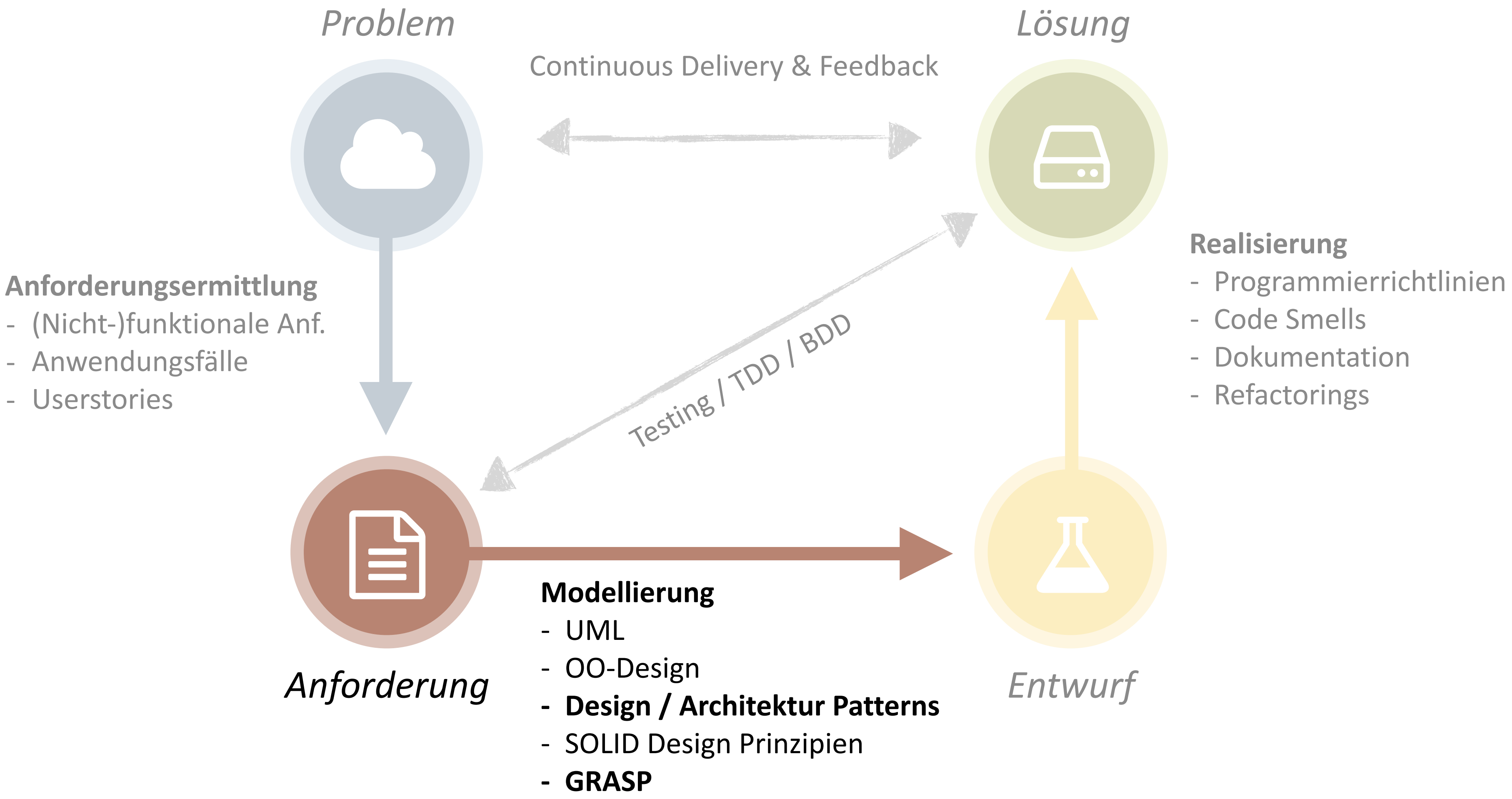
Software Engineering

6. Design Patterns

Jonathan Brachthäuser

Einordnung



Problem

Lösung

Anforderung

Entwurf

Continuous Delivery & Feedback

Testing / TDD / BDD

Anforderungsermittlung

- (Nicht-)funktionale Anf.
- Anwendungsfälle
- Userstories

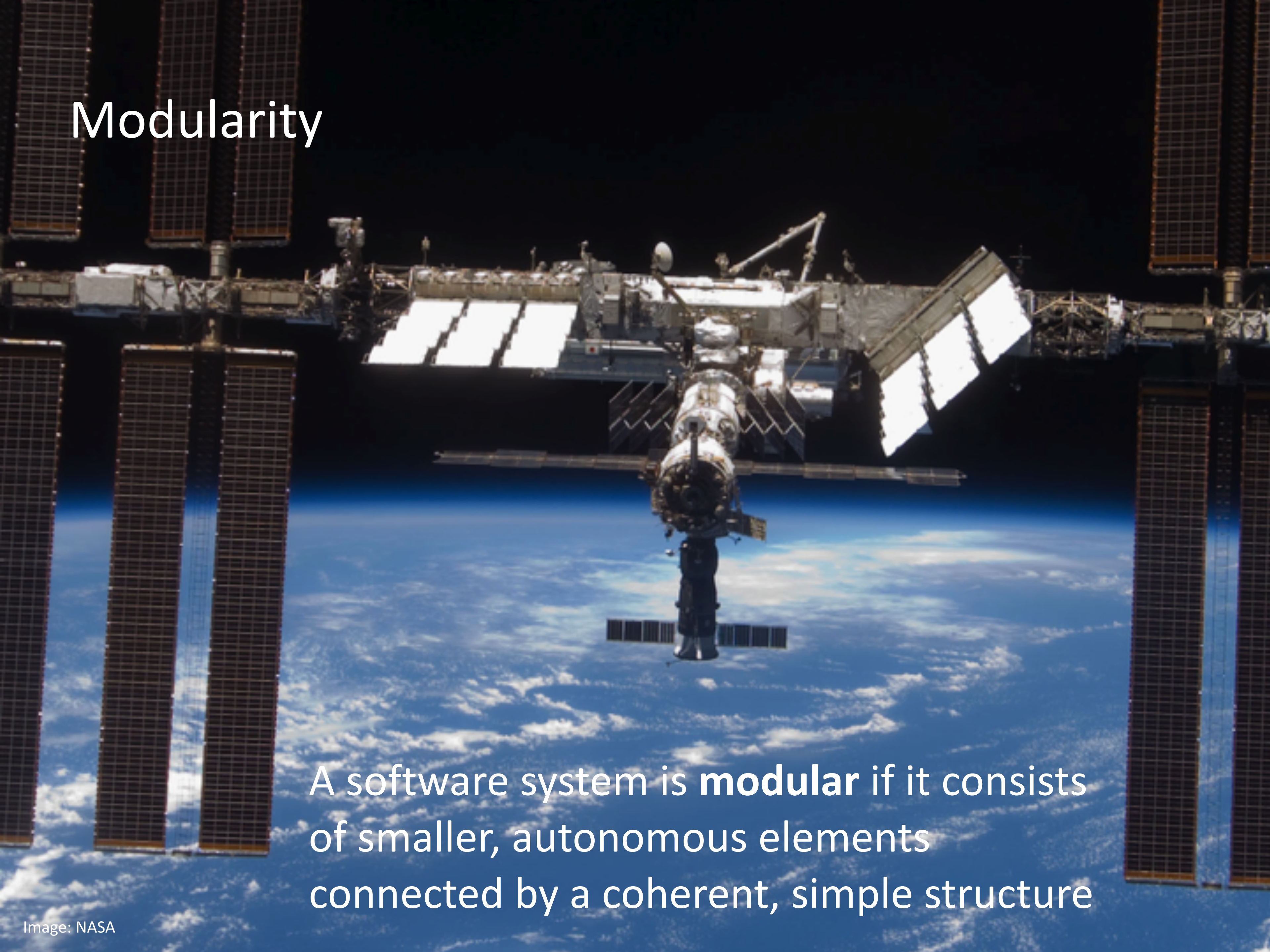
Realisierung

- Programmierrichtlinien
- Code Smells
- Dokumentation
- Refactorings

Modellierung

- UML
- OO-Design
- **Design / Architektur Patterns**
- SOLID Design Prinzipien
- **GRASP**

Modularity



A software system is **modular** if it consists of smaller, autonomous elements connected by a coherent, simple structure

Image: NASA

GRASP (continued)

Nine GRASP patterns:

- Information Expert
- Creator
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Controller

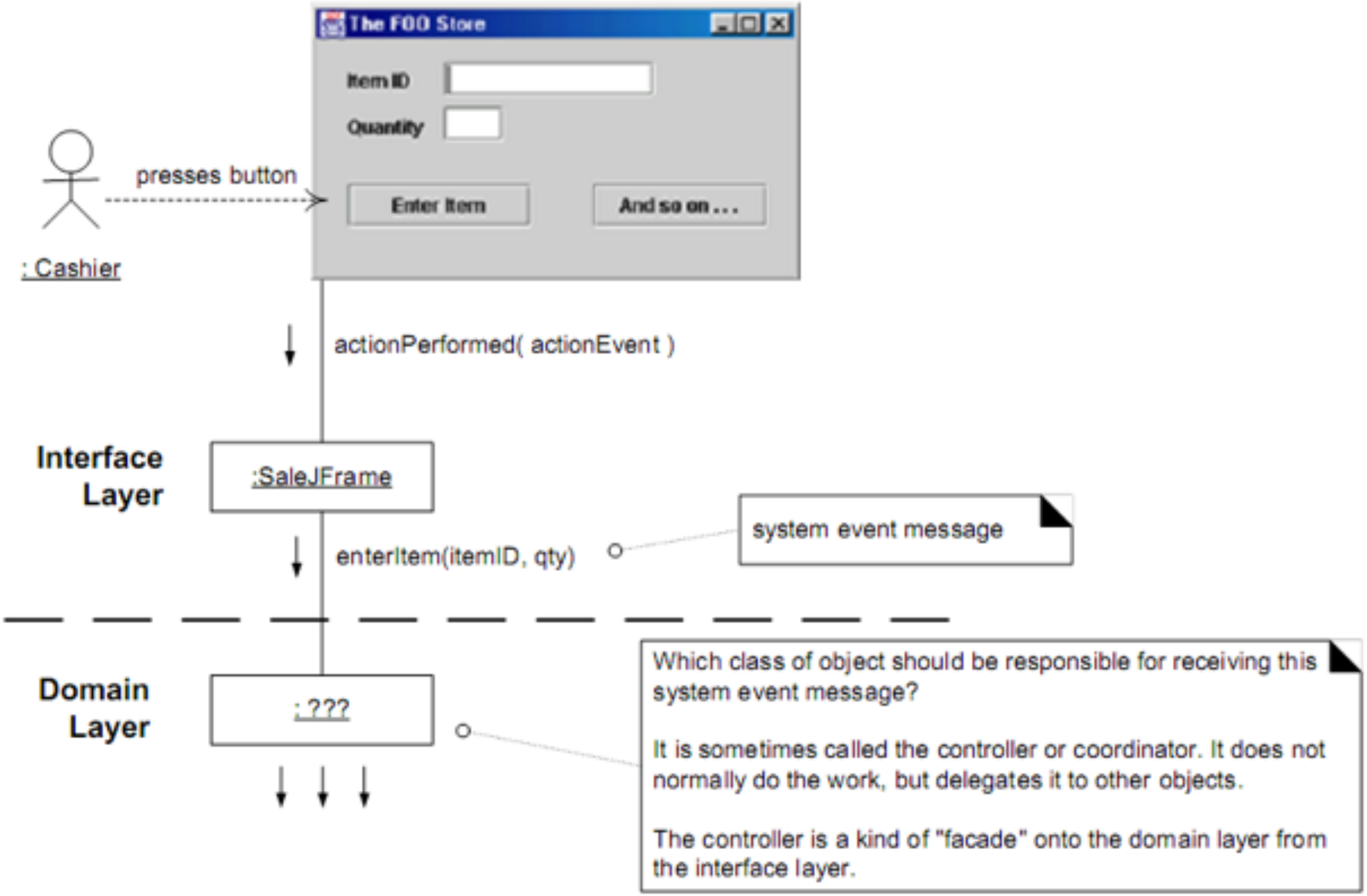
Problem:

Who should be responsible for handling an input system event?

Solution:

Assign the responsibility for receiving or handling a system event message **to a class representing the overall system**, device, or **subsystem** (facade controller) or a **use case scenario** within which the system event occurs (use case controller)

Controller: Example



system event message

Which class of object should be responsible for receiving this system event message?

It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.

The controller is a kind of "facade" onto the domain layer from the interface layer.

Controller: Discussion

- Normally, a controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. It does not do much work itself.
- Facade controllers are suitable when there are not "too many" system events
- A use case controller is an alternative to consider when placing the responsibilities in a facade controller leads to designs with low cohesion or high coupling
  - typically when the facade controller is becoming "bloated" with excessive responsibilities.

Controller: Discussion

- Benefits
  - Increased potential for reuse, and pluggable interfaces
    - No application logic in the GUI
  - Dedicated place to place state that belongs to some use case
    - E.g. operations must be performed in a specific order
- Avoid bloated controllers!
  - E.g. single controller for the whole system, low cohesion, lots of state in controller
  - Split into use case controllers, if applicable
- Interface layer does not handle system events

Nine GRASP patterns:

- Information Expert
- Creator
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Polymorphism

Problem:

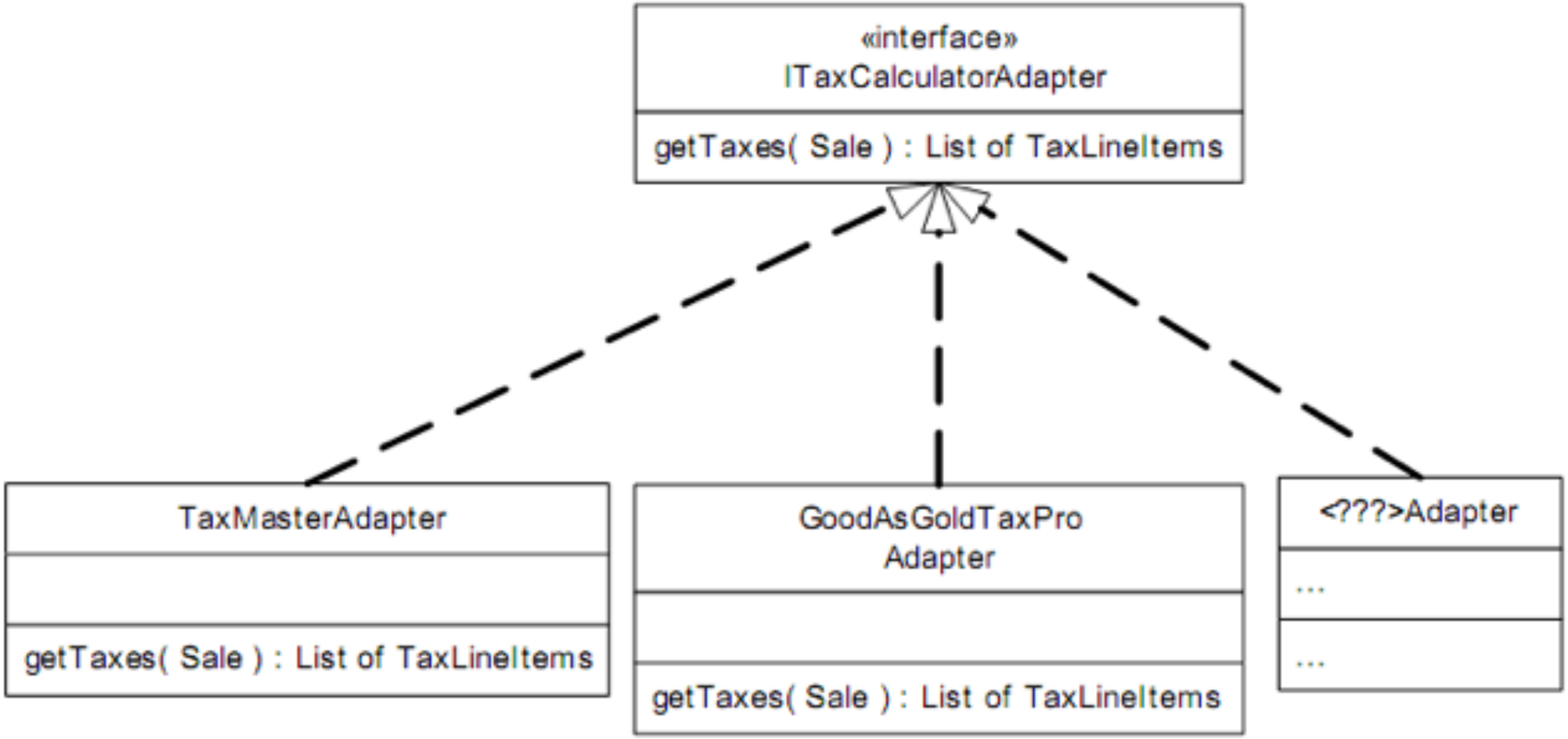
How to handle alternatives based on types?

How to create pluggable software components?

Solution:

When alternate behaviours are selected based on the type of an object, use **polymorphic method call** to select the behaviour, rather than using if/case statement to test the type.

Polymorphism: Example


«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List of TaxLineItems
TaxMasterAdapter
getTaxes(Sale) : List of TaxLineItems
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List of TaxLineItems
<???>Adapter
...
...

Polymorphism: Discussion

- Polymorphism is a fundamental principle in designing how a system is organized to handle similar variations.
- Properties:
  - Easier and more reliable than using explicit selection logic
  - Easier to add additional behaviors later on
  - Increases the number classes in a design
  - May make the code less easy to follow

Nine GRASP patterns:

- Information Expert
- Creator
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Pure Fabrication

Problem:

Adding some responsibilities to domain objects would violate high cohesion/low coupling/reuse

Solution:

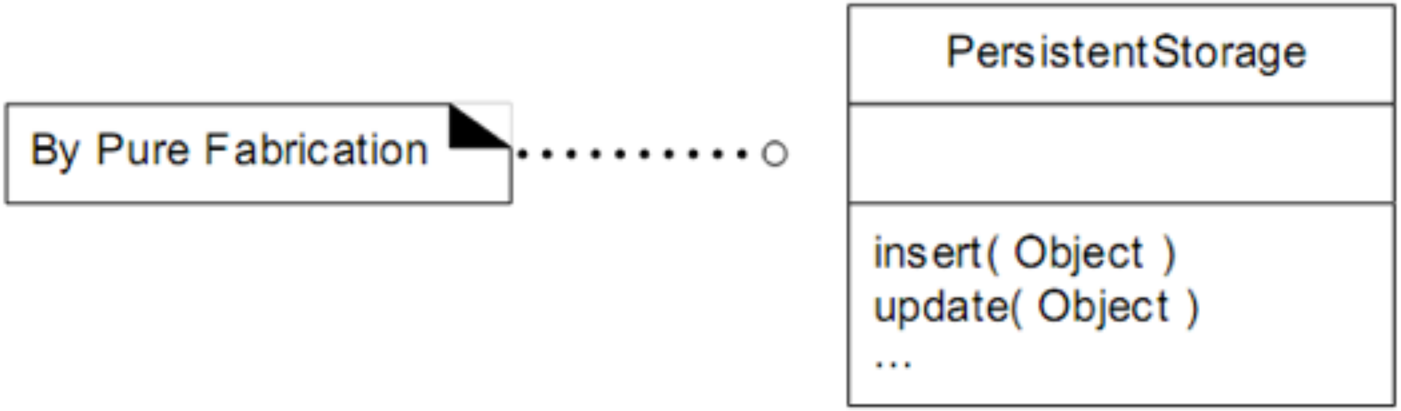
Assign a highly cohesive set of responsibilities to an **artificial or convenience class** that does not represent a problem domain concept—something made up, to support high cohesion, low coupling, and reuse.

Pure Fabrication: Example

- In the point of sale example support is needed to save Sale instances in a relational database.
- By Expert, there is some justification to assign this responsibility to Sale class.
- However, the task requires a relatively large number of supporting database-oriented operations and the Sale class becomes incohesive.
- The sale class has to be coupled to the relational database increasing its coupling.
- Saving objects in a relational database is a very general task for which many classes need support. Placing these responsibilities in the Sale class suggests there is going to be poor reuse or lots of duplication in other classes that do the same thing.

Pure Fabrication : Example

- Solution: create a new class that is solely responsible for saving objects in a persistent storage medium
- This class is a Pure Fabrication



- The Sale remains well-designed, with high cohesion and low coupling
- The PersistentStorageBroker class is itself relatively cohesive
- The PersistentStorageBroker class is a very generic and reusable object

Pure Fabrication: Discussion

- The design of objects can be broadly divided into two groups:
  - Those chosen by representational decomposition (e.g. Sale)
  - Those chosen by behavioral decomposition (e.g. an algorithm object such as TOCGenerator or PersistentStorage)
- Both choices are valid designs, although the second one corresponds less well to the modeling perspective on objects

Nine GRASP patterns:

- Information Expert
- Creator
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Indirection

Problem:

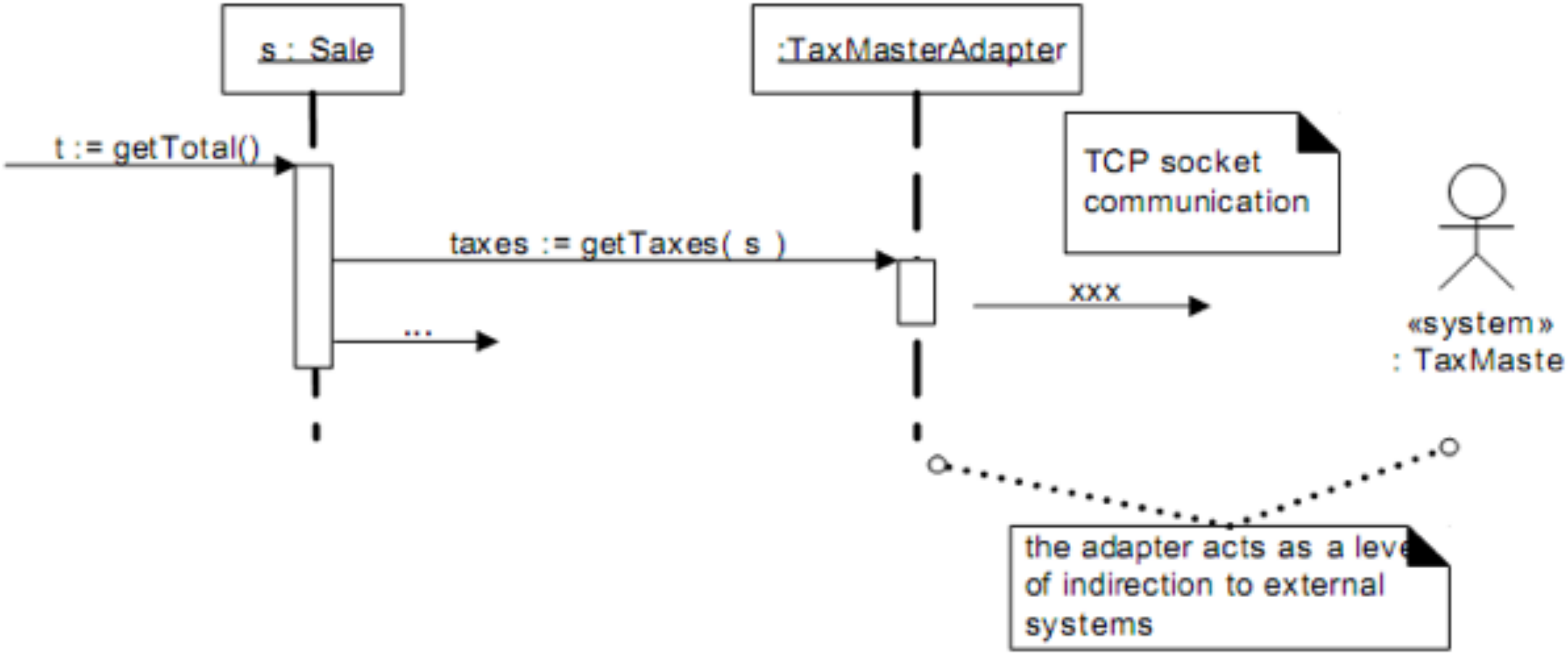
Where to assign a responsibility, to avoid direct coupling between two (or more) things?
How to de-couple objects so that low coupling is supported and reuse potential remains higher?

Solution:

Assign the responsibility to **an intermediate object** to mediate between other components or services, so that they are not directly coupled.

"Most problems in computer science can be solved
by another level of indirection"

Indirection: Example



By adding a level of indirection and adding polymorphism, the adapter objects protect the inner design against variations in the external interfaces

Nine GRASP patterns:

- Information Expert
- Creator
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Protected Variation

Problem:

How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?

Solution:

Identify points of predicted variation or instability; **assign responsibilities to create a stable interface** around them.

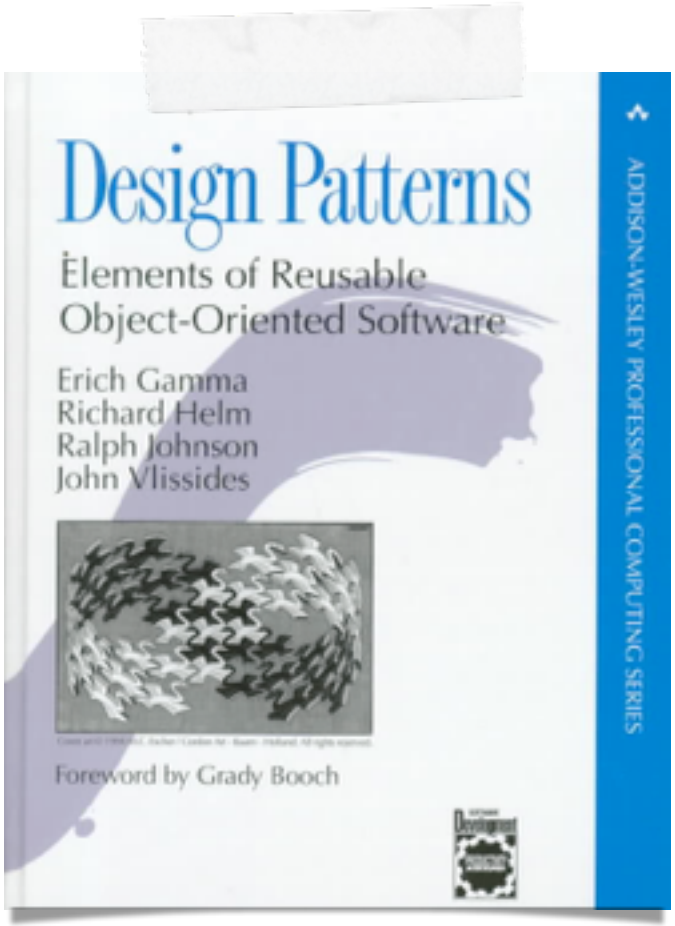
Note: This is basically just another formulation of the information hiding principle.

Protected Variation: Examples

- Data encapsulation, interfaces, polymorphism, indirection, and standards are motivated by PV.
- Virtual machines are complex examples of indirection to achieve PV
- Service lookup: Clients are protected from variations in the location of services, using the stable interface of the lookup service.
- Uniform Access Principle
- Law of Demeter
- …

Design Patterns

Literaturhinweis


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

What is a pattern?

A design pattern describes:

- A **problem that occurs over and over again** in our environment.
- The core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Christopher Alexander, professor of architecture.

What is a pattern?

Aggressive disregard of orginality.

Rule of three:

- Once it is an event.
- Twice it is an accident.
- Thrice it is a pattern.

R. Martin's Chess Analogy

When people begin to play chess they learn the rules and physical requirements of the game. They learn the names of the pieces, the way they move and capture, the board geometry and orientation.
At this point, people can play chess, although they will probably be not very good players.

As they progress, they learn the principles. They learn the value of protecting the pieces, and their relative value. They learn the strategic value of the center squares and the power of a threat…

At this point, they can play a good game. They know how to reason through the game and can recognize "stupid" mistakes.

However, **to become a master** of chess, one must **study games of other masters**. Buried in those games are patterns that must be understood, memorized, and applied repeatedly until they become second nature.

There are thousands upon thousands of these patterns. Opening patterns are so numerous that there are books dedicated to their variations. Midgame patterns and ending patterns are also prevalent, and the master must be familiar with them all.

Elements of Design Patterns

- **Pattern Name**
  - A short mnemonic to increase your design vocabulary.
- **Intent**
  - Description when to apply the pattern (conditions that have to be met before it makes sense to apply the pattern).
- **Solution**
  - The elements that make up the design, their relationships, responsibilities and collaborations.
- **Consequences**
  - Costs and benefits of applying the pattern. Language and implementation issues as well as impact on system flexibility, extensibility, or portability. The goal is to help understand and evaluate a pattern.

Topics of this Lecture

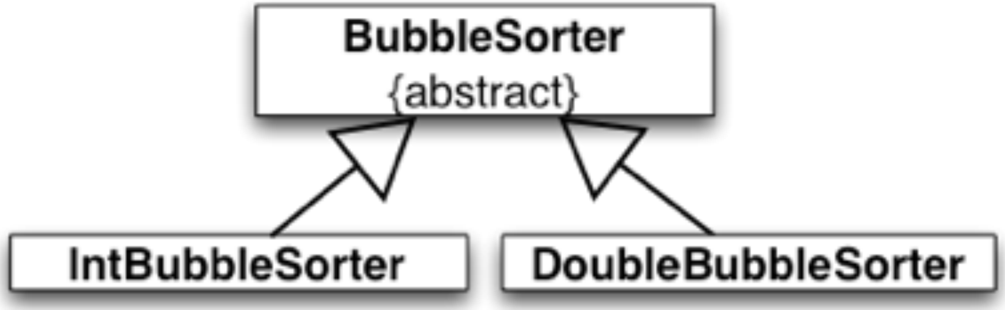
- Design Patterns
  - Template
  - Strategy
  - Decorator
- These design patterns are less general than the GRASP patterns
  - They focus on specific design problems
- These are some of the most common and most important classical design patterns in OO design

Template Method Pattern

- **Intent**
  - Separate policies from detailed mechanisms / invariant and variant parts.
- **Solution**
  - Abstract classes define interfaces and implement high-level policies
  - Detailed mechanisms are implemented in subclasses

- Avoid code duplication.
- The Template Method Pattern is at the core of the design of object-oriented frameworks.

Using the Template Method Pattern for Bubble-Sort

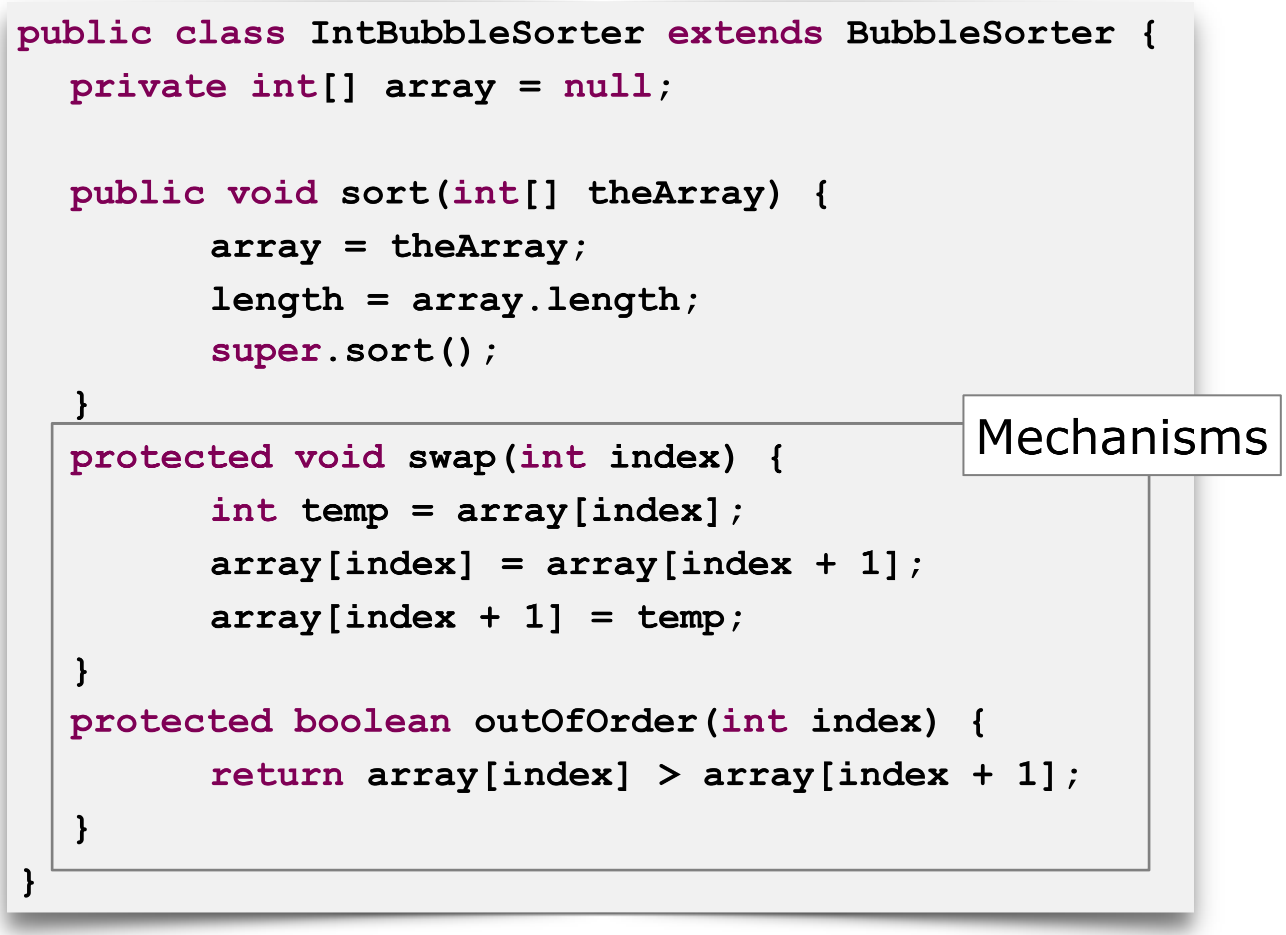


```java
public abstract class BubbleSorter {

    protected int length = 0;

    protected void sort() {
        if (length <= 1) return;
        for (int nextToLast = length - 2; nextToLast >= 0; nextToLast--)
            for (int index = 0; index <= nextToLast; index++)
                if (outOfOrder(index))
                    swap(index);
    }

    protected abstract void swap(int index);
    protected abstract boolean outOfOrder(int index);
}
```

Policy

Mechanisms

Filling the template for a specific sorting algorithmn



```java
public class IntBubbleSorter extends BubbleSorter {
  private int[] array = null;

  public void sort(int[] theArray) {
        array = theArray;
        length = array.length;
        super.sort();
  }
  protected void swap(int index) {
        int temp = array[index];
        array[index] = array[index + 1];
        array[index + 1] = temp;
  }
  protected boolean outOfOrder(int index) {
        return array[index] > array[index + 1];
  }
}
```

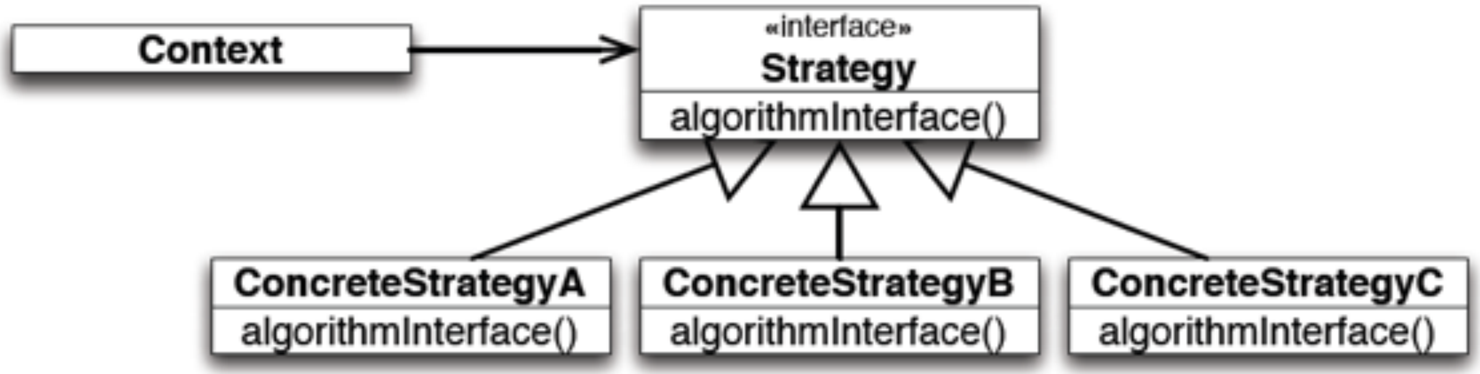
Mechanisms

Consequences

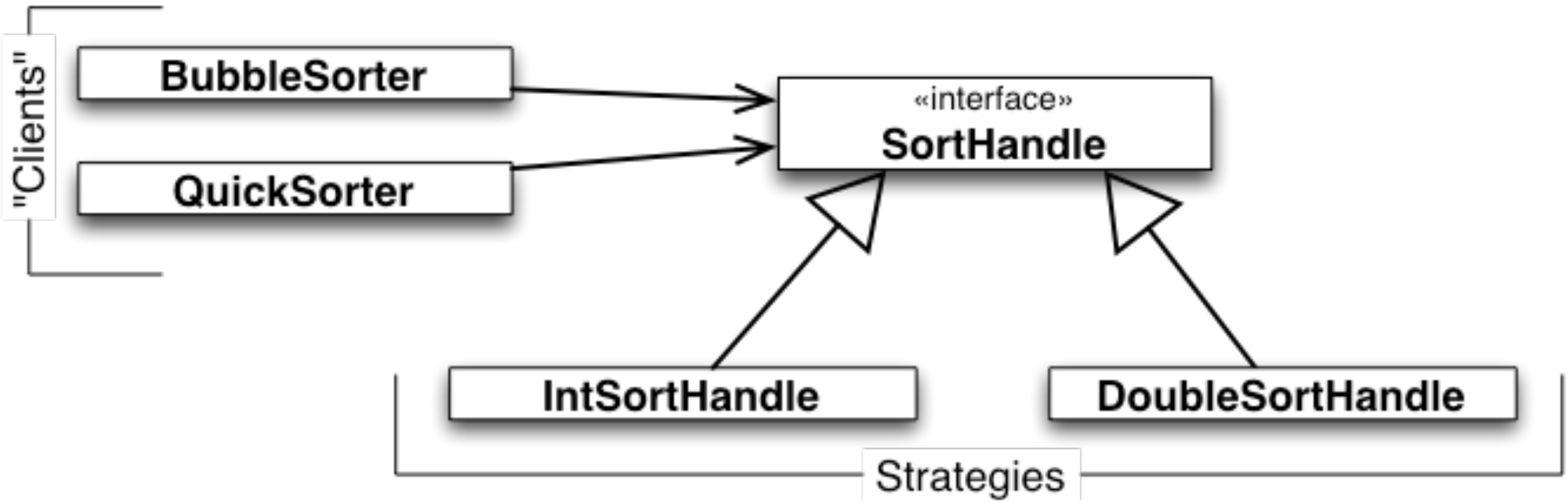
- Template method forces detailed implementations to extend the template class.
- Detailed implementation depend on the template.
- Cannot re-use detailed implementations‘ functionality. (E.g., swap and out-of-order are generally useful.)
- If we want to re-use the handling of integer arrays with other sorting strategies we must remove the dependency
  - this leads us to the Strategy Pattern.

Strategy Pattern

- **Intent**
  - Define a **family of algorithms**, encapsulate each one, and make them **interchangeable**. Strategy lets the algorithm vary independently from clients that use it.
- **Solution**
  - Define an interface for the varying implementation details, implement in concrete strategies which can be passed along.



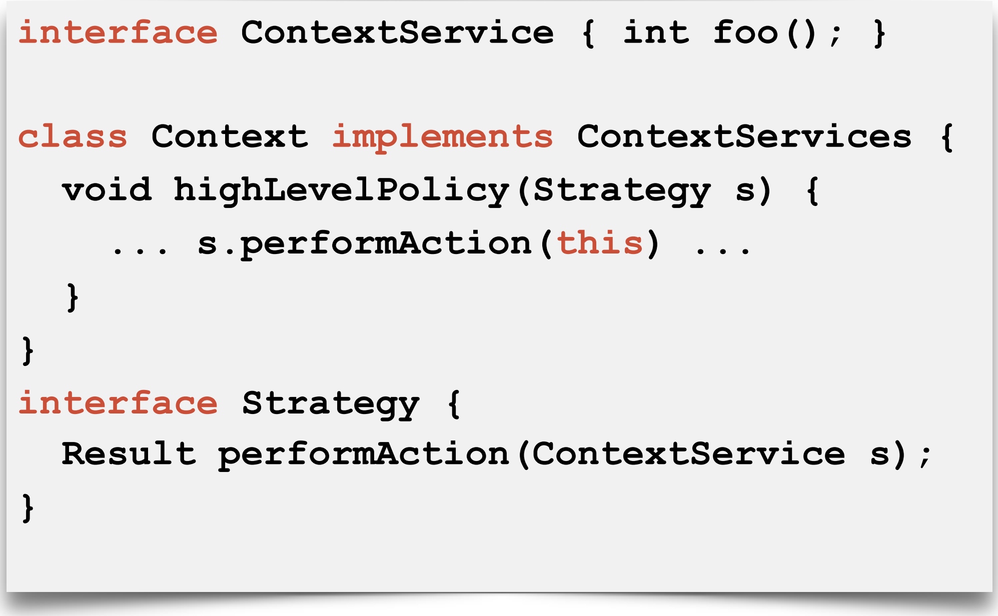
Strategy Pattern: Example



Giving the Strategy Visibility for the Context

- Two possible approaches:
  - **Pass the needed information as a parameter.**
    - Context and Strategy decoupled.
    - Communication overhead.
    - Algorithm can't be adapted to specific needs of context.
  - **Context passes itself as a parameter or Strategy has a reference to its Context.**
    - Reduced communication overhead.
    - Context must define a more elaborate interface to its data.
    - Closer coupling of Strategy and Context.

Example: Passing Context Information to Strategy



```
interface ContextService { int foo(); }

class Context implements ContextServices {
  void highLevelPolicy(Strategy s) {
    ... s.performAction(this) ...
  }
}
interface Strategy {
  Result performAction(ContextService s);
}
```

Strategy Pattern: Discussion

- Use if many related classes only differ in their **behavior** rather than implementing different related abstractions.
  - Strategies allow to configure a class with one of many behaviors.
- Use if you need different variants of an algorithm.
  - Strategies can be used when variants of algorithms are implemented as a class hierarchy.
- Use if you need to modify the behavior of a class at runtime (dynamically).
  - i.e. for instance the "Attack" of a computer game character depends on its equipment which changes at runtime.

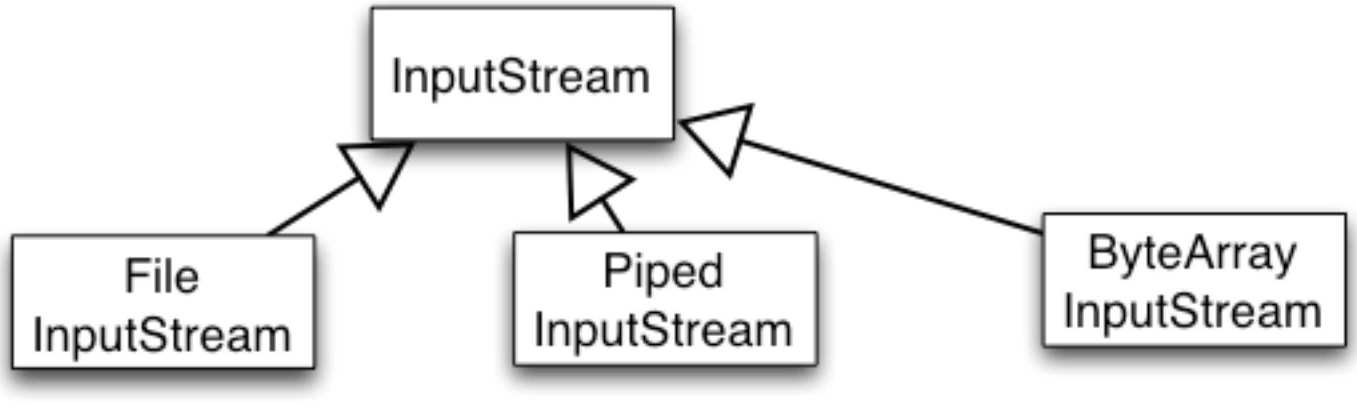
Decorator Pattern

- **Intent**
  - We need to add responsibilities to existing individual objects
  - … dynamically and transparently, without affecting other objects.
  - … responsibilities can be withdrawn dynamically.
- **Solution**
  - Create a decorator (wrapper) around the individual object
  - ... the decorator can modify the behavior of individual methods before (or after) forwarding to the underlying object

Motivation: Limitations of Inheritance

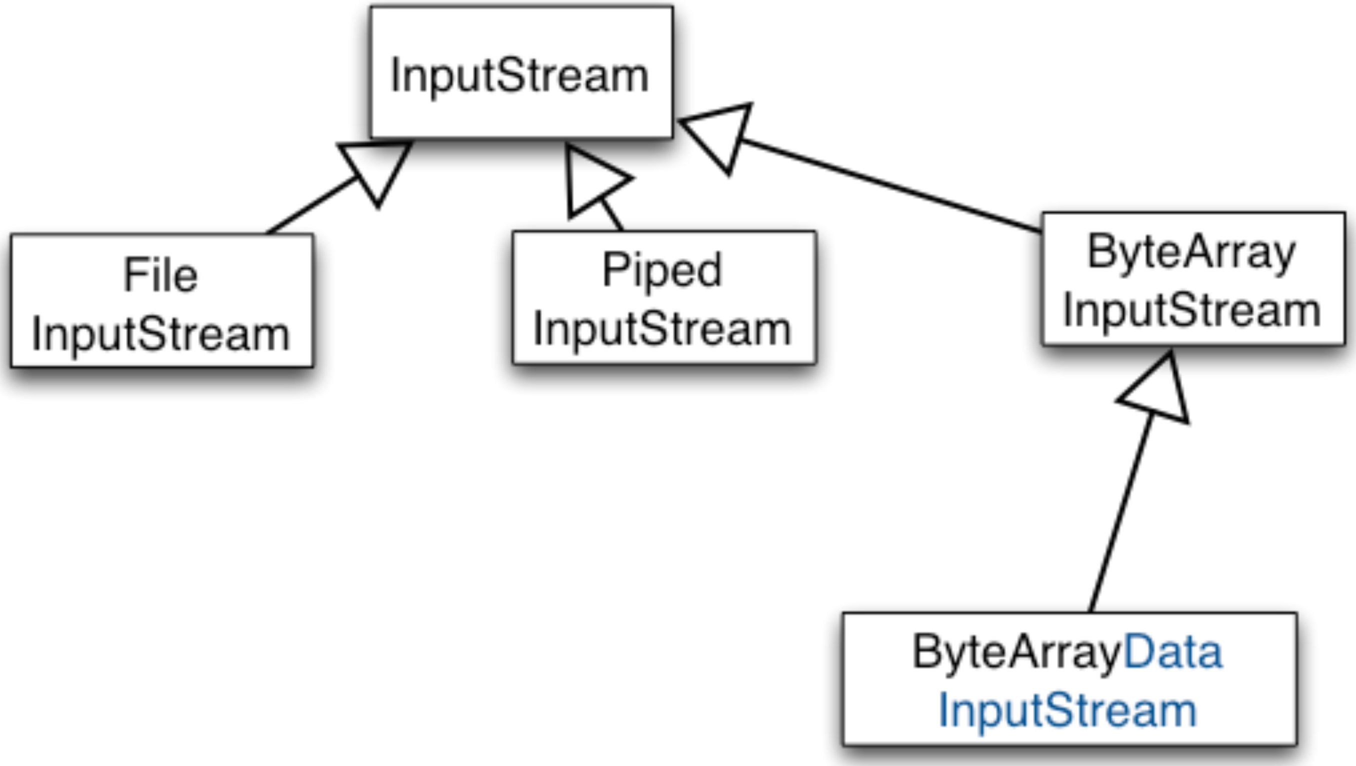
- Only using inheritance would produce an explosion of subclasses to support every combination.
- No support for dynamic adaptation.
- A class definition may be hidden or otherwise unavailable for subclassing
- Cannot change all constructor calls to the class whose object are to be extended
  - A call to `new` fixes the runtime behavior once and for all!

Limitations of Inheritance: Example



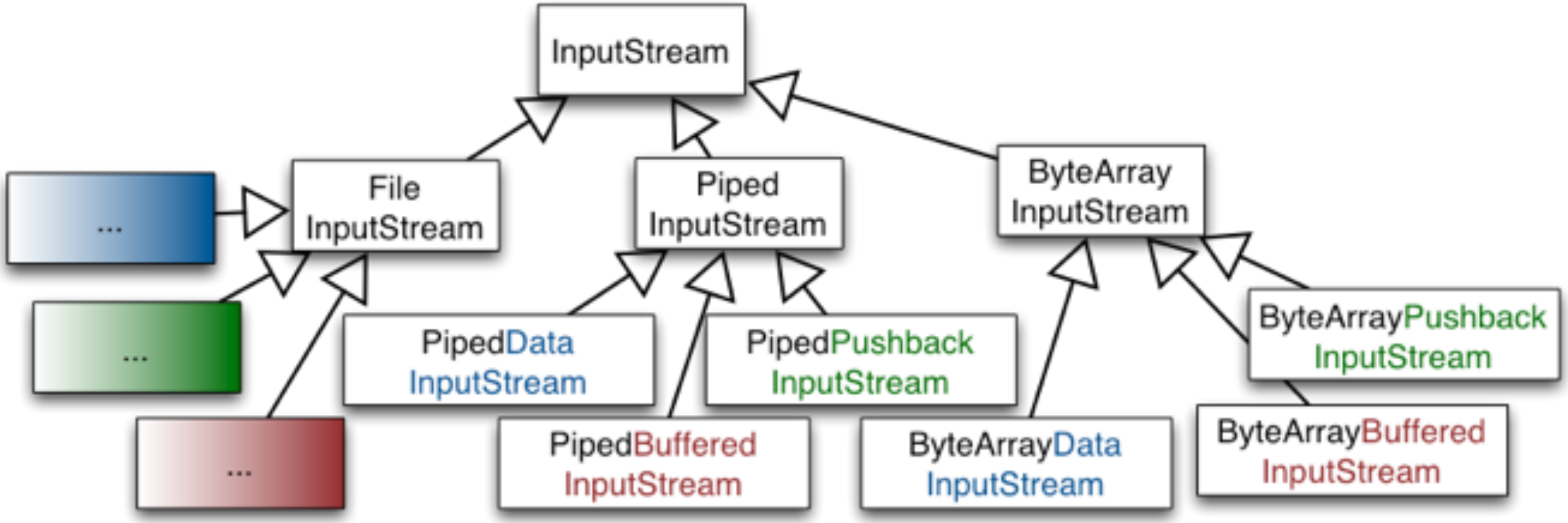
Evolution:
Adding functionality to a ByteArrayInputStream to read whole sentences and not just single bytes.

Limitations of Inheritance: Example



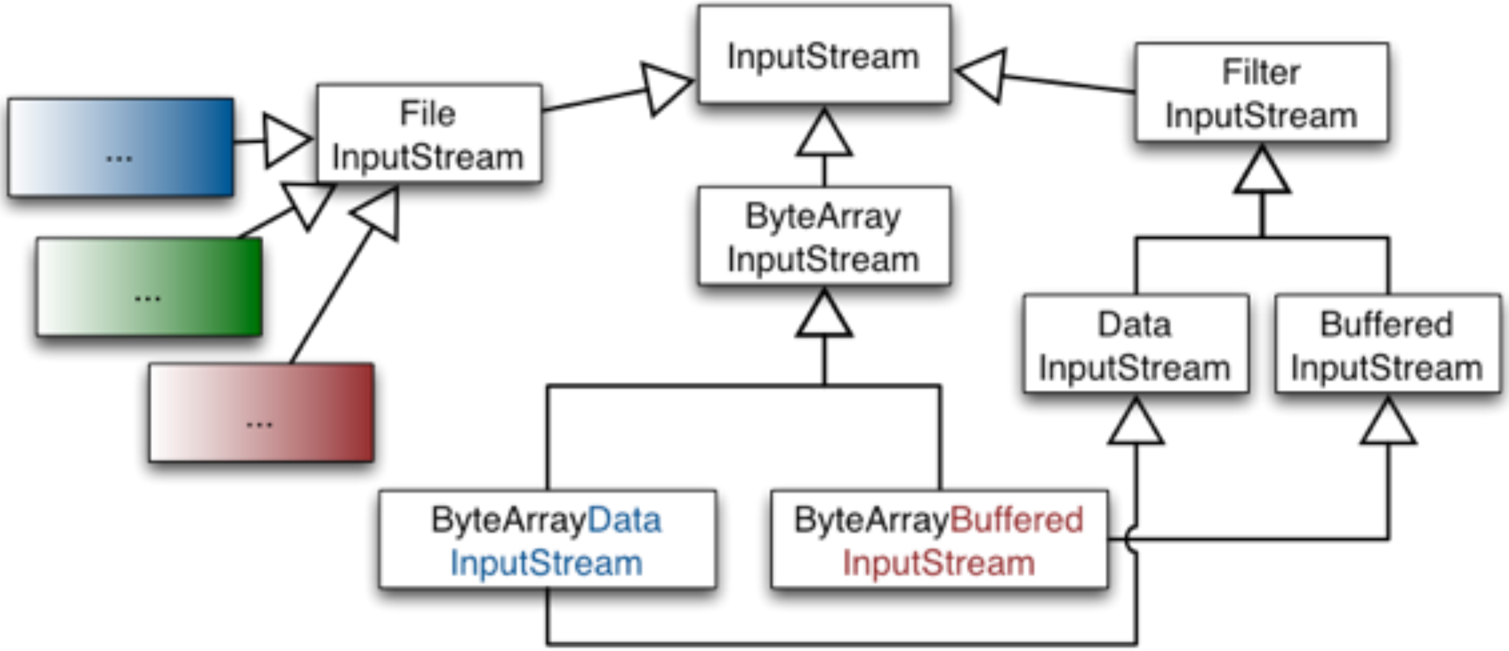
Evolution:
We also want to have the possibility to read whole sentences using FileInputStreams...

After the n-th iteration…


InputStream
File InputStream
Piped InputStream
ByteArray InputStream
...
...
...
PipedData InputStream
PipedPushback InputStream
PipedBuffered InputStream
ByteArrayData InputStream
ByteArrayPushback InputStream
ByteArrayBuffered InputStream

- Problems:
  - … a new class for each responsibility.
  - responsibility mix fixed statically.
    (How about PipedDataBufferedInputStream?)
  - non-reusable extensions; code duplication;
  - maintenance nightmare: exponential growth of number of classes

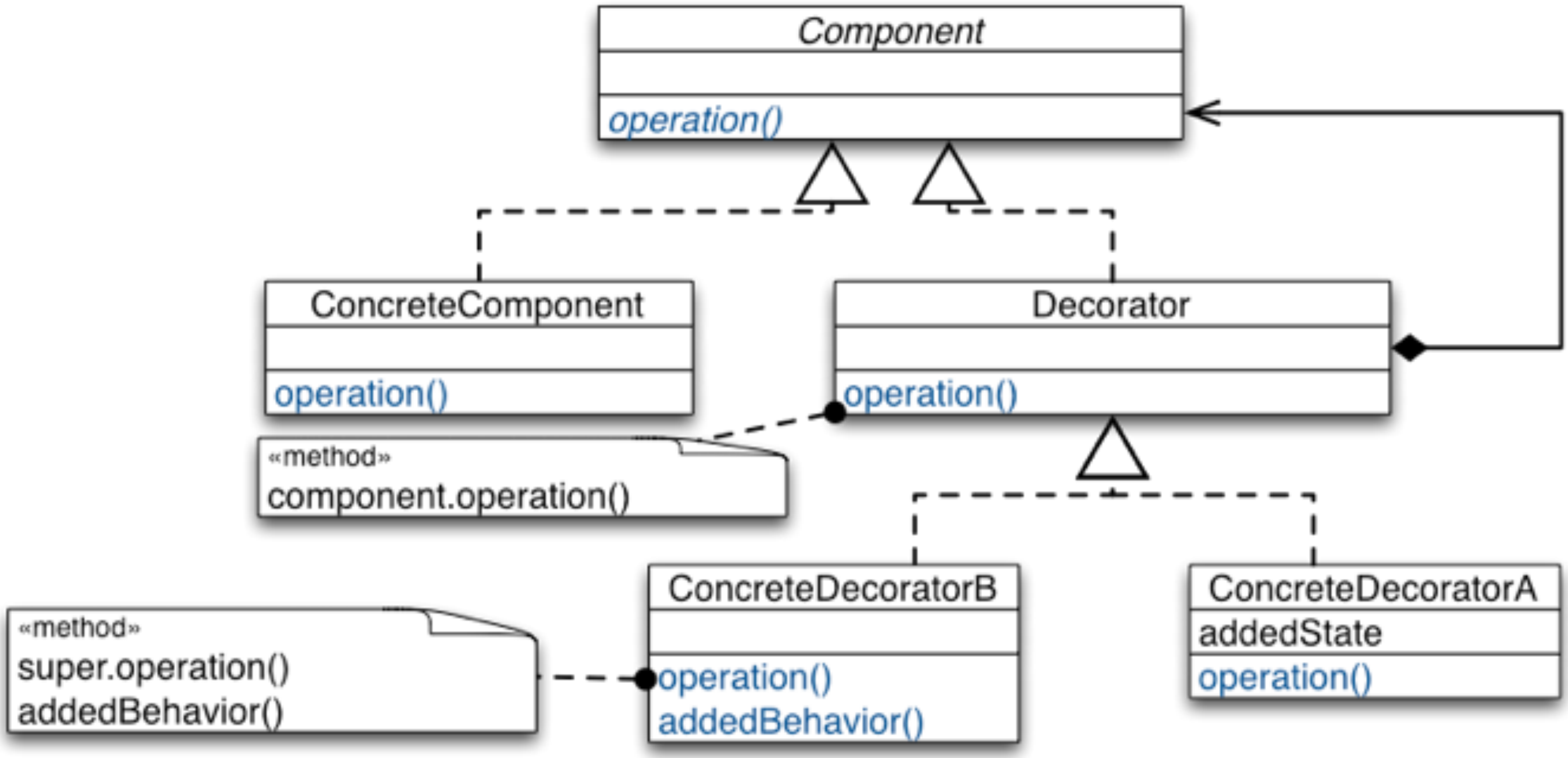
Multiple Inheritance is no Solution Either



- static responsibility mix
- naming conflicts
- hard to dispatch super calls correctly

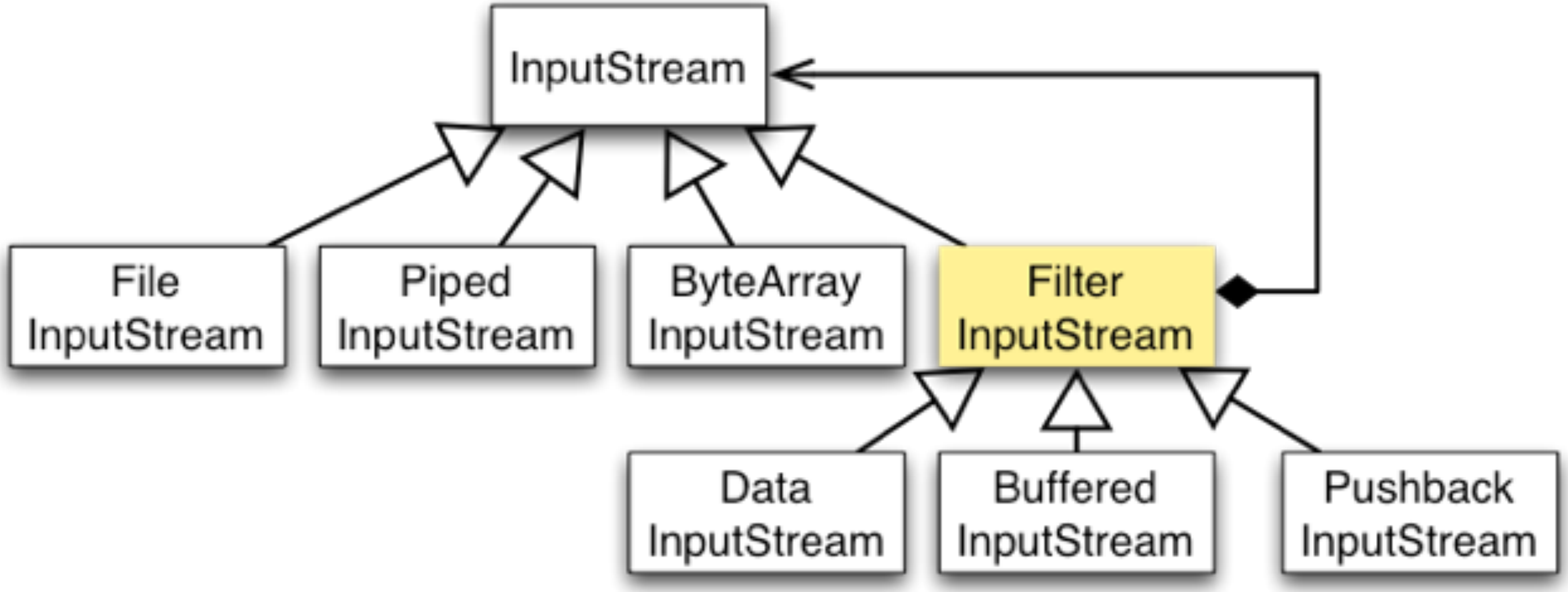
> "Multiple inheritance is good, but there is no good way to do it."
> A. Synder

Structure of the Decorator Pattern



Intent: We need to add responsibilities to existing individual objects dynamically and transparently, without affecting other objects.

Example: Decorator in java.io



- java.io abstracts various data sources and destinations, as well as processing algorithms:
  - Programs **operate on stream objects** ...
  - **independently of** ultimate data source / destination / shape of data.

Decorator Pattern: Discussion

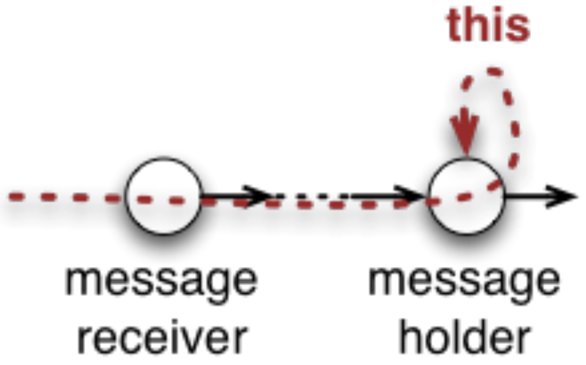
- Decorator enables more flexibility than inheritance:
- Responsibilities can be added / removed at run-time.
- Different Decorator classes for a specific Component class enable to mix and match responsibilities.
- Easy to add a responsibility twice; e.g., for a double border, attach two BorderDecorators
- Decorator avoids incoherent classes:
  - functionality can be composed from simple pieces.
  - an application does not need to pay for features it doesn't use.

Decorator: Problems

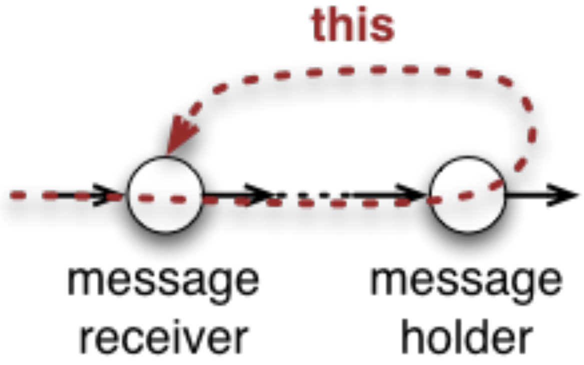
- Lots of little objects
  - A design that uses Decorator often results in systems composed of lots of little objects that all look alike.
  - Objects differ only in the way they are interconnected, not in their class or in the value of their variables.
    Imagine a class to draw a border around a component..
  - Such systems are easy to customize by those who understand them, but can be hard to learn and debug.
- Object identity
  - A decorator and its component aren't identical.
    From an object identity point of view, a decorated component is not identical to the component itself.
  - You shouldn't rely on object identity when you use decorators

Decorator: Problems

- ***No late binding***
  - ***DELEGATION VERSUS FORWARD SEMANTICS***

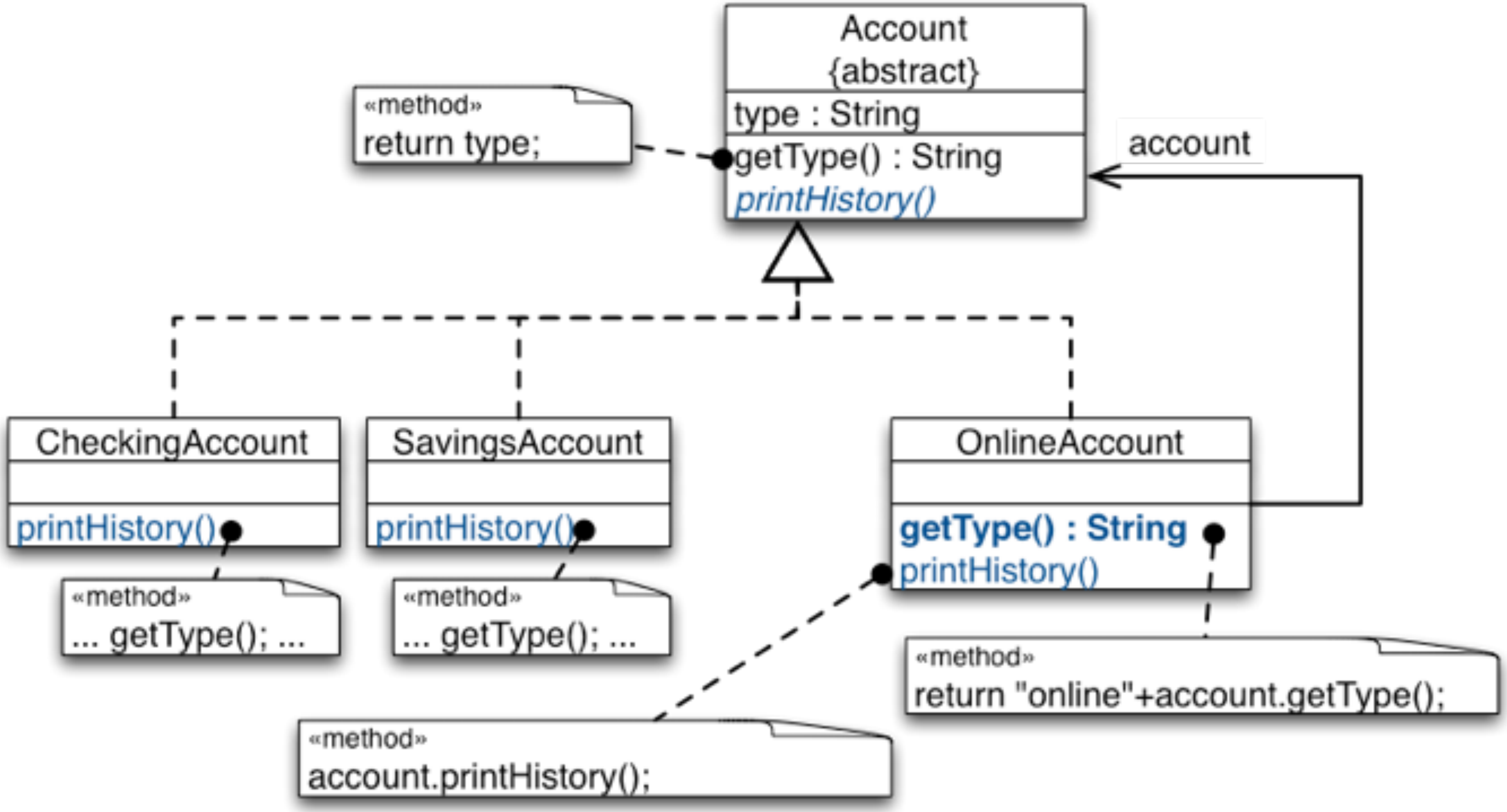


Forwarding with binding of this to method holder; "ask" an **object to do something on its own**.



Binding of this to message receiver: “ask” an object to **do something on behalf of the message receiver**.

No Late Binding: Example



Decorator: Problems

- Need to implement forwarding methods for those methods not relevant to the decorator
  - A lot of repetitive programming work
  - A maintenance problem: What if the decorated class changes
    - Adding new methods or removing methods that are irrelevant to the decorators
    - Decorator classes need to change as well
    - This is a variant of the so-called “fragile base class problem”

Frameworks & Libraries

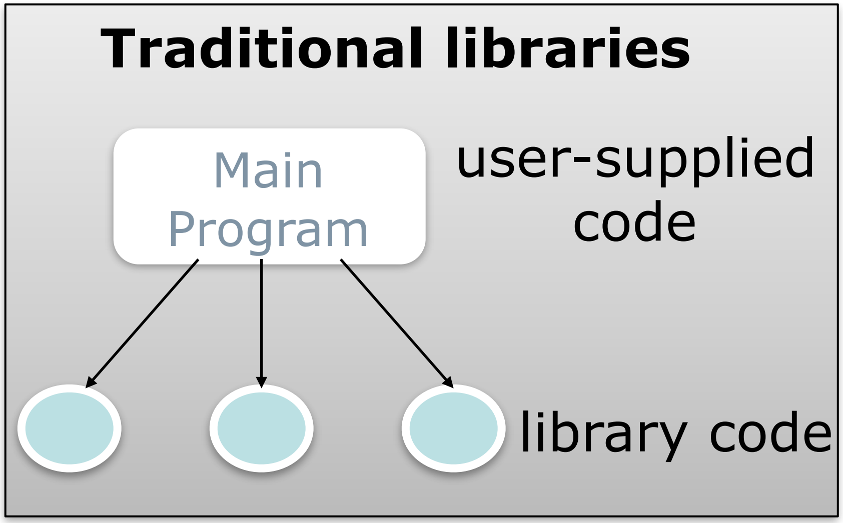
What is an (OO) Framework?

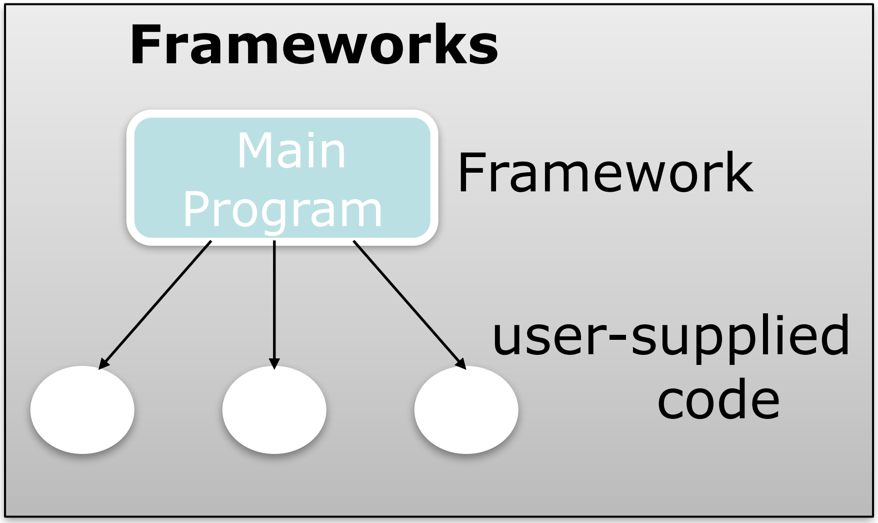
- A **set of cooperating classes** that makes up a reusable design.
- A framework provides architectural guidance by **partitioning the design into abstract classes** and defining their **responsibilities and collaborations**.
- A developer customizes the framework to a particular application by **subclassing** and **composing** instances of framework classes.
- A framework solves **problem in a particular problem** domain.

What is a library?

- A **set of reusable coherent programming abstractions** (classes, methods, functions, data structures)
- Focus on **black-box** reuse.
- Sometimes a library can be seen (and used as) a domain-specific language (DSL).

Libraries vs. Frameworks





- **Control flow** is dictated by the **framework** and is the same for all applications.
- The framework is the main program in coordinating and sequencing application activity. i.e., it manages the object lifecycle

Libraries vs. Frameworks

- **„Traditional“ difference**: Who is in charge of the control flow (Inversion of Control)
- However, this difference is **only well-defined** if one considers libraries that **can only be parameterized by first-order values**
- Libraries that accept **higher-order parameters** (such as first-class functions or objects) are quite similar to frameworks

Libraries vs. Frameworks

- **Remaining difference:** Frameworks are often white-box, glass-box or grey-box, whereas libraries are more black-box
- Frameworks can be adapted in more ways, also ways not anticipated by the framework developer
- Library developers **must anticipate** every extension point, but in turn libraries can be changed more easily without invaliding clients

- No strict discrimination between the two terms possible

Frameworks vs. Design Patterns

- **Patterns are smaller** than frameworks.
  - A framework contains many patterns (Visitor, Decorator etc.).
  - The opposite is not true.

- **Patterns are language independent**.
  - Patterns solve OO language issues (Java, C++, Smalltalk).
  - Frameworks are written in a specific programming language.

- **Patterns are more abstract than frameworks.**
  - Patterns do not solve application domain specific problems.
  - Frameworks provide support for a particular application domain. Frameworks provide reusable code

Frameworks vs. Design Patterns

- Frameworks describe:
  - the **interface of each object** and the **flow of control** between them.
  - how the **responsibilities** are mapped onto its objects
- Again, in other words:
  - A Framework provides **architectural guidance**
  - by **partitioning** the **design into abstract classes** and
  - defining their responsibilities and collaborations.

The high level design is the main intellectual content of software, and frameworks are a way to reuse it!

A Framework is not...

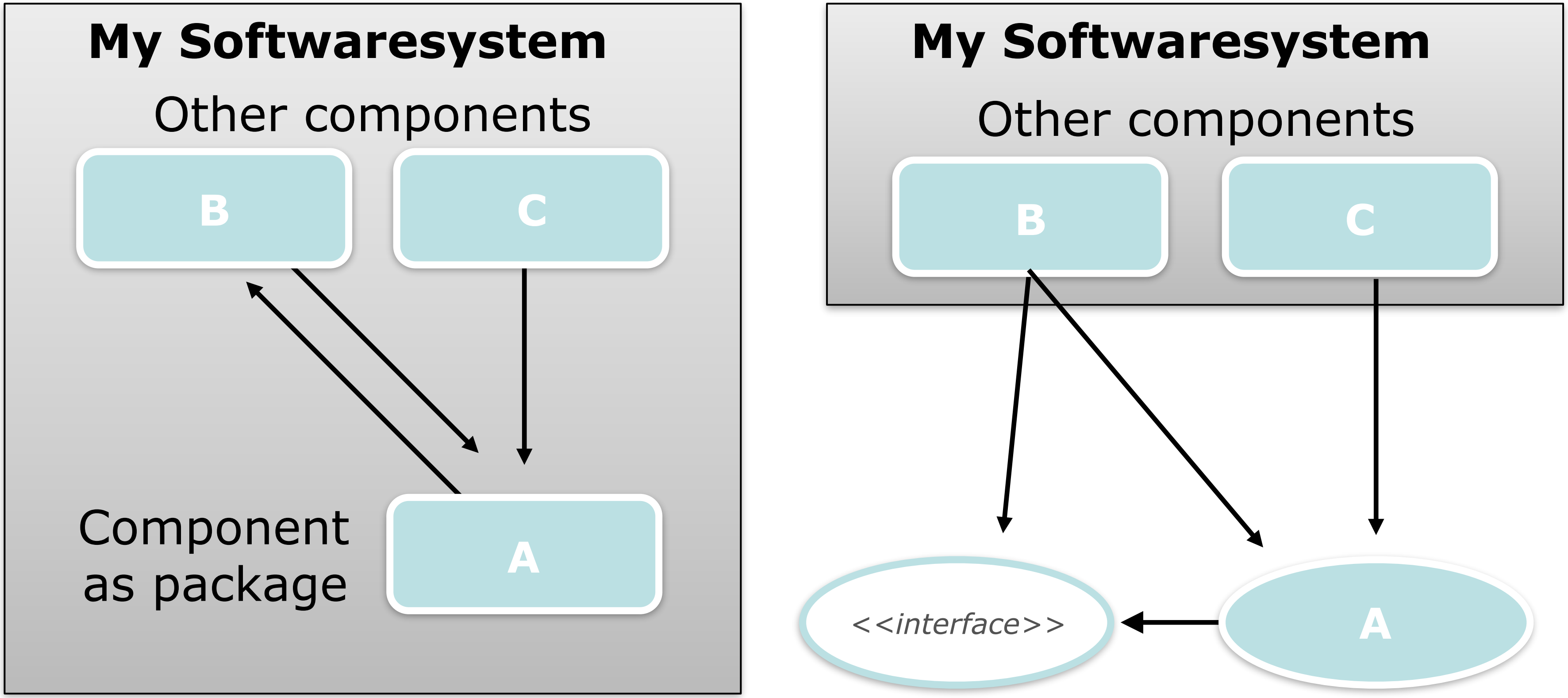
- **... a design pattern.**
  - patterns describe ideas and perspectives;
  - frameworks are implemented software.
- **… an application.**
  - frameworks do not necessarily provide a default behavior, hence they are not executable programs;
  - They can be perceived as a partial design but they do not describe every aspect of an application.
- **… a class library.**
  - applications that use classes from a library invoke predefined methods, whereas frameworks invoke predefined methods supplied by the user.

Library Design Principles

Libraries

- The **oldest, most common,** and **most successful** way of reusing code
- Languages are designed to support libraries
- Works together with static typing, import/export mechanisms, separate compilation, …
- Composability with other libraries
- Support by type and module system
- Information hiding, substitutability, …
- But libraries need a good design to be useful!

(Sub-) Package vs. Library



Component as library

Basic Library Design Principles

- Libraries should be as **context-independent** as possible
  - Every context dependency limits reusability
  - Context dependencies (e.g. on other libraries) should be expressed via interfaces
- Libraries should have a **clean, well-defined scope**
- Library should have a **well-defined interface**
  - To make black-box usage possible
  - Interface should be cleanly separated from implementation details
  - E.g. via separate packages

Basic Library Design Principles (2)

- Library designer has to think about **variability points** of the library
- Different form of variability
  - Parameterization by values
  - Parameterization by types
  - Parameterization by functions/closures or objects

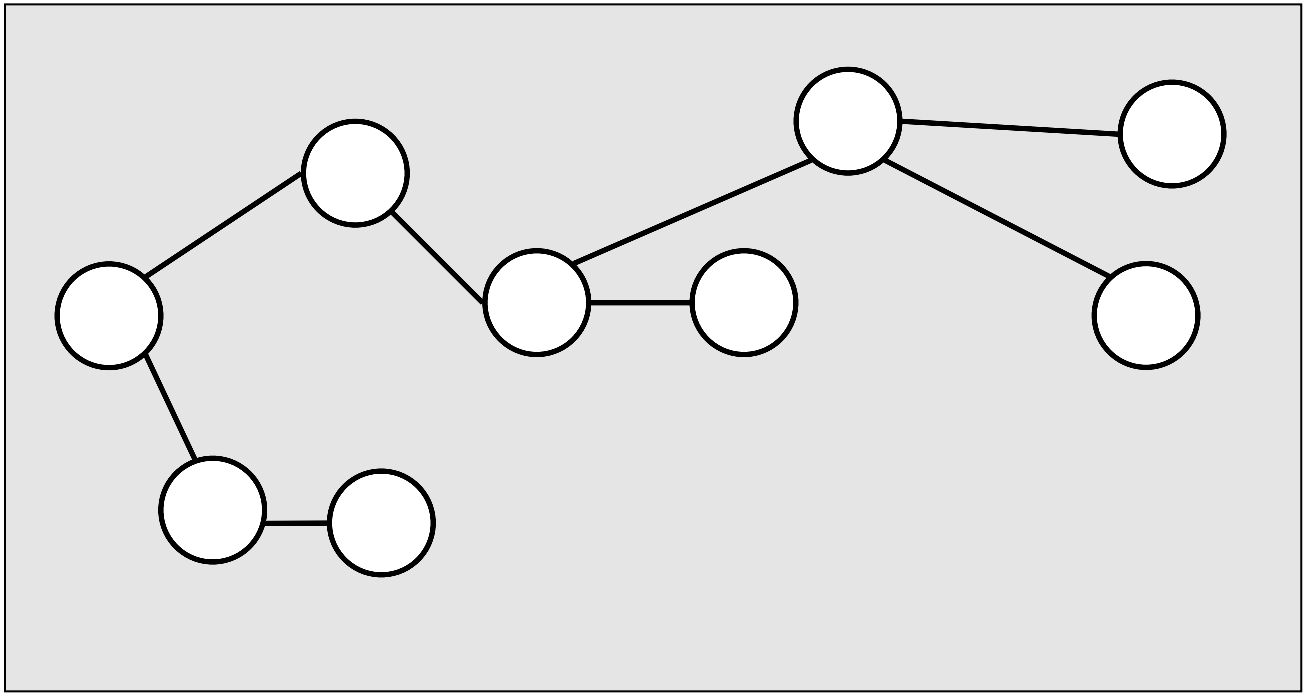
Customizing Frameworks

Customization Points

- So far, we talked about frameworks being semi-complete applications that developers need to extend to make them work as application.
- Thus, the question arises **how** one can customize a framework.

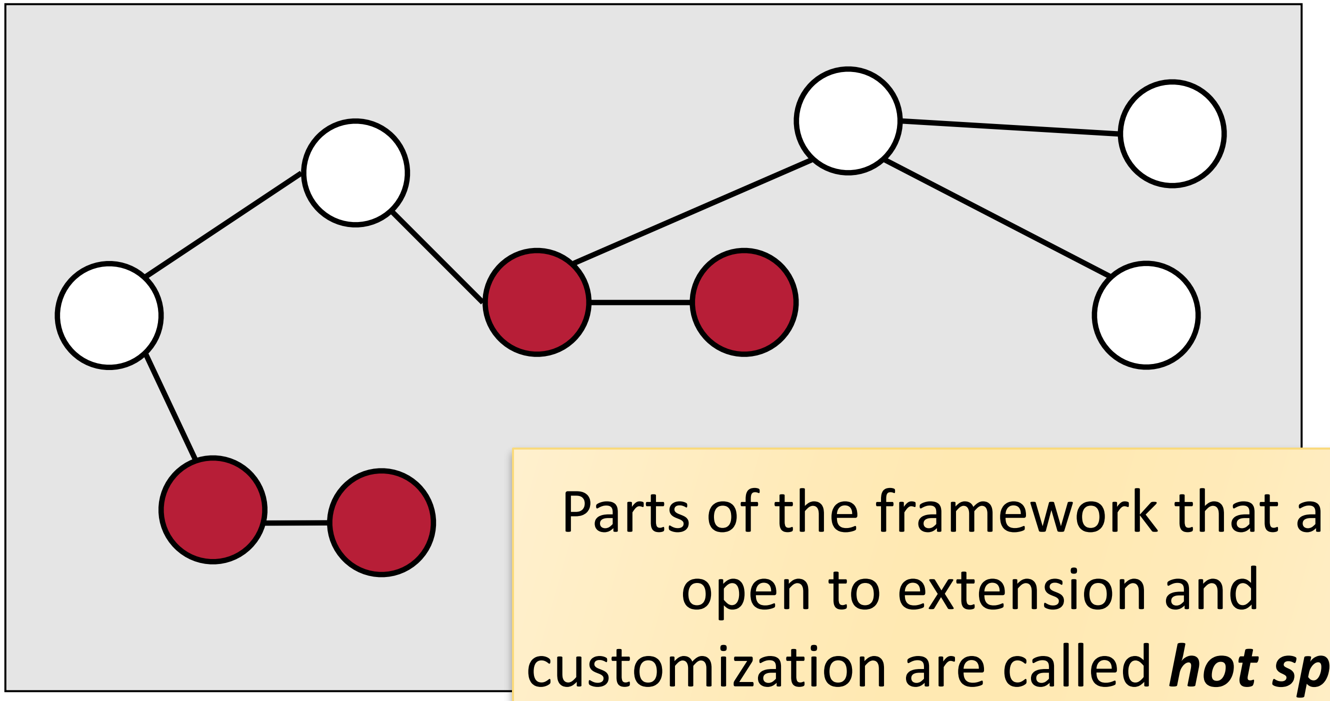
Simplified Representation of a Framework

- Nodes represent classes, links between nodes represent associations between classes used for collaboration between classes.



Framework HotSpots

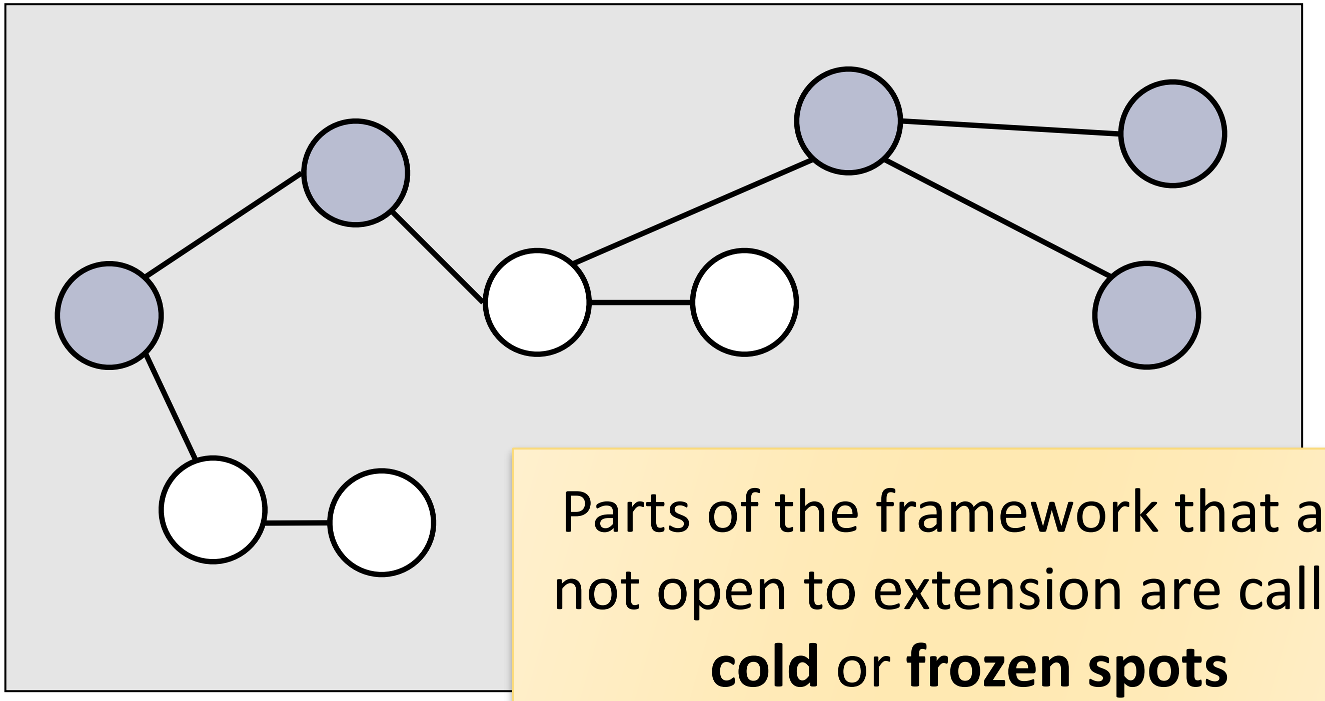
- Since frameworks are incomplete there must be some points in the design allowing a developer to extend the framework. This extension points are called **hot spots**.



Parts of the framework that are open to extension and customization are called ***hot spots***

Simplified Representation of a Framework

- Nodes represent classes, links between nodes represent associations between classes used for collaboration between classes.



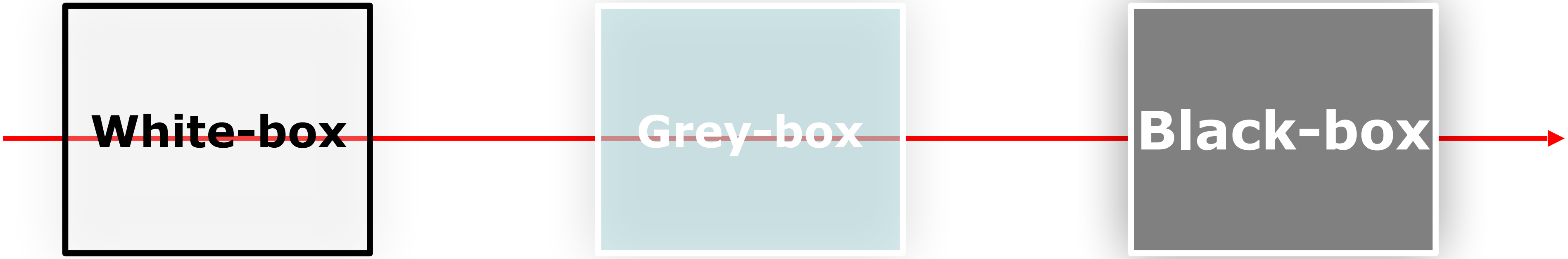
Parts of the framework that are not open to extension are called **cold** or **frozen spots**

How to extend a framework concretely?

- You learned that there are some parts that can be extended and some can’t. But how do you do that actually?
- The short answer: It depends.
- Before explaining that, we need to introduce another classification for frameworks.

Framework Classification by Extensibility

- We distinguish three different kinds of extensibility



White-Box Extensibility

- White-box frameworks can be extended by modifying or adding to the original source code.
- Least restrictive & Most flexible
- We can distinguish between
  - **Open-Box Extensibility**
    - Changes are immediately performed on the original sourcecode
  - **Glass-Box Extensibility**
    - Original source code is known but untouched
    - Predefined hooks are necessary

Black-Box Extensibility

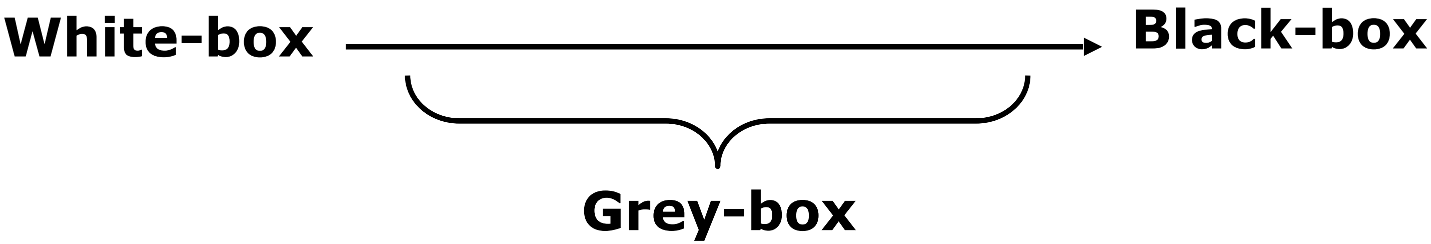
- No internal details about the original system are known
- Only interfaces are known
- Extension points have to be well-preplanned and documented
- Most-restrictive and limited approach
- Easier to learn

Black-box

Grey-Box Extensibility

Grey-box

- Grey-box extensibility uses both **parametrization** and **refinement**
- Frameworks typically evolve from white-box to black-box frameworks over a number of iterations:

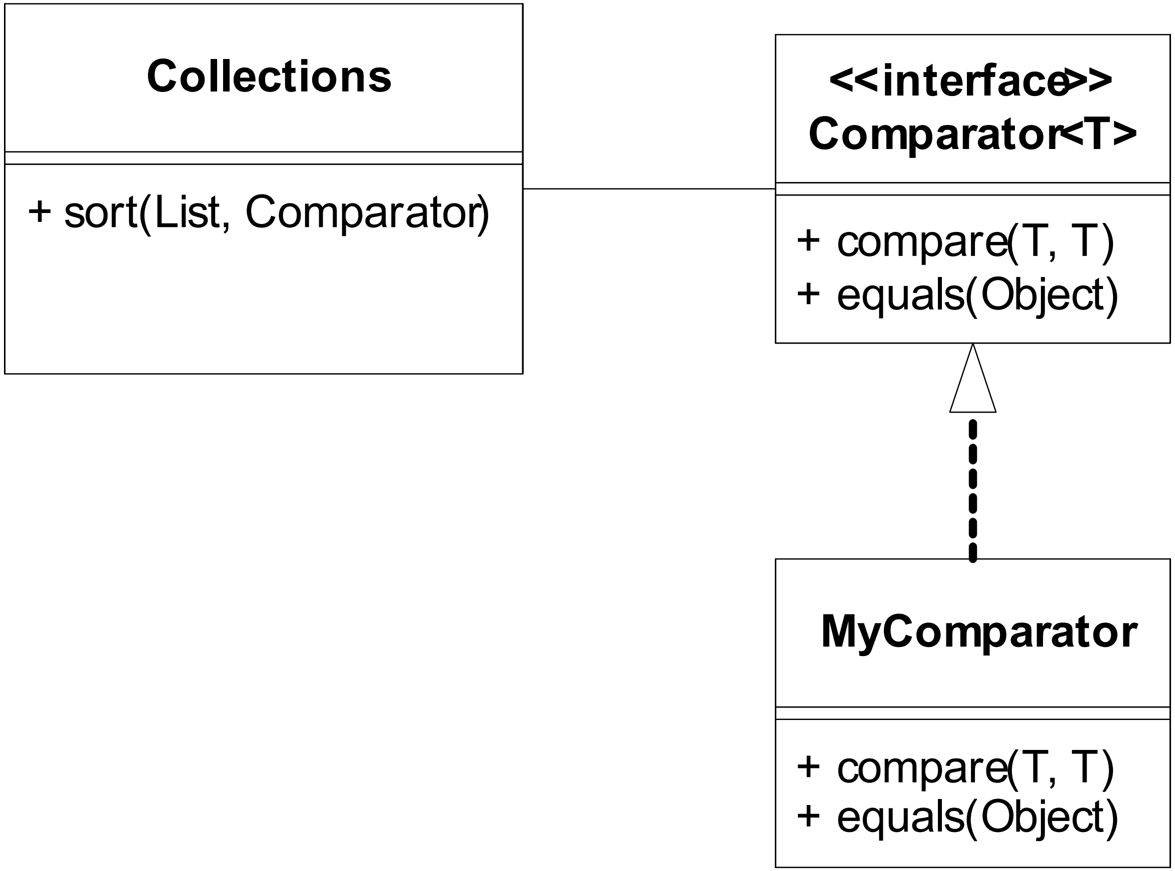


- However, it will be hard to find pure black-box frameworks. Typically, they contain a few white-box elements too.

Inversion of Control, or ...

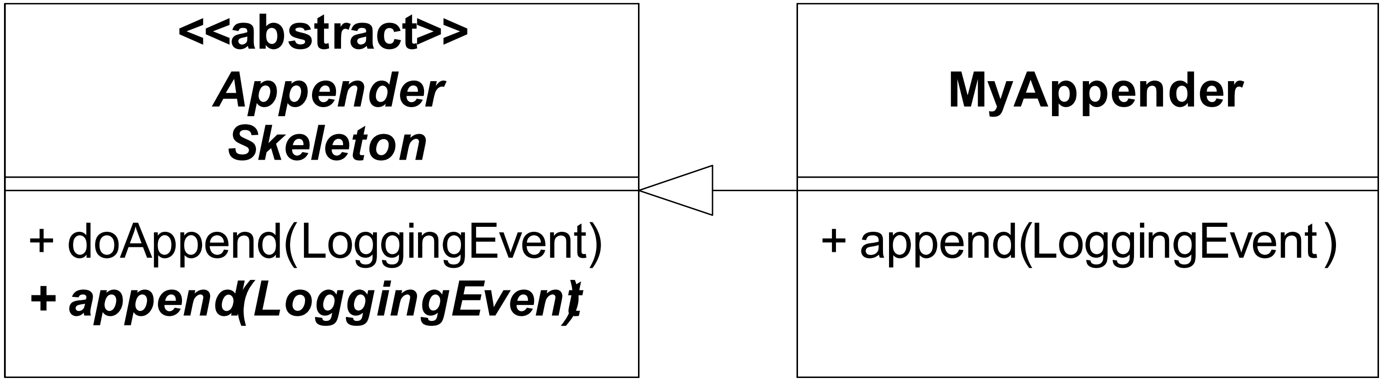
Dependency Inversion via Strategy Pattern

- Dependency Inversion is the most essential principle applied on frameworks.



Dep. Inversion via Hooks and Template Method

- Can also be achieved using the template method pattern



- Which is the template method? Which the hook method?

Benefits of using Frameworks

- **Extensibility**
  - Framework enhances extensibility by providing explicit hook methods.
  - Hook methods systematically decouple the stable interfaces and behaviors of an application domain from a particular context.

- **Inversion of control**
  - IOC leads to reduced coupling between components
  - Increases testability

Weaknesses when using Frameworks

- **Learning curve**
  - it often takes several months become highly productive with a complex framework
- **Integratability**
  - Application development will be increasingly based on integration of multiple frameworks together with class libraries, legacy systems and existing components in one application
- **Maintainability**
  - As frameworks evolve, the applications that use them must evolve with them …
- **Efficiency**
  - In Terms of memory usage, system performance...

Erinnerung

Nächste Woche: Gastvortrag zu TDD & CI