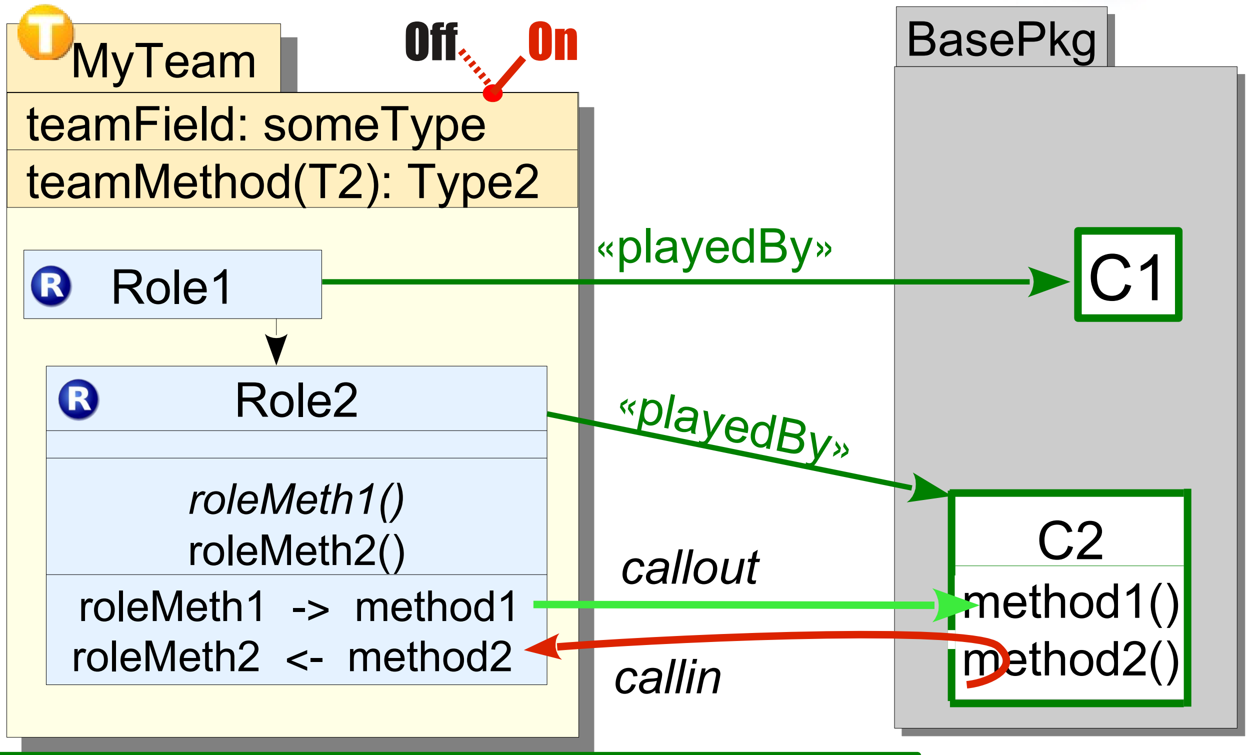
Roles vs. Modules

#4

- **Decapsulation:**
  - defined exceptions to encapsulation
- **Legalizing decapsulation:**
  - visible, controllable (approve/deny per case)
  - less total coupling



Modules for Roles



- **Roles are members of a team**
  - Behavior implemented as interaction among roles
- **Team activation controls all contained roles**
  - no callin trigger into an inactive team
  - on-demand role instances per team instance

Summary

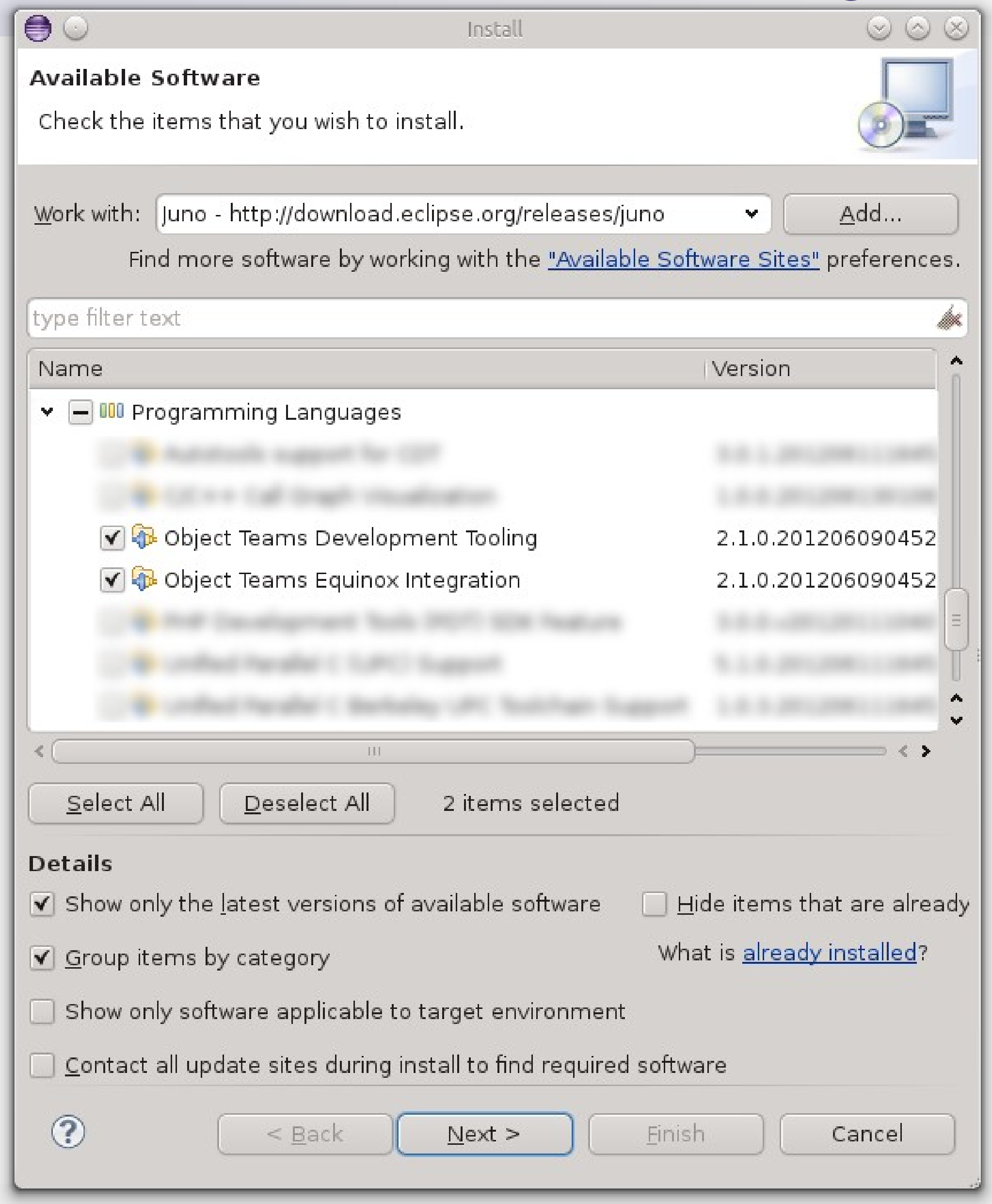
Your shopping cart contains five items:

- **Team classes / packages**
  - unifying classes & packages makes nesting feasible
- **Team inheritance**
  - consistent specialization, deep overriding (“virtual classes”)
- **Role playing**
  - dynamically specialize / compose instances at runtime
- **Decapsulation**
  - admit exceptions from boundary enforcement
- **Teams are modules for roles**
  - consistent (de)activation – affecting callin and role instantiation

To check out these items please visit

http://eclipse.org/objectteams

Install into a recent Eclipse package


Install

Available Software

Check the items that you wish to install.

Work with: Juno - http://download.eclipse.org/releases/juno

Add...

Find more software by working with the "Available Software Sites" preferences.

type filter text

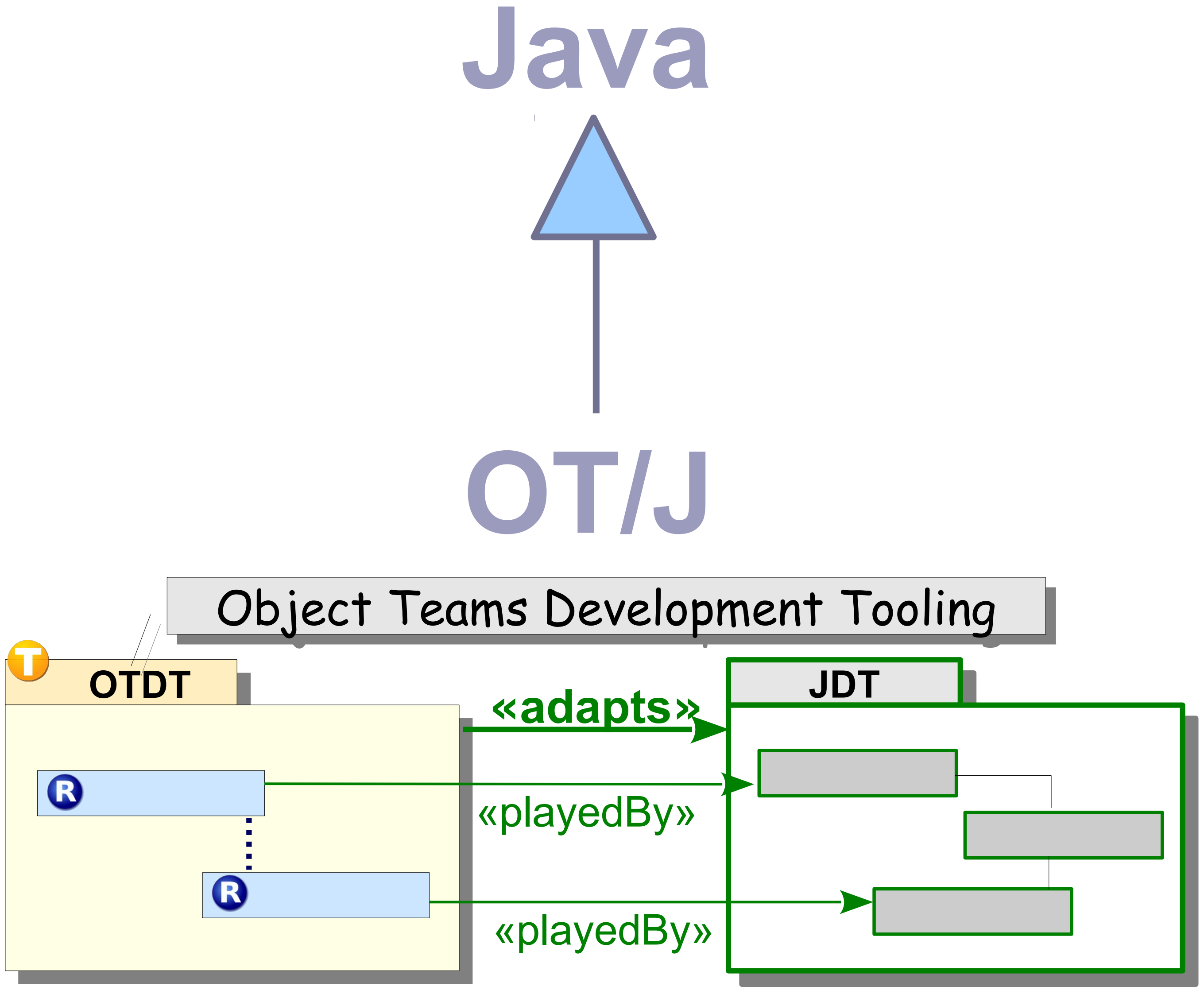
Name	Version
Programming Languages	
☑ Object Teams Development Tooling	2.1.0.201206090452
☑ Object Teams Equinox Integration	2.1.0.201206090452

Select All | Deselect All | 2 items selected

Details

[✓] Show only the latest versions of available software
[ ] Hide items that are already
[✓] Group items by category
What is already installed?
[ ] Show only software applicable to target environment
[ ] Contact all update sites during install to find required software

< Back | Next > | Finish | Cancel


Java
OT/J
Object Teams Development Tooling
OTDT
JDT
«adapts»
«playedBy»
«playedBy»

Credits

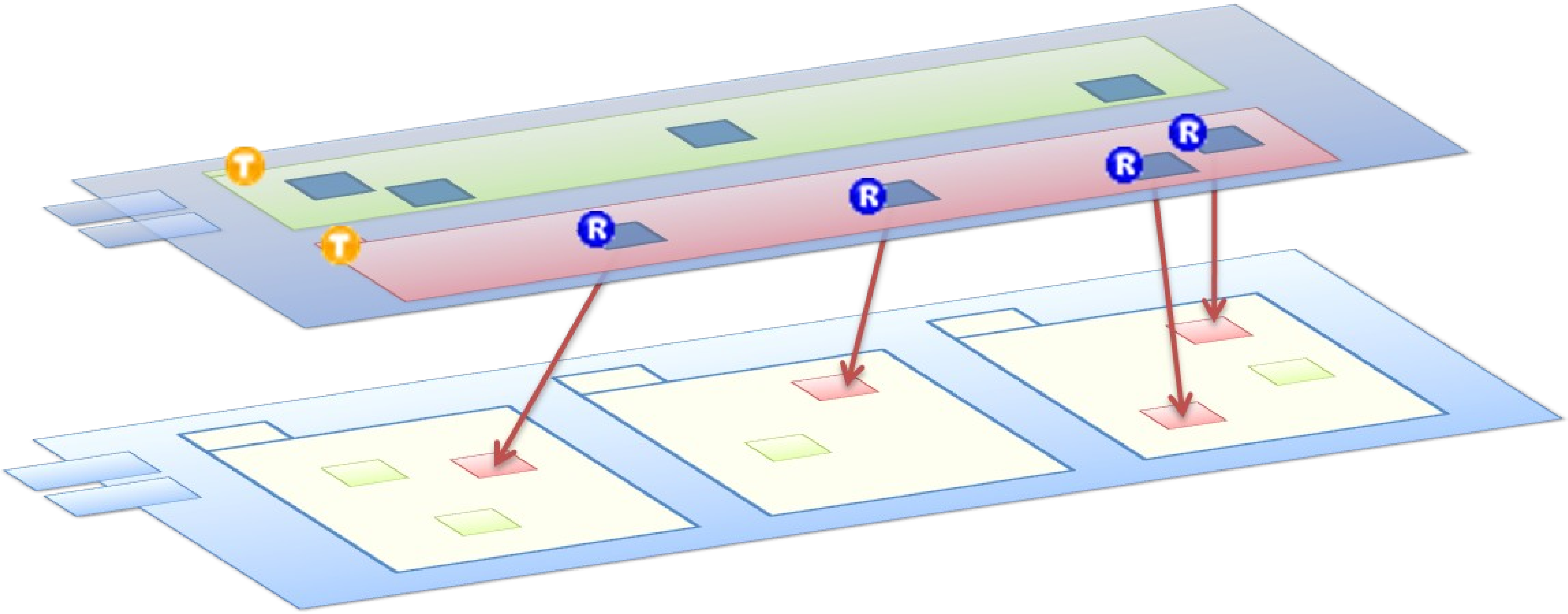
Resources used in this presentation

- http://upload.wikimedia.org/wikipedia/commons/9/93/Spaghetti.jpg by Tim 'Avatar' Bartel

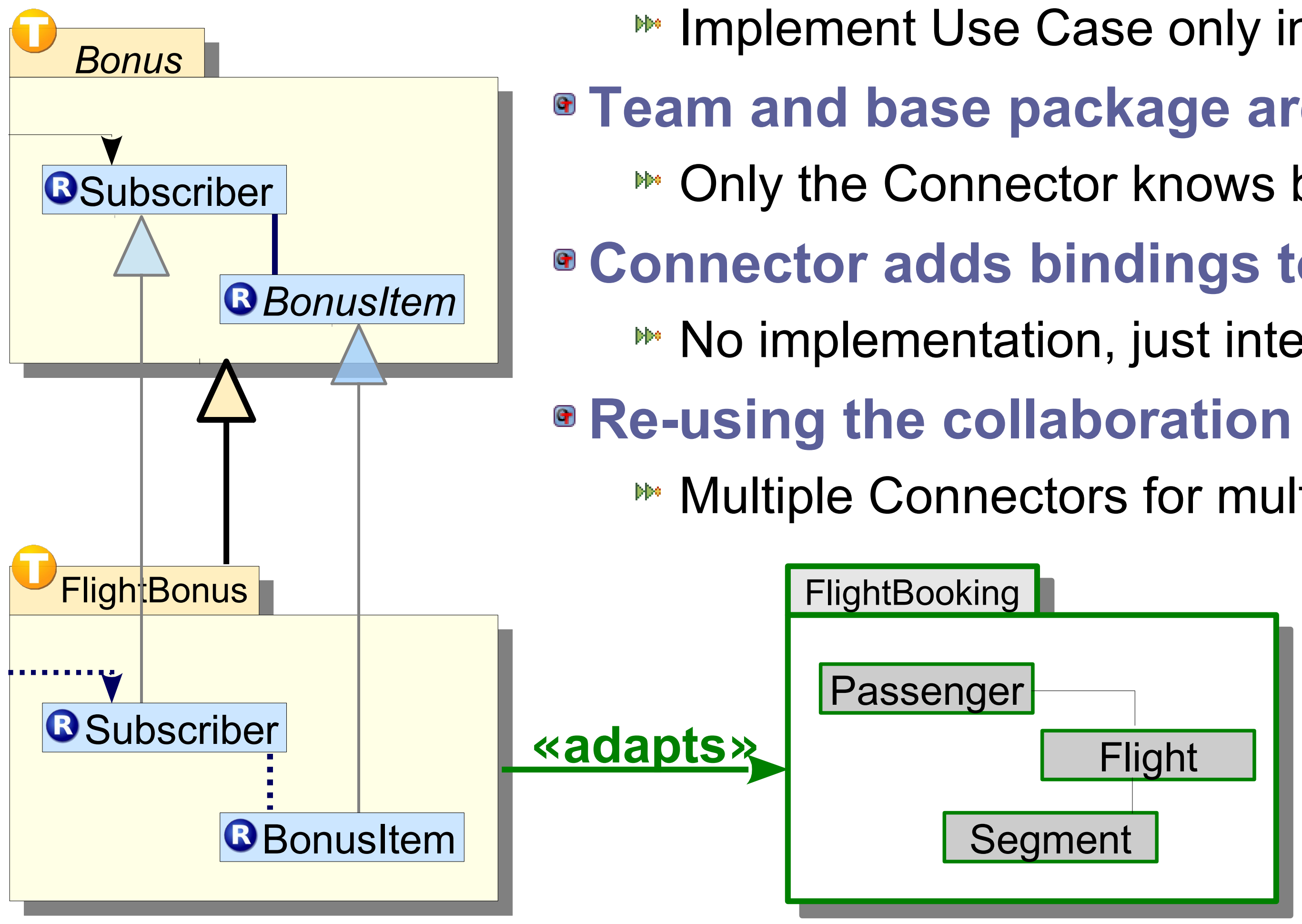
Bonus Material

Adaptation using Object Teams

- **Eat the cake and have it, too**
  - adaptation is a separate module (team and role classes)
  - tightly integrated with existing code
  - minimal coupling

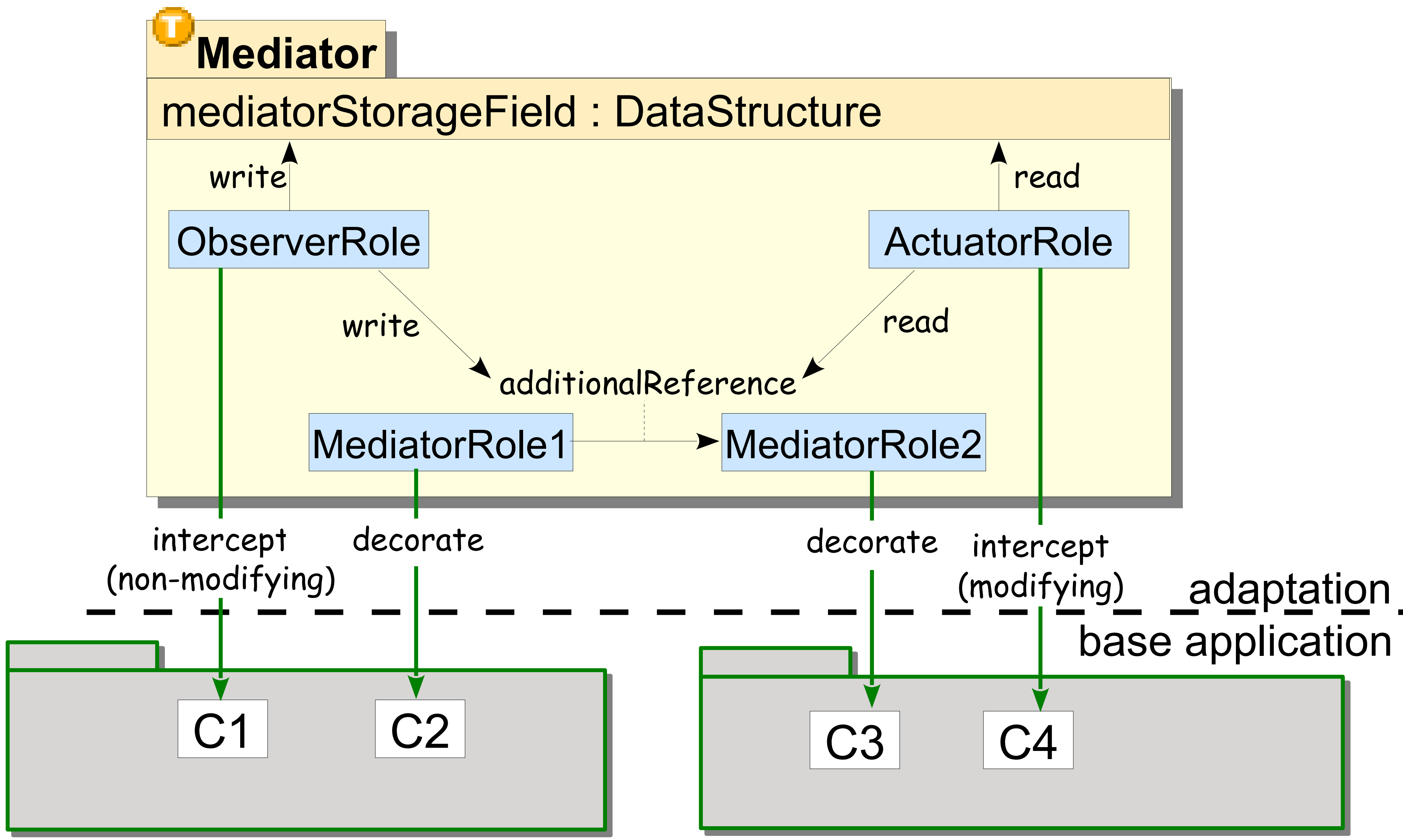


Connector Pattern



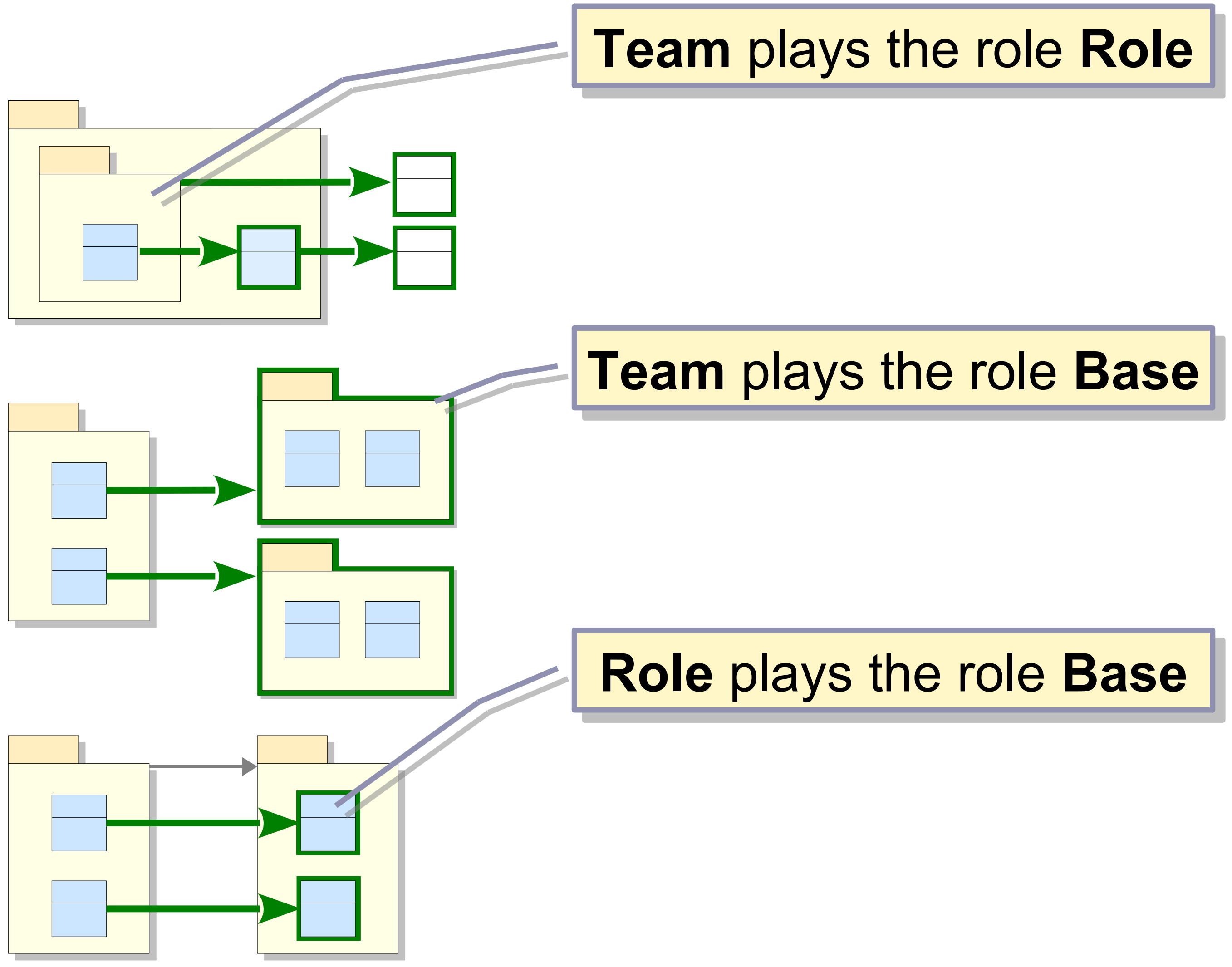
- **Abstract team provides implementation**
  - Implement Use Case only in terms of roles
- **Team and base package are independent**
  - Only the Connector knows both
- **Connector adds bindings to base package**
  - No implementation, just integration
- **Re-using the collaboration**
  - Multiple Connectors for multiple base packages

Observer-Mediator-Actuator

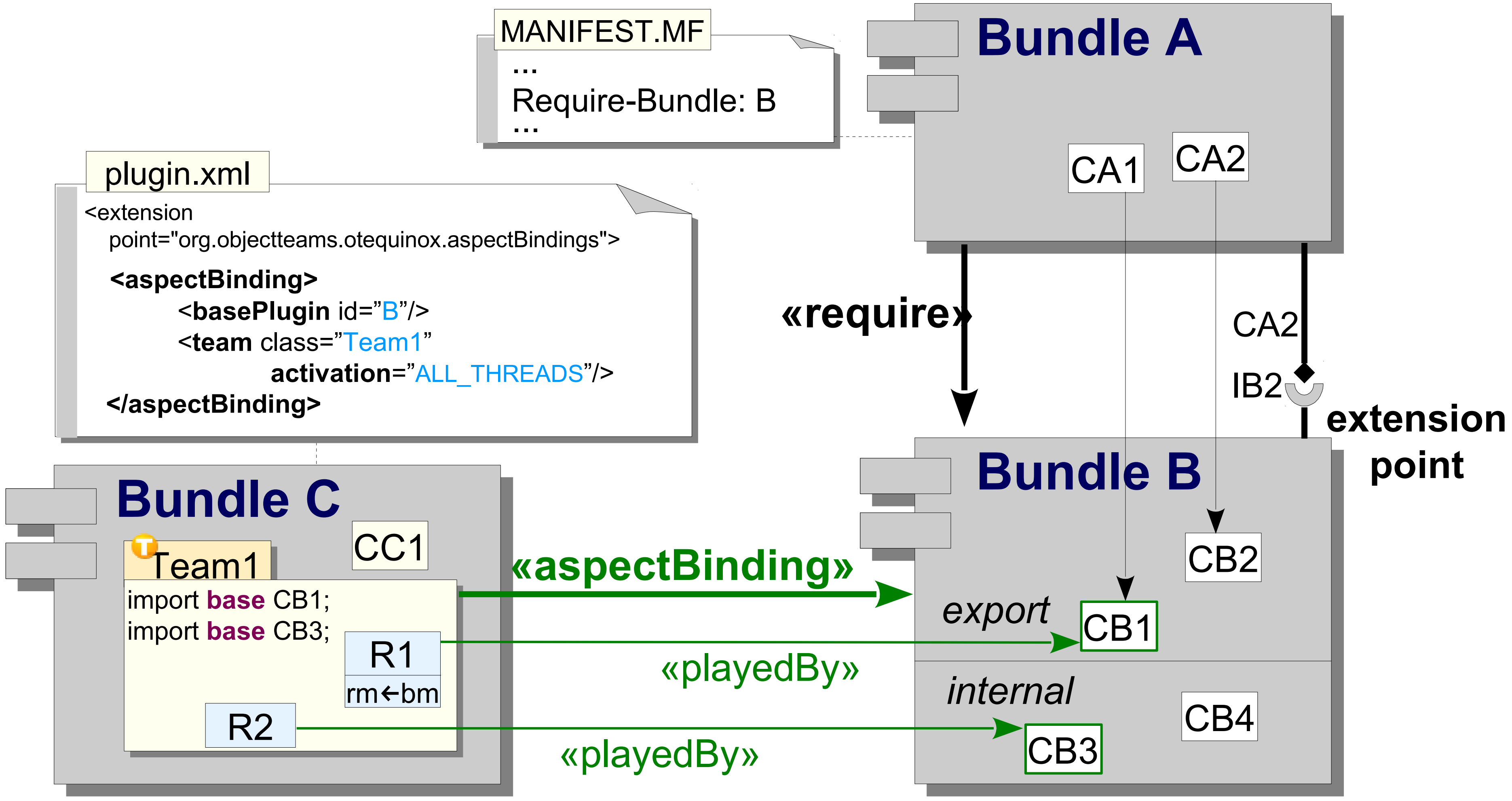


Nesting – Stacking – Layering

- **Nesting**
- **Stacking**
- **Layering**


Team plays the role **Role**
Team plays the role **Base**
Role plays the role **Base**

Components: OT/Equinox


MANIFEST.MF

```
...
Require-Bundle: B
...
```

plugin.xml

```
<extension
   point="org.objectteams.otequinox.aspectBindings">
   <aspectBinding>
         <basePlugin id="B"/>
         <team class="Team1"
                 activation="ALL_THREADS"/>
   </aspectBinding>
```

Team1

```
import base CB1;
import base CB3;
```