Chapter 9, Object Design I
Fundamentals and Design Pattern (first pass)

Object-**O**riented
Software**C**onstruction

Armin B. Cremers

Tobias Rho, Daniel Speicher and Holger Mügge
(based on Bruegge & Dutoit)

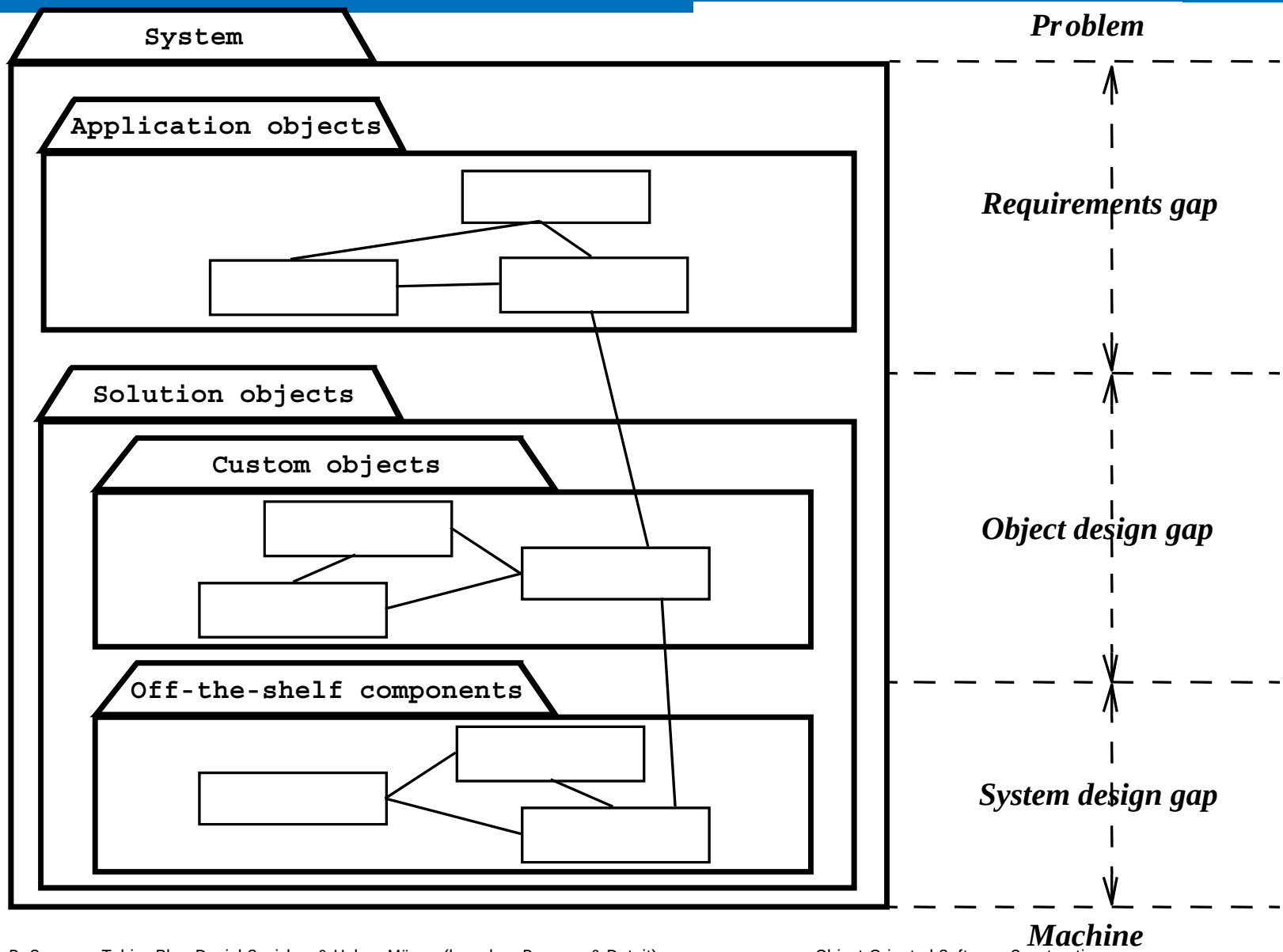
b-it

Object Design

- Process of
  - adding details to the requirements analysis model and
  - making implementation decisions

- Requirements Analysis: Use cases, functional and dynamic model deliver rather coarse-grained operations for object model
- Object Design: Iterates on the models, in particular the object model and refine the models
- Object Design serves as the basis of implementation

Object Design: Closing the Gap

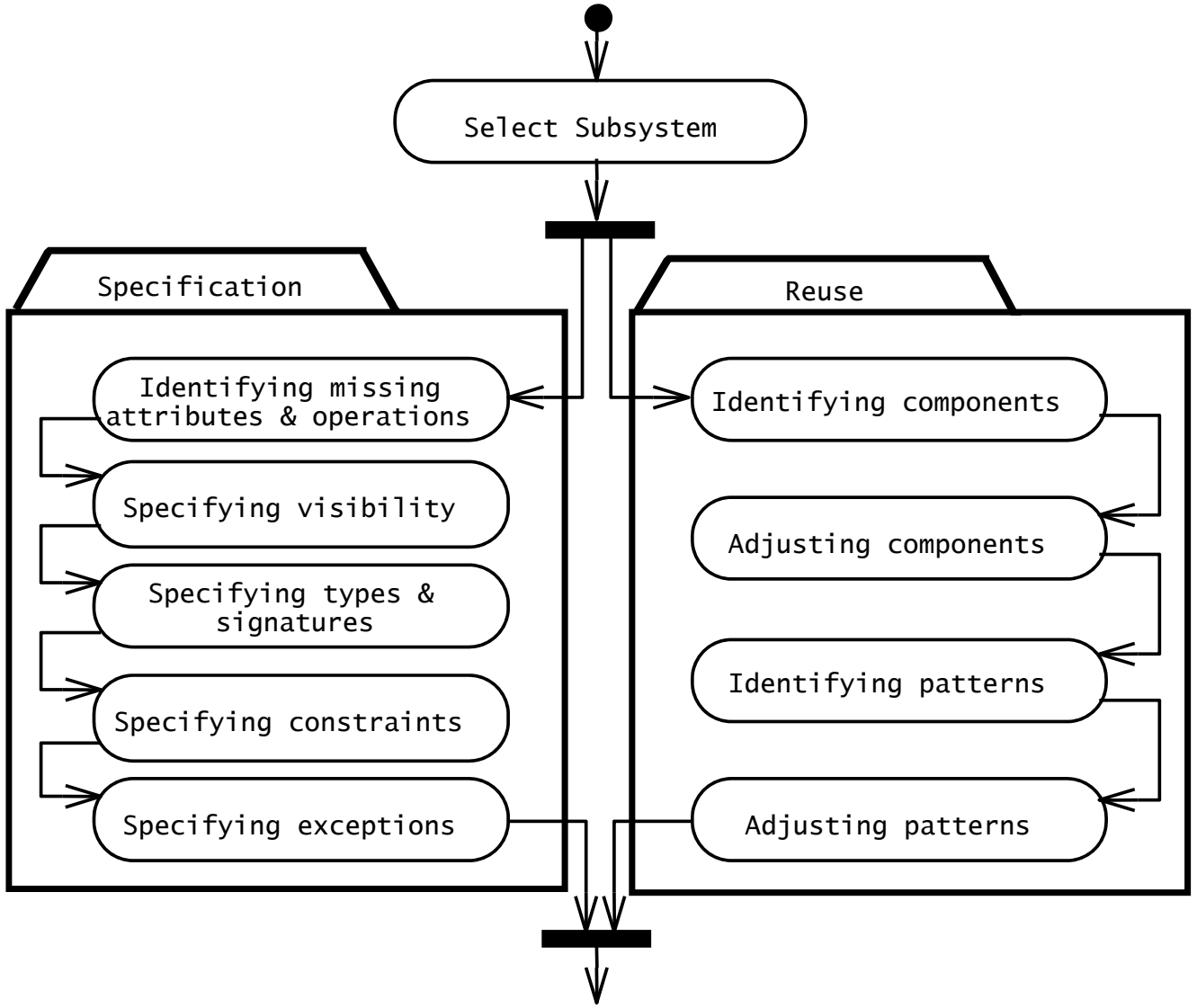


Typical Object Design Activities

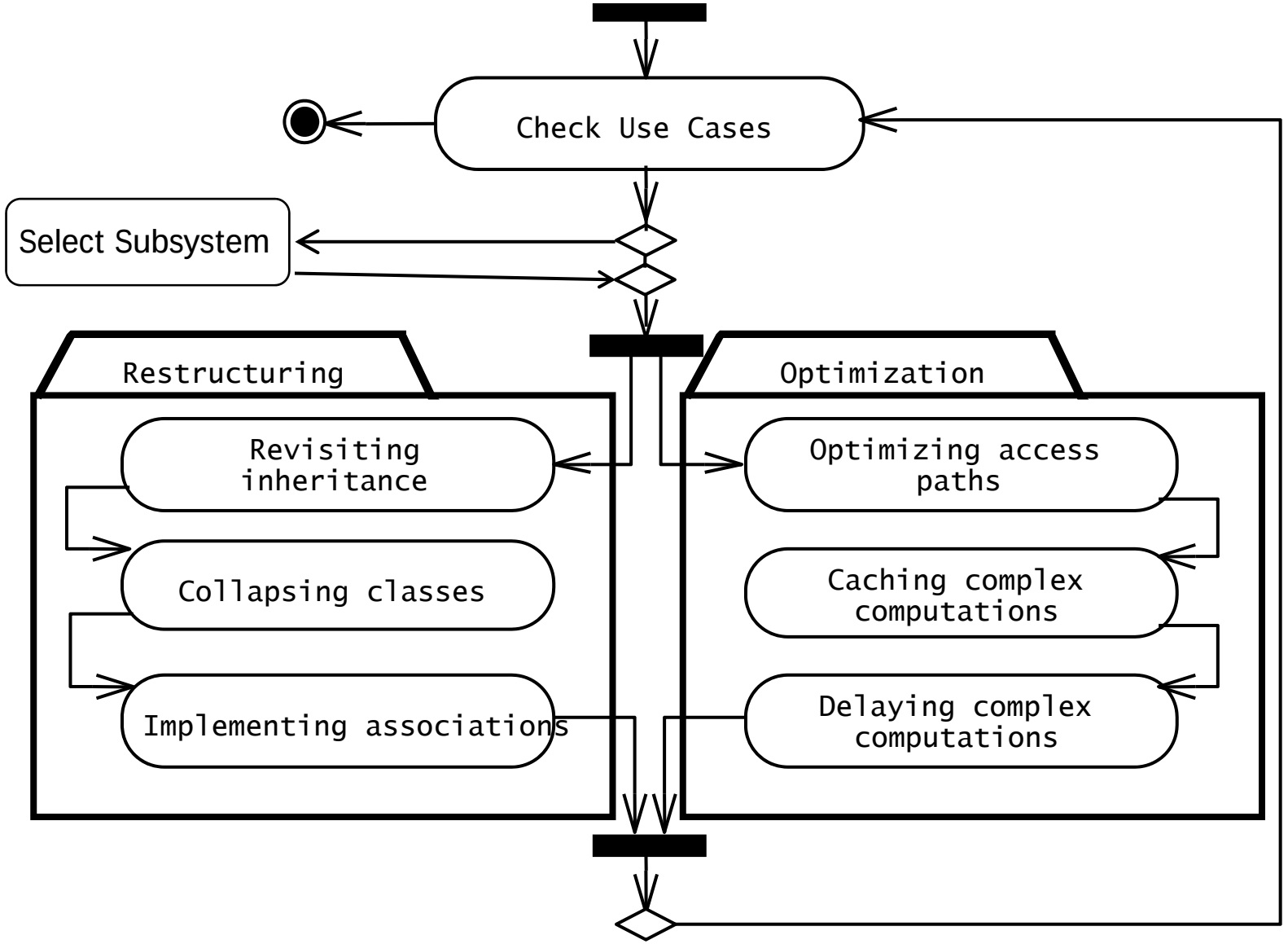
- Service Specification
  - Describe class interfaces precisely
- Reuse Activities
  - Identify off-the-shelf components, design patterns and frameworks to make use of existing solutions
- Object Model Restructuring
  - Adaptation of object model to improve understandability and extensibility
- Object Model Optimization
  - Adapt object model to address performance issues

  - **=> How are these activities put into practice?**

A More Detailed View of Object Design Activities



Detailed View of Object Design Activities



Plan for the next Lectures

1. Reuse: Identification of existing solutions
   - Use of inheritance
   - Off-the-shelf components and additional solution objects
   - Design patterns
2. Interface specification
   - Describes precisely each class interface
3. Object model restructuring
   - Transforms the object design model to improve its understandability and extensibility
4. Object model optimization
   - Transforms the object design model to address performance criteria such as response time or memory utilization.

Object Design lectures

Mapping Models to Code lecture

Outline of today's and next lecture(s)

Object Design I: Fundamentals and Design Patterns (first pass)

- Reuse Concepts
- Types of Inheritance
- Delegation as an alternative way for reuse
- Design Patterns (first pass)

Object Design II: Design Pattern (second pass)

Object Design III: Interface Specification

Reuse-based software engineering

- Need for reusable and flexible designs
  - Achieve better software, more quickly and at lower cost
- Application system reuse
  - An application system may be reused either by
    - incorporating it without change into other systems (COTS reuse) or
    - by configuring the application for different clients
- Component and class reuse
  - Components of an application from *sub-systems* to *classes* may be reused.
- Function/Operation reuse
  - A single well-defined function may be reused

Reuse on class level

- Look for existing classes in class libraries
- Select data structures appropriate to the algorithms
  - Container classes
  - Arrays, lists, queues, stacks, sets, trees, ...
- Adjust the class (libraries)
  - Change the API if you have the source code.
  - Use *adapter objects* if you don't have access

- Object-oriented Reuse on a class-level is often too fine-grain and too specialized
- Object-oriented Reuse is best supported through larger-grain abstractions (→ frameworks)

Frameworks

- Reusable partial application that can be specialized to produce custom applications
- Targeted to
  - particular technologies, such as data processing or cellular communications, or
  - to application domains, such as user interfaces or real-time applications.
- How to use a framework:
  - **Hook methods** provide access to framework
    - Can be used or overridden by an application
  - **Call back methods**
    - Must be defined by applications
    - Are called in response to events recognized by the framework
- Frameworks can also be classified by the techniques used to extend them.
  - Whitebox frameworks (providing hook methods)
  - Blackbox frameworks (specifying call back methods)

Concept Reuse

- The reuse of program or design components often imply design decisions made by the original developer of the component.

  → Limit the opportunities for reuse.

- A more abstract form of reuse is concept reuse:
  - Described a particular approach in an implementation independent way
  - The implementation is then developed based on this way

- The main approach to design concept reuse are **Design patterns** ...

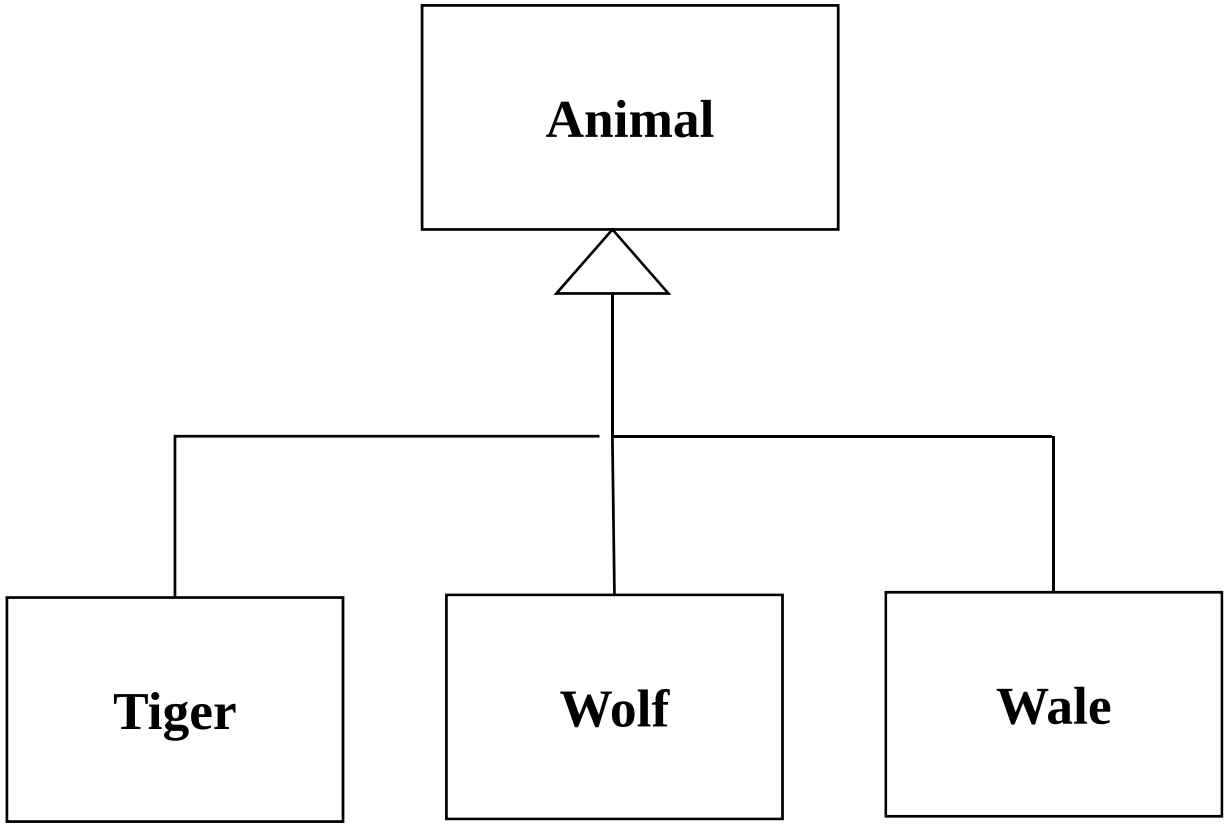
The use of inheritance

- Inheritance is used to achieve two different goals
  - Description of Taxonomies
  - Reuse of classes

- Description of taxonomies
  - Used during requirements analysis.
  - Activity: identify application domain objects that are hierarchically related
  - Goal: make the analysis model more understandable
  - Inheritance is found either by specialization or generalization

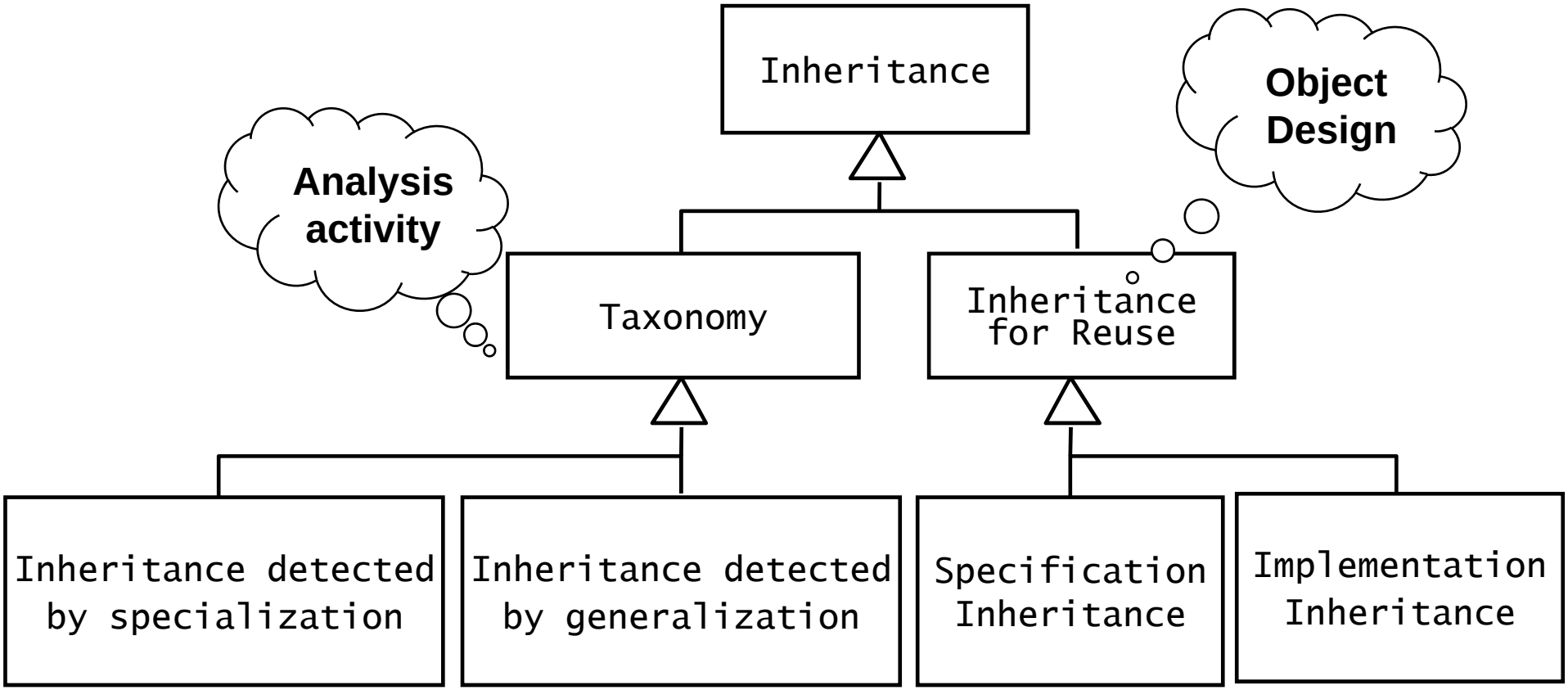
- Reuse of classes
  - Used during object design
  - Activity: factor out redundant behavior to superclass
  - Goal: increase reusability and extensibility, reduce redundancy

Taxonomy Example



Meta model for Inheritance

- Inheritance is used during analysis and object design



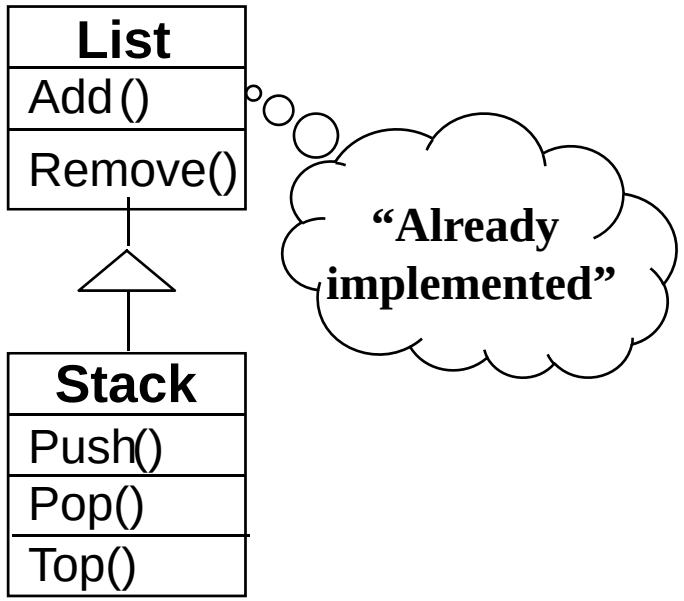
Implementation Inheritance vs. Specification Inheritance

- Implementation inheritance
  - Also called class inheritance
  - Goal: Extend an application's functionality by reusing functionality in parent class to avoid inconsistencies
  - Inherit from an existing class with *some* or all operations already implemented
- Specification inheritance
  - Also called subtyping
  - Inherit from an abstract class with *all* operations specified, but not yet implemented
  - Override existing functionality (methods...)

Implementation Inheritance

- Implementation Inheritance is a generalization technique, in which the behavior of a superclass is shared by all its subclasses. Sometimes it is misused as an implementation technique.
- A very similar class is already implemented that does almost the same as the desired class implementation.

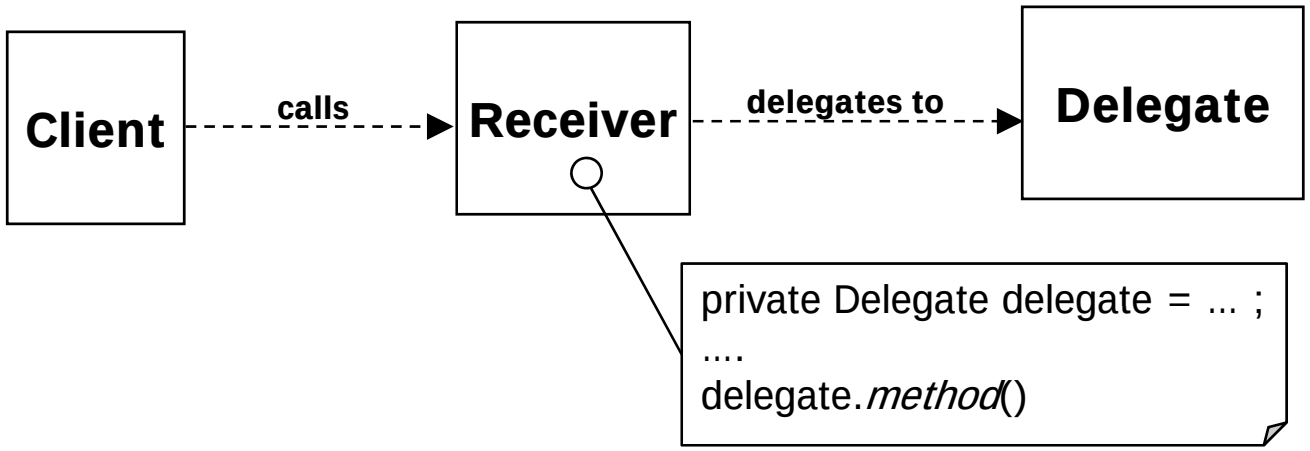
  - Example: I have a **List** class, I need a **Stack** class. How about subclassing the **Stack** class from the **List** class and providing three methods, **Push()** and **Pop(), Top()**?



- Problem: What happens if the Stack user calls Remove() instead of Pop()?

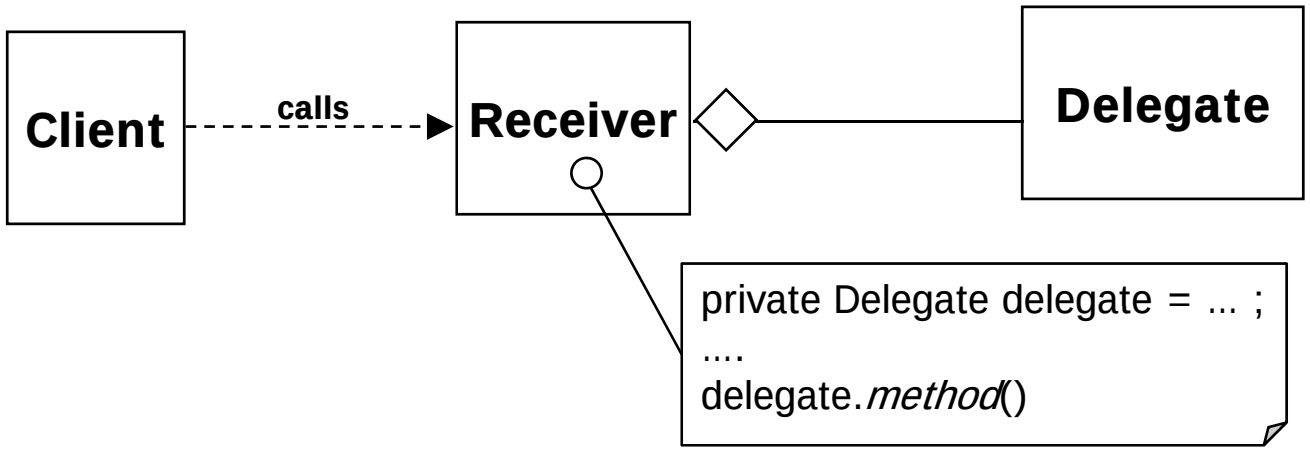
Delegation as alternative to Implementation Inheritance

- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- In Delegation, two objects are involved in handling a request
  - A **receiving object** delegates operations to its **delegate**.
  - **Advantage**: The developer can make sure that the receiving object does not allow the client to misuse the delegate object
- Modeling Delegation (use dependency):



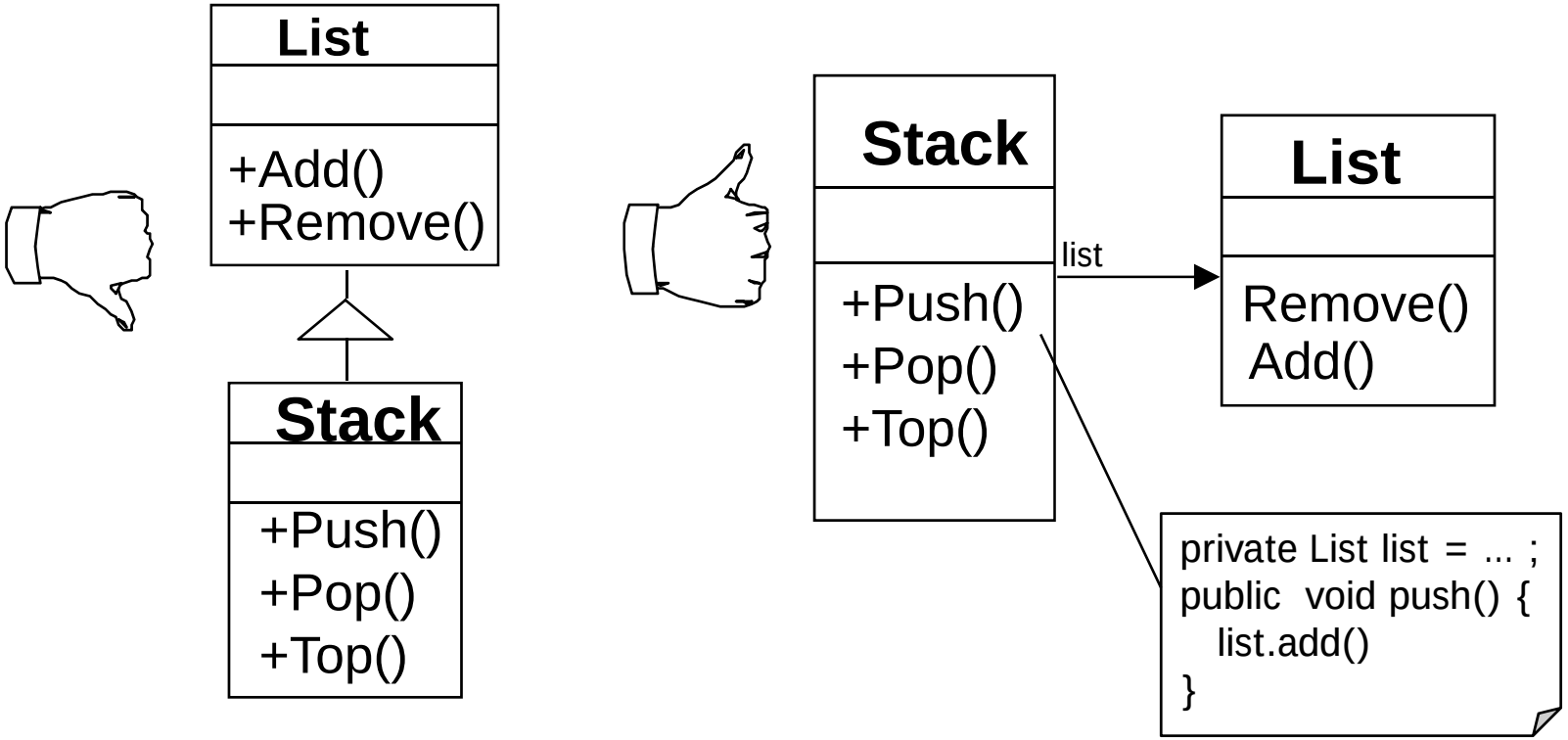
Delegation as alternative to Implementation Inheritance

- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- In Delegation, two objects are involved in handling a request
  - A **receiving object** delegates operations to its **delegate**.
  - **Advantage**: The developer can make sure that the receiving object does not allow the client to misuse the delegate object
- Modeling Delegation (as aggregation):



Delegation instead of Implementation Inheritance

- Inheritance: Extending a Base class by a new operation or overwriting an operation.
- Delegation: Catching an operation and sending it to another object.
- Which of the following models is better for implementing a stack?



Comparison: Delegation vs. Implementation Inheritance

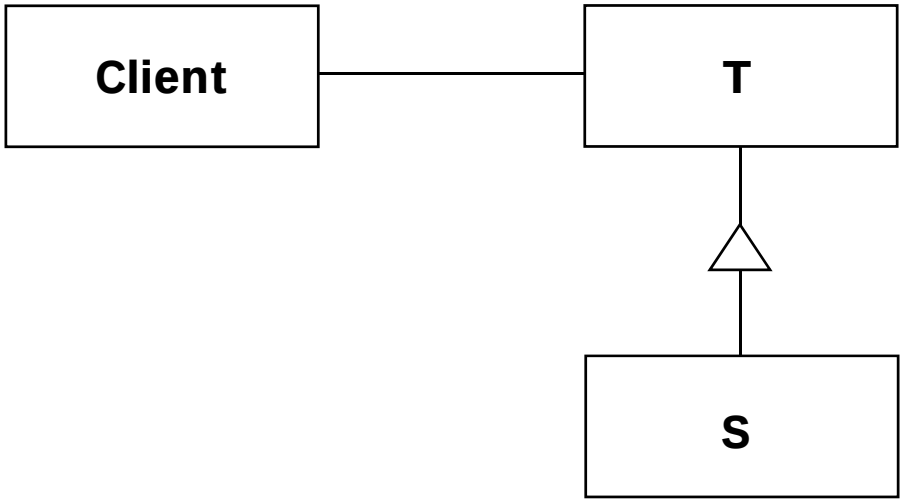
- Delegation
  - Pro:
    - Flexibility: Any delegate object can be replaced at run time by another one (as long as it has the same type)
    - Details of the delegate can be hidden
    - Low coupling from client to delegate class
  - Con:
    - Inefficiency: Many objects available, flat hierarchy
    - Highly parameterized → difficult to understand, risk of having inconsistencies in the architecture

Comparison: Delegation vs. Implementation Inheritance

- Inheritance
  - Pro:
    - Straightforward to use, supported by many languages
    - Static specification of a hierarchy at *compile time* ensures consistent architecture
    - Easy to implement (override) new functionality
    - Good for creating and maintaining different implementations according to a common interface
  - Con:
    - Inheritance exposes (undesired) details of the parent class
    - High coupling from client to superclass possible
    - Inheritance hierarchy cannot be change *at runtime*
    - Substitution of the sub classes for a super class in any of the client code possible
      → may lead to unforeseeable behavior, if some methods do not provide the service as *expected* by the client

The Liskov Substitution Principle

- "If an object of type *S* can be substituted in all places where an object of Type *T* is *expected*, then *S* is a subtype of *T*"



- Also formal definition for Specification Inheritance
- Client do not has to change its code, if object T is substituted by S
- More information in Object Design III

Outline of today's lecture

Object Design I: Fundamentals and Design Patterns (first pass)

- Reuse Concepts
- Types of Inheritance
- Delegation as an alternative way for reuse
- **Design Patterns (first pass)**

Design Patterns
An approach to realize flexible software

- A design pattern is a *template* solution that developers have refined over time to solve a range of recurring problems [Gamma et al. 1995]
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Make software modifiable and extensible to minimize the cost of future changes
- Patterns merely rely on object characteristics such as:
  - inheritance
  - delegation
  - dynamic binding.

Classes of Design Patterns

- Creational
  - Concern the process of object creation
  - Examples: **Abstract Factory**, Builder, Prototype, Singleton
- Structural
  - Deal with the composition of classes of objects towards higher level constructs
  - Examples: **Adapter**, **Bridge**, **Composite**, Decorator, **Façade**, Flyweight, Proxy
- Behavioral
  - Characterize the ways in which classes or objects interact and distribute responsibility
  - Examples: Command, Iterator, Chain of Responsilibility, Mediator, Observer, State, **Strategy**, Visitor

Ingredients of a Design Pattern
(according to [Gamma et al.], extract)

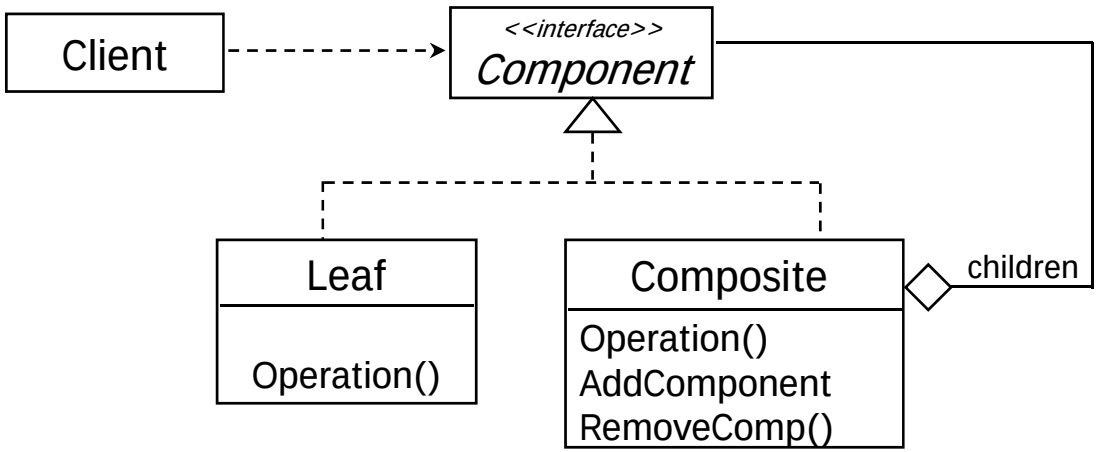
- Name
  - A meaningful pattern identifier.
- Intent
  - What does the pattern do? What is its rationale?
- Motivation (Problem Description)
  - A scenario that illustrates a design problem and how the class structures in the pattern solve the problem
- Applicability
  - What are situations in which the pattern can be applied?
- Structure (Solution)
  - Description of the pattern by means of standardized representation techniques (class diagrams, UML...)
- Consequences
  - The results and trade-offs of applying the pattern.
- Sample Code

Introducing the Composite Pattern

- Models tree structures that represent part-whole hierarchies with arbitrary depth and width.
- The Composite Pattern lets client treat individual objects and compositions of these objects uniformly

Composite Pattern (1/2): Representing Recursive Hierarchies

Name	Composite Design Pattern
Problem Description	Represent a hierarchy of variable width and depth so that leaves and composites can be treated uniformly through a common interface.
Solution	The Component interface specifies the services that are shared among Leaf and Composite. A Composite has an aggregation association with Components and implements each service by iterating over each contained Component. The Leaf services do the actual work.



Composite Pattern (2/2): Representing Recursive Hierarchies

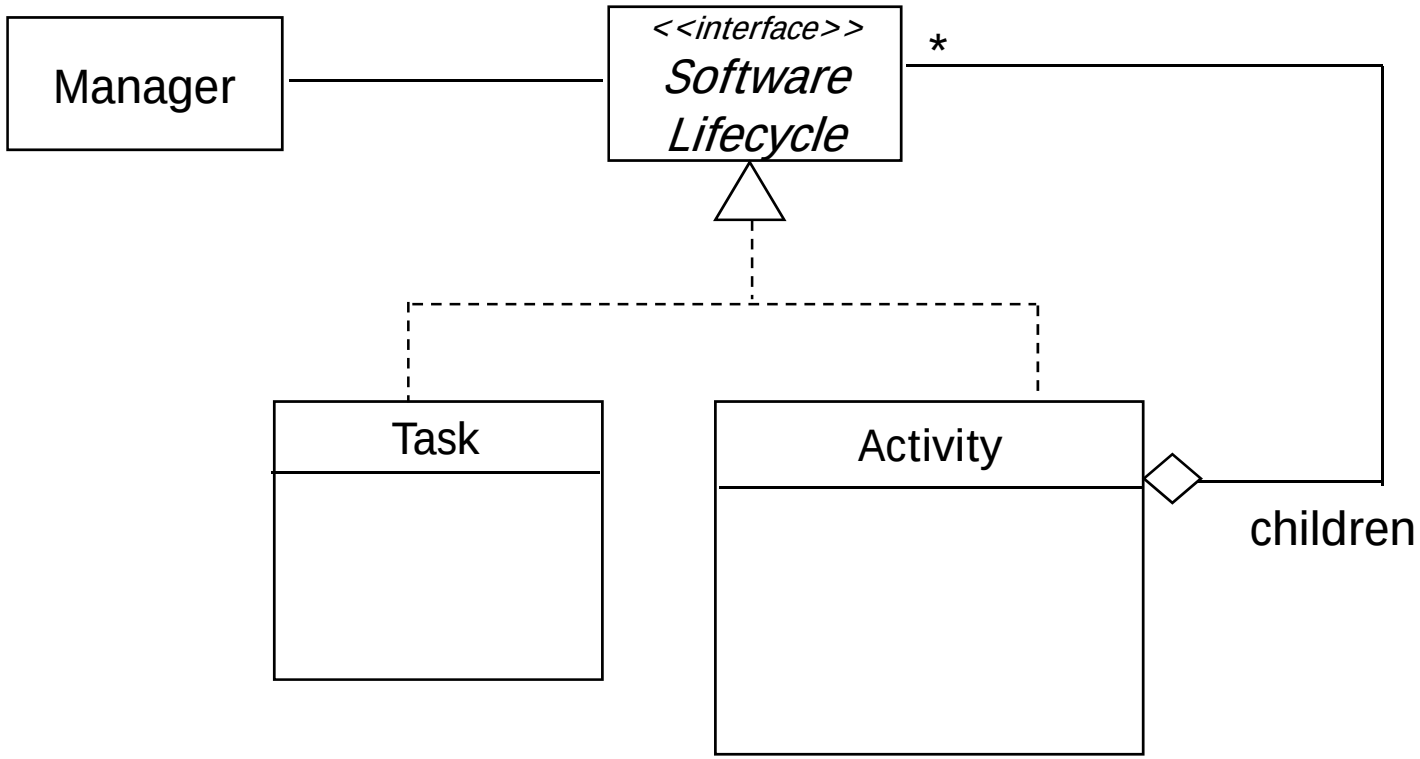
....	
Consequences	• Client uses the same code for dealing with Leaves and Composites • Leaf-specific behavior can be modified without changing the hierarchy. • New classes of leaves can be added without changing the hierarchy
Examples	• Groups of drawable elements • Hierarchy of files and directories • Subsystem decomposition • Describing hierarchies of activities

Application of Composite Pattern

- Software System:
  - Definition: A software system consists of subsystems which are either other subsystems or collection of classes
  - Composite: Subsystem (A software system consists of subsystems which consists of subsystems , which consists of subsystems, which...)
  - Leaf node: Class

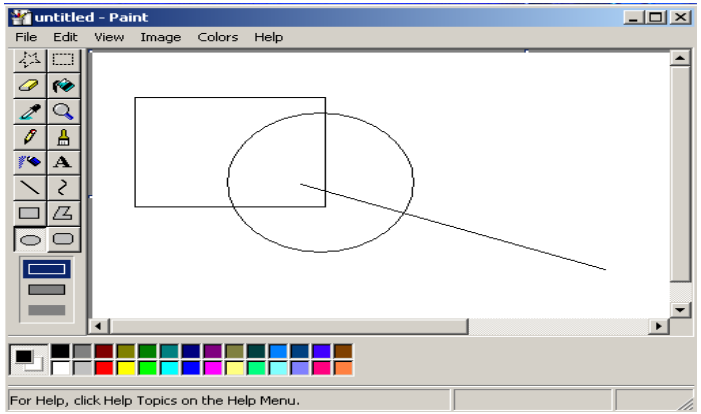
- Software Lifecycle:
  - Definition: The software lifecycle consists of a set of development activities which are either other activities or collection of tasks
  - Composite: Activity (The software lifecycle consists of activities which consist of activities, which consist of activities, which....)
  - Leaf node: Task

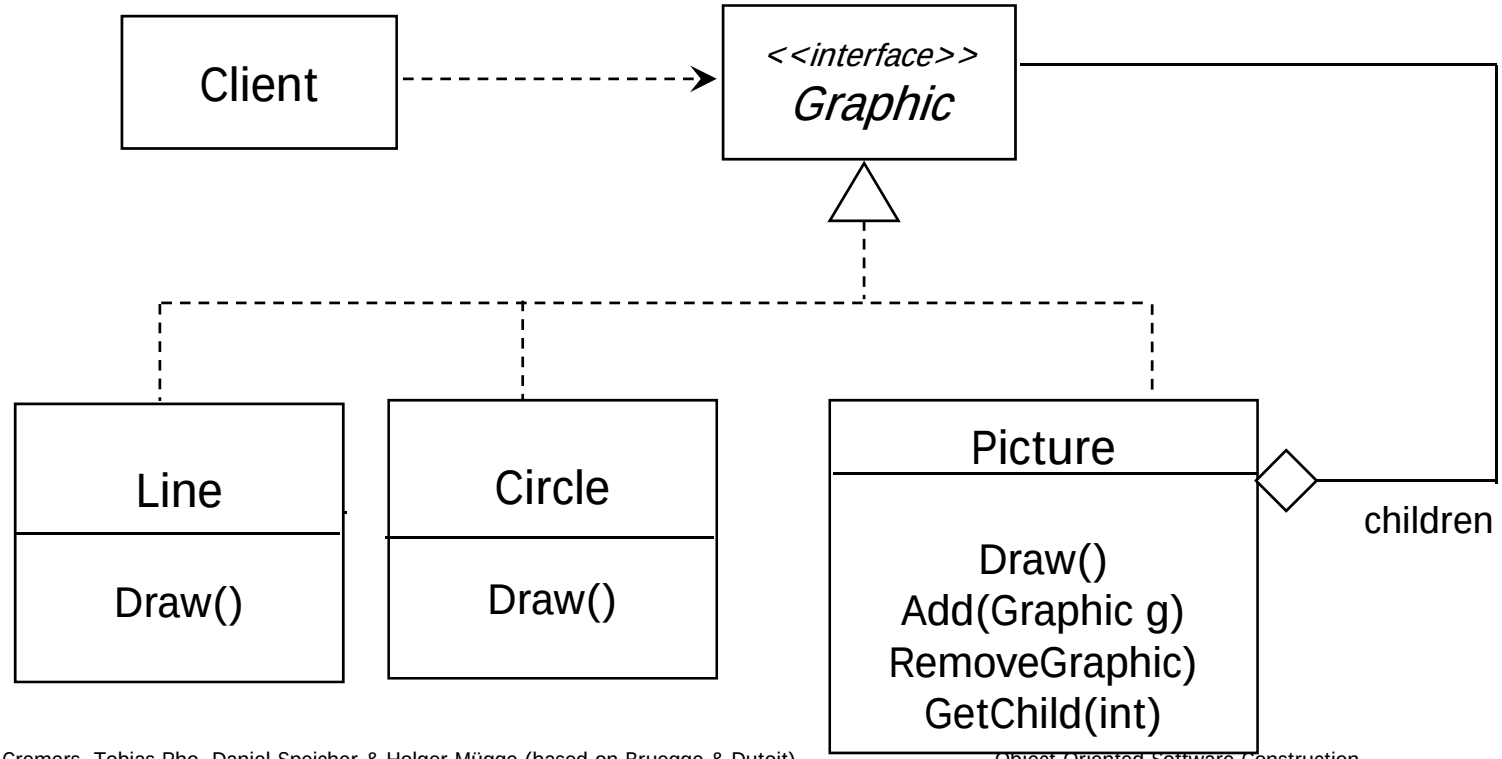
Modeling the Software Lifecycle with a Composite Pattern



Graphic Applications use the Composite Pattern

- The Graphic Class represents both primitives (Line, Circle) and their containers (Picture)



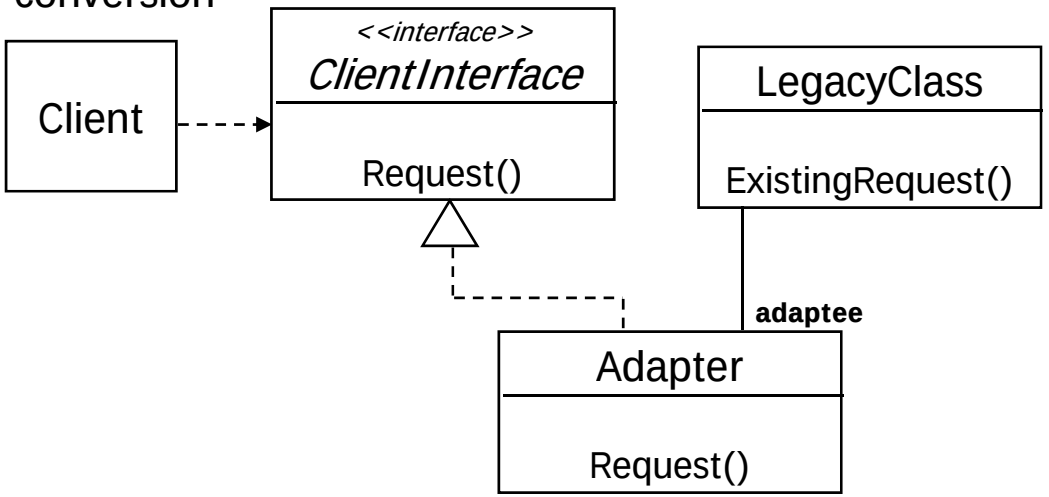


Adapter Pattern

- Applied to integrate existing applications (Legacy Applications, COTS, objects...)
- “also known as” Mediator, Wrapper ...

Adapter Pattern (1/3)
Wrapping around Legacy Code

Name	Adapter Design Pattern
Problem Description	Convert the interface of a legacy class into a different interface expected by the client, so that client and legacy class can work together without changes
Solution	An Adapter class implements the ClientInterface expected by the client. The Adapter delegates requests from the client to the LegacyClass and performs any necessary conversion



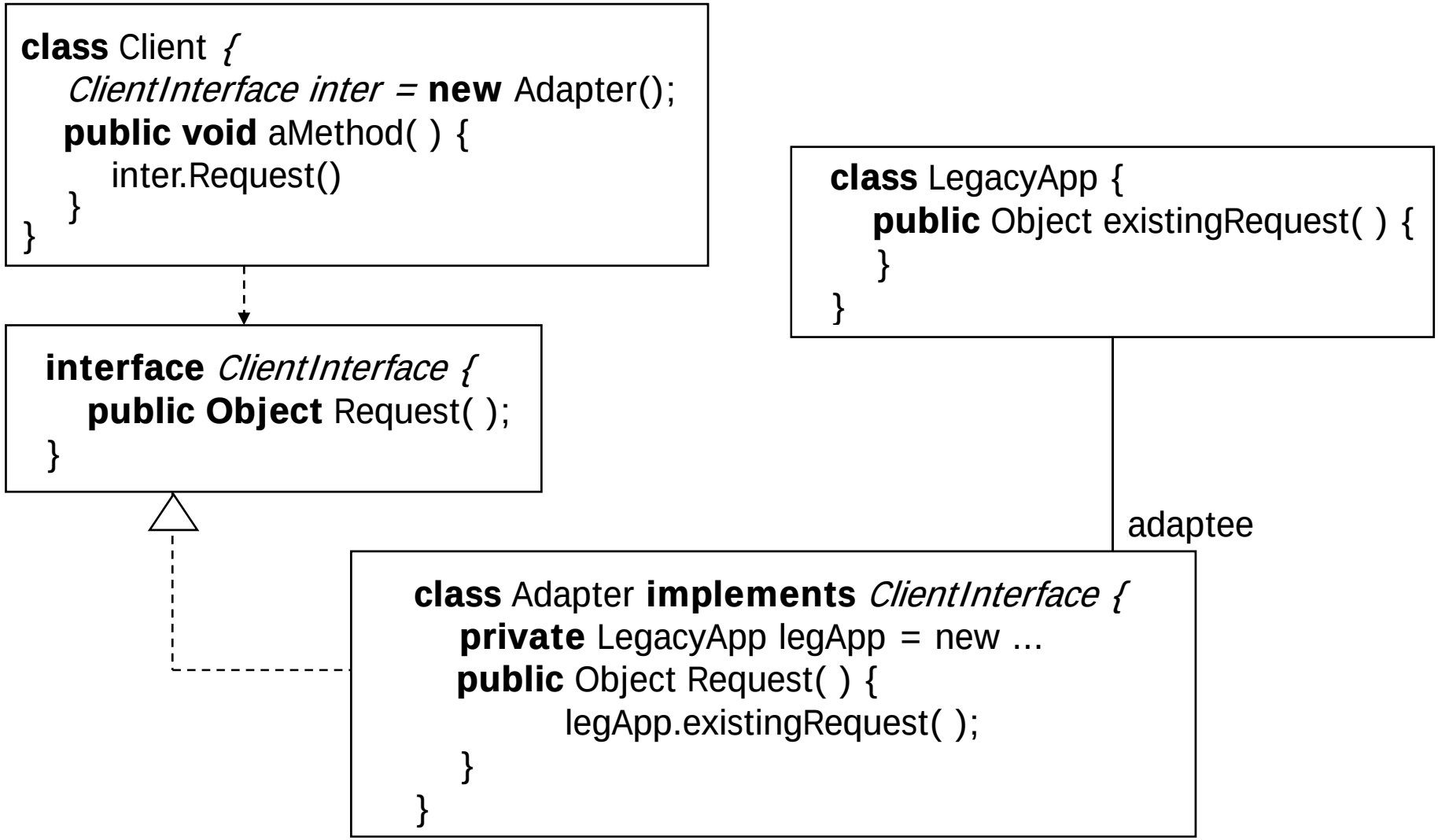
Adapter Pattern (2/3)
Wrapping around Legacy Code

....

Consequences	• Client and LegacyClass work together without modification of neither Client nor LegacyClass • Adapter works with LegacyClass and all of its subclasses. • A new Adapter needs to be written for each specialization (subclass) of ClientInterface

Adapter Pattern (3/3)
Wrapping around Legacy Code

Sample Code



Bridge Pattern

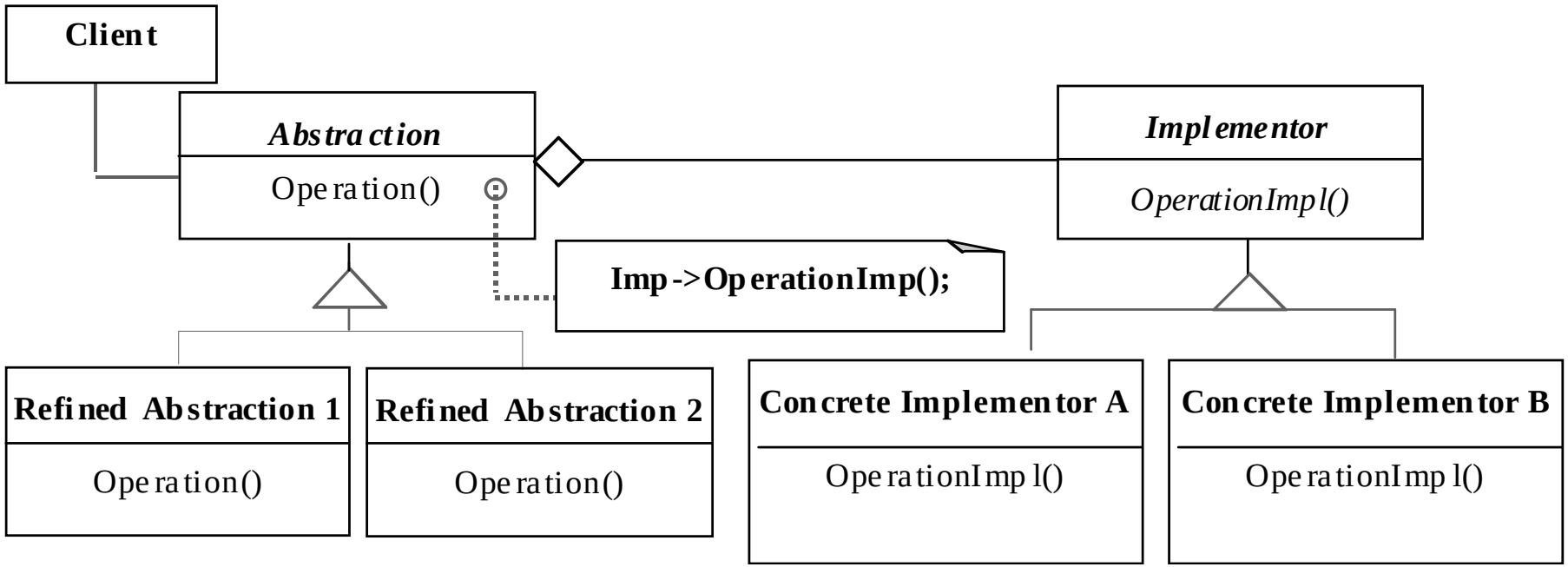
- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently”. (From [Gamma et al 1995])
- Allows different implementations of an interface to be decided upon dynamically.
  - Examples: Interface to a component that is incomplete, not yet known or unavailable during testing (e.g. storage medium)
  - Goal: The abstraction should be exchanged too (e.g. GUI interface)

Using a Bridge
Allowing for alternate Implementations

Name	Bridge Design Pattern
Problem Description	Decouple an interface from an implementation so that implementations can be substituted, possibly at runtime.
Solution	The Abstraction Class defines the interface visible to the client. The Implementor is an abstract class that defines the lower-level methods available to the Abstraction. An Abstraction instance maintains a reference to its corresponding Implementor instance. Both Abstraction and Implementor can be refined independently.

Using a Bridge
Allowing for alternate Implementations

Solution

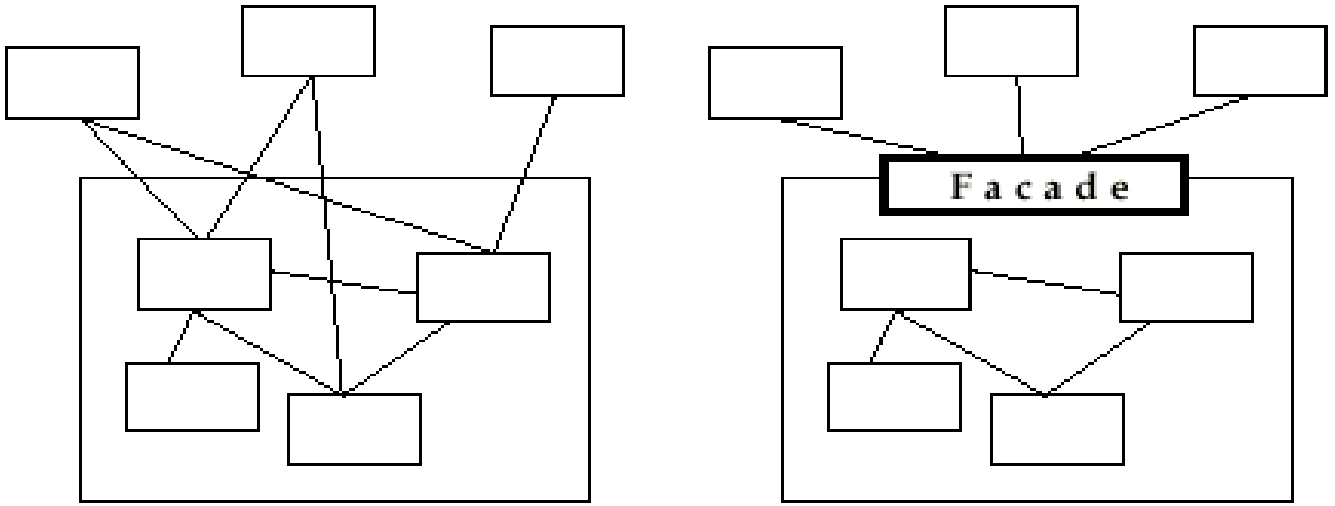


Consequences

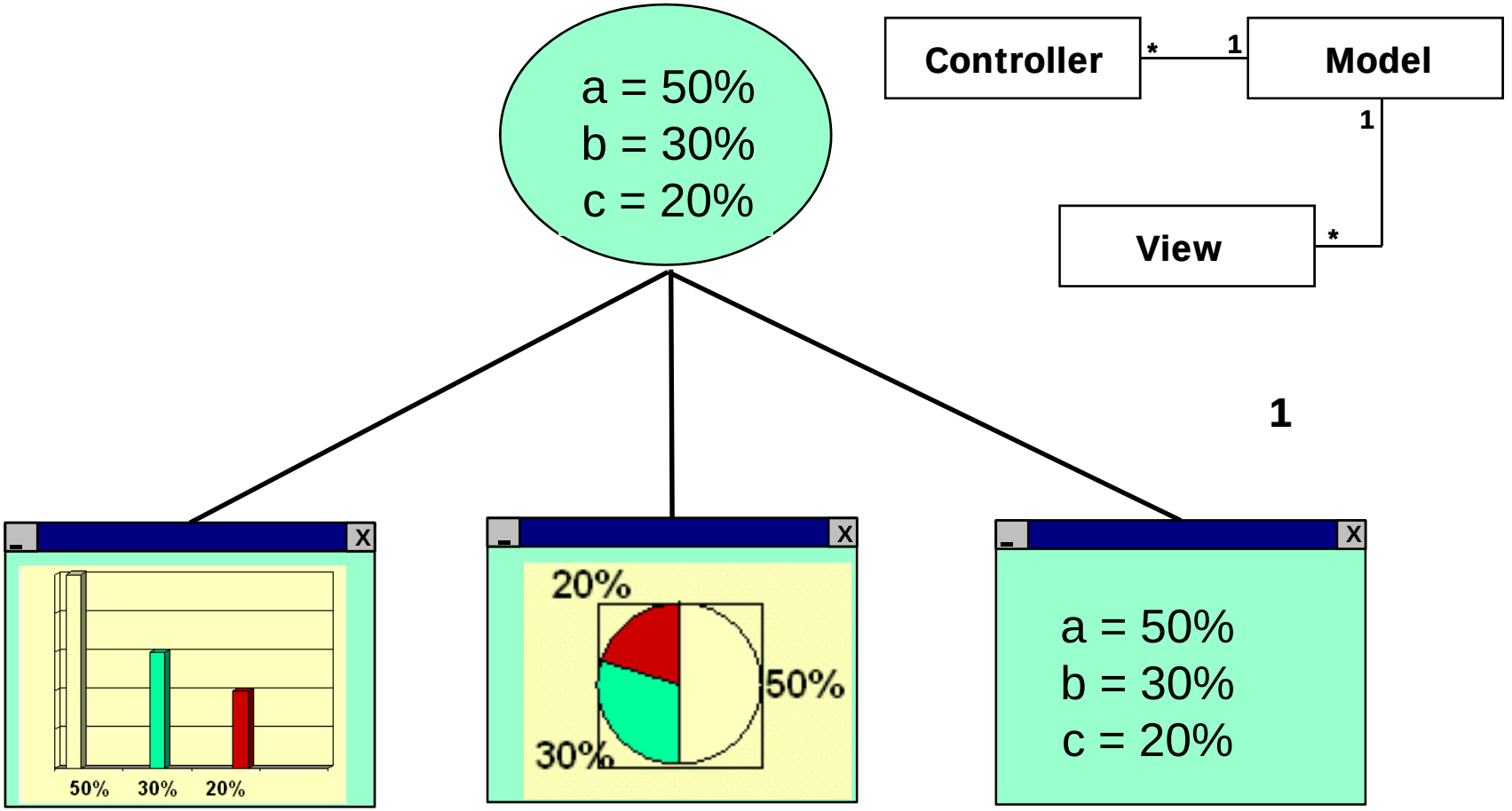
- Client is shielded from abstract and concrete implementations
- Interfaces and Implementations can be refined independently

Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- Facades allow us to provide a closed architecture



Model View Controller



- Multiple views can exist at the same time
- New views can be added without any changes to the model

Selected Patterns and the changes they anticipate

Design Pattern	Anticipated Change
Bridge	**New Vendor, new technology, new implementation:** decoupling the interface from its implementation. Same as adapter, but developer is not constrained by an existing component
Adapter	**New Vendor, new technology, new implementation:** encapsulating a piece of legacy code that was not designed to work with the system
Composite	**New Complexity, new functions, new activities**: encapsulates hierarchies by providing a common superclass for aggregate and leaf nodes. New leafs can be added without modifying existing code.
Model-View-Controller (MVC)	**New Views**. Decouples data models from their actual representation (view). If data is changed, all views are updated consistently. New Views can be added independently from changing any view or the model

Heuristics for identifying common design patterns

- Similar to Abbott's heuristics, key phrases can be used to identify candidate design patterns:

"Must comply with existing interfaces" "Must reuse existing legacy component"	→	Adapter
"Must support future protocols or DBs"	→	Bridge
"Must support aggregate structures" "Must allow for hierarchies of variable depth"	→	Composite
"Must support new views or visualizations"	→	MVC

(Possible) Subsystem Design with Façade, Adapter, Bridge, MVC

- The ideal structure of a subsystem consists of
  - A set of interface (boundary) objects
    - Some of the boundary objects are interfaces to existing systems
  - a set of application domain objects (entity objects) modeling real entities or existing systems
  - one or more control objects

- We can use design patterns to implement this subsystem structure
  - Implementation of the boundary object: Façade, provides the interface to the subsystem
  - Interface to existing or future subsystem: Adapter or Bridge
  - Use MVC for realizing user interface objects
    - Connection to data models (entity objects..)

Summary

- Object design closes the gap between the requirements and the machine.
- Object design is the process of adding details to the requirements analysis and making implementation decisions
- Object design activities include:
  - ✓ Identification of Reuse
  - ✓ Identification of Inheritance and Delegation opportunities
  - ✓ Component selection
  - Interface specification
  - Object model restructuring
  - Object model optimization
- Design patterns are partial solutions to common problems such as
  - separating an interface from a number of alternate implementations
  - wrapping around and selecting between a set of legacy classes
  - protecting a caller from changes associated with specific platforms.