Software Engineering

Detailed Design

Peter Müller

Chair of Programming Methodology

The slides in this section are partly based on the lecture
“Software Engineering I” by Prof. Bernd Brügge, TU München

Spring Semester 10

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

5. Detailed Design

Bloopers

- Speed
  - Harry's partner shoots Harry in the right leg
  - Throughout the movie, Harry limps on the left leg

- Star Wars
  - At the end of Episode V, Han Solo is frozen into carbonite
  - When being frozen, Han Solo is wearing a dark jacket
  - When thawed, he is wearing a white shirt

Why do Movies Contain Bugs?

Scenes shot out of sequence

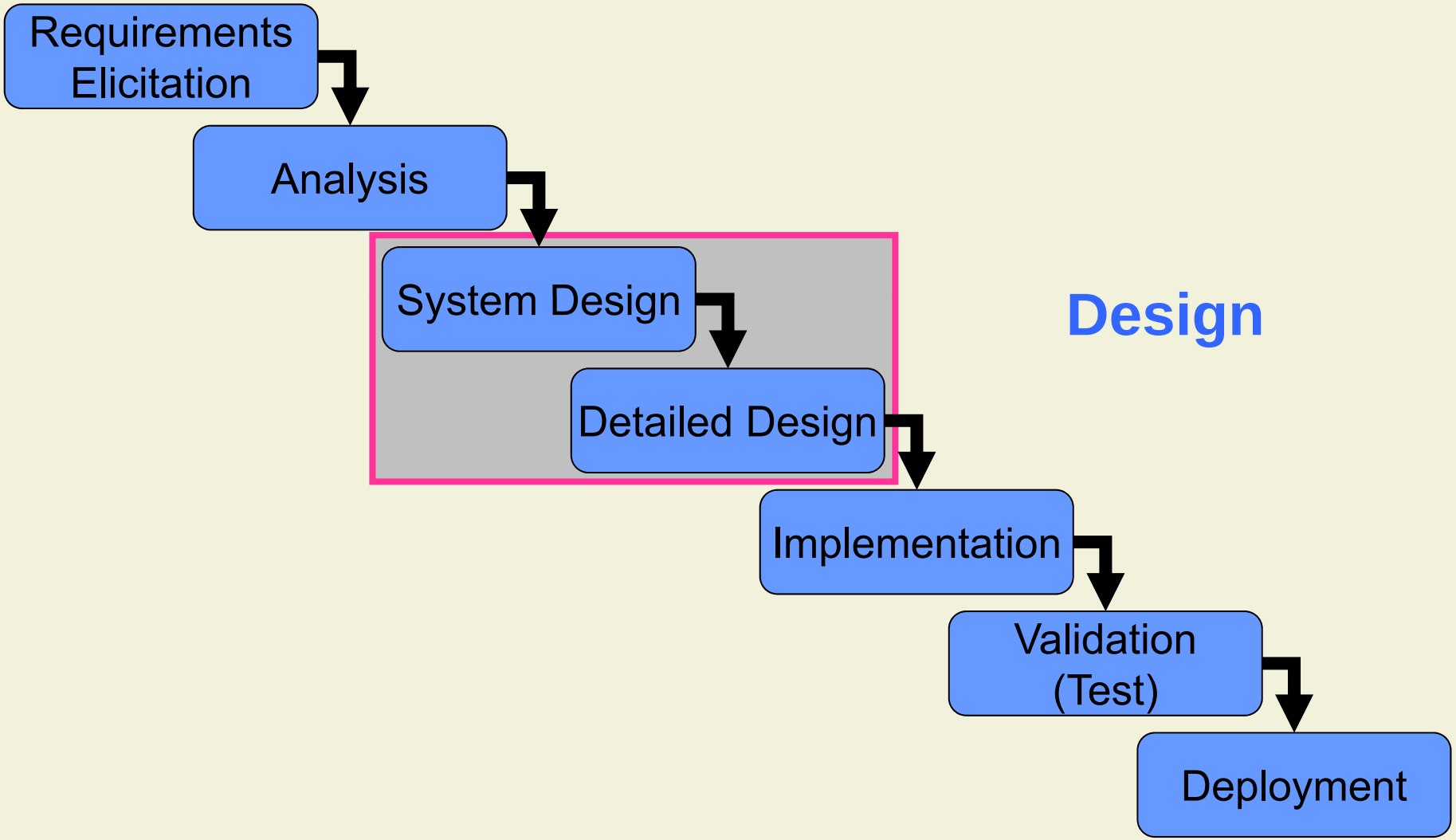
High pressure of release date during editing

Details (props, costumes) changed during production

Cooperation of many different people

Some scenes re-shot out of schedule

Waterfall Model of Project Life Cycle

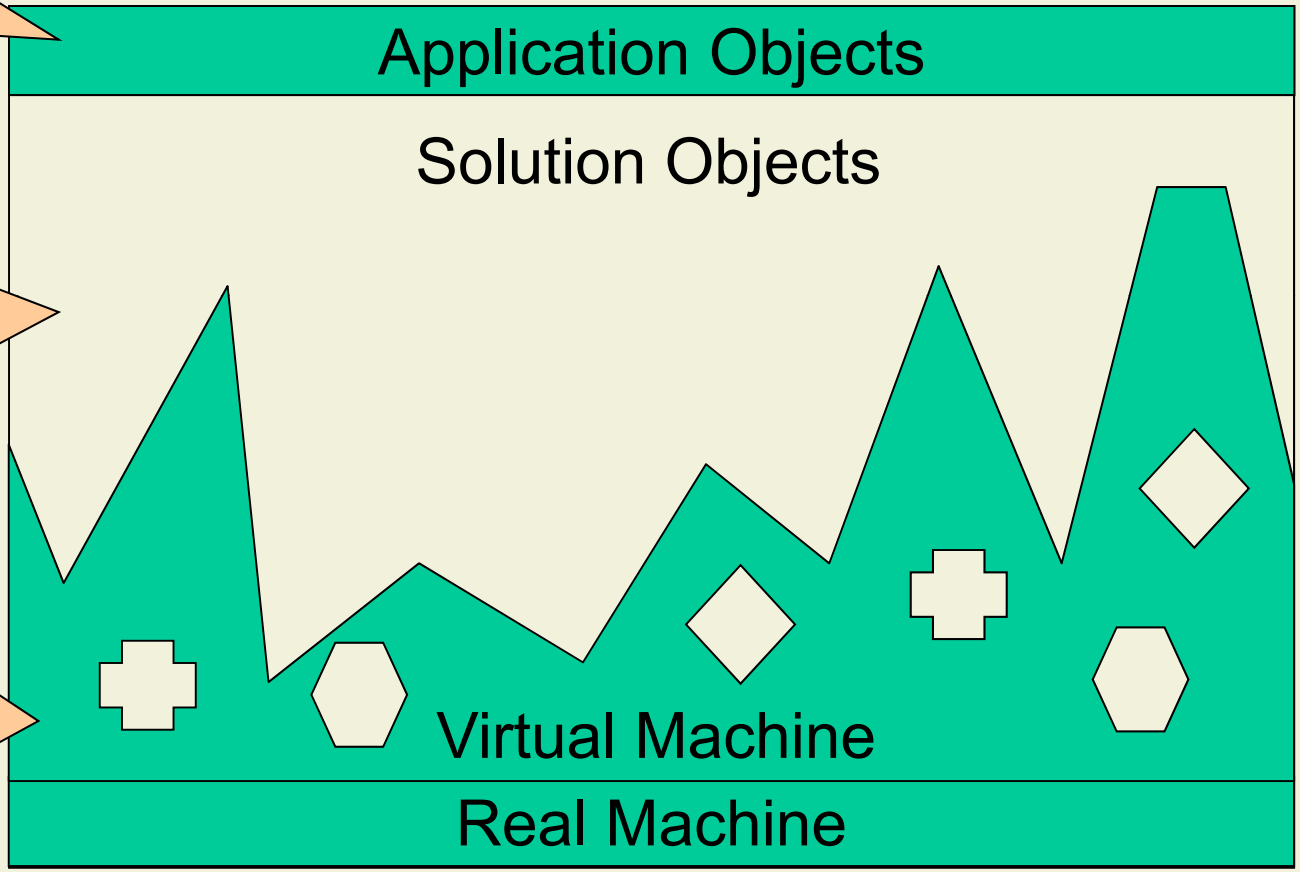


Detailed Design: Closing the Gap

Analysis defines application objects

Detailed design identifies new objects and adjusts components

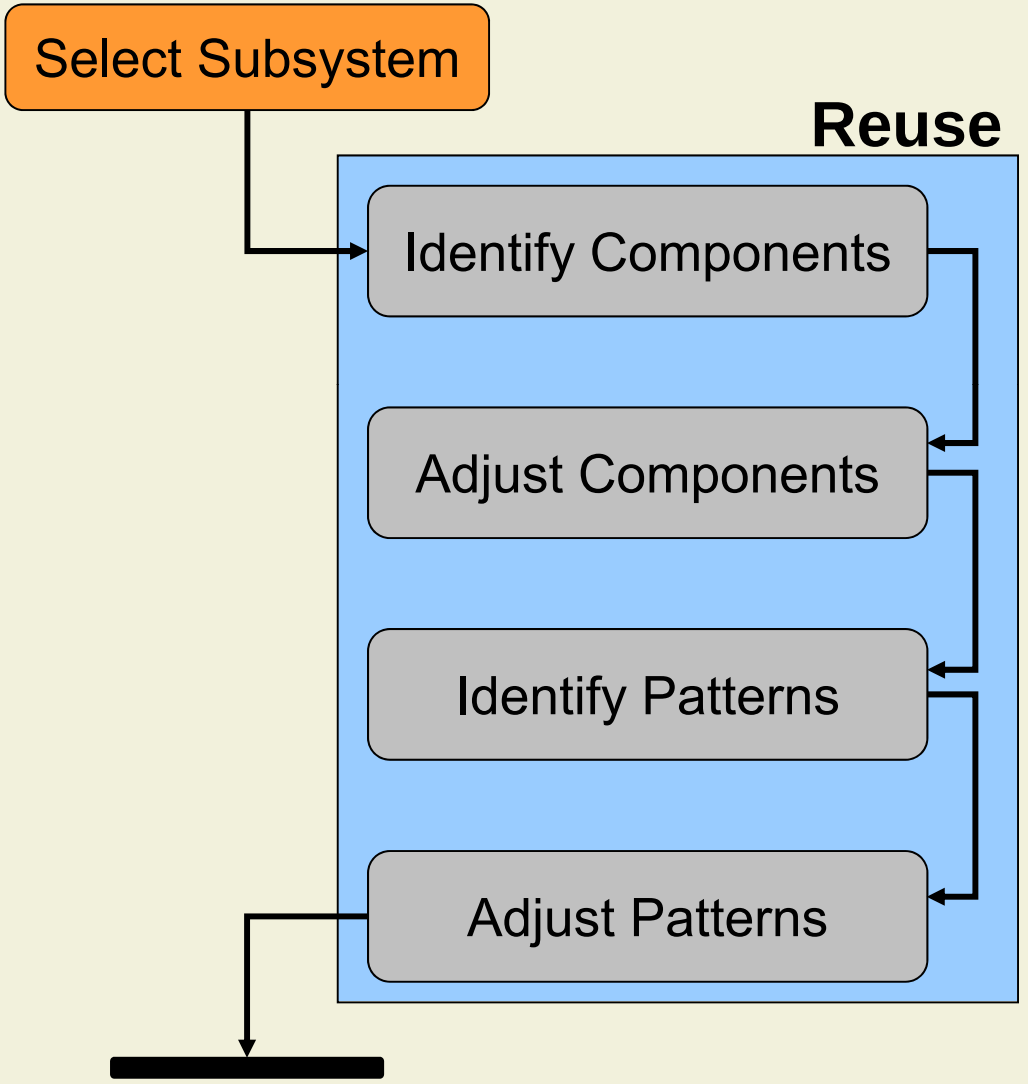
System design selects off-the-shelf components and frameworks



Detailed Design

- **Adding details** to the requirements analysis and system design, and making **implementation decisions**
- **Choosing** among different ways to implement the analysis model and system design
  - Goals: minimize execution time, memory, and other measures of cost
- Providing the **basis for implementation**

Detailed Design Activities



Software Systems Contain Similar Problems

Classes implemented independently

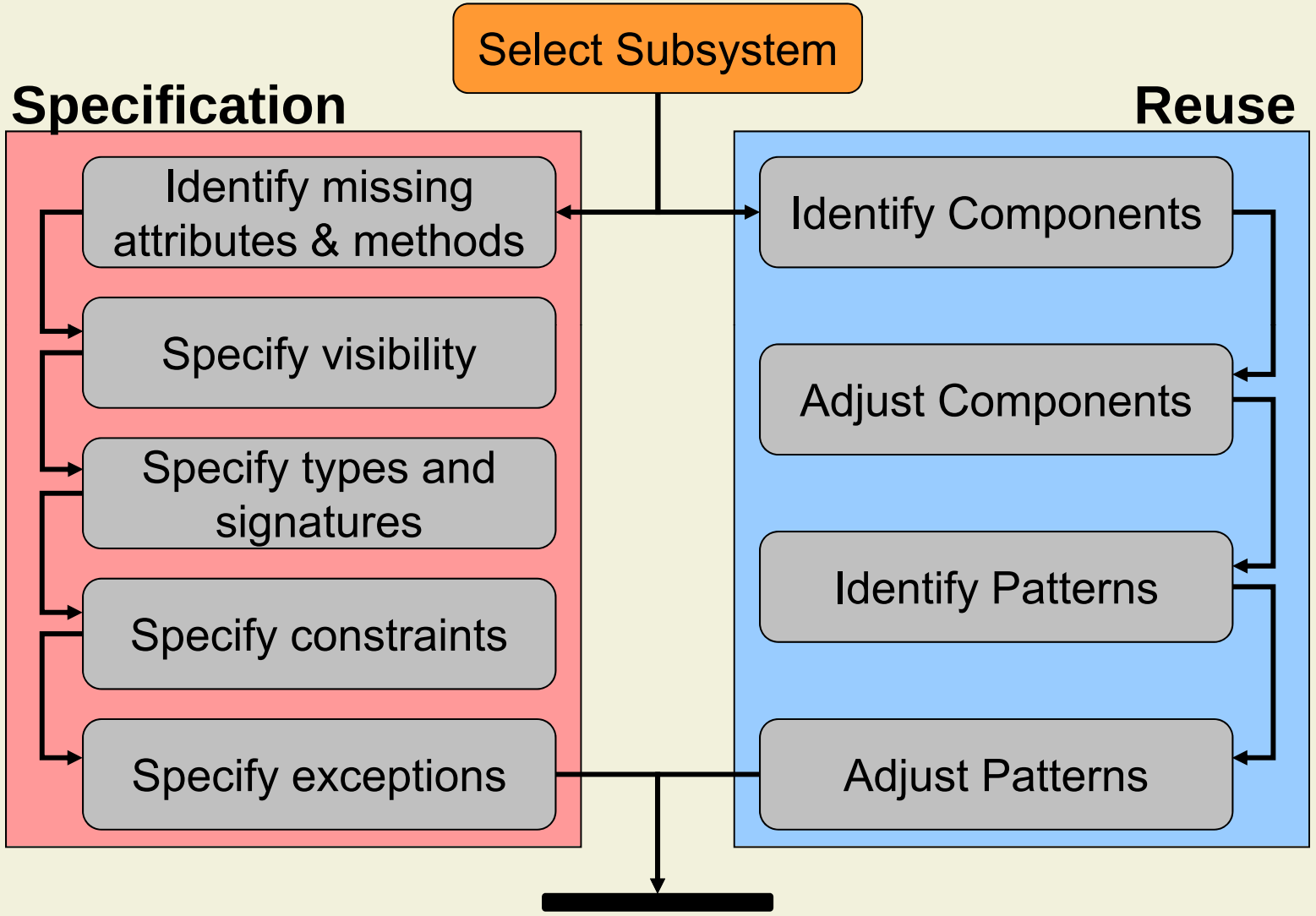
High pressure of release date

Details (interfaces, contracts) changed during development

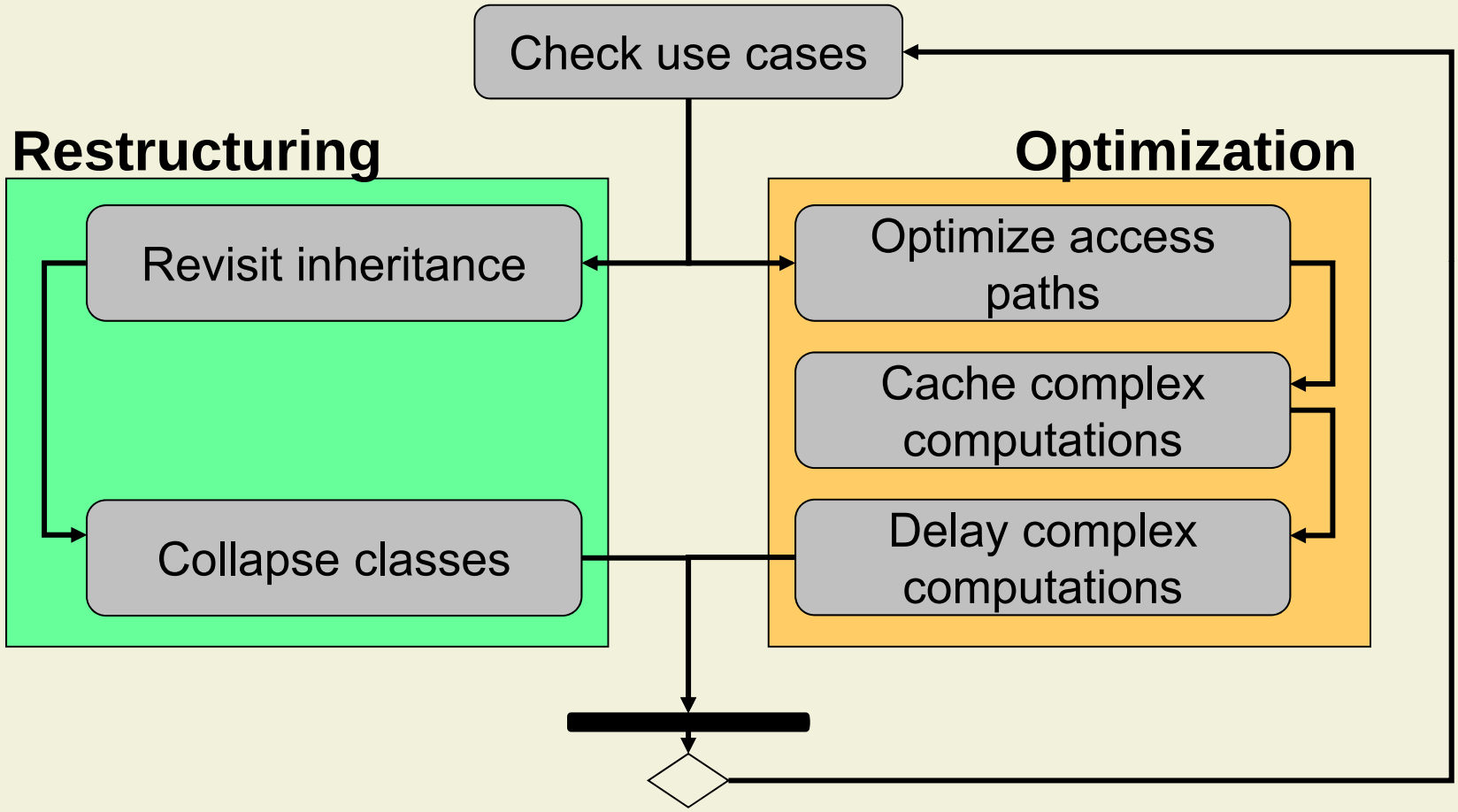
Cooperation of many different people

Some classes re-designed out of schedule

Detailed Design Activities (cont'd)



Detailed Design Activities (cont'd)



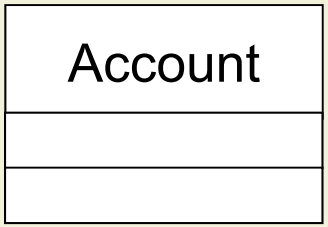
5. Detailed Design

Implementation of Application Domain Objects

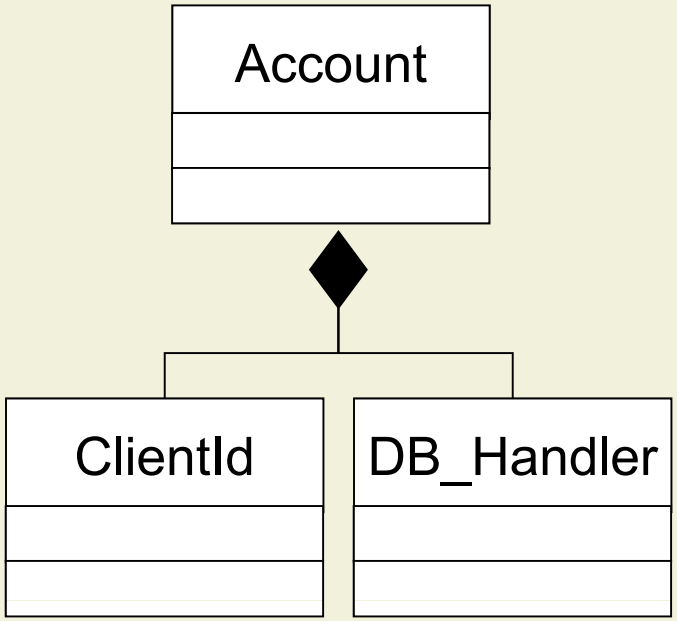
- New classes are often needed during detailed design
- The implementation of algorithms may necessitate objects to hold values (e.g., arrays)
- New low-level operations may be needed during the decomposition of high-level operations

Application vs. Solution Objects: Example

Requirements Analysis
(Language of application domain)



Detailed Design
(Language of solution domain)



Application vs. Solution Objects

- Application objects
  - Also called domain objects
  - Represent relevant **concepts of the domain**
  - Are identified by application domain specialists and by end users
- Solution objects
  - Represent concepts that have **no counterpart in the application domain**
  - Are identified by developers
  - Examples: persistent data stores, user interface objects, middleware

Finding Solution Objects

"*Many objects in a design come from the analysis model. But object-oriented designs often end up with classes that have no counterparts in the real world. […] Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's. The abstractions that emerge during design are key to making a design flexible.*"

[Gamma et al., 1995]

- There is a need for **reusable** and **flexible** designs
- **Design knowledge** complements application domain knowledge and solution domain knowledge

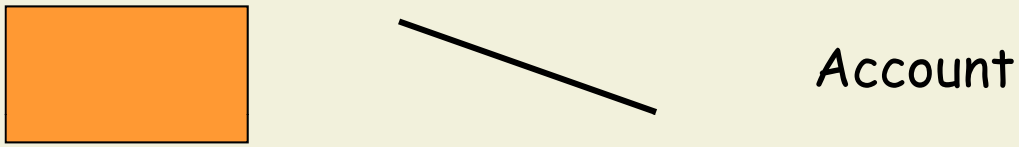
Design Patterns

"*Design patterns help you identify less-obvious abstractions and the objects that can capture them. For example, objects that represent a process or algorithm don't occur in nature, yet they are a crucial part of flexible design. […] These objects are seldom found during analysis or even the early stages of design; they're discovered later in the course of making a design more flexible and reusable.*"

[Gamma et al., 1995]

Composite Pattern: Motivation

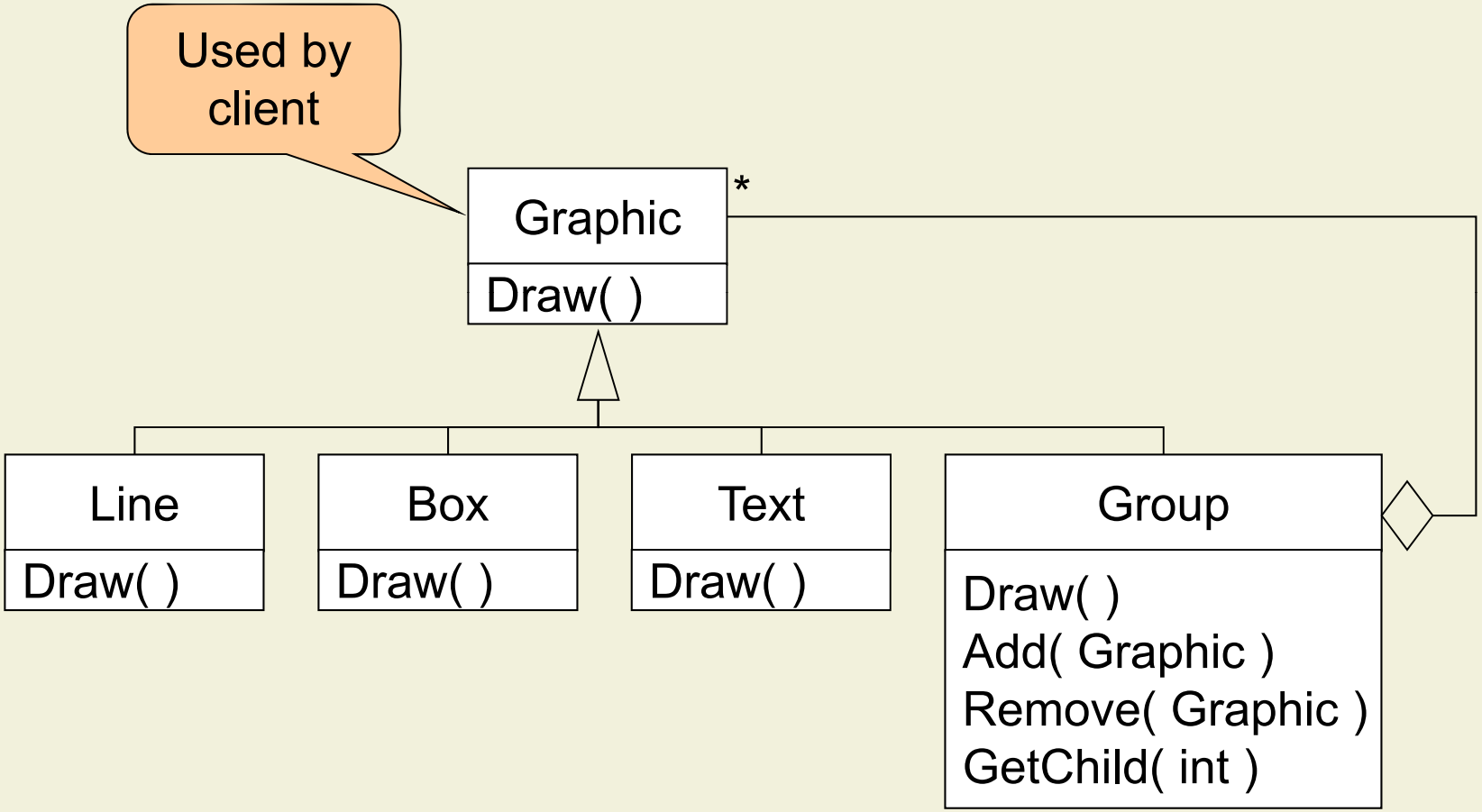
- A program manipulates
  - Individual units (e.g., graphical objects)



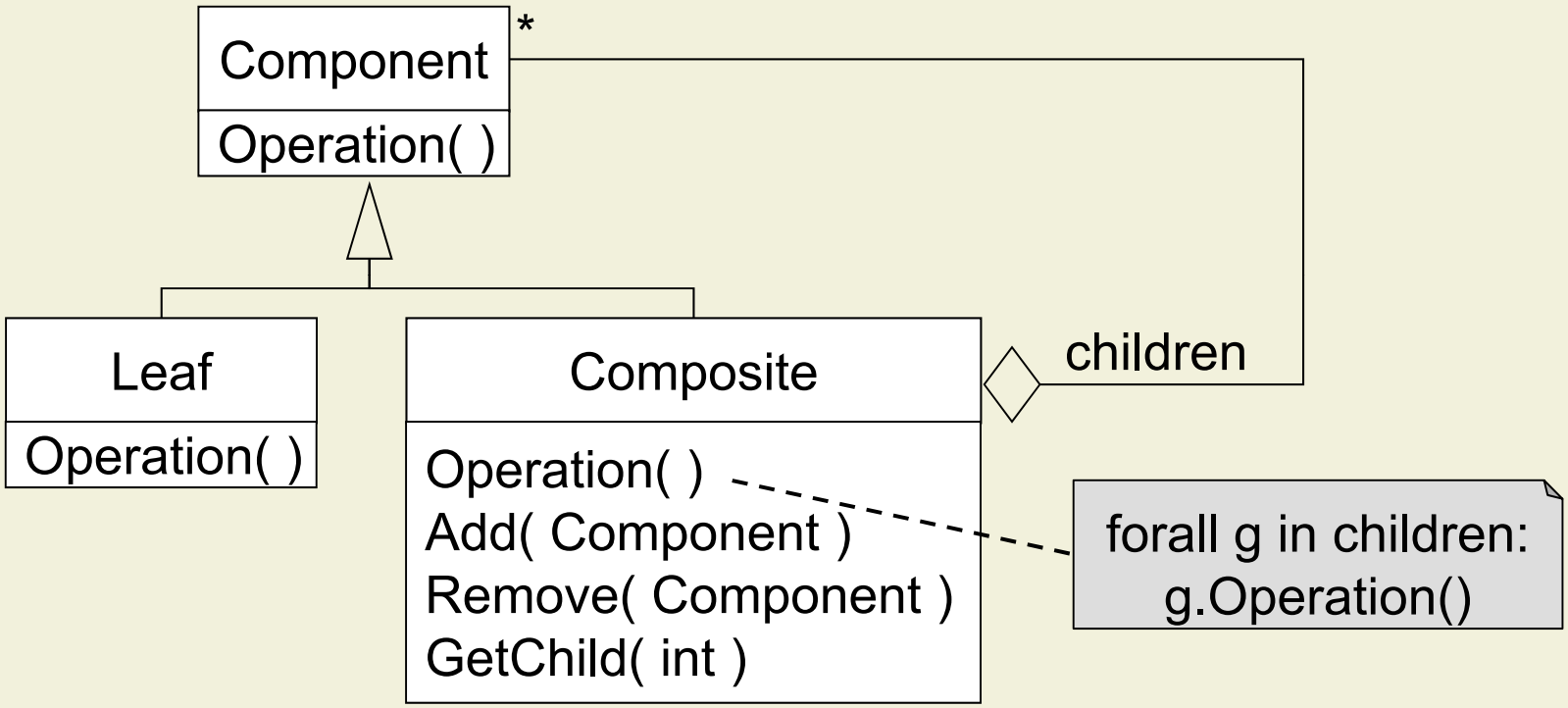
  - Groups of units

- Wanted: a design that allows algorithms to deal with single units and groups in a uniform way

Composite Pattern: Example

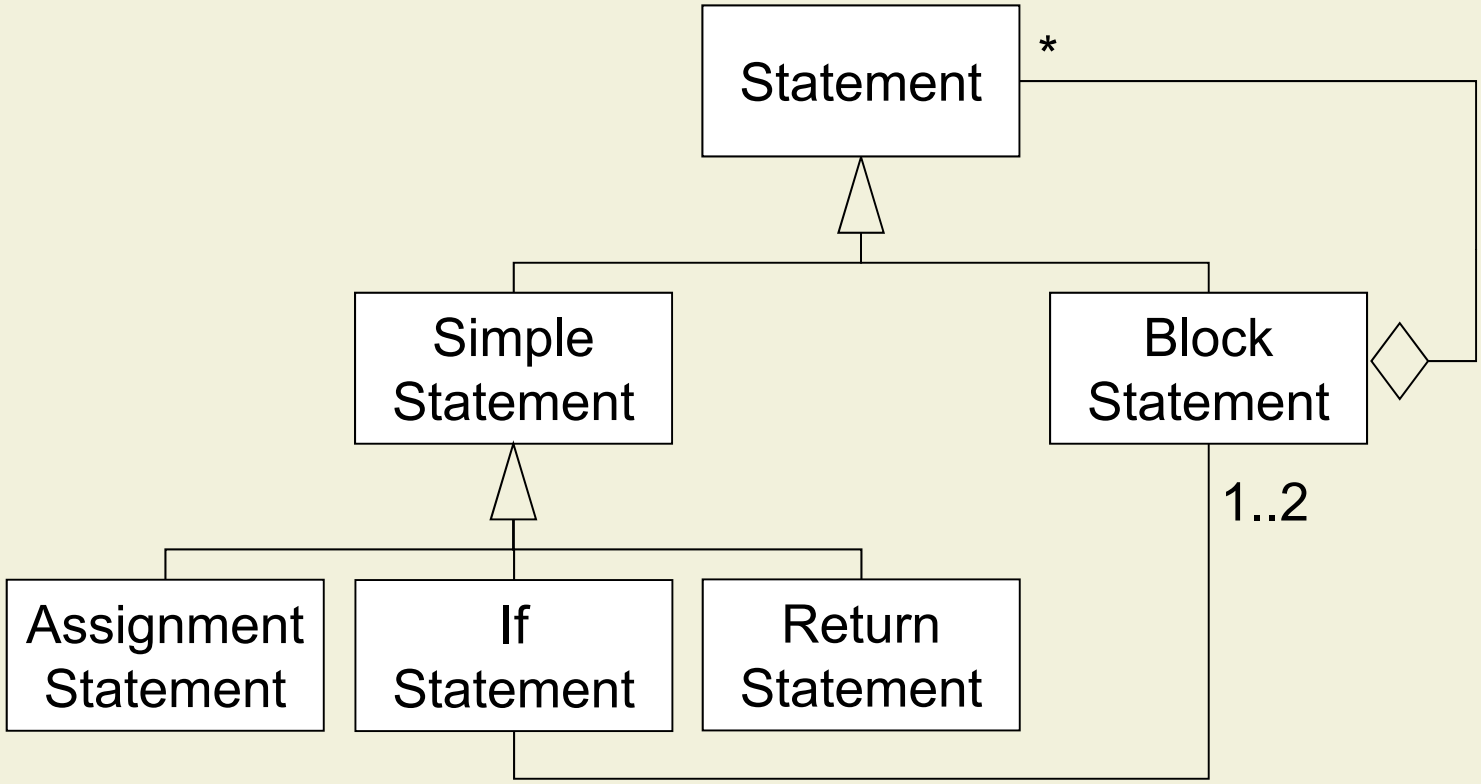


Composite Pattern: Structure



- Allows hierarchical grouping of components

Composite Pattern: Statement Syntax



Composite Pattern: Properties

- Defines class hierarchies consisting of **primitive** objects and **composite** objects
  - Objects can be composed hierarchically
  - Composite objects can be used like primitive objects
- Makes client simple
- Makes it easier to add new kinds of components
- Can make the design overly general
  - Difficult to restrict composites
  - Example: no **return** statement in a block

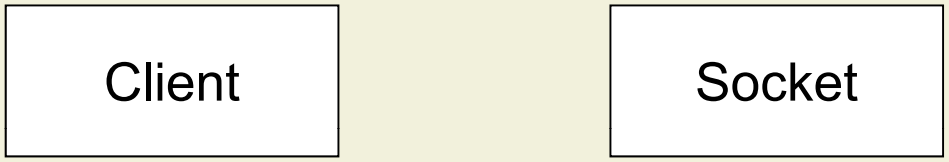
Floral Patterns

Composite Pattern: Implementation Issues

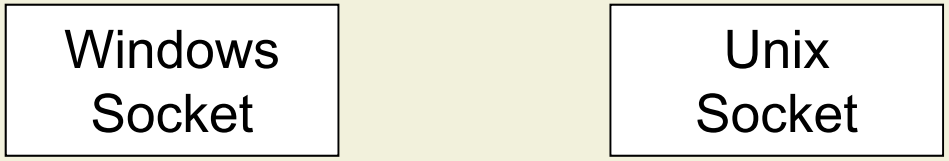
- Explicit parent references
  - Simplifies traversal and deletion of components
- Sharing components
  - Reduces storage requirements
- Child ordering
  - Might be required by the design (e.g., Block Statement)
- Caching to improve performance
  - Improves performance (e.g., bounding box for Group)
- Data structure for storing components
  - Affects performance (lists, trees, arrays, hash tables)

Abstract Factory Pattern: Motivation

- A client class wants to create sockets for network communication

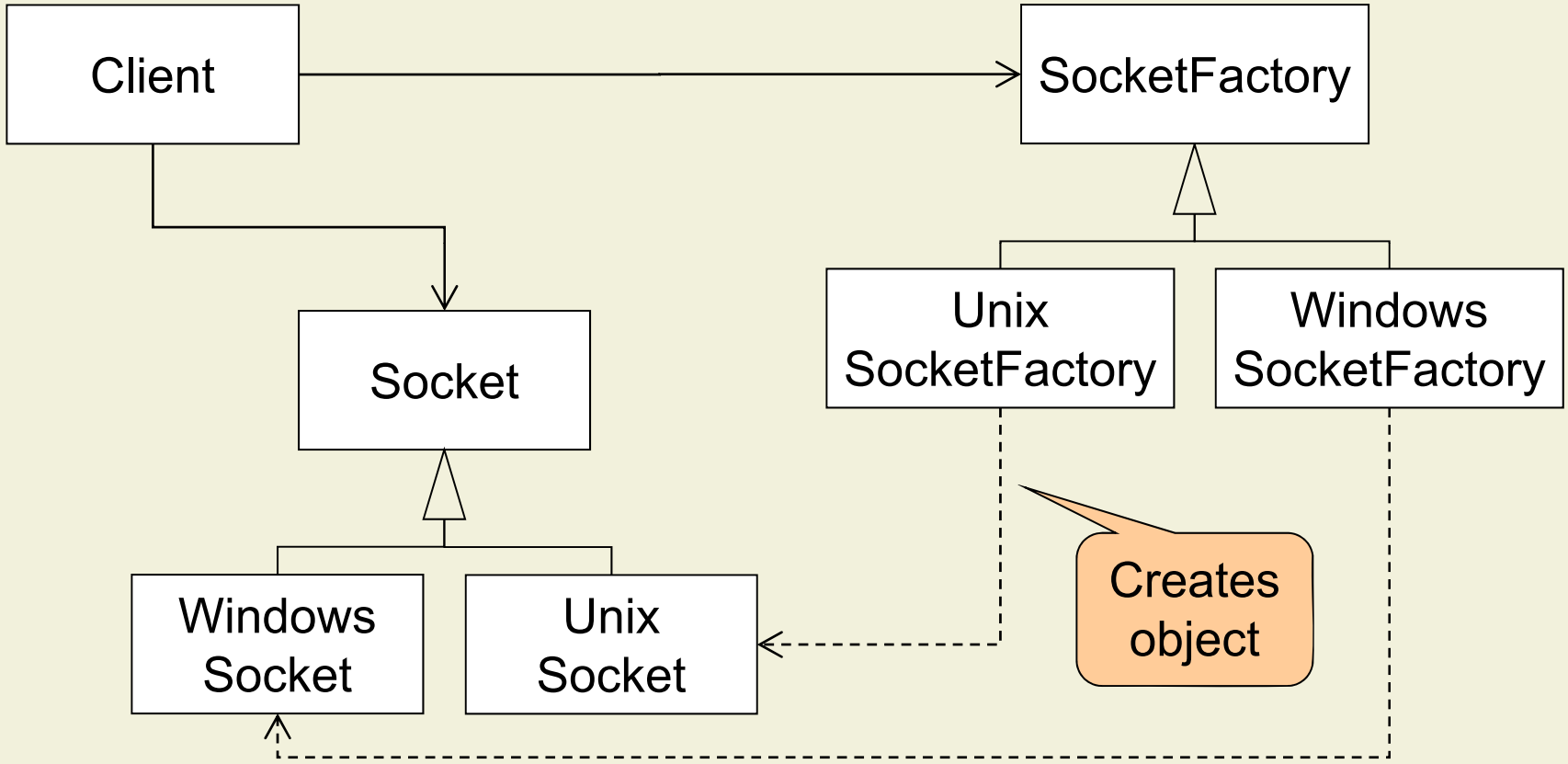


- The concrete implementation of the socket depends on the operating system

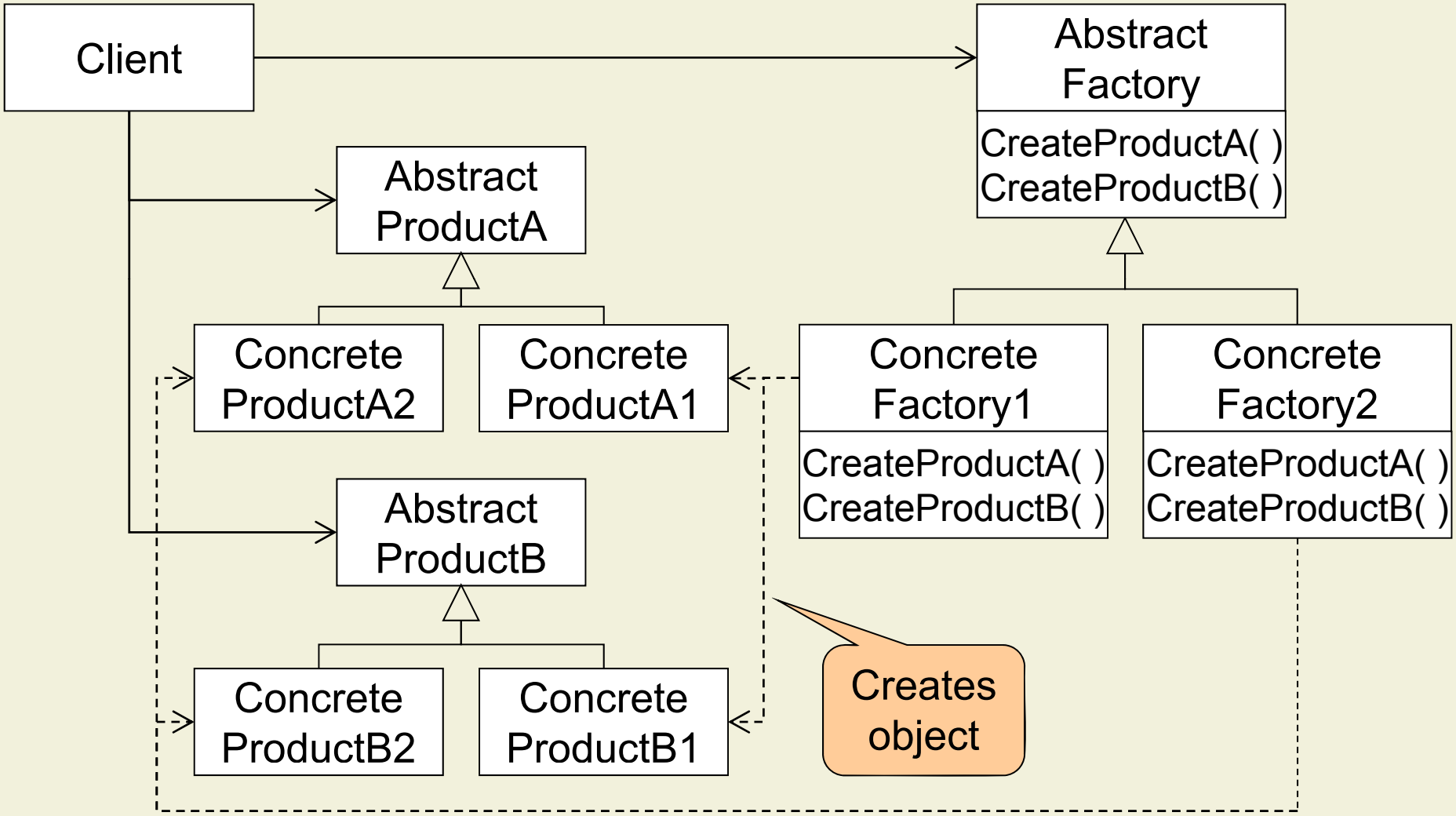


- The client class should be platform-independent

Abstract Factory Pattern: Example



Abstract Factory Pattern: Structure

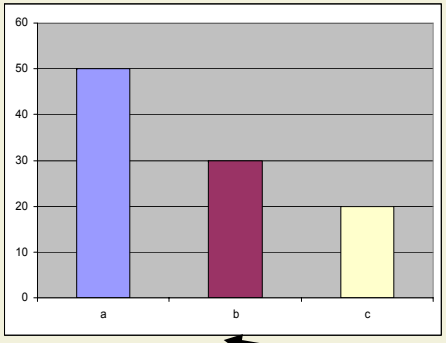


Abstract Factory Pattern: Properties

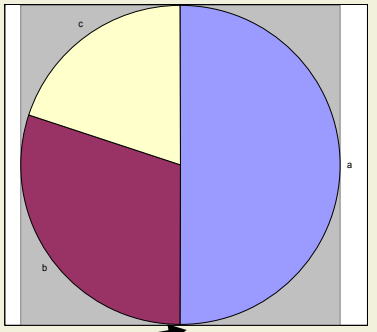
- Isolates concrete classes
  - Helps control what classes are instantiated
  - Isolates clients from implementation classes (clients manipulate objects through interfaces)
- Makes exchanging product families easy
  - Class of concrete factory appears only once in program
- Supporting new kinds of products is difficult
  - Affects interface of abstract factory and all concrete factories

Observer Pattern: Motivation

- Maintaining **consistency** between **loosely coupled** objects
- Many dependent objects have to be informed when one object changes its state



	a	b	c
X	60	30	10
Y	50	30	20
Z	80	10	10

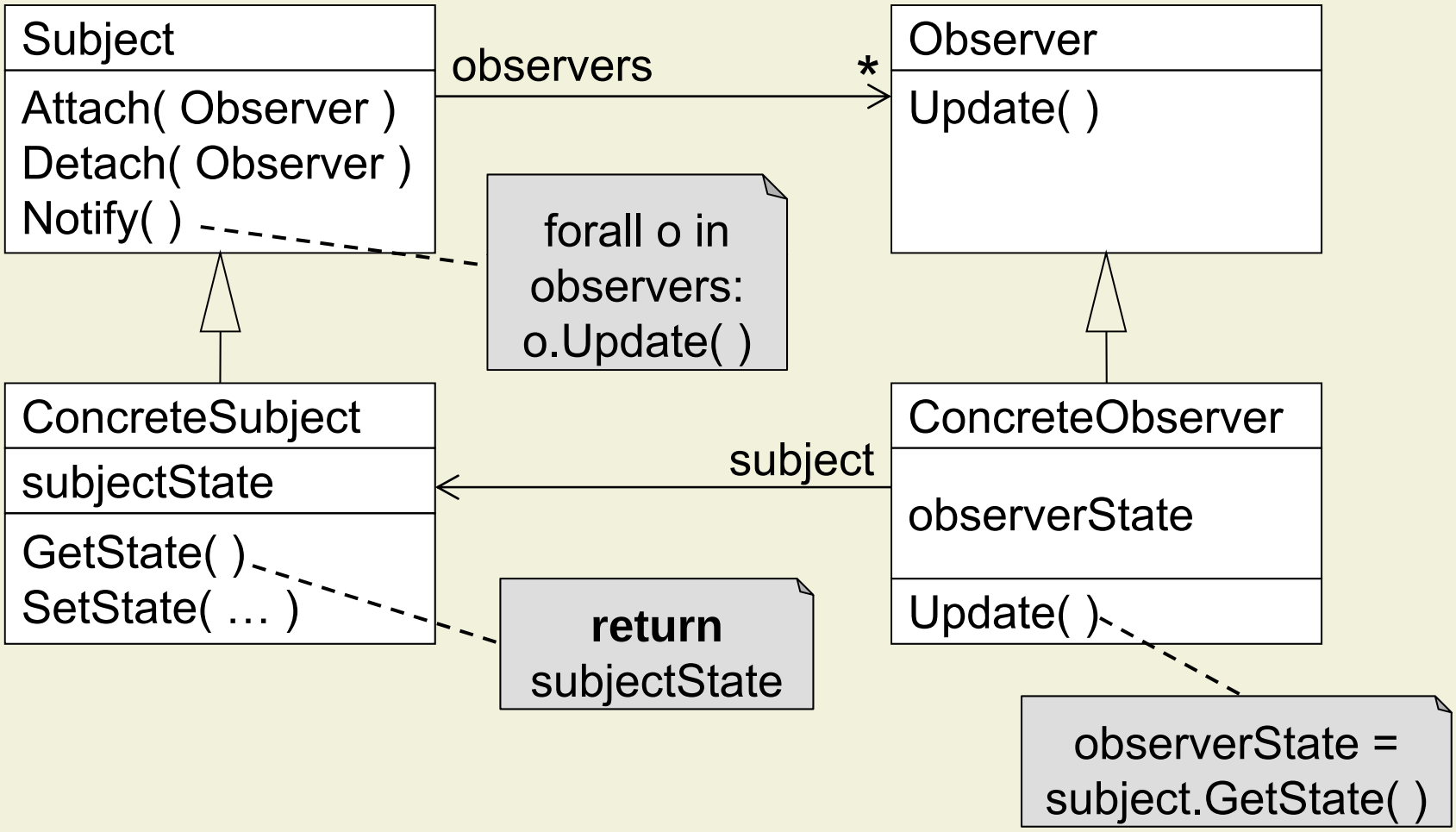


a = 50%
b = 30%
c = 20%

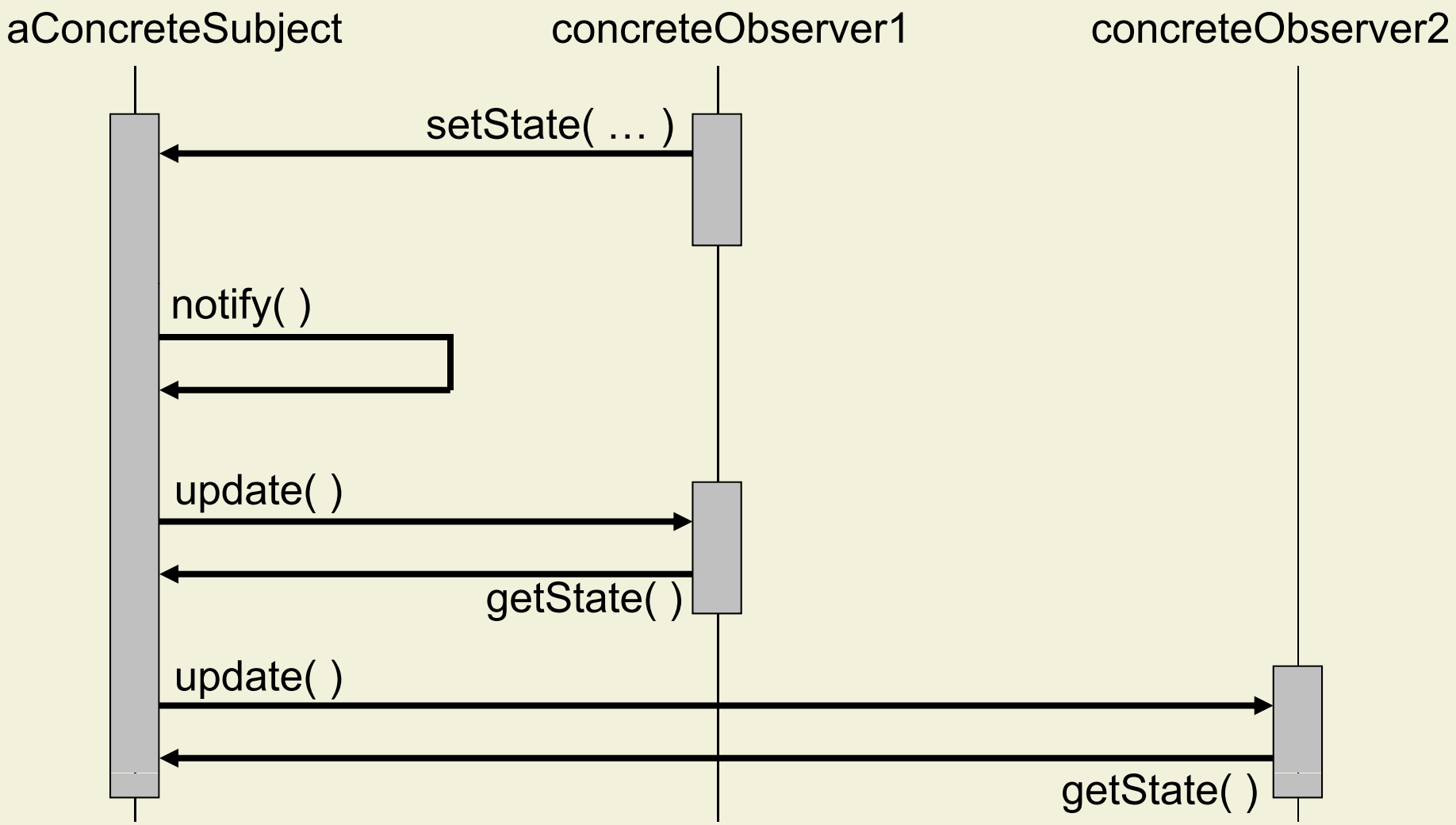
Change notification

Requests, modifications

Observer Pattern: Structure



Observer Pattern: Collaborations



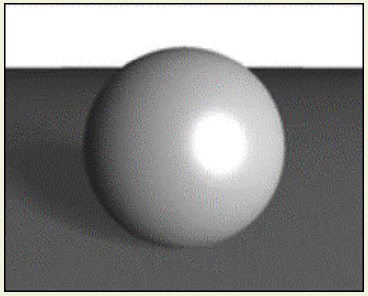
Observer Pattern: Properties

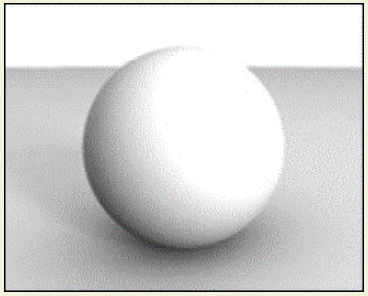
- Abstract coupling between subject and observer
  - Subject does not know concrete class of observer
- Support for broadcast communication
  - Freedom to add and remove observers

- Example
  - Debuggers (subject) broadcasts event when it reaches a breakpoint
  - Editor (observer) shows line of code
  - Stack tracer (observer) shows stack trace.

Strategy Pattern: Motivation

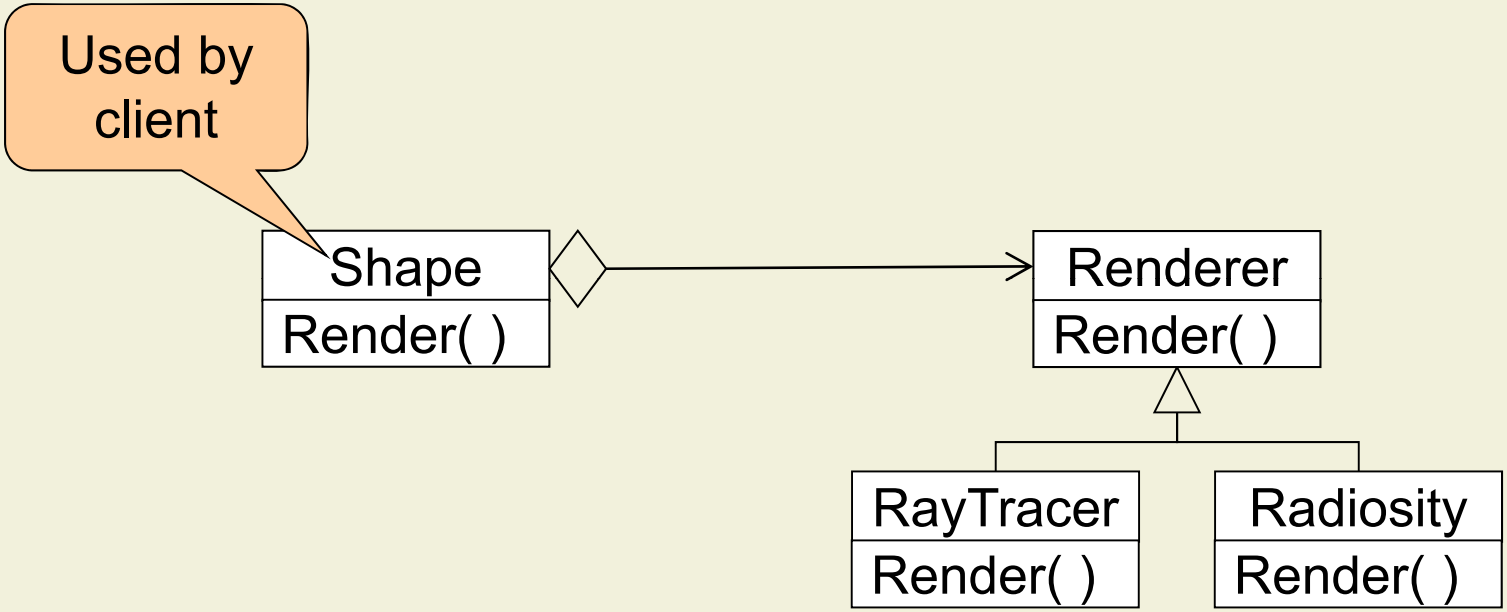
- A program uses 3D-shapes that can be rendered
  - Rendering code too complex to be included in Shape
- **Different** rendering **algorithms** are appropriate at different times
  - Do not implement the ones we do not use



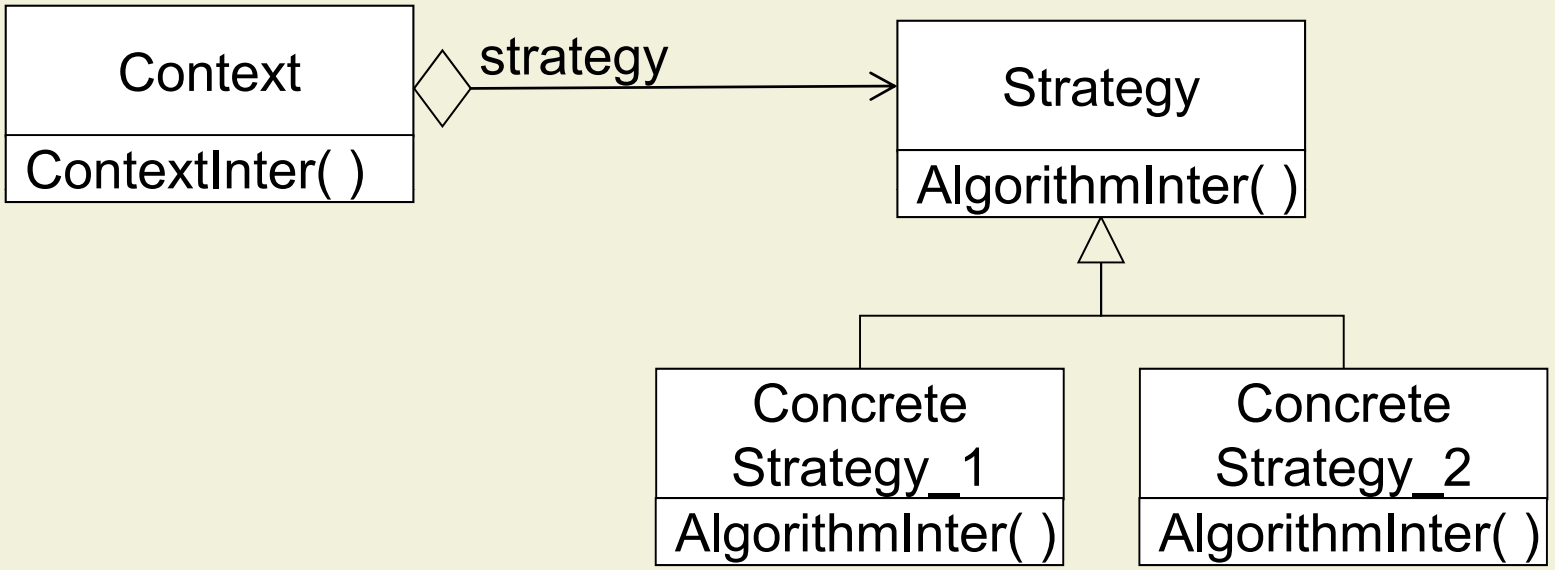


- Rendering algorithm should **not** be **hard-wired**
  - New algorithms may be added

Strategy Pattern: Example



Strategy Pattern: Structure

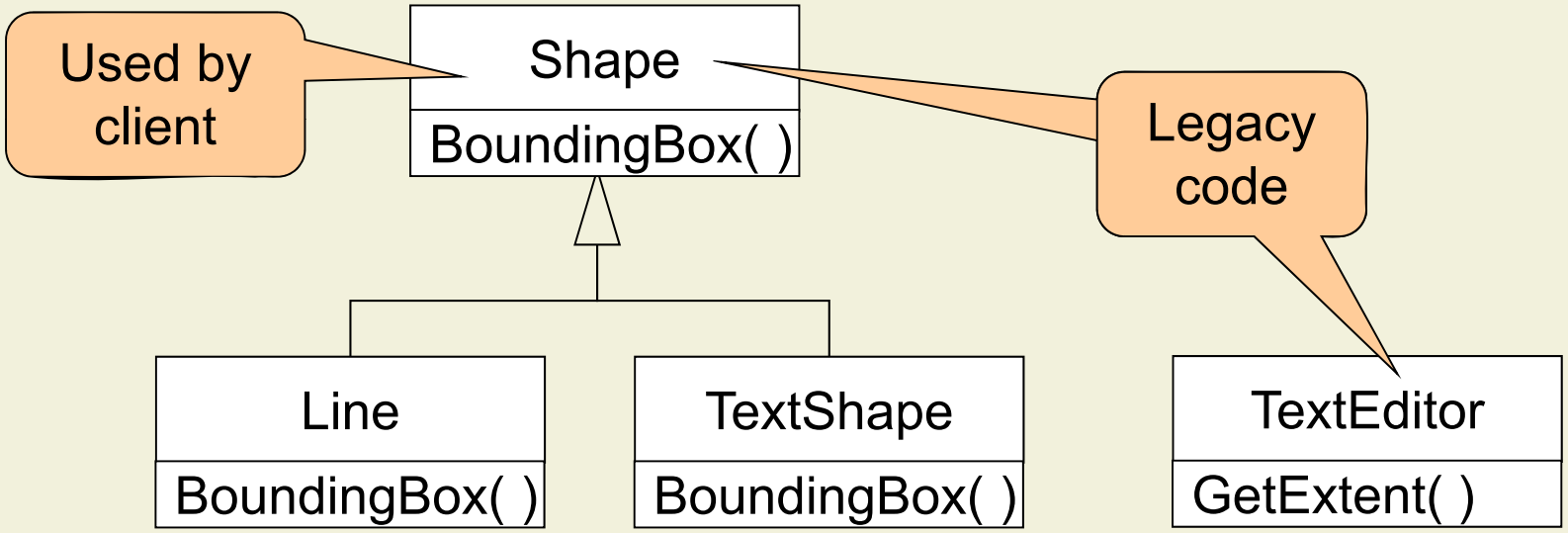


Strategy Pattern: Properties

- Supports families of algorithms
  - Sorting, line breaking, layouting, etc.
  - Clients have a choice (e.g., different space and time trade-offs)
- Alternative to inheritance
  - Behavior not hard-wired into context (dynamic exchange)
  - Separates context from algorithm (easier to maintain)
- Communication overhead
  - Arguments must be passed to strategies

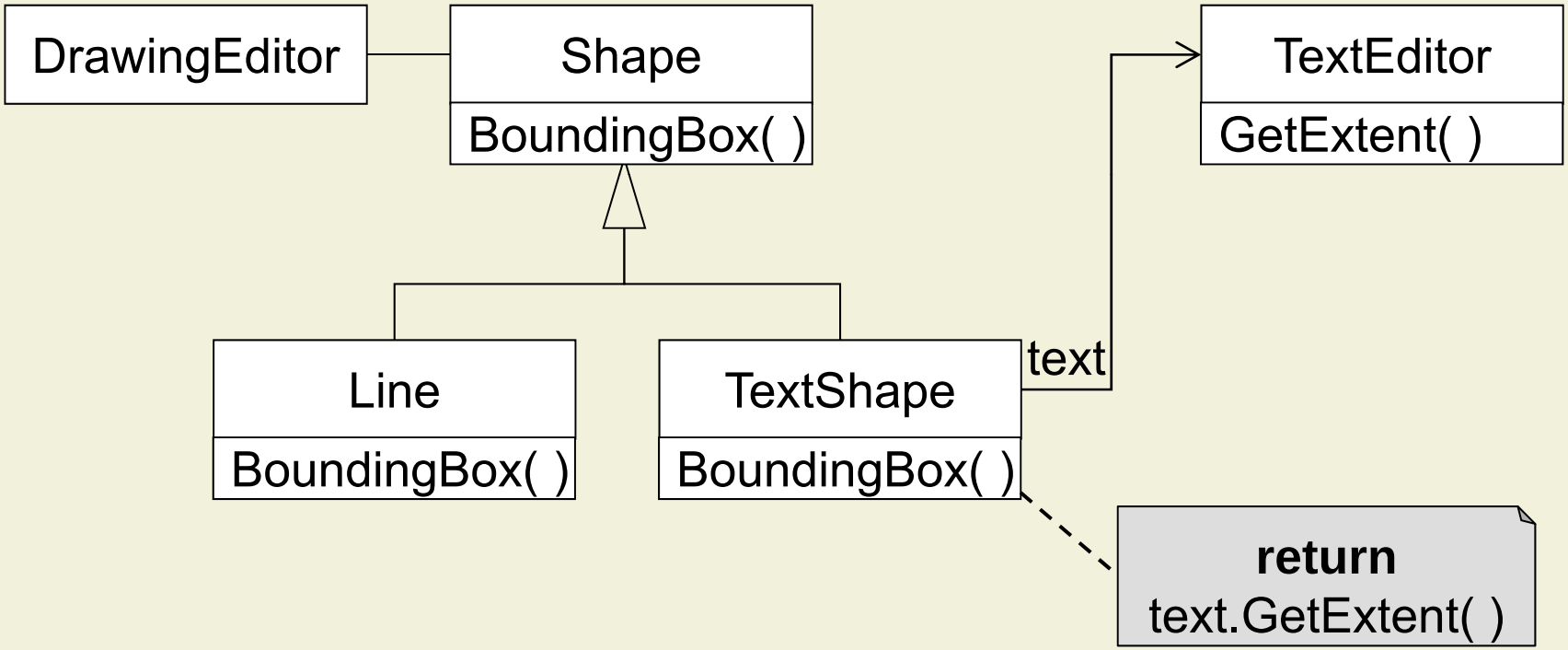
Adapter Pattern: Motivation

- A program expects an **interface** that **is incompatible** with the interface of a reusable class

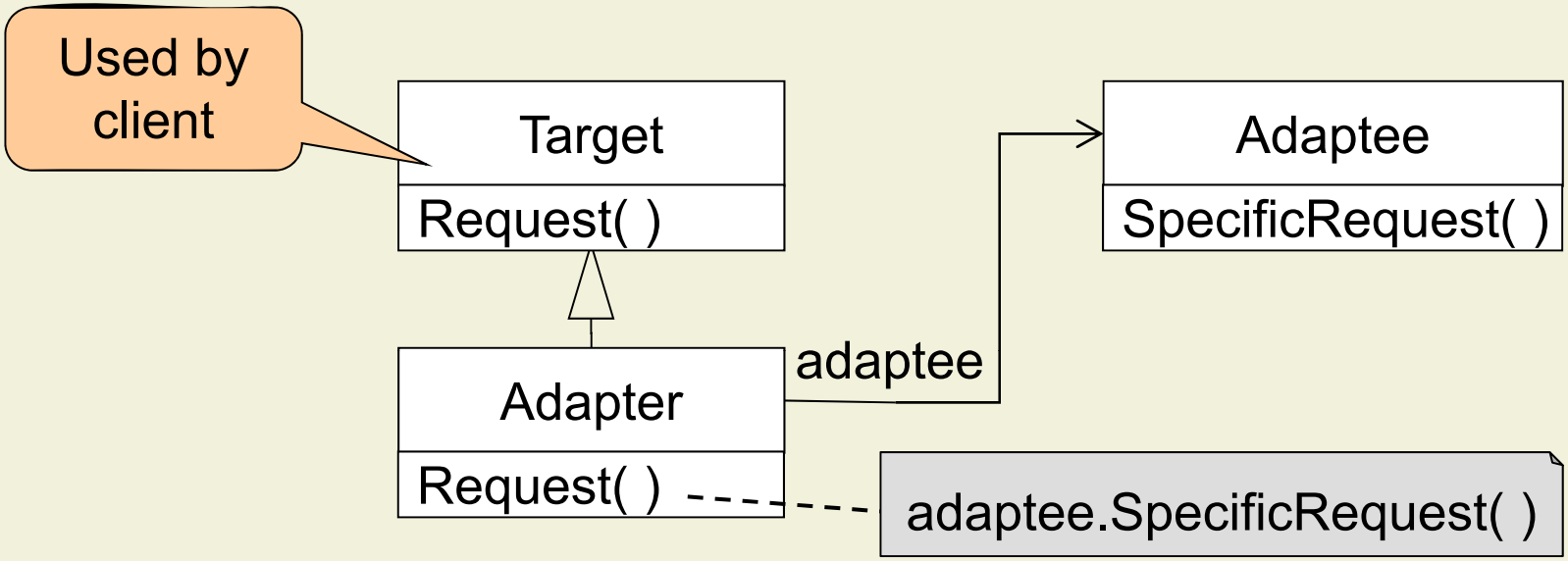


- Common problem with legacy code
- Also known as **wrapper**

Adapter Pattern: Example



Adapter Pattern: Structure



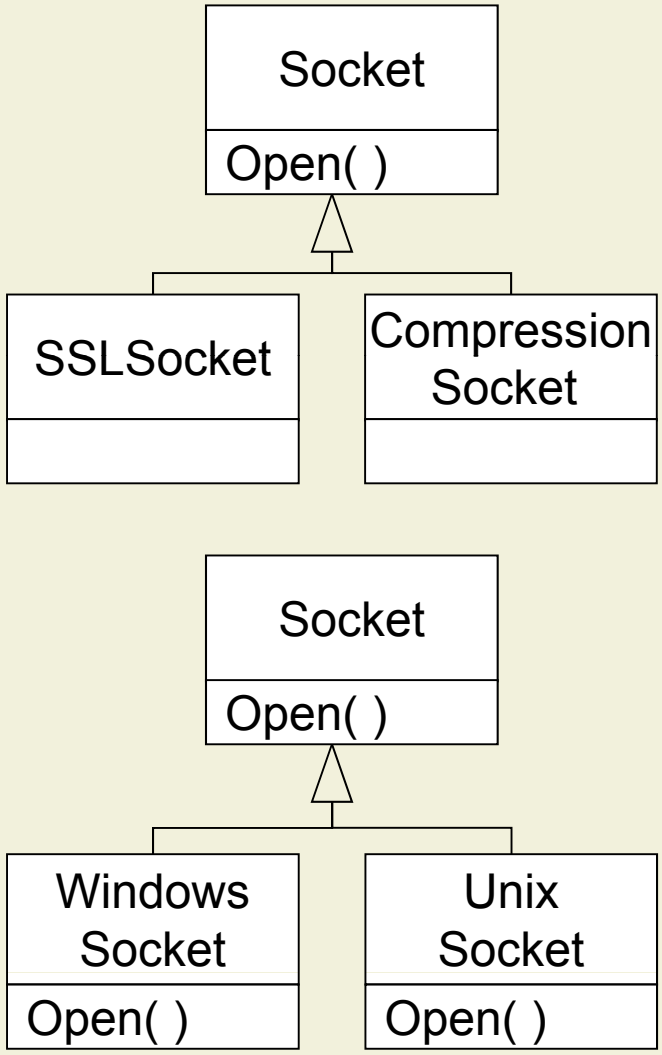
- **Delegation** used to bind Adapter and Adaptee
- Subtyping used to specify interface of Adapter
- Target and Adaptee exist before Adapter
- Target may be realized as interface in Java

Adapter Pattern: Properties

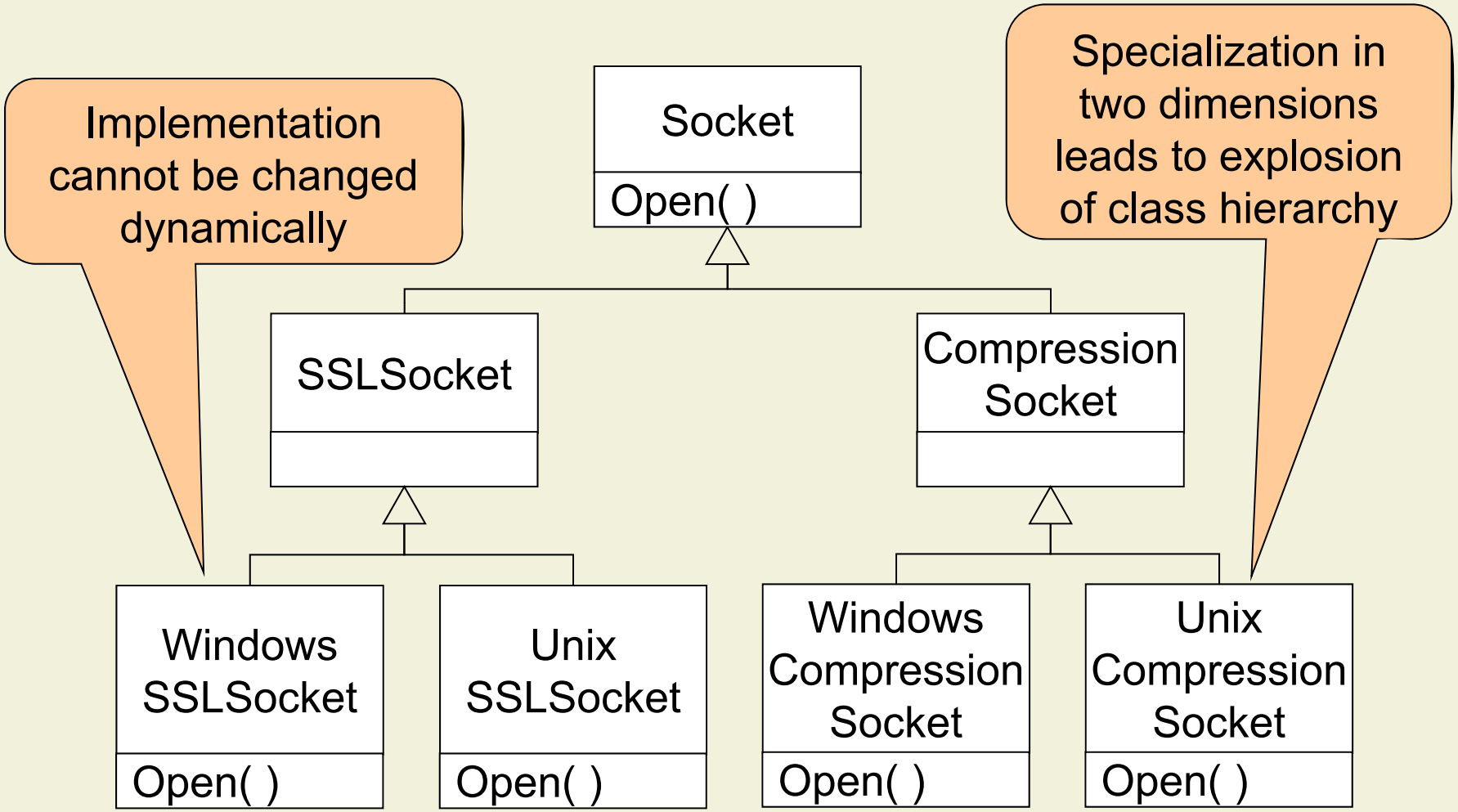
- How much adaptation does an adapter do?
  - From simple interface conversion (renaming) to entirely different set of operations
- Variant: class adapter
  - Adapter inherits from Target and Adaptee
  - No aggregation and delegation
  - Requires multiple inheritance if Target is a class

Bridge Pattern: Motivation

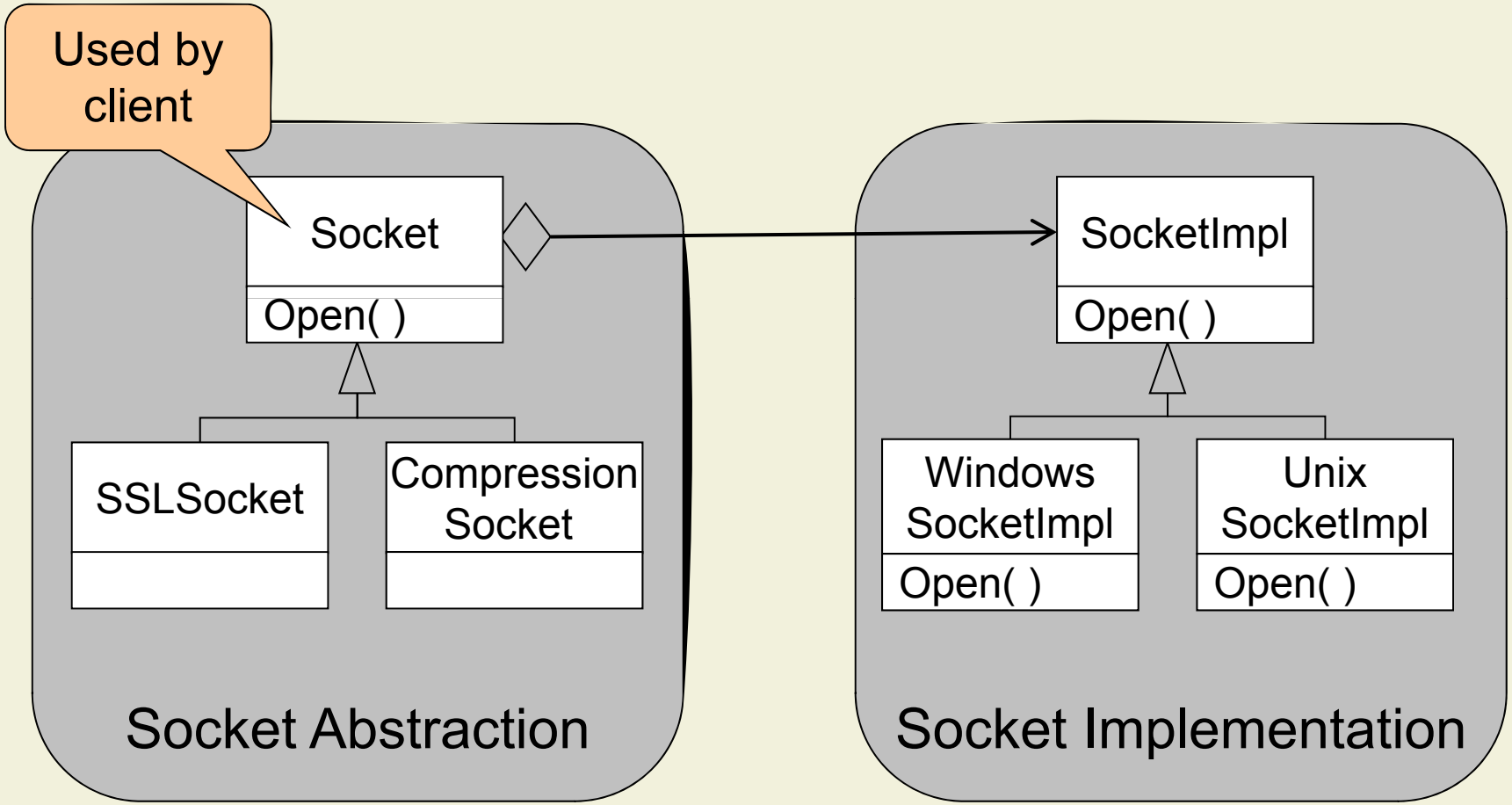
- A program uses socket abstractions to communicate
- Different socket abstractions
- Different socket implementations



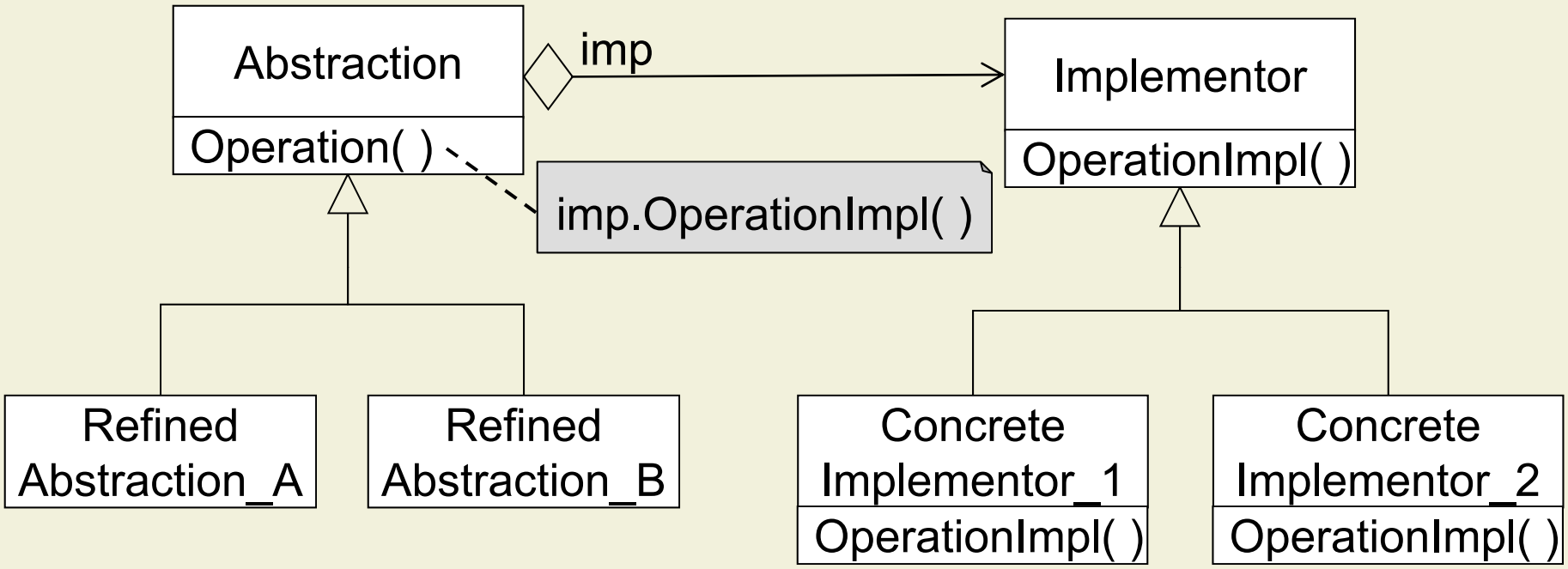
Bridge Pattern: Motivation (cont'd)



Bridge Pattern: Example



Bridge Pattern: Structure



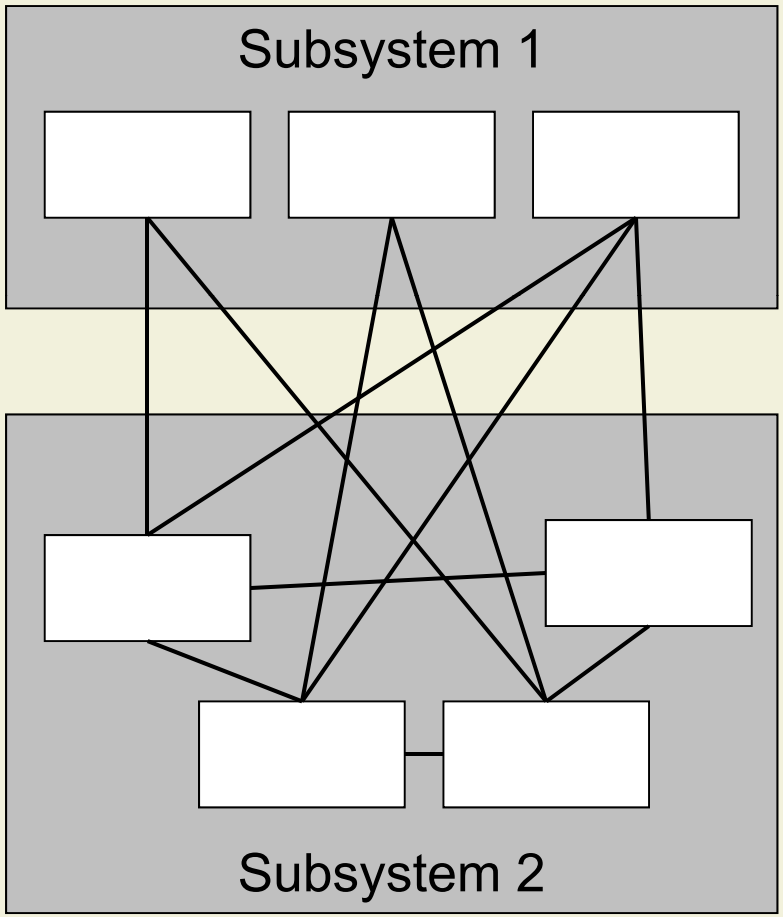
- **Decouples** an abstraction from its implementation
- Allows different **implementations** of an interface to be **exchanged dynamically**

Adapter vs. Bridge

- Both are used to hide the details of the underlying implementation
- Adapter pattern
  - Makes **unrelated components** work together
  - Applied to systems **after they are designed** (reengineering, interface engineering)
- Bridge Pattern
  - **Used up-front in a design** to let abstractions and implementations vary independently
  - Green field engineering of an “extensible system”

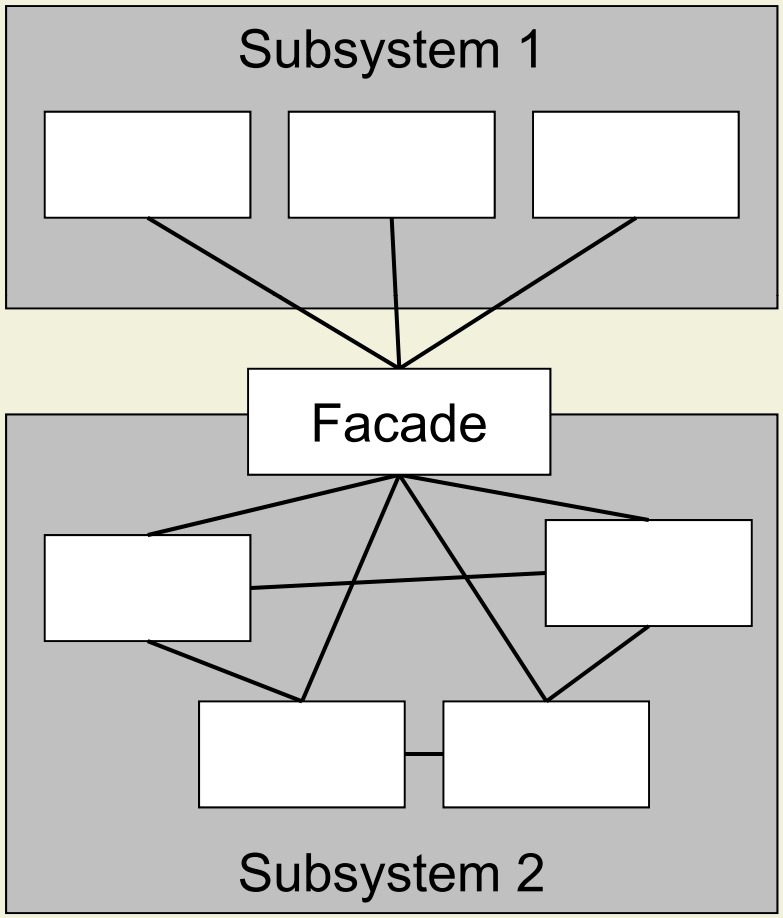
Facade Pattern: Motivation

- Subsystem 1 can call operations on any component of Subsystem 2
- Advantages
  - Efficiency
- Disadvantages
  - Caller does not understand how the subsystem works
  - Subsystem will be misused, leading to non-maintainable code



Facade Pattern: Example

- Provides a unified interface to a set of objects in a subsystem
- Defines a higher-level interface that makes the subsystem easier to use
- **Reduces coupling**
- Does not prevent direct usage of objects in a subsystem



Subsystem Design with Facade and Adapter

- Ideal structure of a subsystem
- An interface object (boundary object)
- A set of application domain objects (entity objects) modeling real entities or existing systems
  - Some of the entity objects are interfaces to existing systems
- One or more control objects

Facade

Interface to existing systems: Adapter

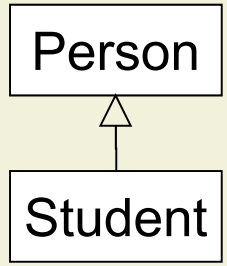
- Provides interface to existing system
- Existing system is not necessarily object-oriented!

Design Patterns Encourage Reusable Designs

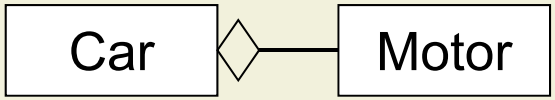
- **Facade** should be used by **all subsystems**
  - Defines all the services of the subsystem
  - Delegates requests to components within the subsystem
  - Most of the time the facade need not be changed when the component is changed
- **Adapters** should be used to **interface** to **existing components**
- **Bridges** should be used to interface sets of objects
  - Where the full set is not completely known at design time
  - When the subsystem must be extended later after the system has been deployed (dynamic extension)

The “Ingredients” of Design Patterns

- Inheritance (subclassing)
  - Establishes **“is-a” relation**
  - Enables subtype **polymorphism**

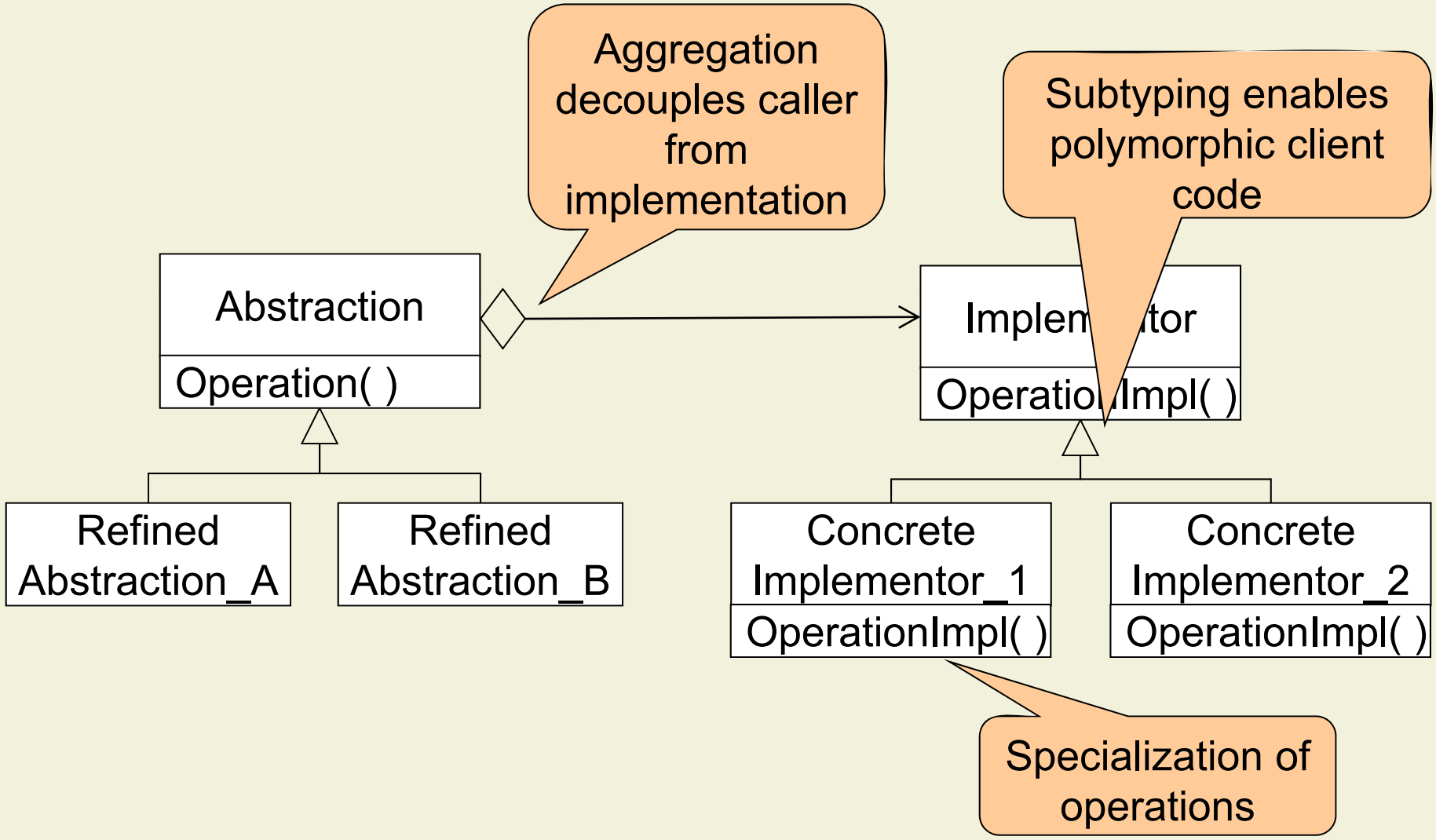


- Aggregation
  - Establishes **“has-a” relation**
  - **No subtyping** in general



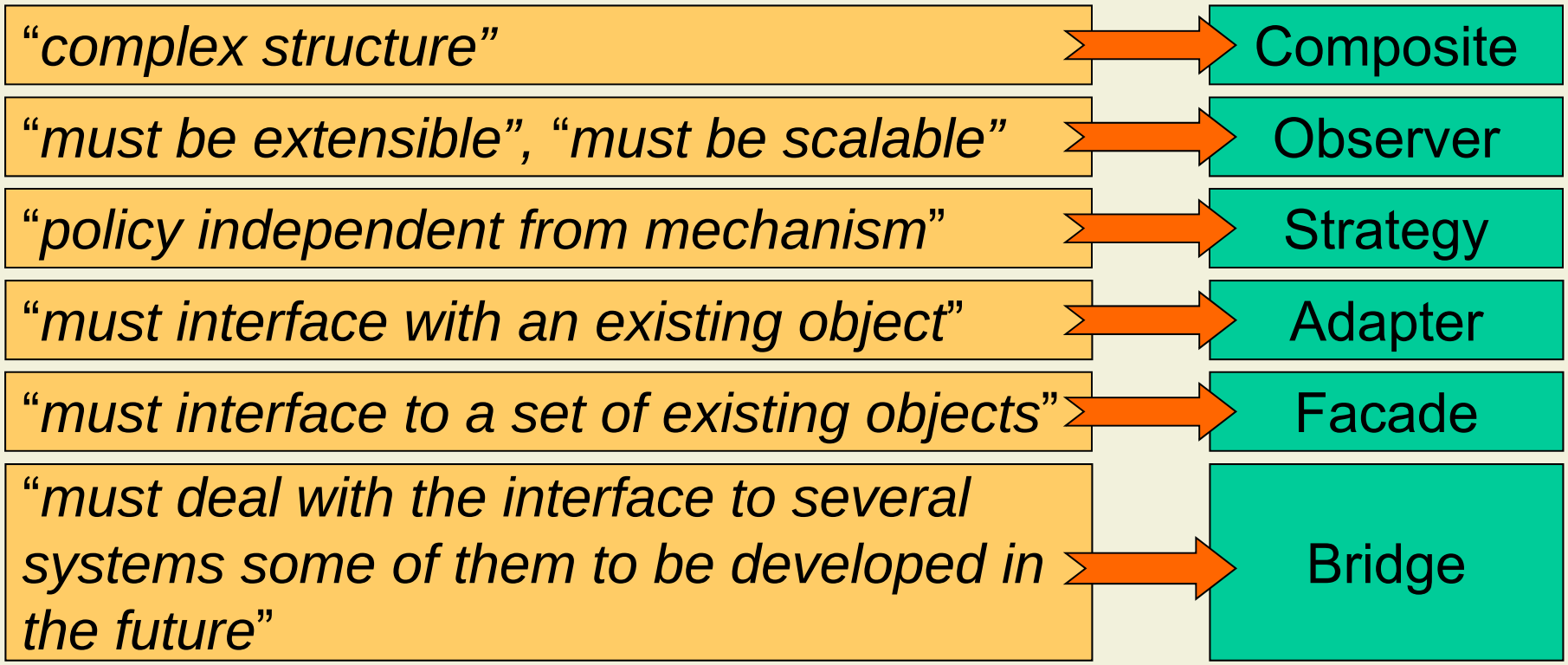
- Design patterns provide guidance how to use inheritance and aggregation

Inheritance and Aggregation: Bridge Pattern



Textual Clues in Nonfunctional Requirements

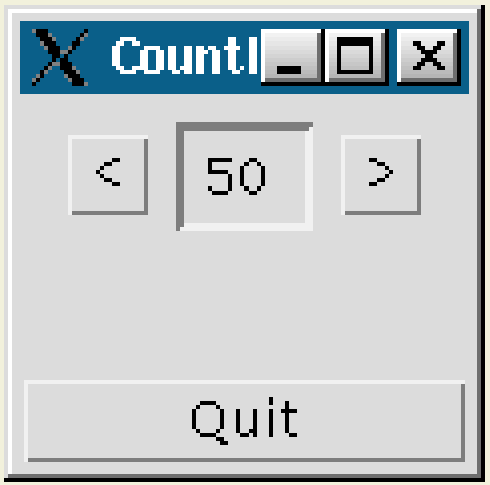
- Use textual clues to identify design patterns
  - (similar to Abbot's technique in analysis)



5. Detailed Design

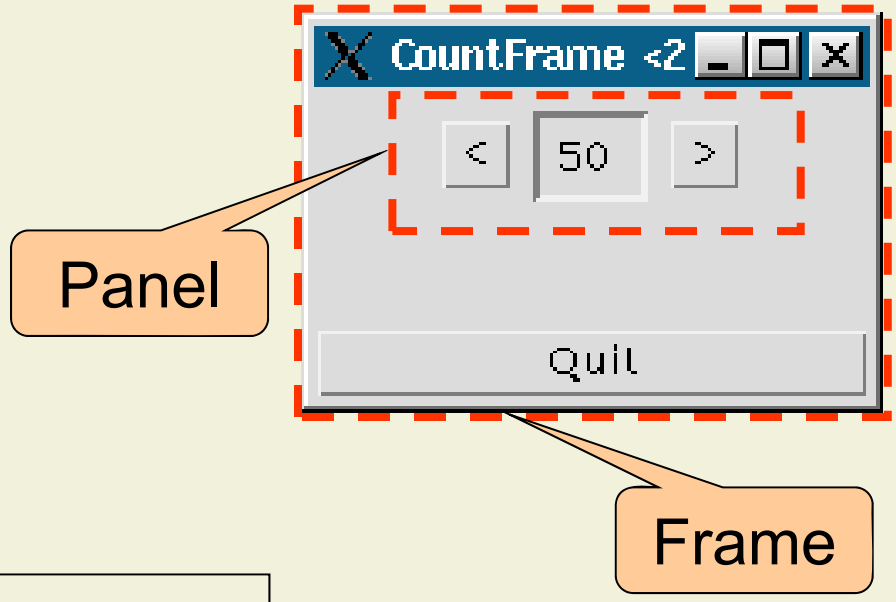
AWT: Overview

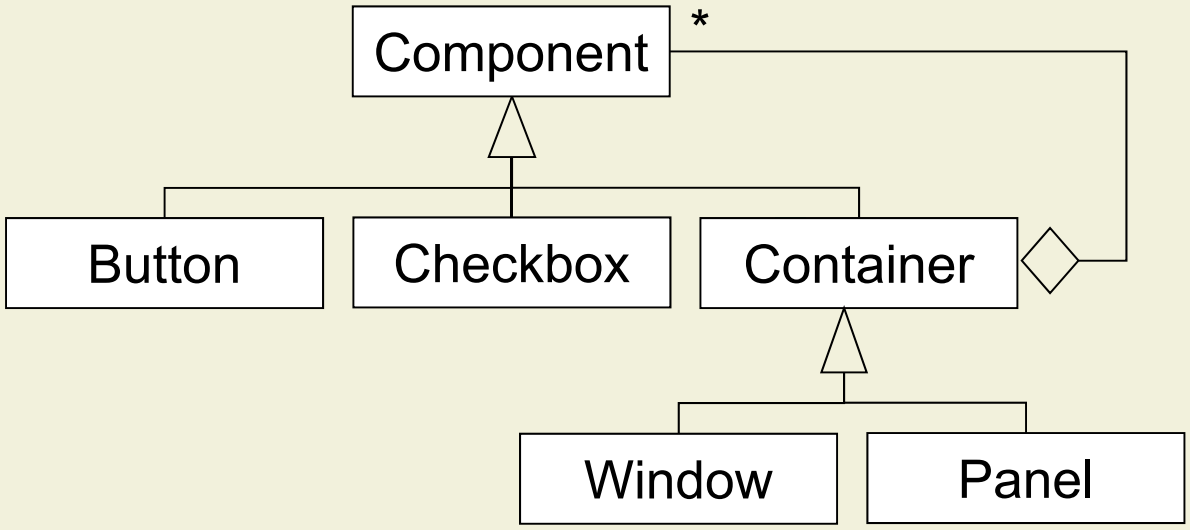
- AWT: Abstract Window Toolkit
- Elements of the GUI are represented by **components**
- **Display** and **layout** of the components have to be specified
- Components receive **events** from the window system and propagate them to so-called **listeners**



Component Hierarchy: Composite Pattern

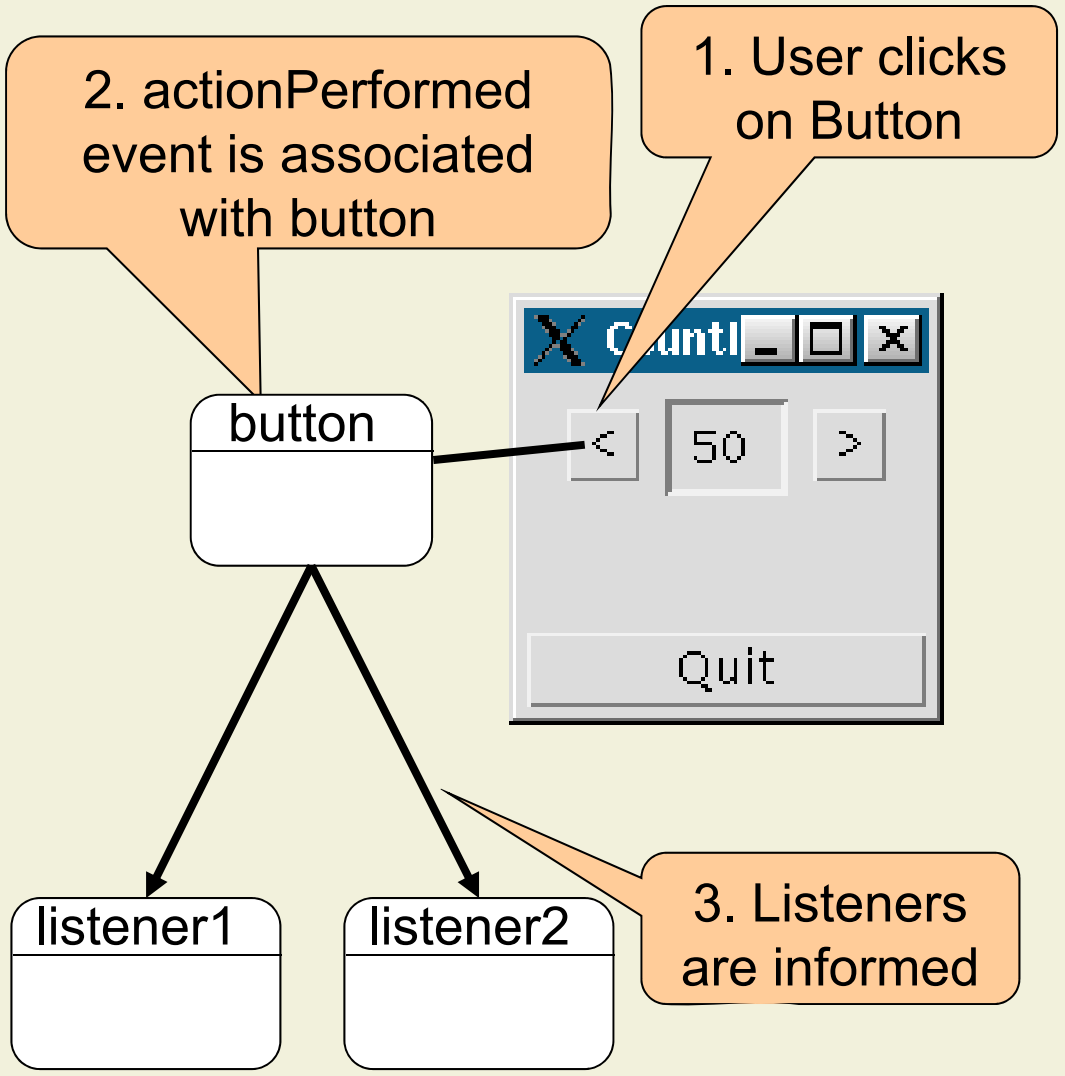
- Components can be grouped into containers
- Containers are also components



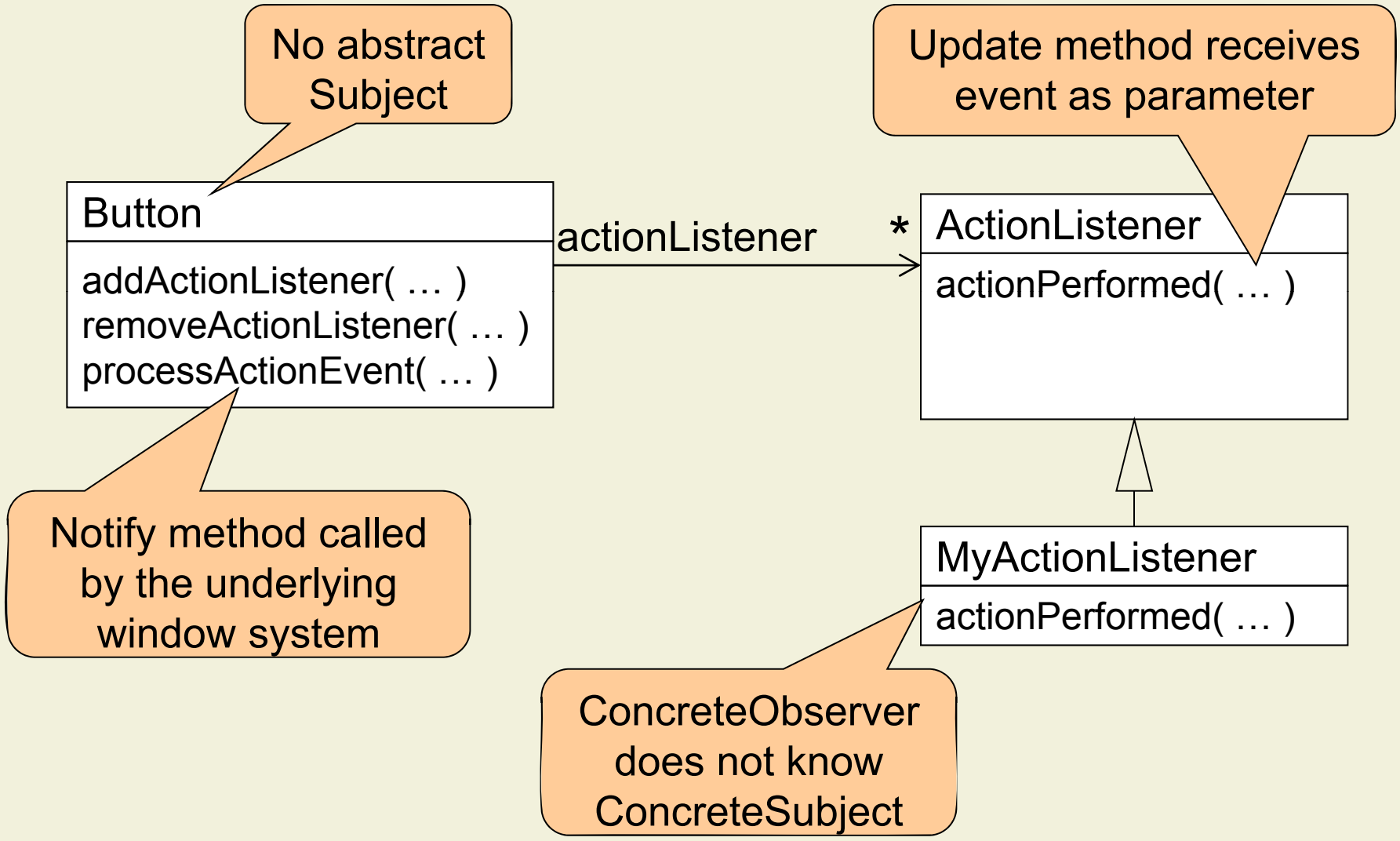


Event Communication

- Objects can register at a component as observer (listener) for one or several event types
- Upon occurrence of an event, the event source informs all registered objects by invoking a method

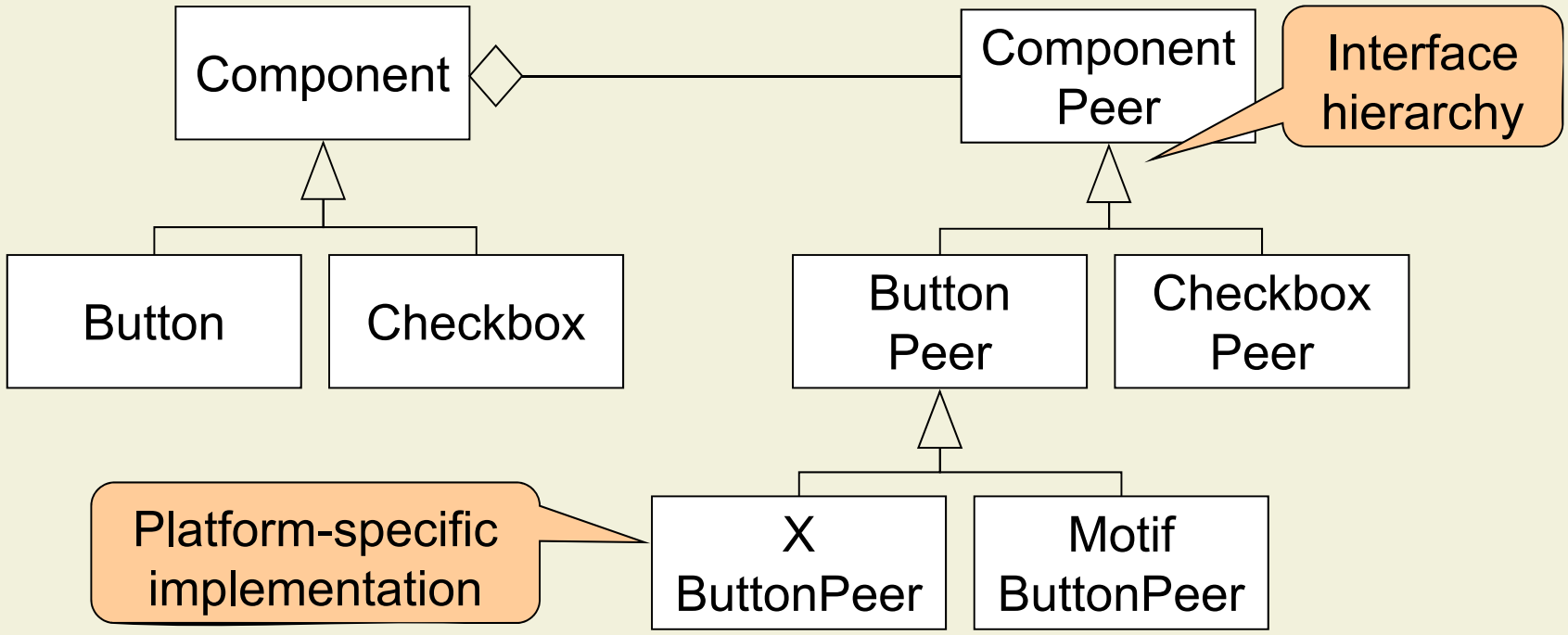


Component / Listener: Observer Pattern



Platform Independence: Bridge Pattern

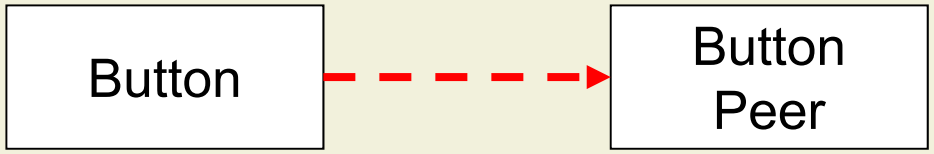
- AWT components are platform-independent
- Operations that depend on the window system are delegated to platform-specific peer objects



Platform Independence: Peer Creation

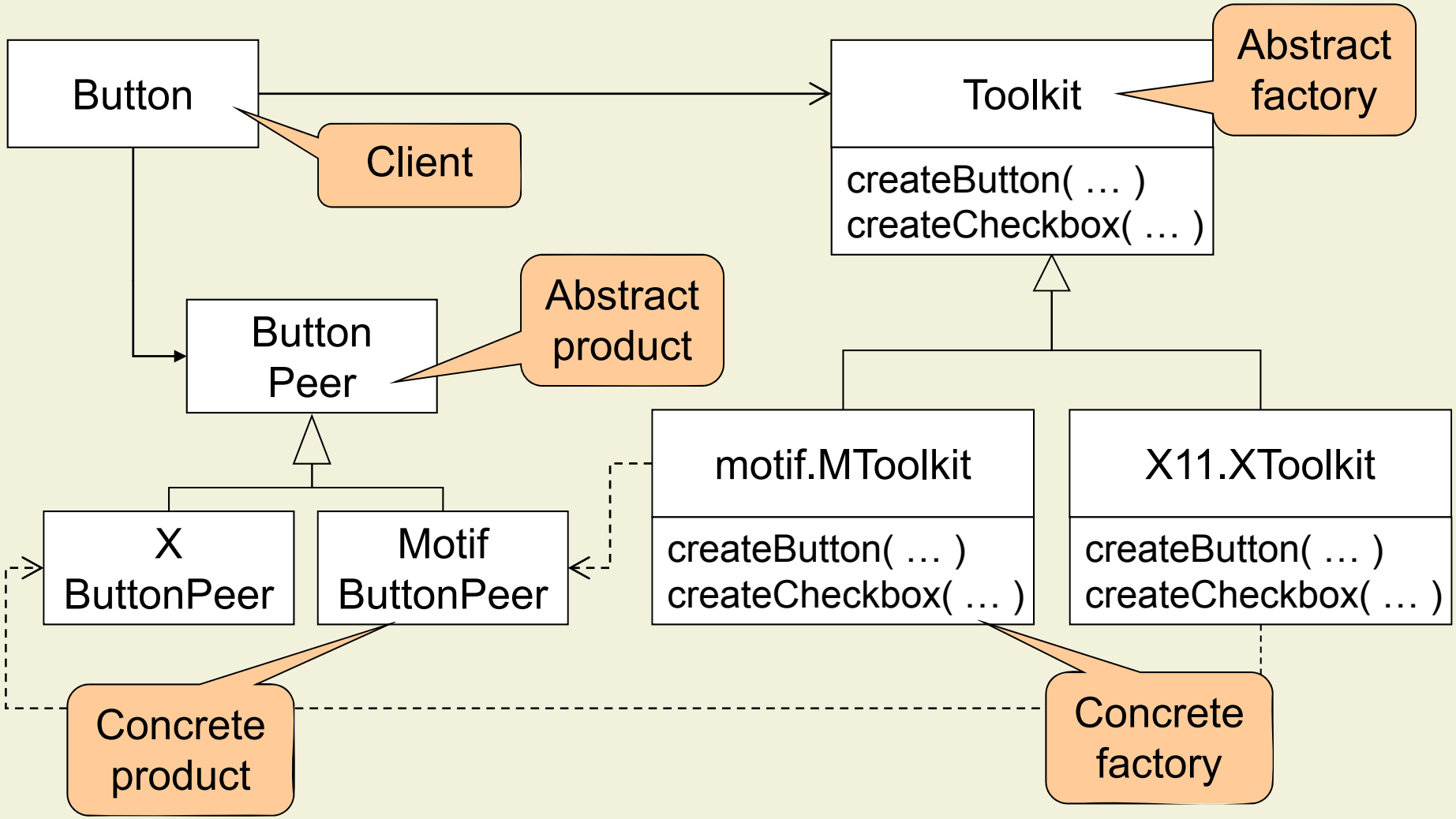
- Component objects have references to their peers

- Platform-independent components cannot instantiate platform-dependent peers



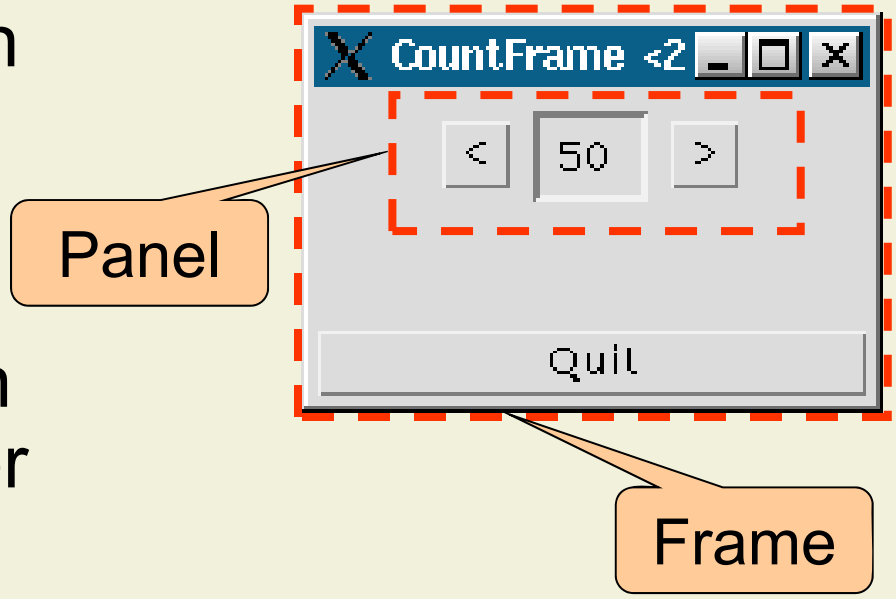
- Solution: abstract factory

Platform Independence: Abstract Factory

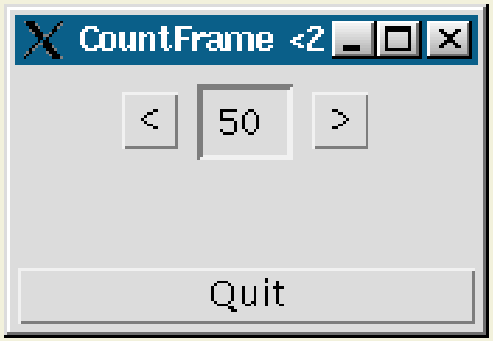


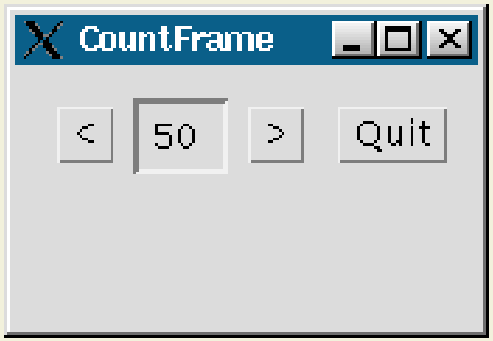
Displaying Containers: Layout Managers

- Layout of components in one container is computed by a layout manager
- The layout manager can be set for each container



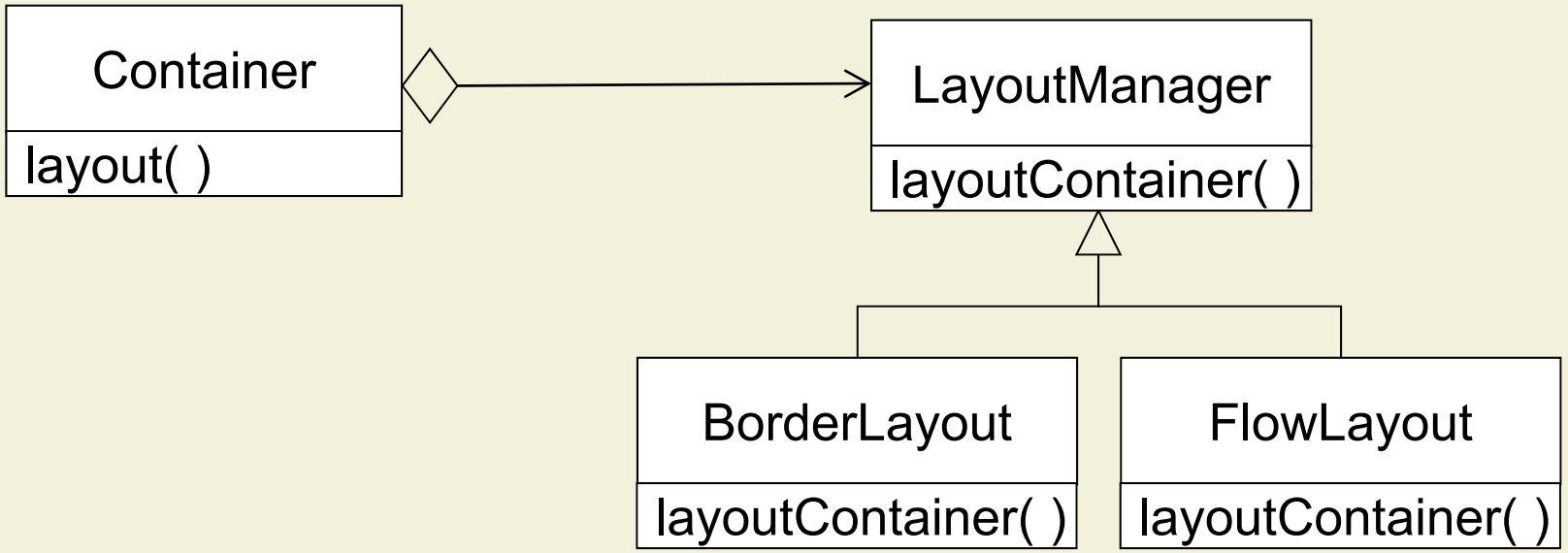
Border-Layout





Flow-Layout

Layout Managers: Strategy Pattern

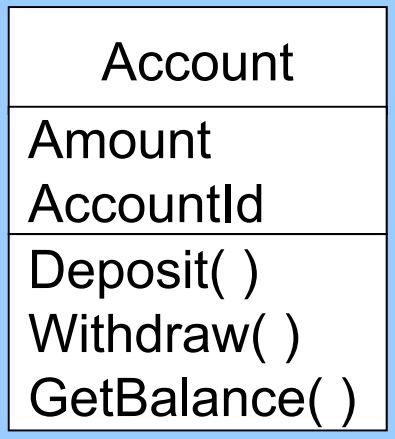


5. Detailed Design

Specifying Interfaces

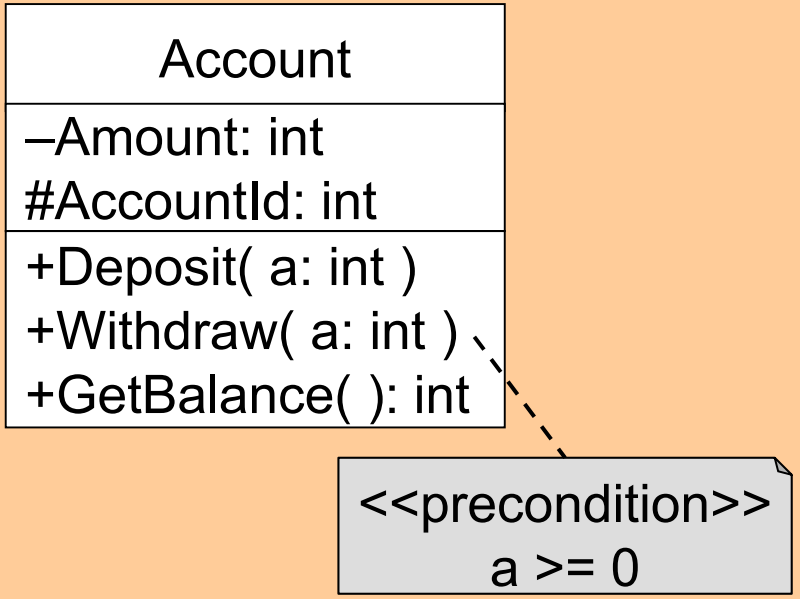
Requirements Analysis

- Attributes
- Operations without parameters and types



Detailed Design

- Visibility
- Signatures
- Contracts



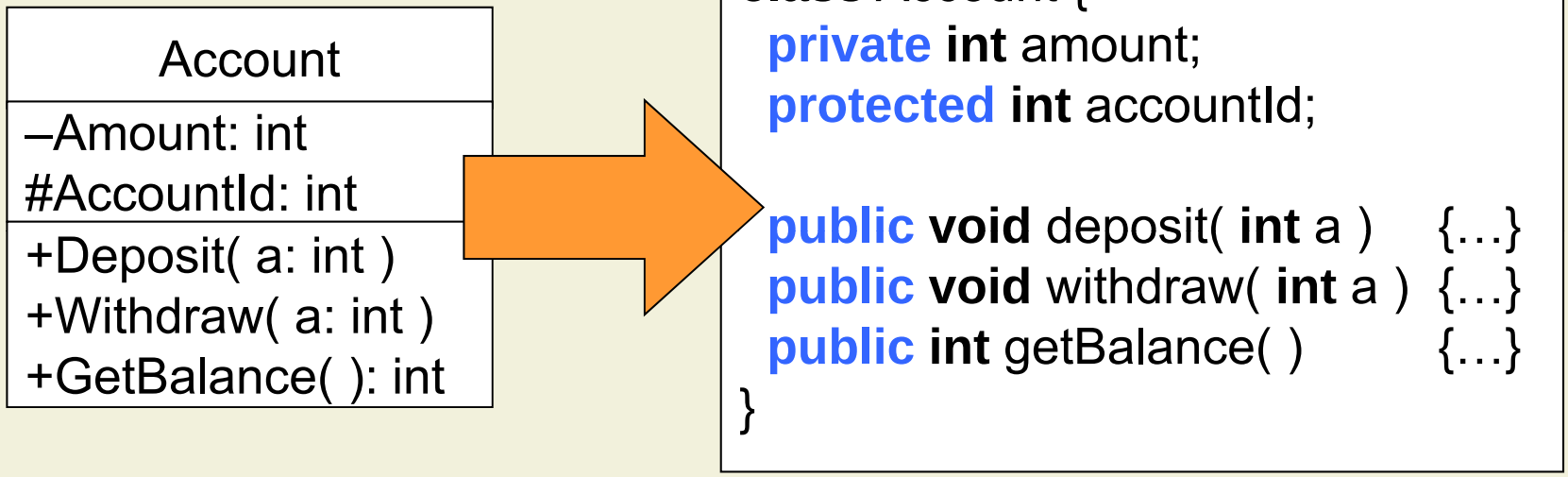
Information Hiding

- Definition
  Information hiding is a technique for reducing the dependencies between modules:
  - *The intended client is provided with all the information needed to use the module **correctly**, and **with nothing more***
  - *The client uses only the (publicly) available information*

Visibility Information

- UML defines **three levels of visibility**
  - Similar to C++, Java, and C#
- Private (**implementation interface**): "–"
  - Private features can be accessed only by the class in which they are declared (not even subclasses)
- Protected (**subclass interface**): "#"
  - Protected features can be accessed by the class in which they are defined and by any descendent of the class
- Public (**client interface**): "+"
  - Public features can be accessed by any class

Implementation of UML Visibility in Java

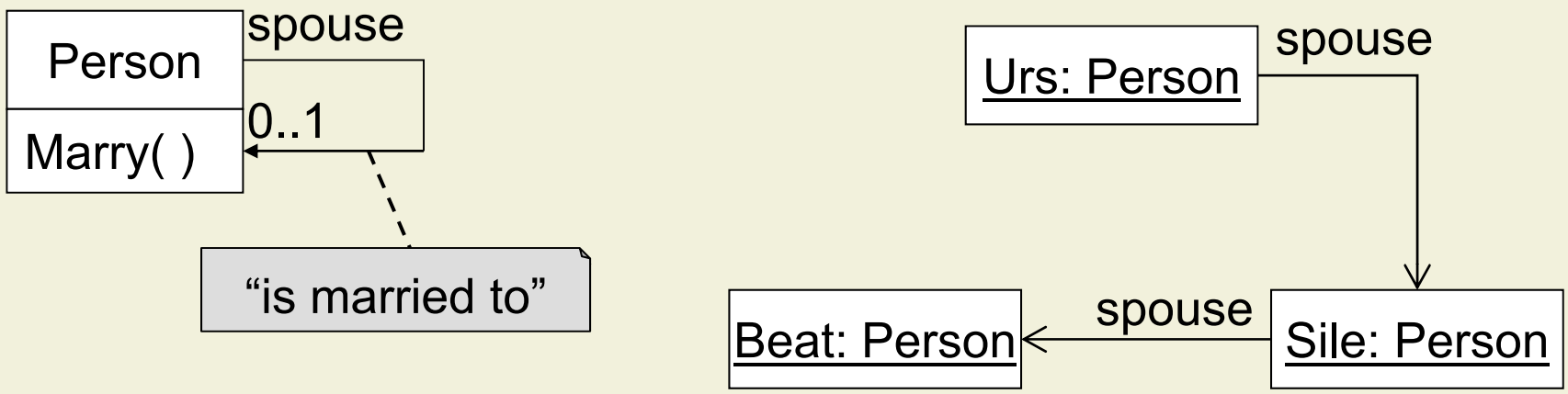


- **protected** has a slightly different meaning in Java
  - Also visible to classes in the same package
- Eiffel provides more fine-grained visibility control

Information Hiding Heuristics

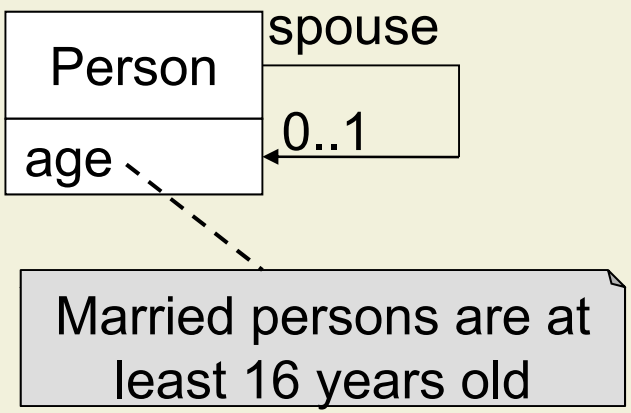
- Public interface for classes and subsystems
  - Use the facade pattern
  - Define abstract interfaces that mediate between system and external world as well as between subsystems
- The less an operation knows the less likely it will be affected by any changes
- Access attributes only via operations
  - Only the operations of a class should manipulate its attributes (no public attributes)
  - Trade-off: Information hiding vs. efficiency

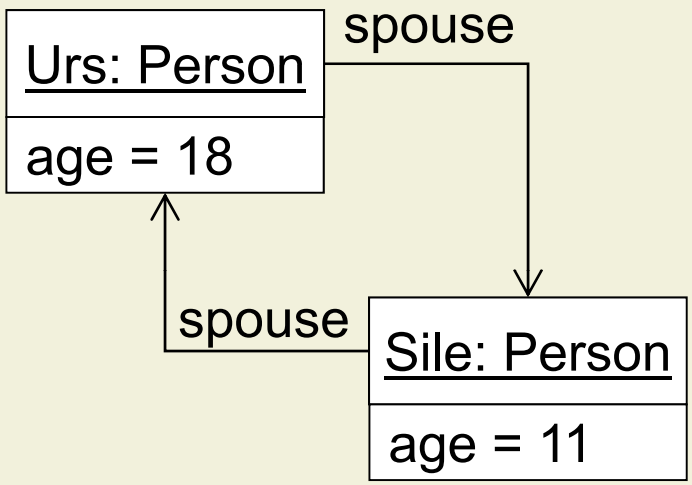
UML is not Enough



- Urs is married to Sile, Sile is married to Beat, and Beat is not married at all
- A valid instantiation of the class diagram!
- Associations describe relations between classes

UML is not Enough (cont'd)





- Urs is married to Sile, who is only eleven
- A valid instantiation of the class diagram!
- Class diagrams do not restrict values of attributes

Expressing Contracts

- Natural language
  - Advantage: Easy to understand and use
  - Disadvantage: Ambiguous
- Mathematical notation
  - Advantage: Precise
  - Disadvantage: Difficult for normal customers
- Contract language
  - Formal, but easy to use
  - Examples: Eiffel, JML

spouse expresses "is married to"

$$\text{spouse}: \text{Person} \leftrightarrow \text{Person}$$
$$\text{spouse} = \text{spouse}^{-1}$$
$$\text{souse} \cap \text{id} = \emptyset$$

$$\forall p: \text{Person}: p \in \textbf{dom}(\text{spouse}) \Rightarrow \text{spouse}(p) \in \textbf{dom}(\text{spouse}) \wedge p \neq \text{spouse}(p) \wedge p = \text{spouse}(\text{spouse}(p))$$

spouse /= **Void implies**
spouse /= **Current and**
spouse.spouse = **Current**

Contracts in Eiffel: Object Invariants

- Associated with classes
- Describe **consistency criteria** of objects and object structures
- Hold for all instances of a class

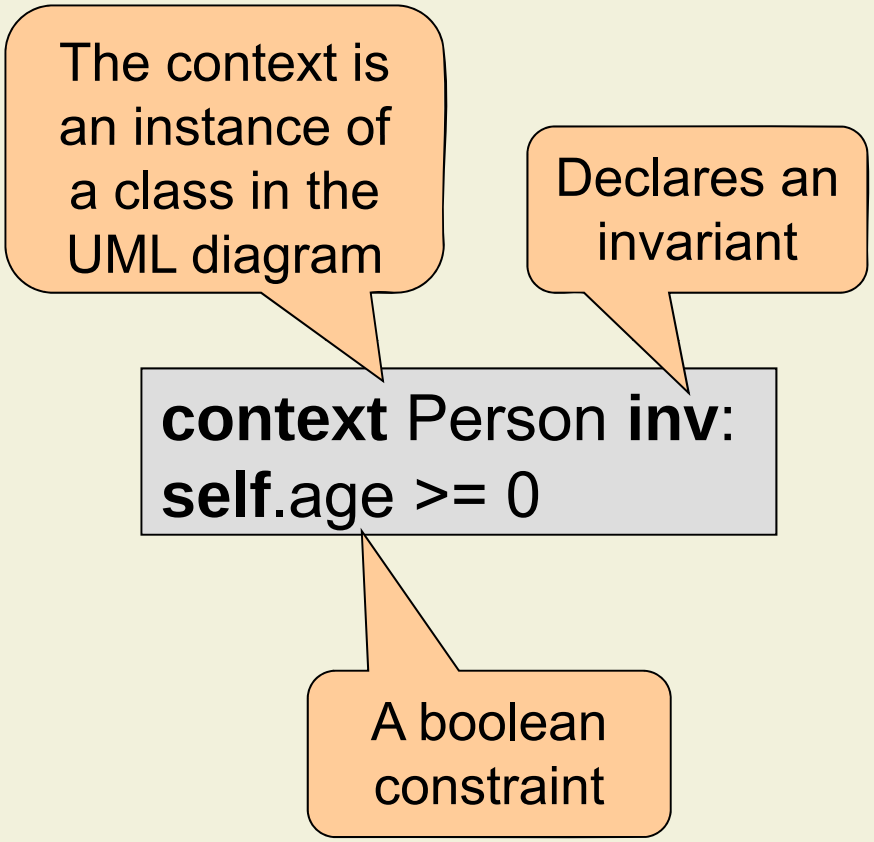
```
class PERSON feature

  age: INTEGER
  spouse: PERSON

  invariant
    spouse /= Void
    implies
      spouse /= Current
    and
      spouse.spouse = Current
    and
      age >= 16

end -- class PERSON
```

Object Constraint Language – OCL

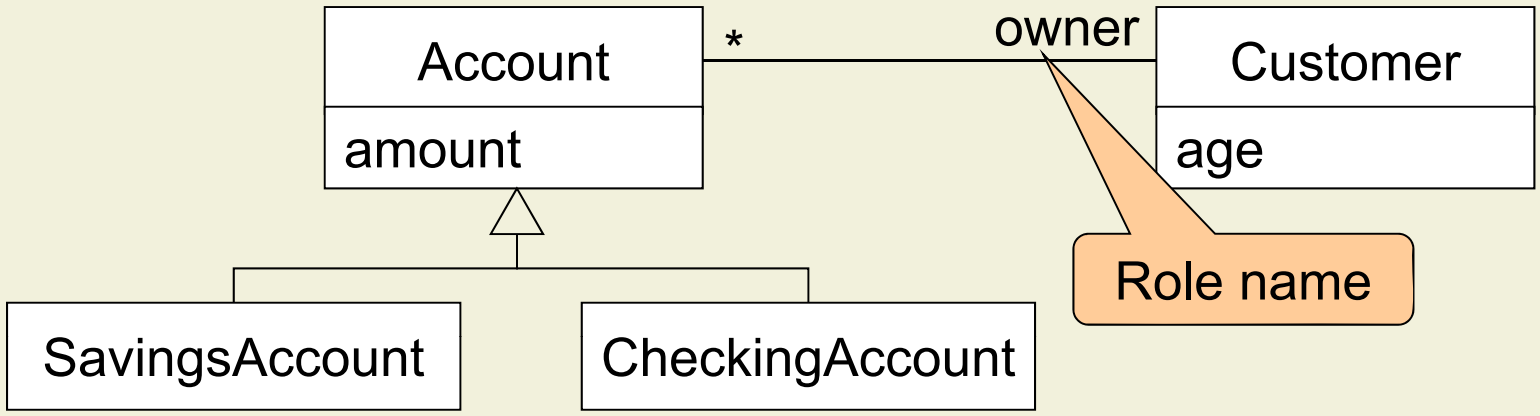
- The contract language for UML
- Used to specify
  - Invariants of objects
  - Pre- and postconditions of operations
  - Guards (for instance, in state diagrams)
- Special support for
  - Navigation through UML class diagram
  - Associations with multiplicities

Form of OCL Invariants

- Constrains can mention
  - **self**: the contextual instance
  - Attributes and role names
  - Side-effect free methods (stereotype <<query>>)
  - Logical connectives
  - Operations on integers, reals, strings, sets, bags, sequences
  - Etc.



OCL Invariants: Example



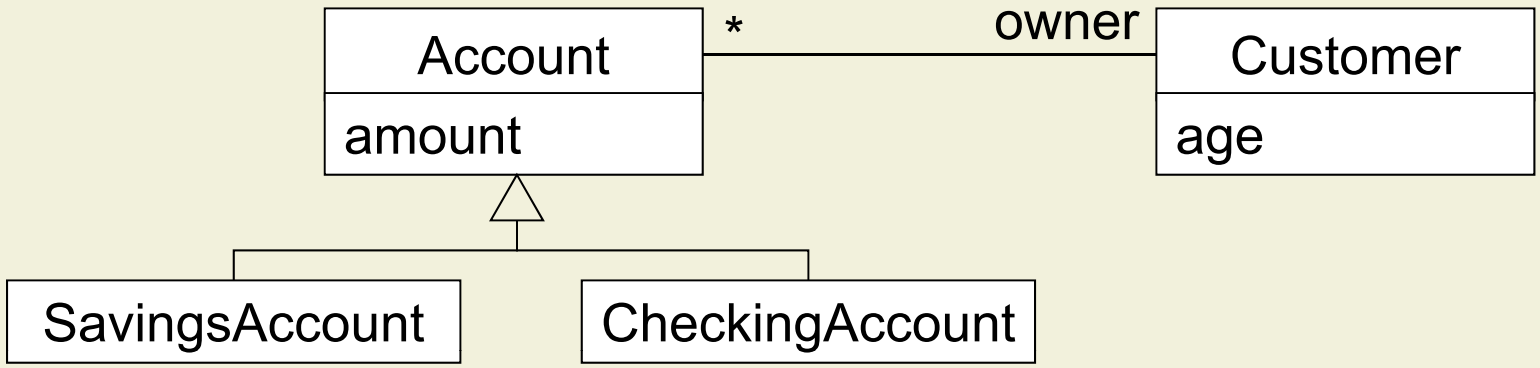
- A savings account has a non-negative balance

context SavingsAccount **inv**:
self.amount >= 0

- Checking accounts are owned by adults

context CheckingAccount **inv**:
self.owner.age >= 18

OCL Invariants: Contexts



- Checking accounts are owned by adults

context CheckingAccount **inv**:
self.owner.age >= 18

- Accounts are owned by adults

context Account **inv**:
self.owner.age >= 18

- Customers are adults

context Customer **inv**:
self.age >= 18

Collections

- OCL provides three predefined collection types
  - Set, Sequence, Bag
- Common operations on collections

size( )	Number of elements in the collection
includes(object)	True iff the object is an element
isEmpty( )	True iff collection contains no elements
exists(expression)	True iff expression is true for at least one element
forAll(expression)	True iff expression is true for all elements

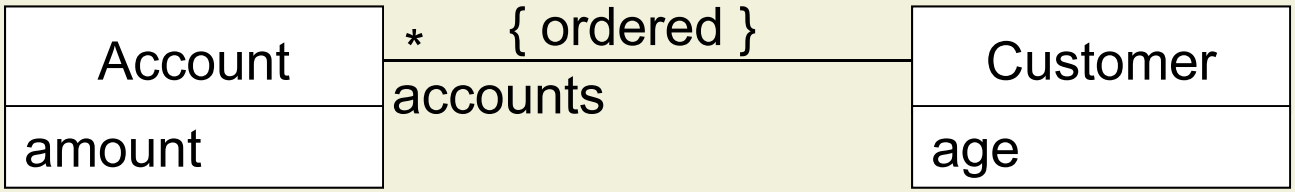
Generating Collections

- Explicitly enumerating the elements

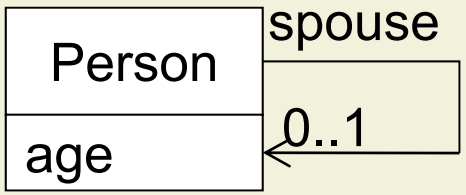
Set { 1, 7, 16 }

- By navigating along 1:n associations
  - Navigation along a single 1:n association yields a Set
  - Navigation along a single 1:n association labeled with the constraint { ordered } yields a Sequence

self.accounts



Example: Multiplicity Zero or One



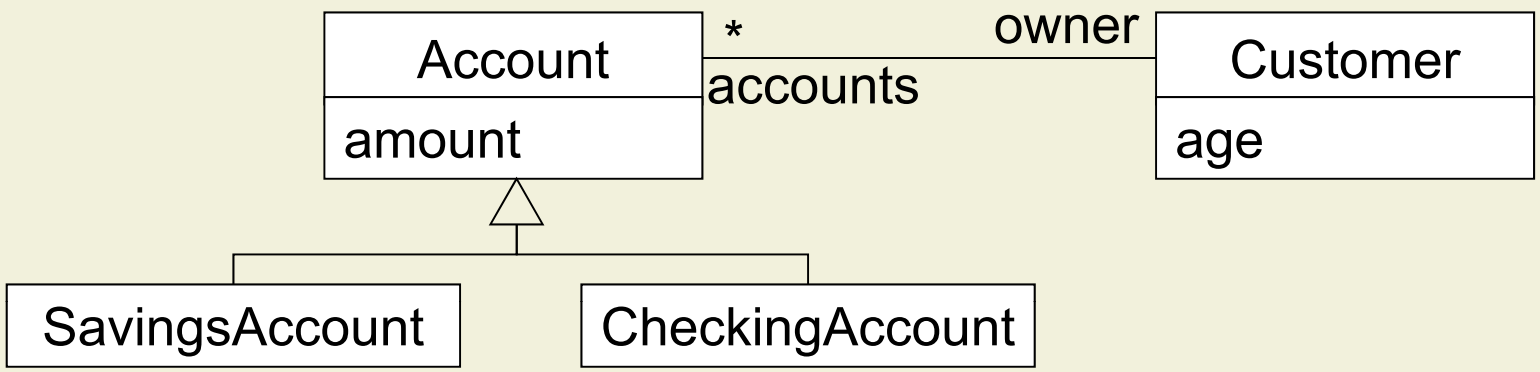
self can be omitted

spouse used as set

```
context Person inv:
spouse->size( ) = 1 implies
age >= 16 and spouse.spouse = self and spouse <> self
```

spouse used as object

Example: Quantification and Type Information

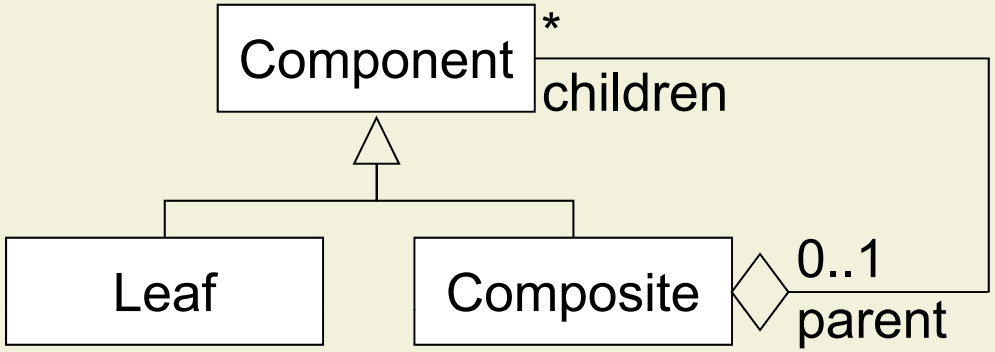


Subtype relation

context Customer **inv**:
age <= 18 **implies**
accounts->forAll(a | a.oclIsKindOf(SavingsAccount))

$\forall a \in accounts$: a.oclIsKindOf(Savingsaccount)

Example: Composite Pattern



- A composite is the parent of its components

context Composite **inv**:
children->forAll(c | c.parent = **self**)

- A component is contained in its parent composite

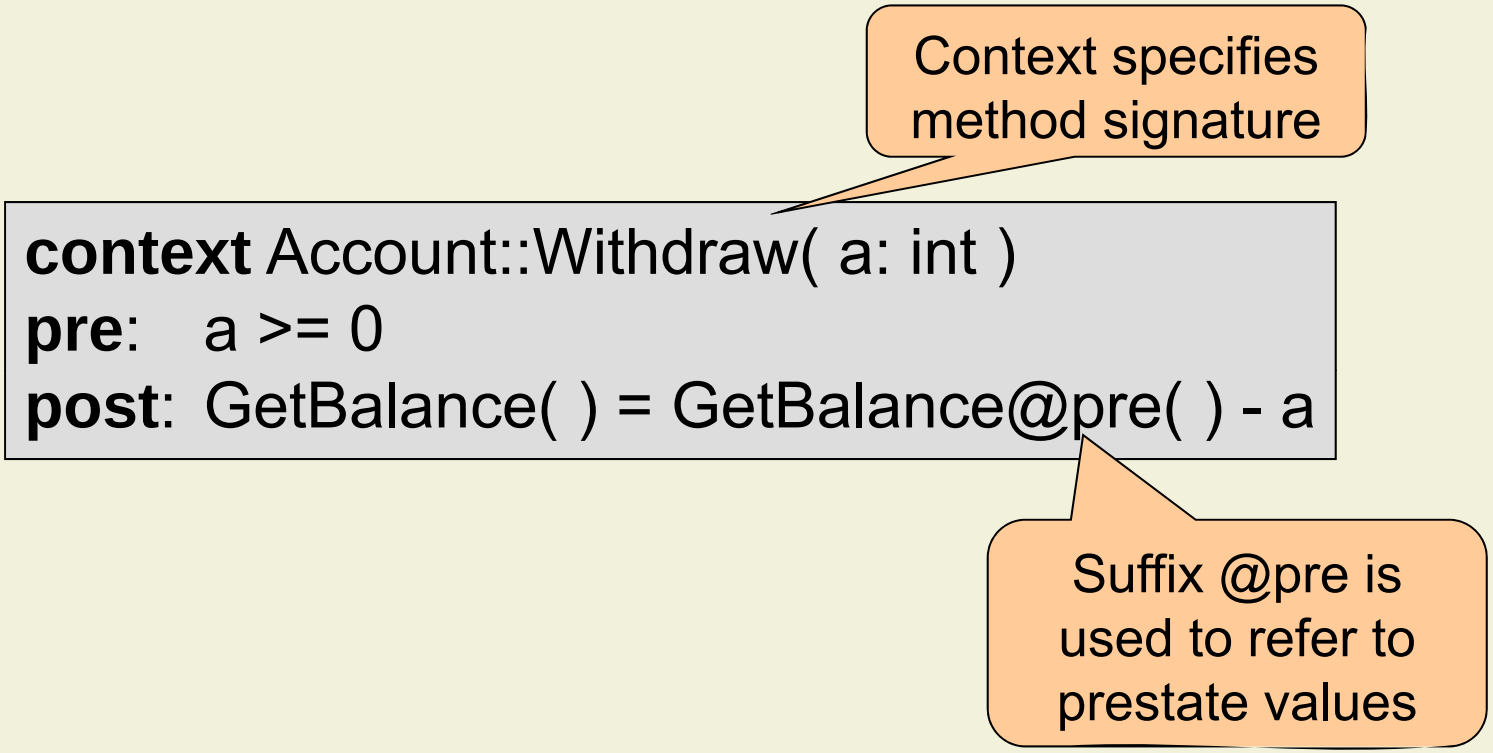
context Component **inv**:
parent->size( ) = 1 **implies**
parent.children->includes(**self**)

Contracts in Eiffel: Method Specifications

- Method precondition
  - Must be true before the method is executed
- Method postcondition
  - Must be true after the method terminates
  - old expressions is used to refer to values of the pre-state

```
class interface ACCOUNT feature

  withdraw ( a: INTEGER ) is
    require a >= 0
    ensure GetBalance( ) = old( GetBalance( ) – a )

end
```

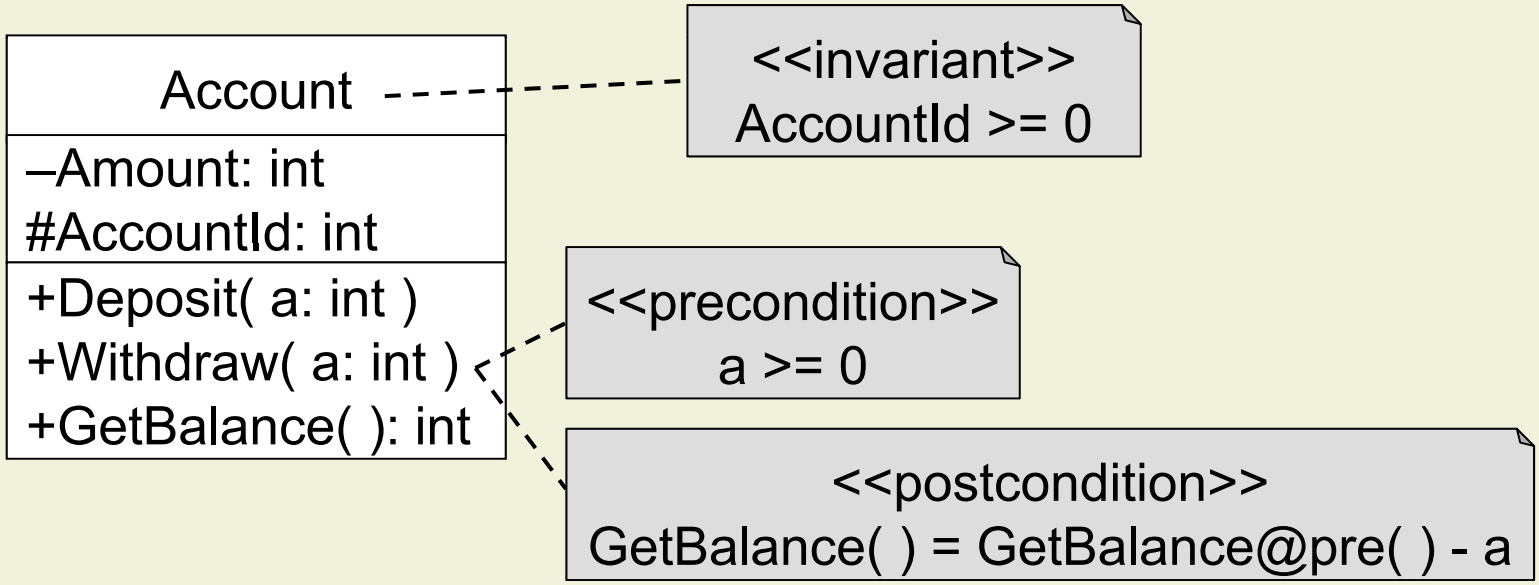
Pre- and Postconditions in OCL



- **result** is used to refer to return value
- Pre- and postconditions can be named (like in Eiffel)

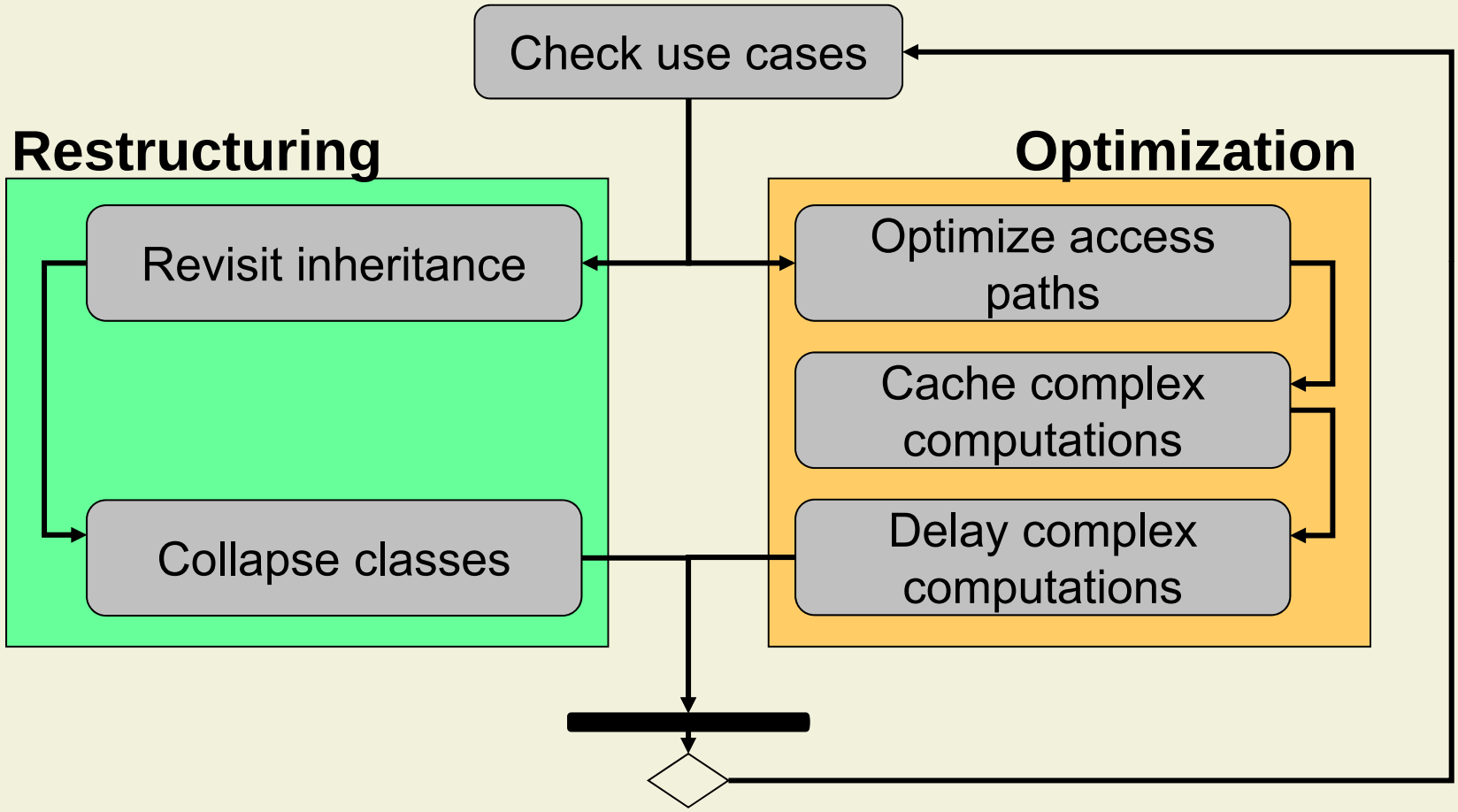
Alternative Notation

- Contracts can be depicted as notes in diagrams
  - Stereotypes instead of keywords **inv**, **pre**, **post**



5. Detailed Design

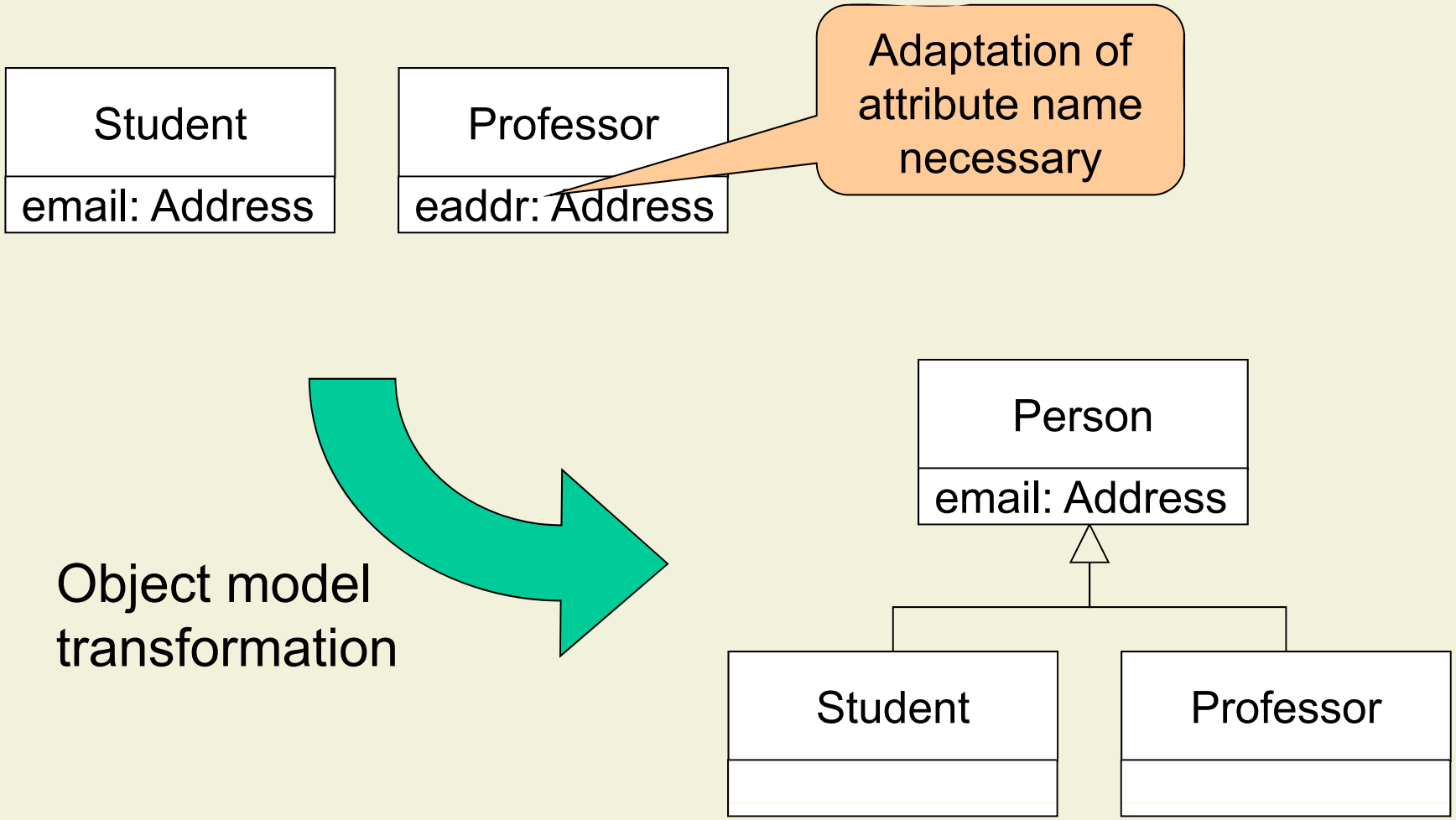
Object Model Restructuring and Optimization



Increasing Inheritance

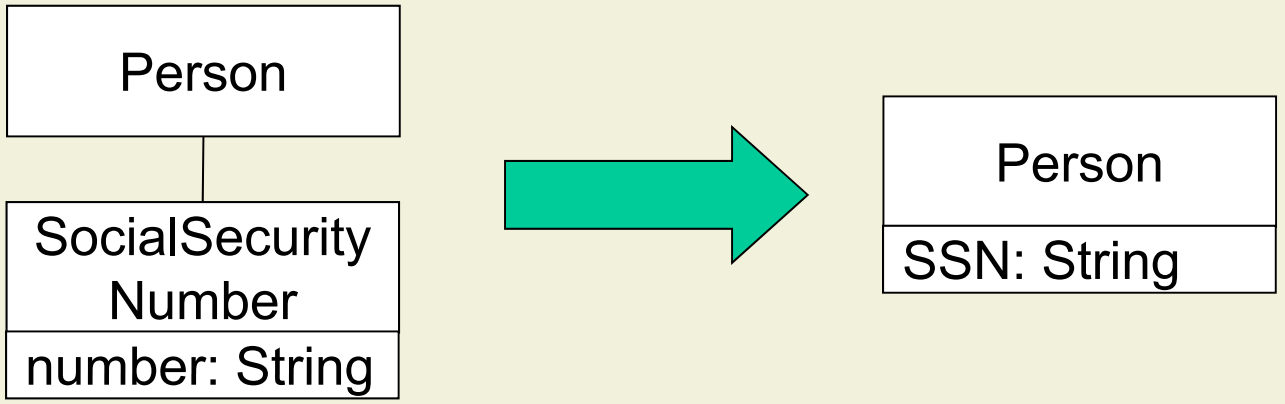
- **Rearrange** and **adjust** classes and operations to prepare for inheritance
  - Generalization
  - Specialization
- **Generalization** is a common modeling activity
  - Abstracts common behavior out of a group of classes
  - If operations or attributes are repeated in 2 classes the classes might be instances of a more general class
- Superclasses are desirable
  - Increase of modularity, extensibility, and reusability

Increasing Inheritance: Example



Collapsing Classes

- Collapse a **class** without interesting behavior **into an attribute**
  - If the only operations defined on the attributes are Set( ) and Get( )



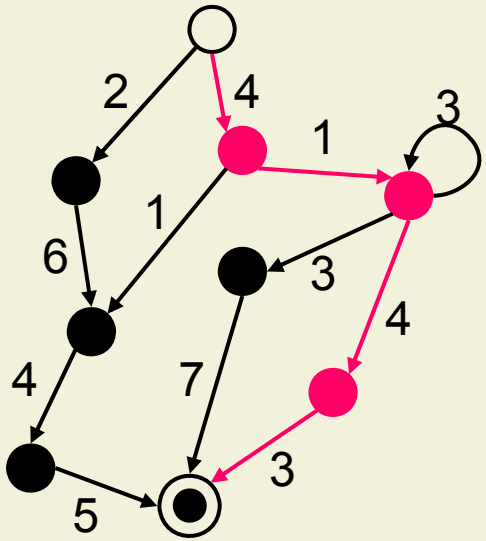
- Associations are more flexible than attributes but often introduce unnecessary indirection

Optimizing Access Paths

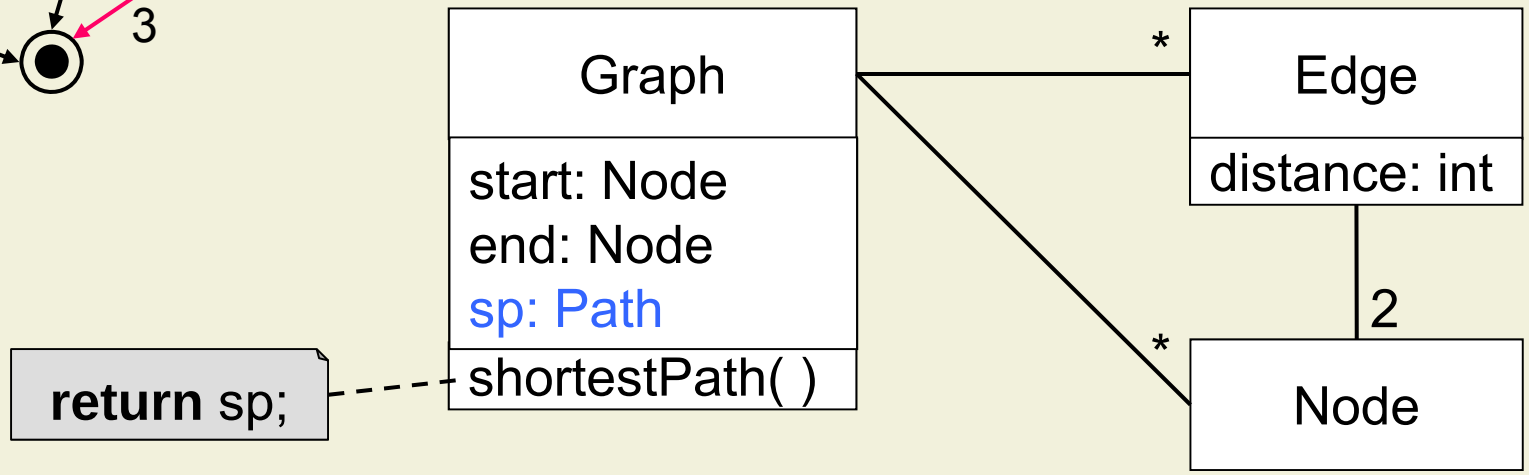
- Add **redundant associations** to minimize access cost
  - What are the most frequent operations?
  - How often is the operation called? (30 times a month, every 50 ms)

- Turn classes into attributes (collapse classes)

Caching Complex Computations



- ShortestPath is an expensive operation
- Result can be cached



Keeping Caches Up-to-Date: Eager Update

- Operations that change the state of the data structure **update the cache**
- Possible if cache update is cheap or state changes are rare
- Also called push solution

```
void addEdge( Node n, Node m ) {
  // add (n,m) to edges
  sp = computeShortestPath( );
}

Path shortestPath( ) {
  return sp;
}
```

Keeping Caches Up-to-Date: Lazy Update

- Operations that change the state of the data structure **increment a version counter** or set a flag
- Access to cached value updates cache if cache is outdated
- Also called pull solution

```
void addEdge( Node n, Node m ) {
  // add (n,m) to edges
  sp = null;  // invalidate cache
}

Path shortestPath( ) {
  if ( sp == null )
    sp = computeShortestPath( );
  return sp;
}
```

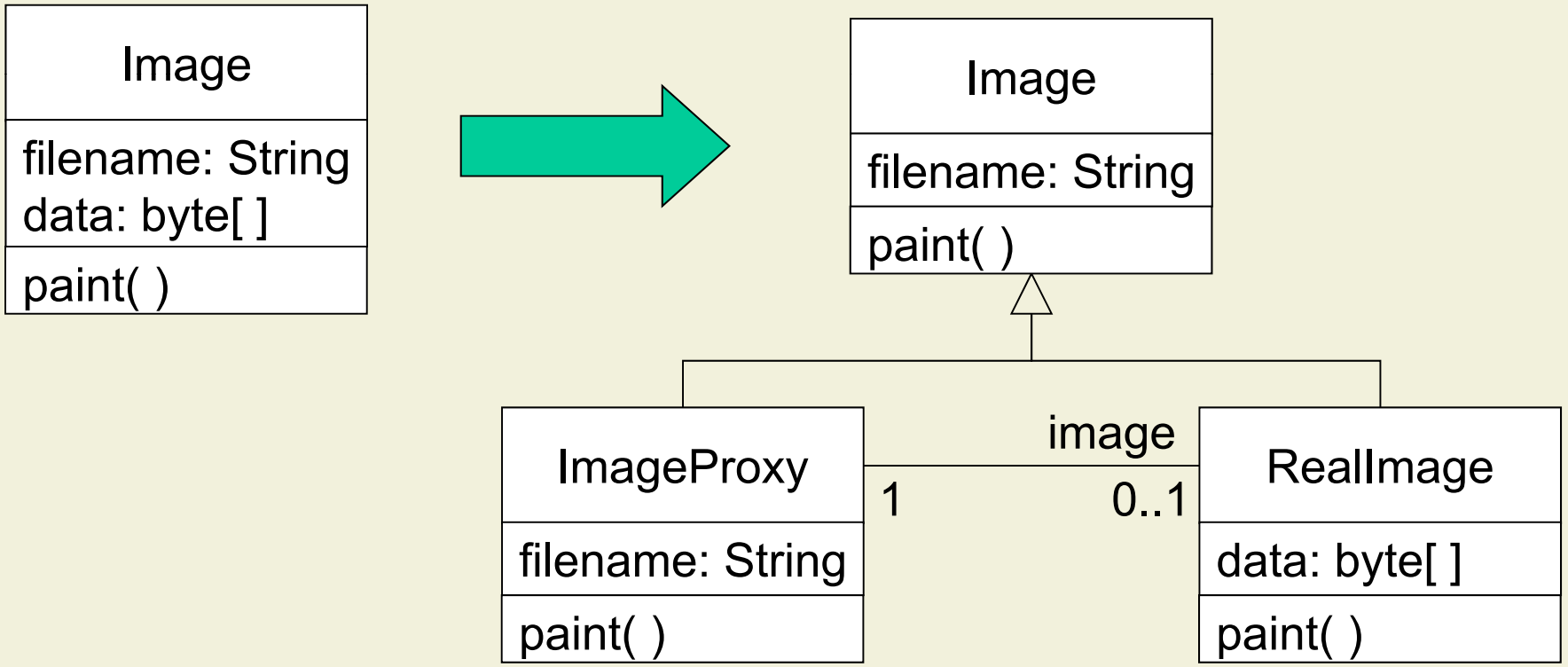
Keeping Caches Up-to-Date: Active Values

- Observer pattern
  - Active value is subject
  - Cache is observer
- Operations that change the state of the data structure **trigger an event** (notify)
- Cache can be updated eagerly or lazily

```
void addEdge( Node n, Node m ) {
  // add (n,m) to edges
  notify( );  // trigger event
}

void update( ) {  // eager update
  sp = computeShortestPath( );
}

Path shortestPath( ) {
  return sp;
}
```

Delaying Complex Computations

- Computation is delayed until result is accessed
- Example: lazy object initialization



Design Optimizations: Summary

- Design optimizations are an important part of the detailed design phase
  - The requirements analysis model is semantically correct but often too inefficient if directly implemented
  - Strike a balance between efficiency and clarity