Detailed Design

Gerson Sunyé
Université de Nantes
http://sunye.free.fr

Agenda

- Introduction
- Design for extensibility
- Design heuristics

Introduction

Detailed Design

- The goal of the detailed design is to specify the internal structure and behavior of components.
- The design is often driven by a quality factor:
  - extensibility, performance, portability, testability, etc.

Design Approach

- During domain analysis, we used abstraction to hide technical details and simplify models.

- This abstraction was needed to understand the problem domain, without any influence from implementation details.

- Conversely, during the detailed design, we will do either:

  - add details to domain models or

  - create models from scratch.

Goal

- Create a model that can be translated to code effortlessly.

Design for Extensibility

Design for Extensibility

- Extensibility is the ability of a software to allow and accept significant extension of its functionalities without major changes in its design.

- Extensibility principles. An extensible design:
  - is open to changes.
  - addresses a class of problems instead of a single problem.
  - does not contain immature/fuzzy concepts.

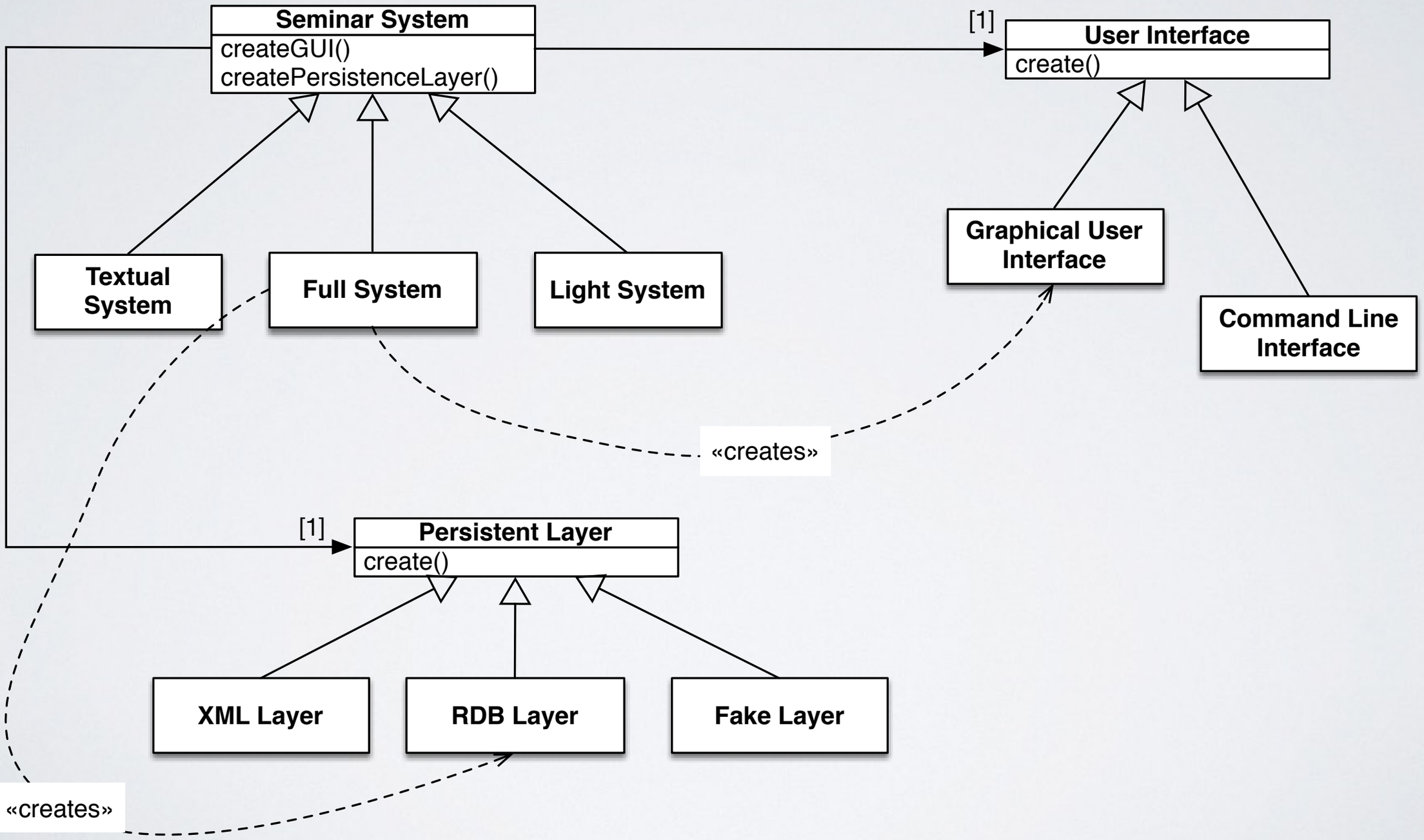
Design Patterns for Extensibility

What varies	Design Patterns
Algorithm	Strategy, Visitor
Actions	Command
Implementation	Bridge
Reactivity to changes	Observer
Interaction between objects	Mediator
Object creation	Fabric, Prototype
Created structure	Builder
Traversal algorithm	Iterator
Object interface	Adapter
Object behavior	Decorator, State

Reify Variants

- Apply the Abstract Factory design pattern to dynamically configure products from the same family.

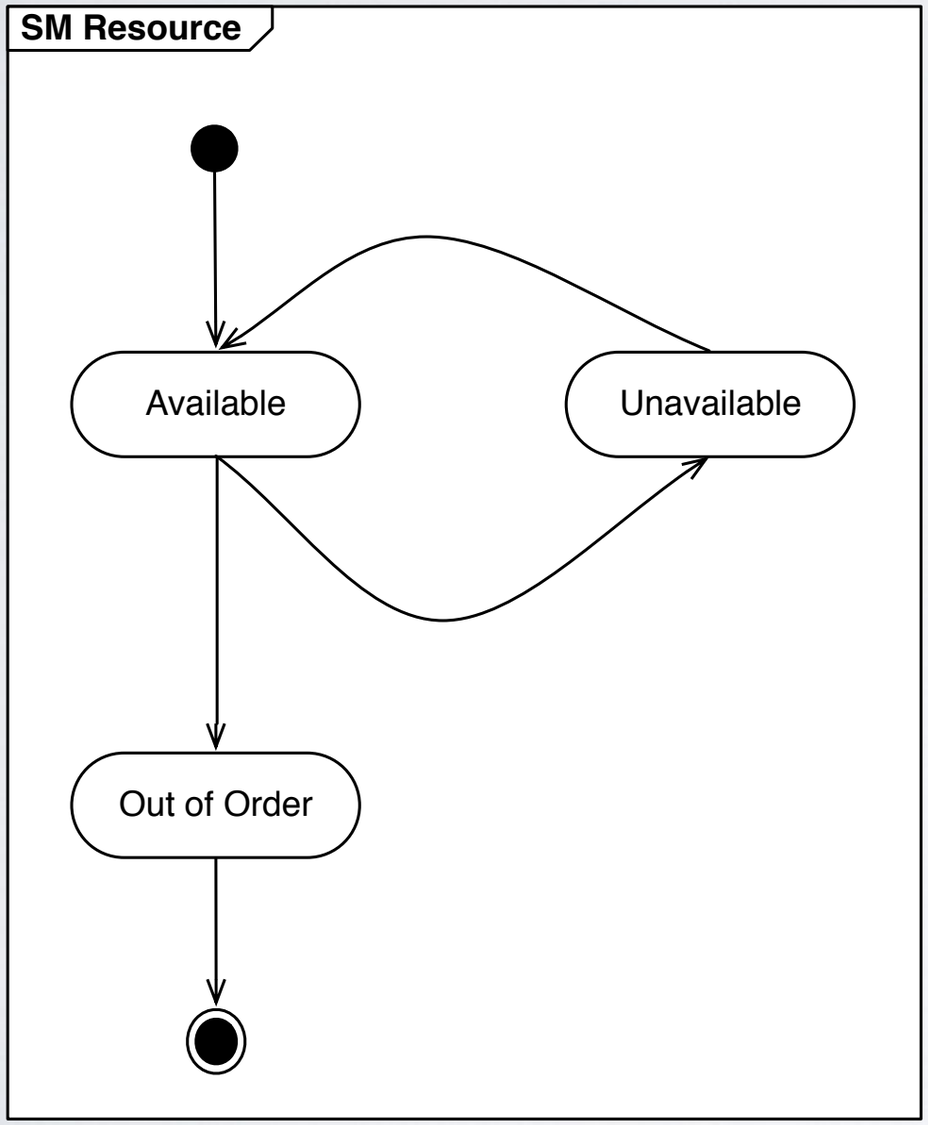
Applying the Abstract Factory

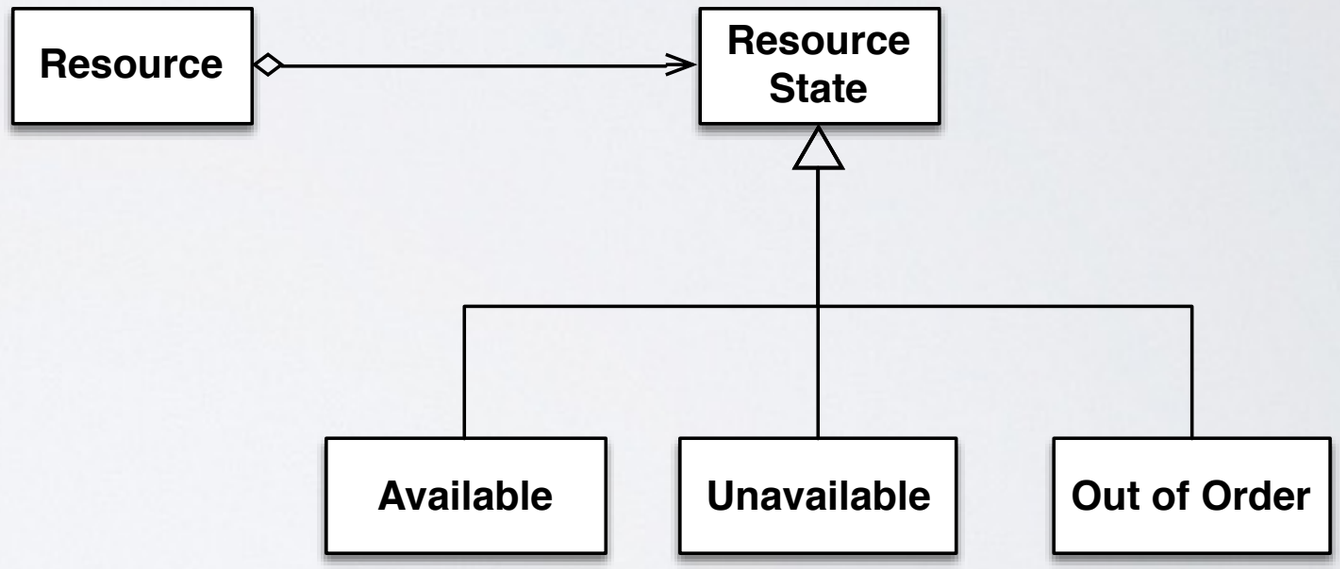


Reify States

- Apply the State design patterns for each state chart.

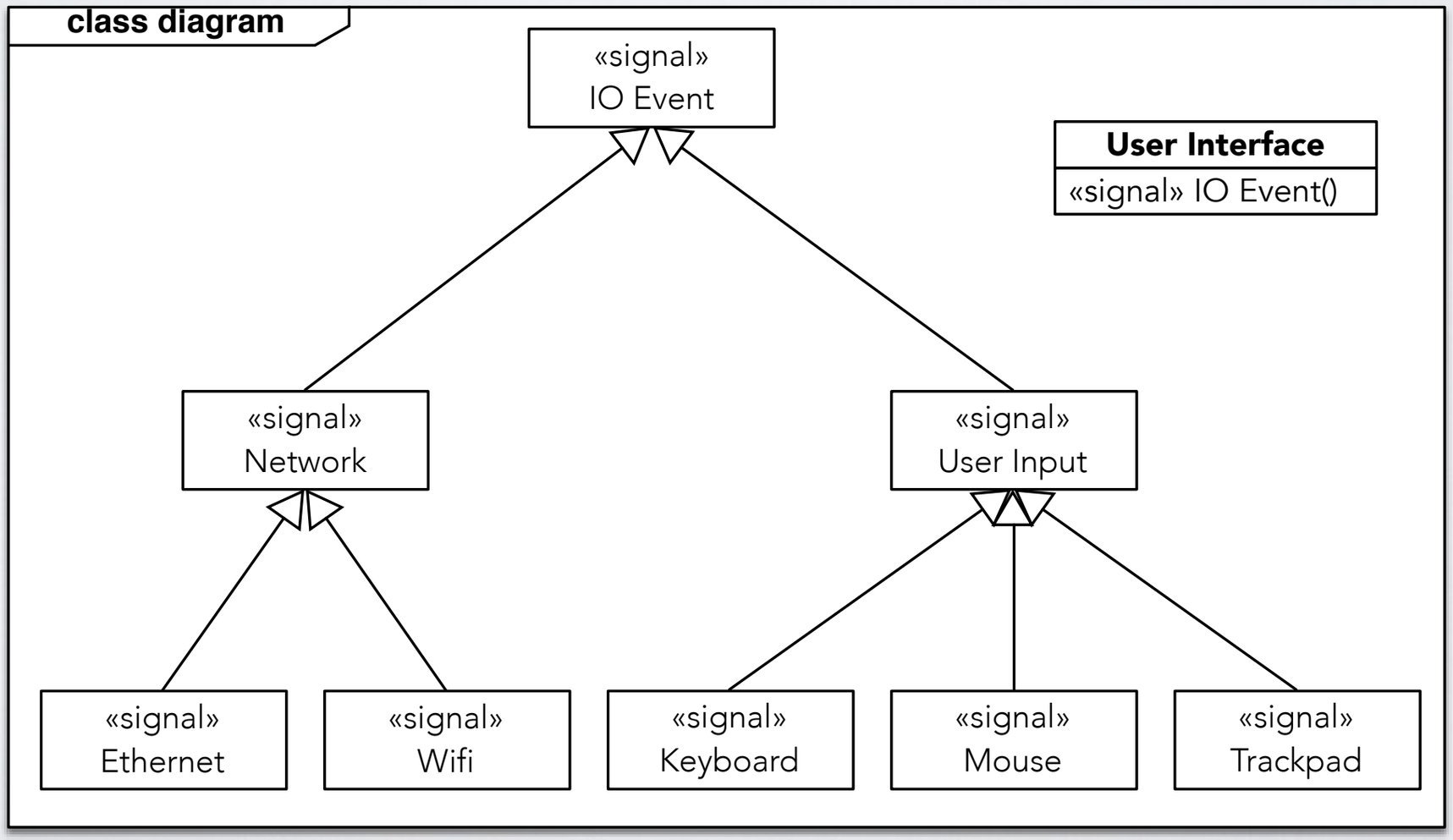
Applying the State Pattern



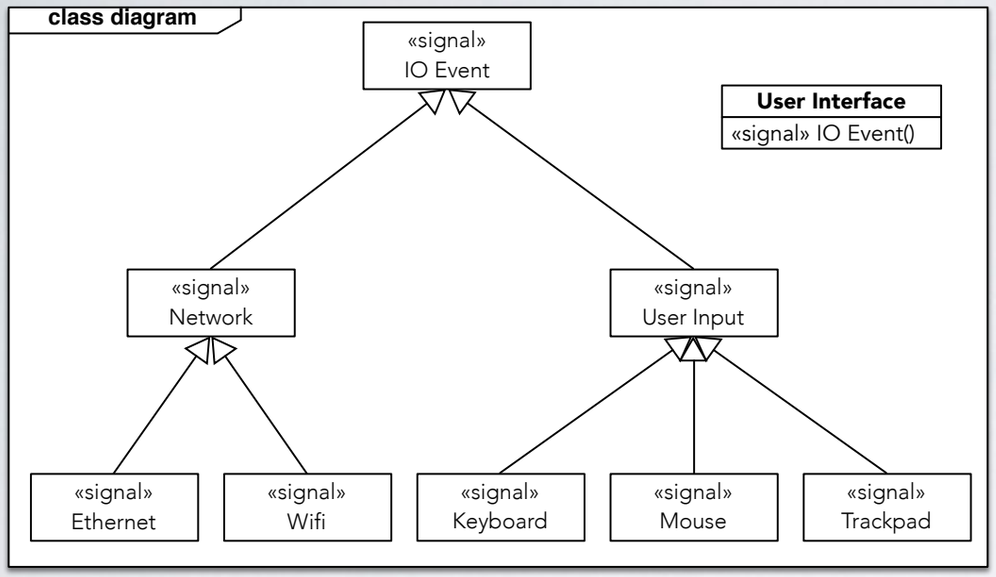


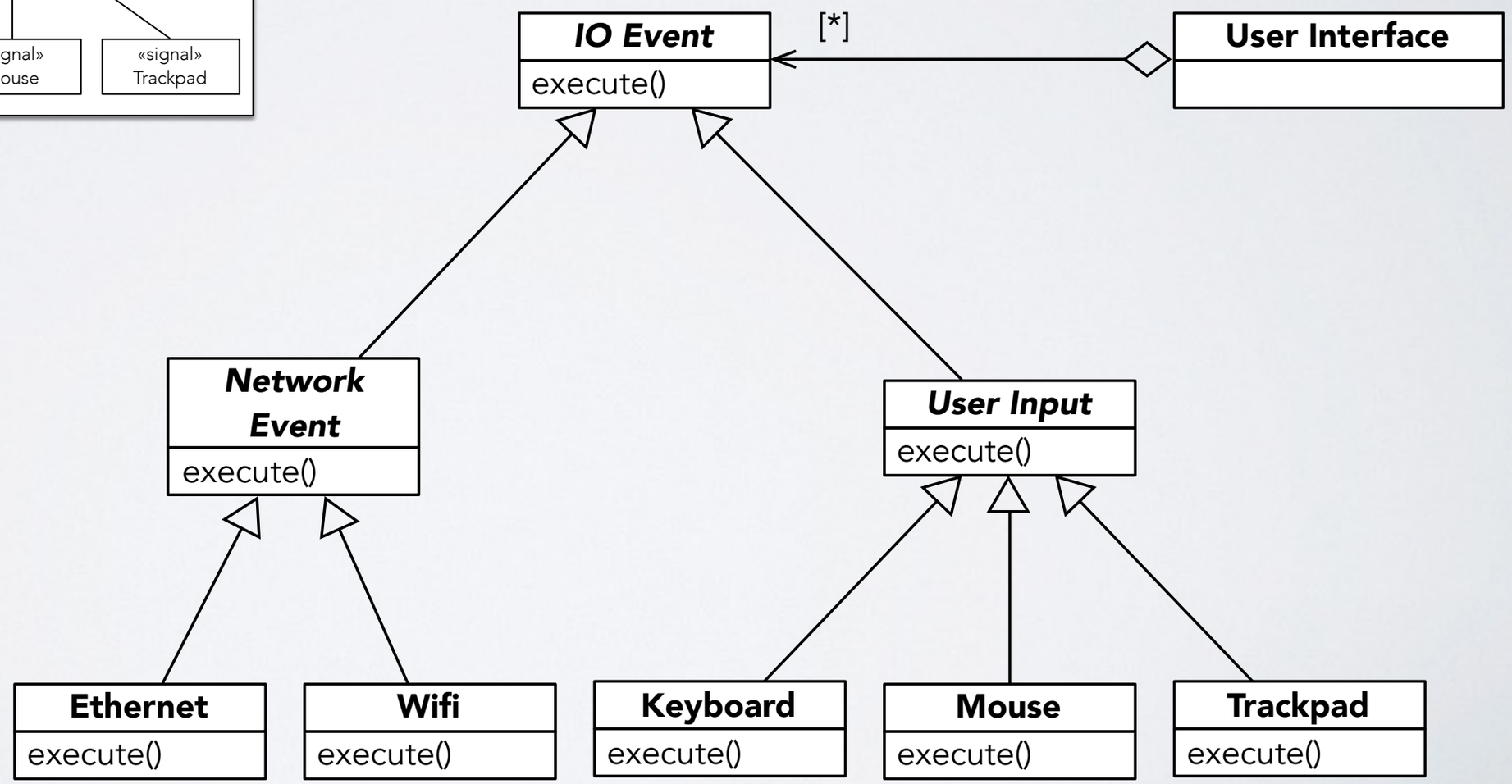
Reify Events

- Apply the Command design pattern to implement events.

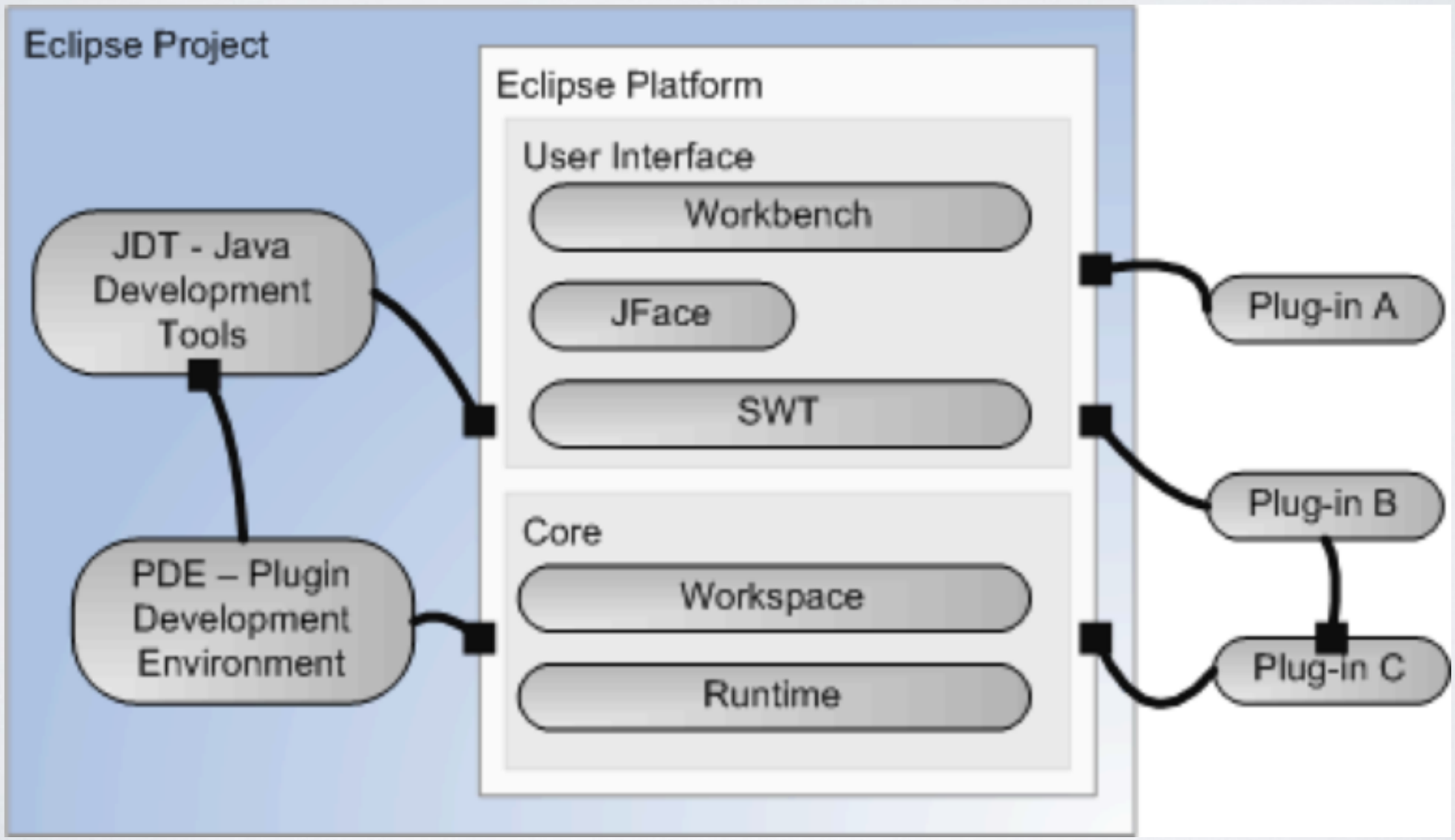


Applying the Command Pattern





Extensibility Example
The Eclipse Project



Plugin Configuration

```
<extension point="org.eclipse.ui.actionSets">
    <actionSet
      id="de.korayguclu.eclipse.actionSet"
      label="Sample Action Set"
      visible="true">
      <menu id="sampleMenu"
            label="Sample &amp;amp;Menu">
        <separator name="sampleGroup"/>
      </menu>
      <action
        id="de.korayguclu.eclipse.actions.SampleAction">
        class="de.korayguclu.eclipse.actions.SampleAction"
        menubarPath="sampleMenu/sampleGroup"
        toolbarPath="sampleGroup"
       label="&amp;amp;Sample Action"
        icon="icons/sample.gif"
        tooltip="Hello, Eclipse world">
     </action>
   </actionSet>
   </extension>
<extension point="org.eclipse.ui.perspectiveExtensions">
    <perspectiveExtension
      targetID="org.eclipse.ui.resourcePerspective">
    <actionSet id="de.korayguclu.eclipse.actionSet"/>
    </perspectiveExtension>
   </extension>
```

Design for Performance

Design for Performance

- Performance is the degree to which a system or component accomplishes its designated functions within given constraints regarding processing time and throughput rate.

- Efficiency is the capability of the software product to provide appropriate performance, relative to the amount of resources used under stated conditions.

Design for Performance Principles

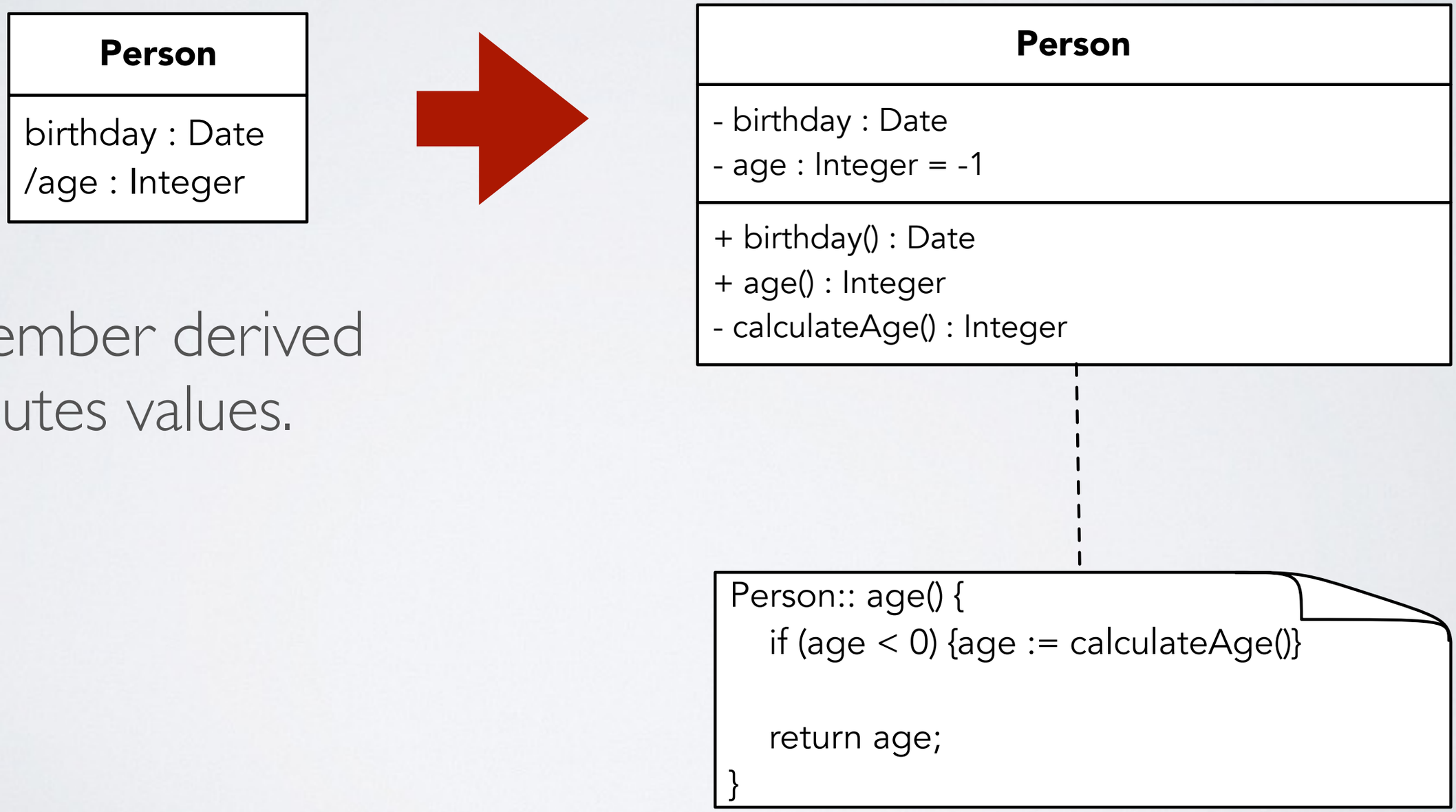
- Implement correctly at first, then measure performance and optimize it, if necessary.
- Maintain consistency between different representations:
    - Verify that the optimized version sis equivalent to the non-optimized one.
- Use, when necessary, different programming paradigms:
    - Create interfaces to separate the two paradigms.
    - Do not mix paradigms: do not pollute one side with other side concepts.
        - Examples: Swig, JNI, P / Invoke.

Code Optimization Rules [M. Jackson]

- First Rule of Program Optimization: Don't do it.
- Second Rule of Program Optimization (for experts only!): Don't do it yet."
- Third Rule of Code Optimization: Profile first.

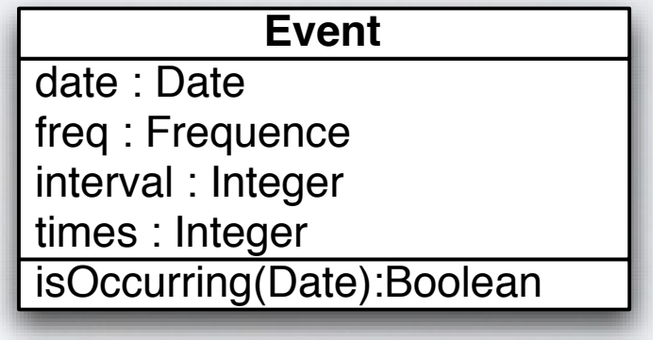
Optimization of Attributes

- Remember derived attributes values.

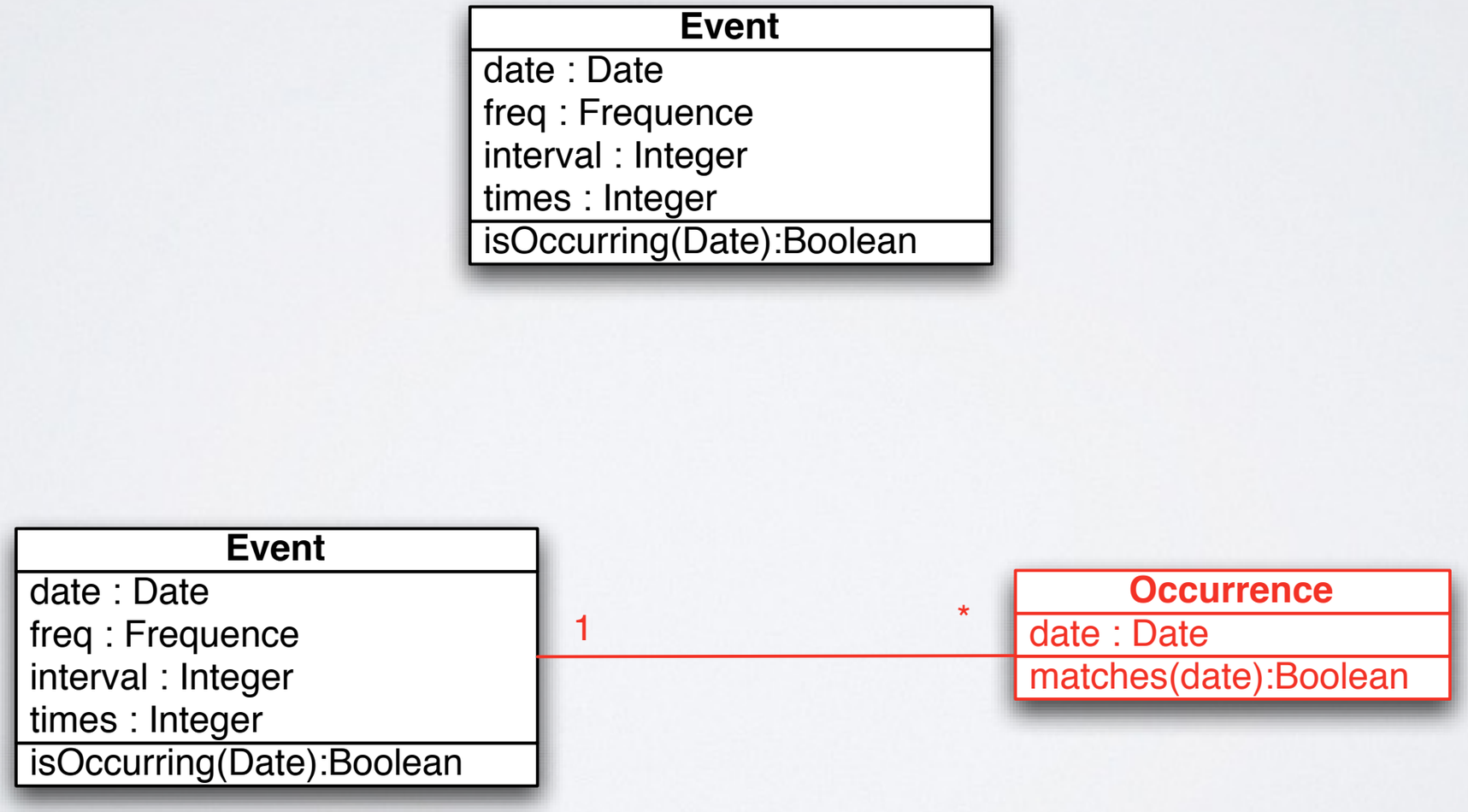


Optimization of Operations

- Add (possibly) redundant attributes and associations to speed up calculation.

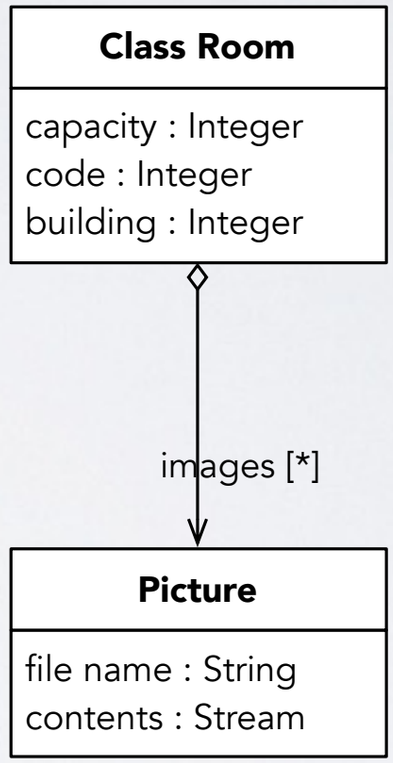


Optimization of Operations

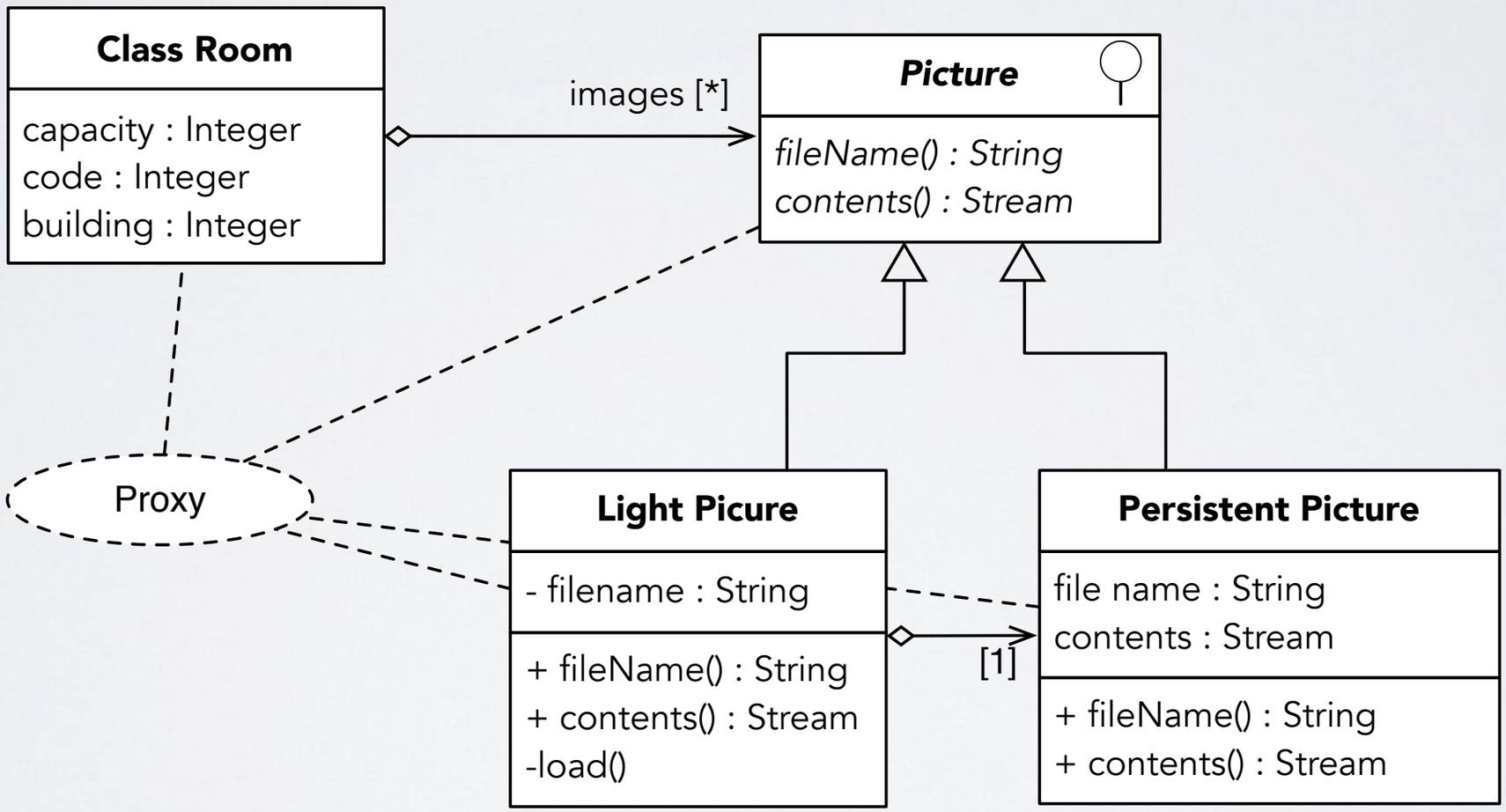


Lazy Loading of Objects

- Load an object only when necessary.
- Apply the Proxy pattern.



Proxy Pattern Application



Design for Portability

Design for Portability

- Portability is the ease with which the software product can be transferred from one hardware or software environment to another.

- Levels of portability:
  - Code: the software is adapted in its source-level, then recompiled for the new target environment.
  - Virtual Machine: the software is compiled into intermediate form (byte code), and executed on platform-specific virtual machines.
  - Binary: This the software is ported directly in its executable form, usually with little adaptation.

Portability Principles

- Control the interfaces: identify all interfaces to the environment, and cast them in a standard form wherever possible.

- Isolate dependencies: recognize the portions of a software unit which must be adapted, and isolate these portions.

- Think portable: the designer must be constantly aware of his part of the likelihood of future porting, and the impact on portability of all design decisions.

Portability at Code Level

- Portability is often treated at code level. Some examples:
  - Java, XUL (XML User Interface Language).
  - Posix (Portable Operating System Interface).
  - Corba (Common Object Request Broker Architecture), etc.
- Limits:
  - Same user interface for different graphical environments.
  - No integration with other software (calendar, address book, etc.)

Portability at Design Level

- Develop different version of the same component.
- For instance, a GUI for: KDE, Gnome, Windows, etc.

Portability in Firefox

- XUL, a powerful widget-based markup language.
- Includes a set of cross-platform widgets
- Based on existing standards
  - Cascading Style Sheets (CSS)
  - Document Object Model (DOM)
  - JavaScript, including E4X (ECMAScript for XML)
  - XML, SVG, MathML
- XUL user interfaces can run on Windows, Mac and Linux (Platform portability).
- Easy localization: locale specific resources are easily separated from presentation and code.

Design for Testability

Design for Testability

- Testability is the capability of the software product to enable modified software to be tested.

- Factors that influence testability:
  - Controllability: the better we can control the software (in isolation), the more and better testing can be done, automated, and optimized.
  - Observability: what you see is what can be tested. Ability to observe the inputs, outputs, states, internals, error conditions, resource utilization, and other side effects of the system under test.

Other Factors

- Suitability: clarity of specifications.
- Stability: how often the code changes
- Performance: how fast it works
- Diagnosability: how fast defects can be localized.

Automated Tests

- A testable software allows automated tests.
- Good automated tests are:
  - Repeatable
  - Easy to write
  - Easy to understand
  - Fast

Improving Controllability

- Use interfaces:
  - they simplify integration test by creating mock objects and stubs.
- Separate Input/Output Interfaces from the rest of the code:
  - they cannot be directly automatized.

Improving Observability

- Use the public visibility when possible
  - private operations cannot be tested.
  - private attributes cannot be read.
- Use a Logging module
  - Define levels (Verbose, Info, Warning, Error, Critical).
  - Use the last two for urgent notifications.
- Use observable middleware (e.g. Corba bus).

Improving Diagnostic

- Use unit tests (small granularity tests).

- Use assertions in code.

```
Person::setAge(i : Integer) {
  assert i >= 0 :“Invalid age”;
```

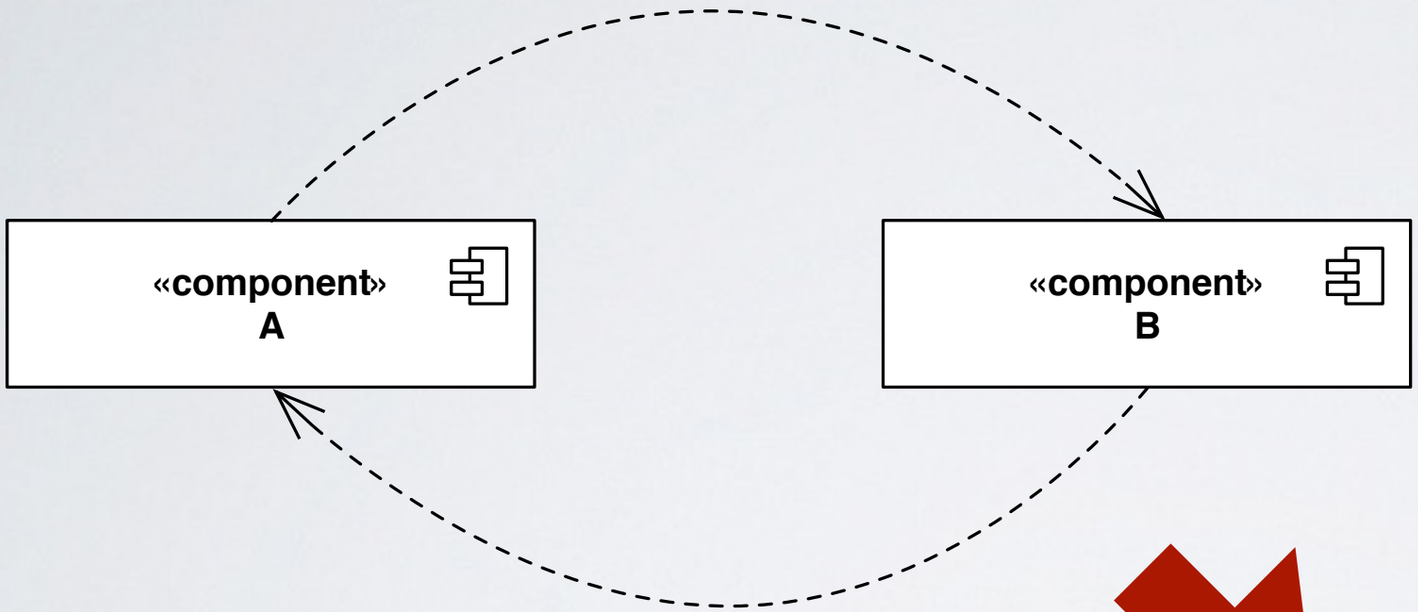
Improving Integration Test

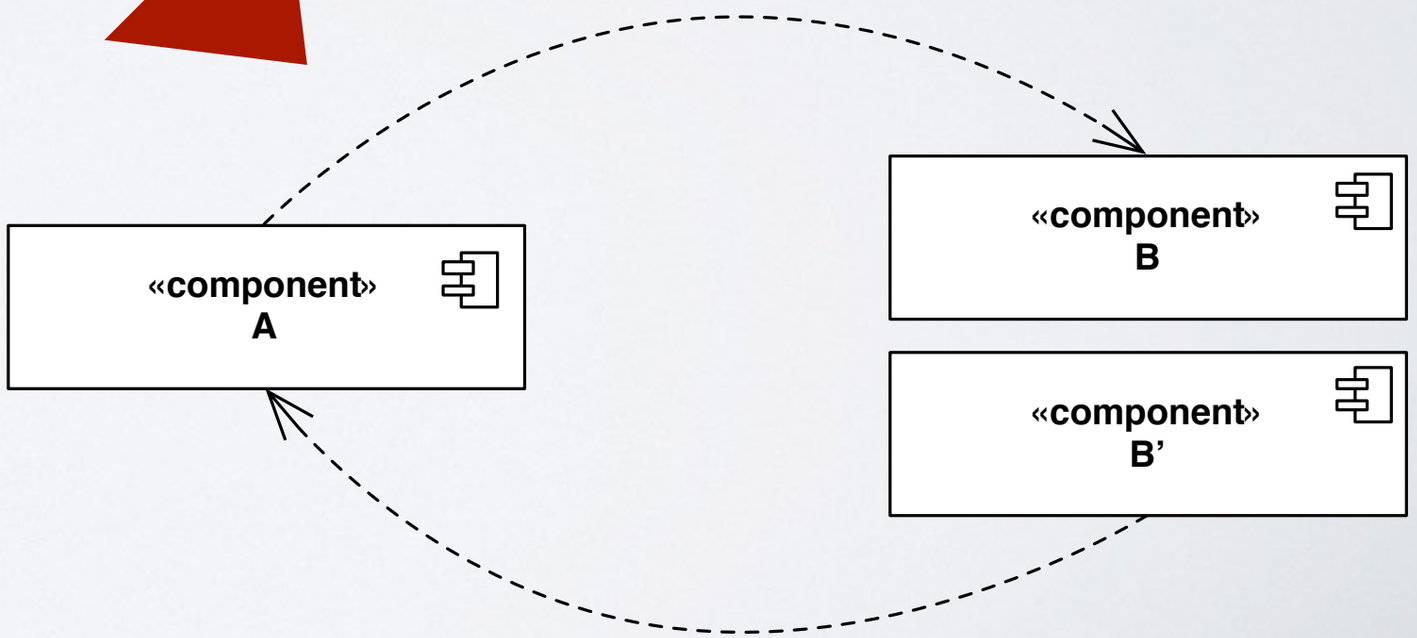
- Use dependencies on class, component, and package diagrams: «uses», «creates», etc.

- Class coupling highlight.

- Simplification of the integration plan.

Avoid Cycles





Adding Details to the Domain Model

Specialization/Generalization Adjustment

- Look for reuse: make similar what is almost similar:
  - possible changes of operation signatures and attribute types.
- Abstract common behaviors and properties.

Attributes

- Precise:
  - Default values
  - Multiplicities: [0..1], [*], etc.
  - Constraints (OCL expressions).
  - Modifiers: readOnly, redefines, etc.
  - Visibility.
  - Use OCL to precise derived attribute.

Association Roles

- Precise the relationships between different roles: union, subsets <x>, redefines <y>, ordered, unique, etc.

- Precise navigability.

Operation Specification

- Use OCL expressions to specify pre- and post-conditions.
- Use the Action Semantics (xUML) to specify operation body.
- Precise parameters directions (in, out, in/out).
- Precise return parameter properties: readOnly, ordered, etc.

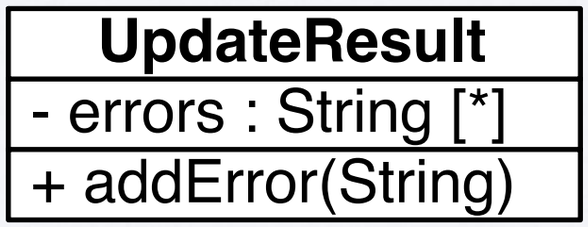
Error Handling

- Error handling alternatives:
  - Use return values: use a value convention (e.g. C language)
  - Use of exceptions: handle exceptions at the lowest possible level.
  - Separate errors from exception: the latter are expected, the former no.

Collecting Parameters Pattern

[Beck96]

...when you need to collect results over several methods, you should add a parameter to the method and pass an object that will collect the results for you...”



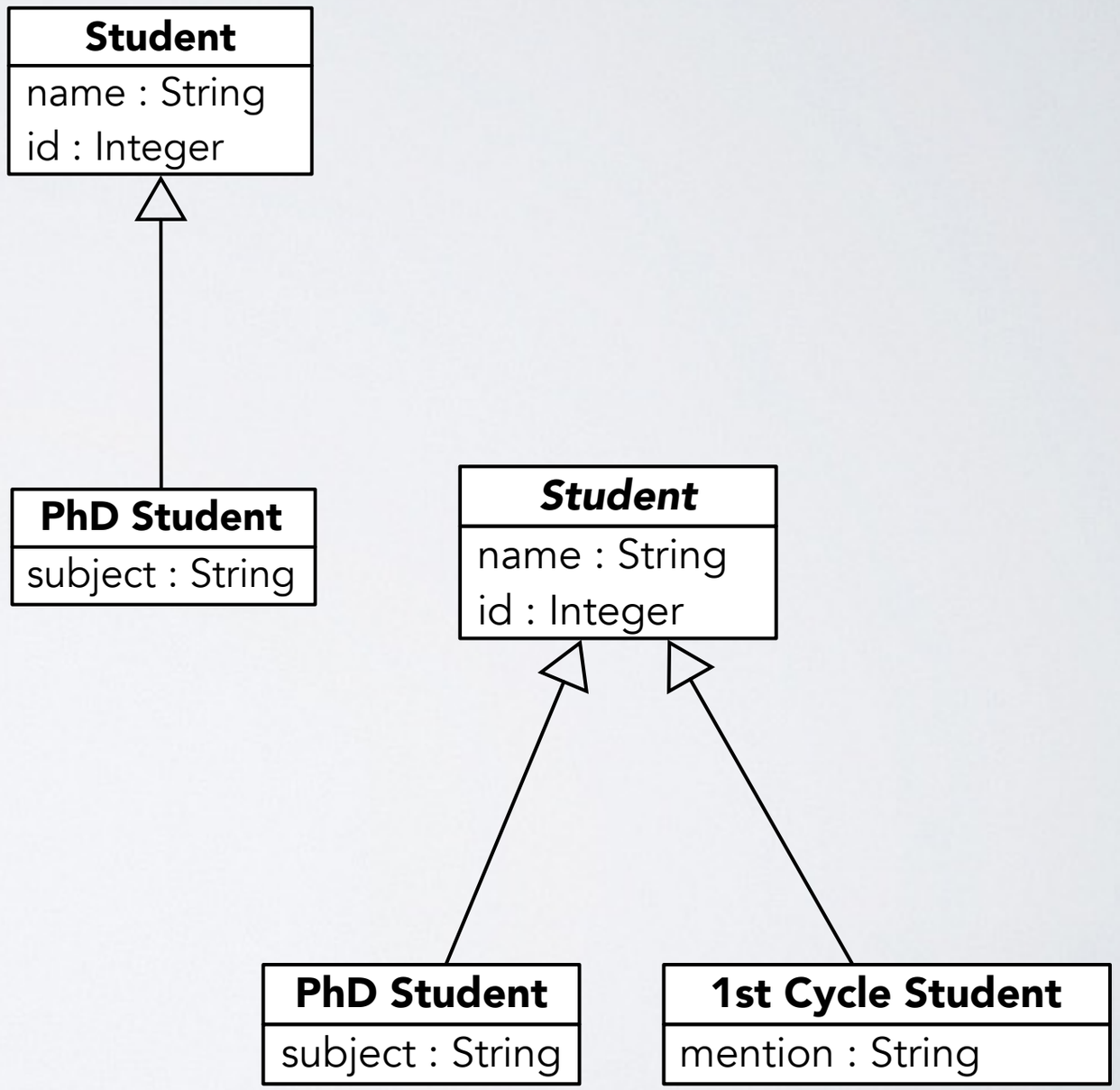
Design Heuristics

Heuristics

- Avoid god classes:
  - systems composed of a class that controls everything and of several data classes.
- Distinguish specialization and aggregation:
  - if the choice is possible, use aggregation instead of specialization.

Specialization

- Avoid concrete classes specialization.
- Superclasses should be abstract.
- Super-classes should not know subclasses.
- Specialization hierarchies should be shallow.



Specialization (2/2)

- If two classes share the same properties but not the same behavior, those properties should be moved to a third class.

- the behavior related to these properties will intuitively follow them.

Attributs

- Attributes should be hidden
  - Setter and Getter methods are not essentials: they prevent good design choices.
- Prefer private attributes rather than protected and protected rather than public.

Classes

- Clients of a class must be dependent on its public interface, but a class should not be dependent on its clients.

- Define a minimal interface for all classes. For instance: copy(), deepCopy(), equals(), fromString(), toString(), etc.

Classes (2/2)

- A class should represent one and only one abstraction.
- The classes are abstractions of the real and not the different roles played by an object.
- Be suspicious with classes whose names are verbs:
  - they may be an operation.
  - in some cases, they are indeed an operation (see Command and Strategy patterns).

Operations

- Feature envy: operations of a class should use properties (and other operations) of this same class.

- Common behavior of public operations should be placed in private methods.

Conclusion

- Separate interface and implementation.
- Separate what is common from what is variable.
- Use a common interface to allow variable implementations to be replaced.
- Use design patterns for ensuring variability and extensibility.

References

- "Object-Oriented Design Heuristics". Arthur J. Riel, May 10, 1996.
- "Pattern-Oriented Software Architecture: Patterns for Concurrent & Networked Objects". Schmidt et al., Wiley and Sons, 2000, ISBN 0-471-60695-2.
- "Pattern-Oriented Software Architecture -- a System of Patterns". Buschmann, Muenier, Rohnert, Sommerlad, and Stal, John. Wiley & Son, 1996.
- "Bringing Portability to the Software Process". James D. Mooney, West Virginia University.. Technical Report. 1997.

Detailed Design