Object-Oriented Software Engineering
Using UML, Patterns, and Java

Object Design I: Reuse

Where are we? What comes next?

- We have covered:
  - Introduction to Software Engineering (Chapter 1)
  - Modeling with UML (Chapter 2)
  - Requirements Elicitation (Chapter 4)
  - Analysis (Chapter 5)
  - System Design (Chapter 6 and 7)
- Today and next class
  - Object Design (Chapter 8).

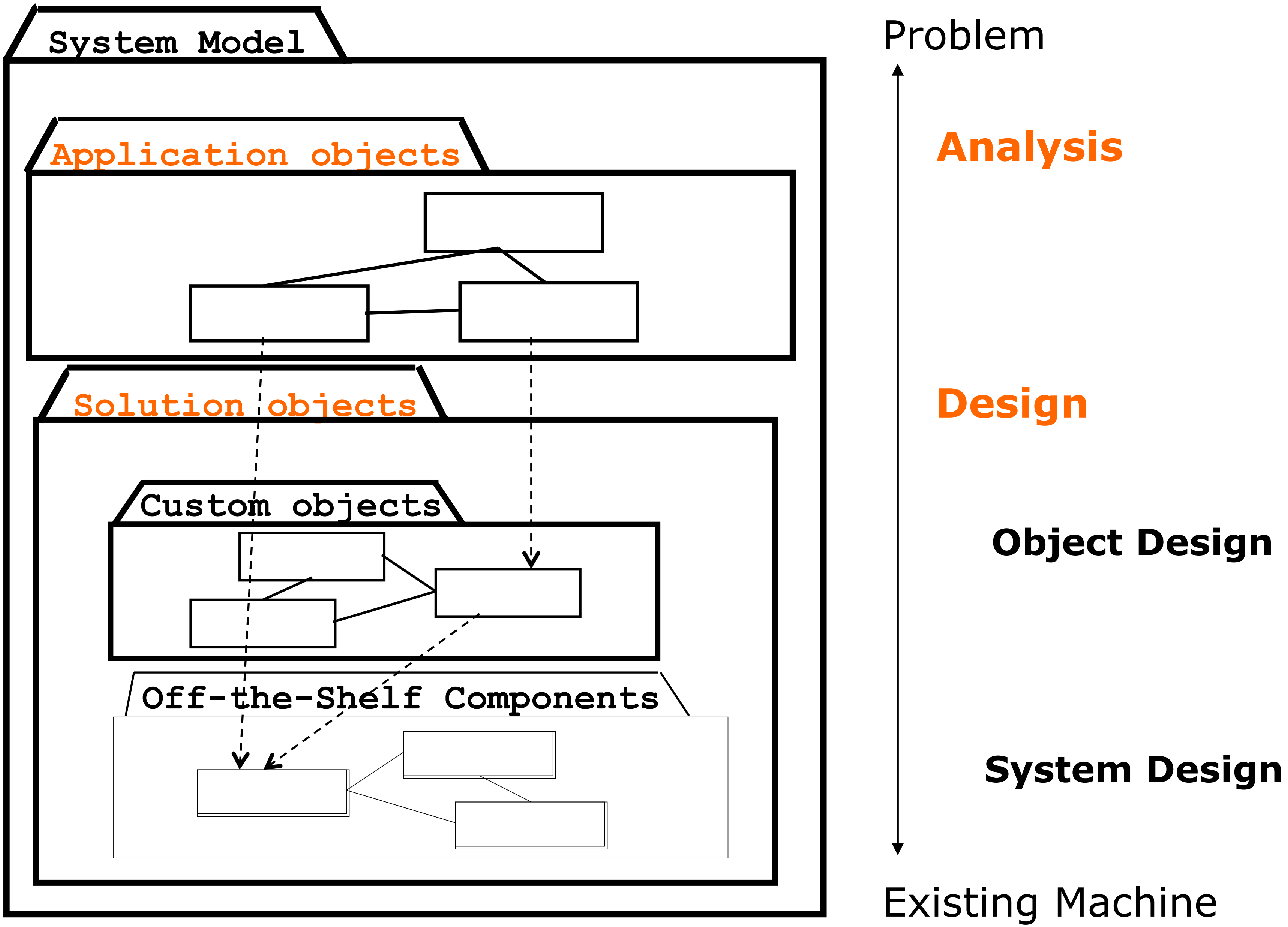
Outline of Today

- Definition and Terminology: Object Design vs Detailed Design
- System Design vs Object Design
- Object Design Activities
- Reuse examples
    - Whitebox and Blackbox Reuse
- Object design leads also to new classes
- Implementation vs Specification Inheritance
- Inheritance vs Delegation
- Class Libraries and Frameworks

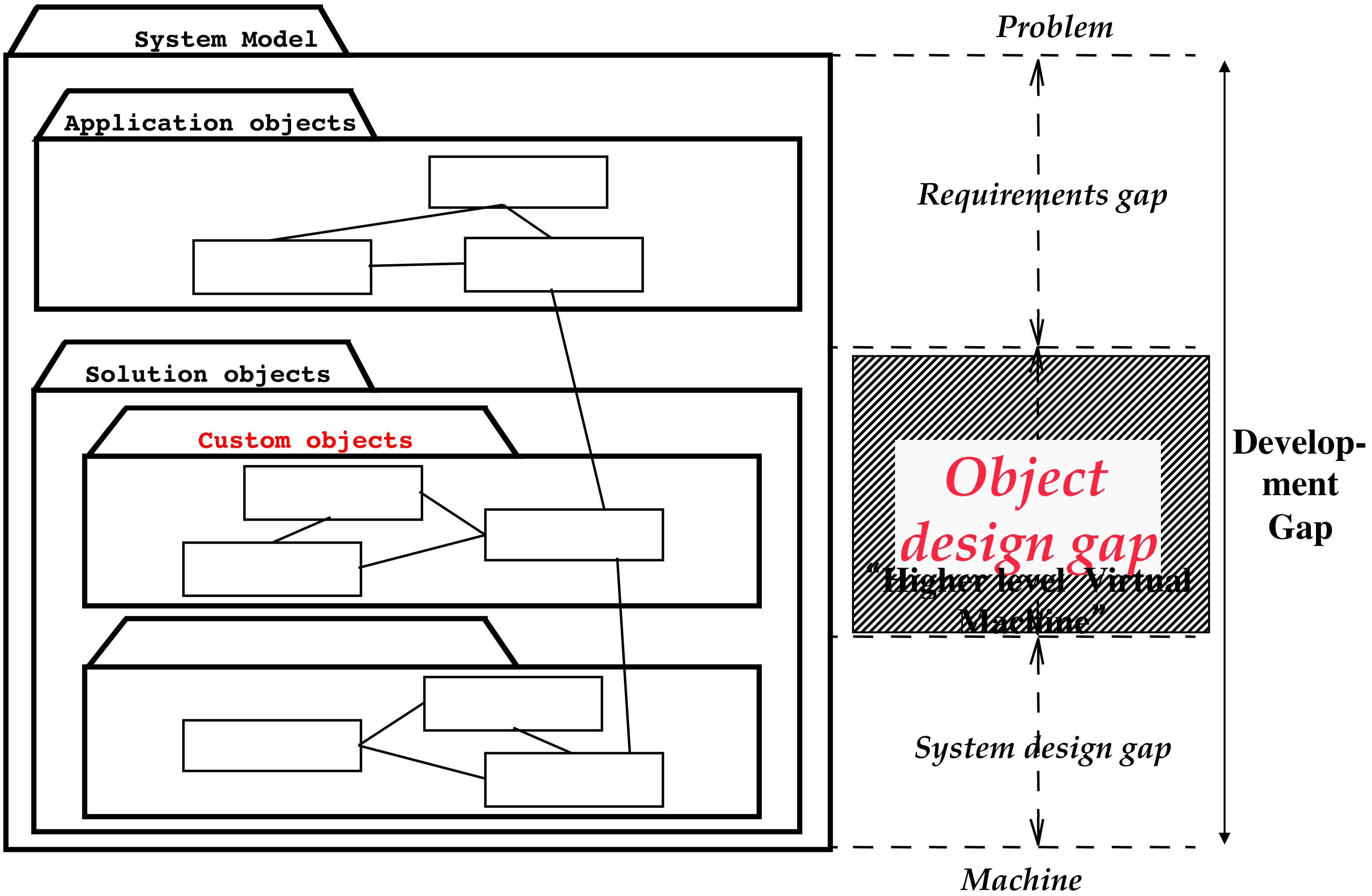
Object Design

- Purpose of object design:
  - Prepare for the implementation of the system model based on design decisions
  - Transform the system model (optimize it)
- Investigate alternative ways to implement the system model
  - Use design goals: minimize execution time, memory and other measures of cost.
- Object design serves as the basis of implementation.

System Development as a Set of Activities



Design means "Closing the Gap" between Problem and Existing Machine



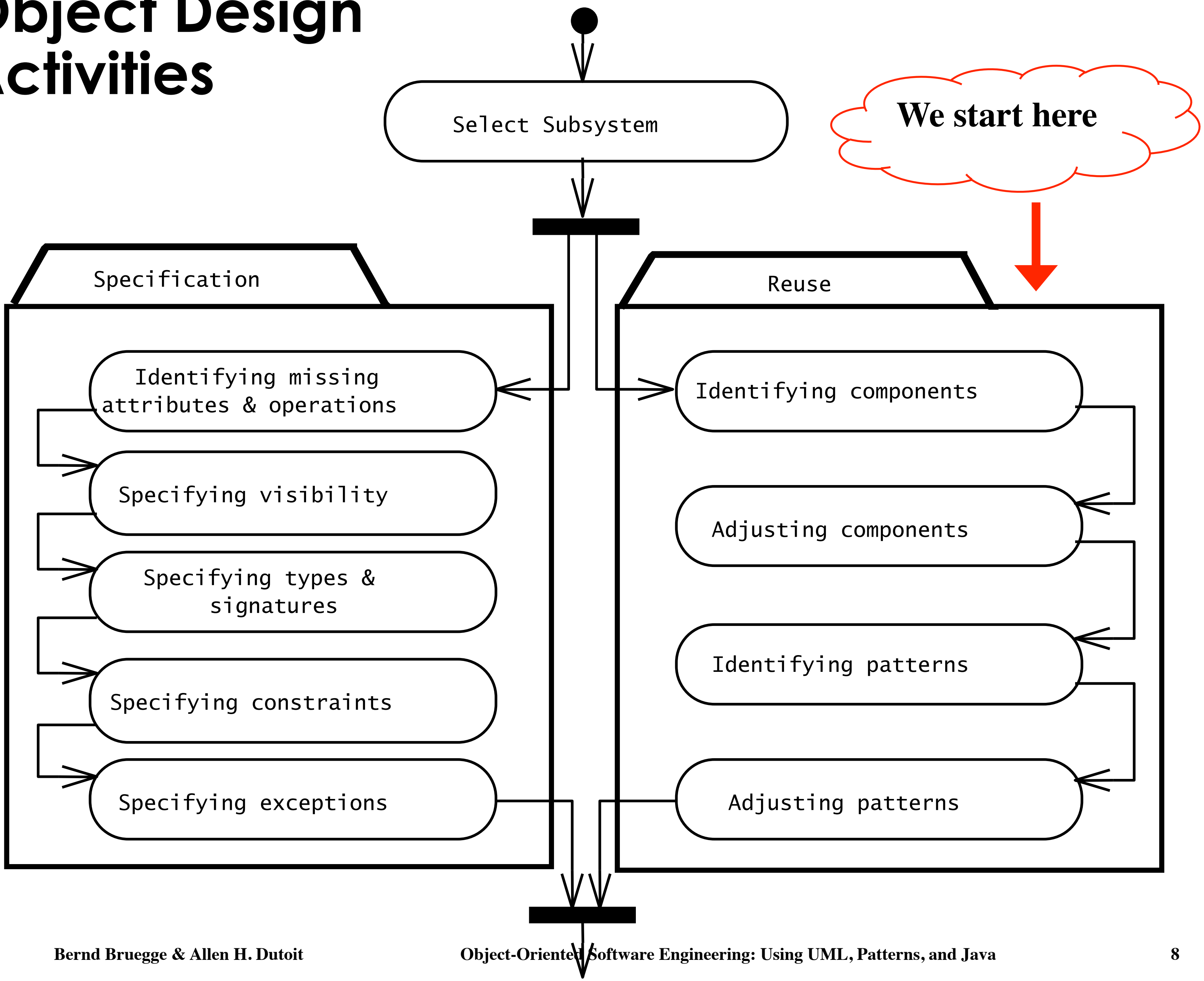
Object Design Activities consists of 4 Activities

1. Reuse: Identification of existing solutions
   - Use of inheritance
   - Off-the-shelf components and additional solution objects
   - ~~Use of 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.

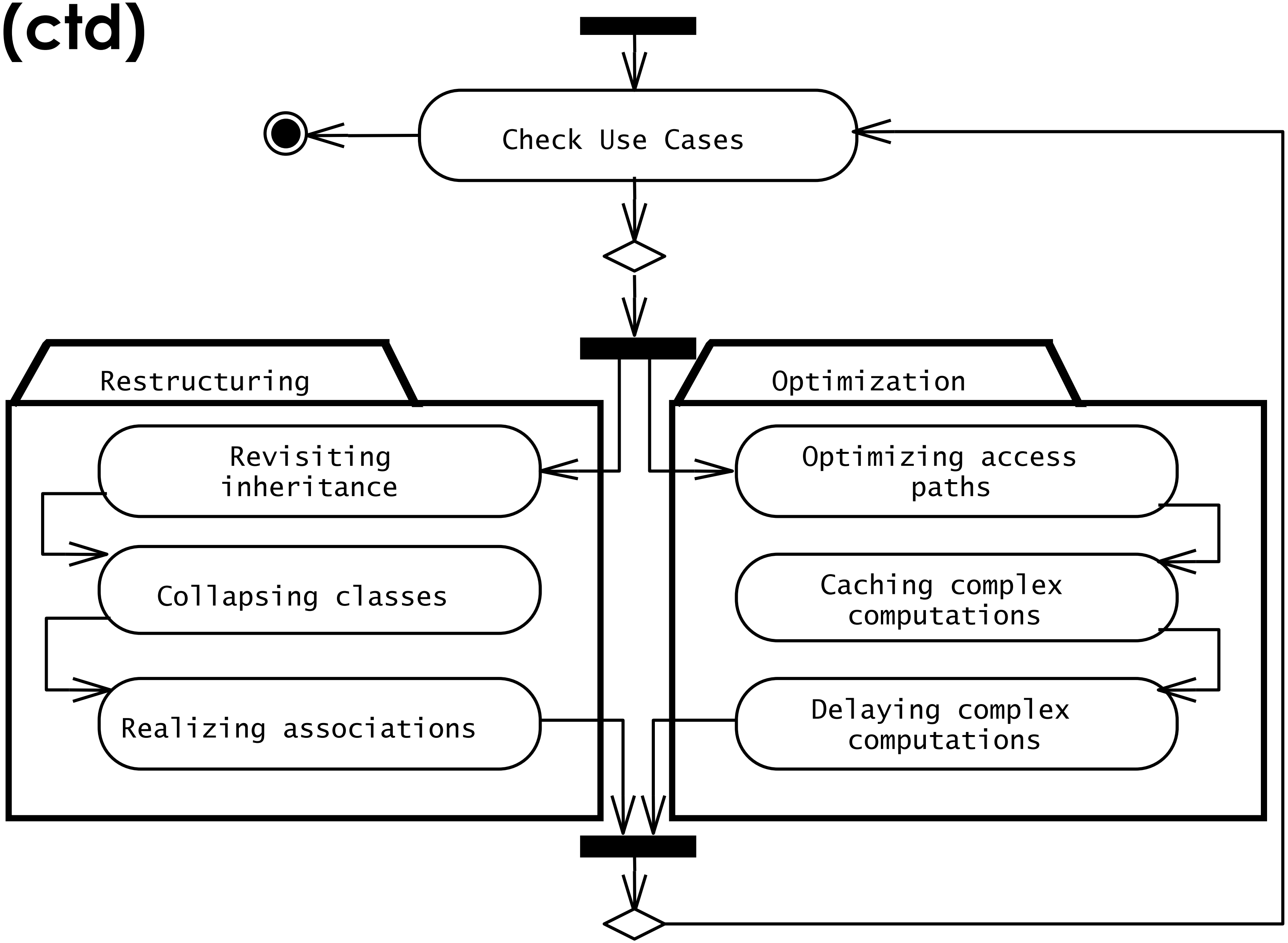
Focus on Reuse and Specification

Towards Mapping Models to Code

Object Design Activities



Detailed View of Object Design Activities (ctd)



One Way to do Object Design

1. Identify the missing components in the design gap
2. Make a build or buy decision to obtain the missing component

=> Component-Based Software Engineering:

The design gap is filled with available components ("0 % coding")

- Special Case: COTS-Development
  - COTS: Commercial-off-the-Shelf
  - The design gap is filled with commercial-off-the-shelf-components.

=> Design with standard components.

Modeling of the Real World

- Design knowledge such as the adapter pattern complements application domain knowledge and solution domain knowledge
- Modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's
- There is a need for reusable and extendable ("flexible") designs.

Review: Design pattern

A design pattern is…

…a reusable template for solving a recurring design problem

- Basic idea: Don't re-invent the wheel!

… design knowledge

- Knowledge on a higher level than classes, algorithms or data structures (linked lists, binary trees...)
- Lower level than application frameworks

…an example of *modifiable design*

- Learning how to design starts by studying other designs.

Why are modifiable designs important?

A modifiable design…

…enables an iterative and incremental development

- concurrent development
- risk management
- flexibility to change

… minimizes the introduction of new problems when fixing old ones

… allows to easily add more functionality after the delivery of the system

What makes a design modifiable?

- Low coupling and high cohesion
- Clear dependencies
- Explicit assumptions

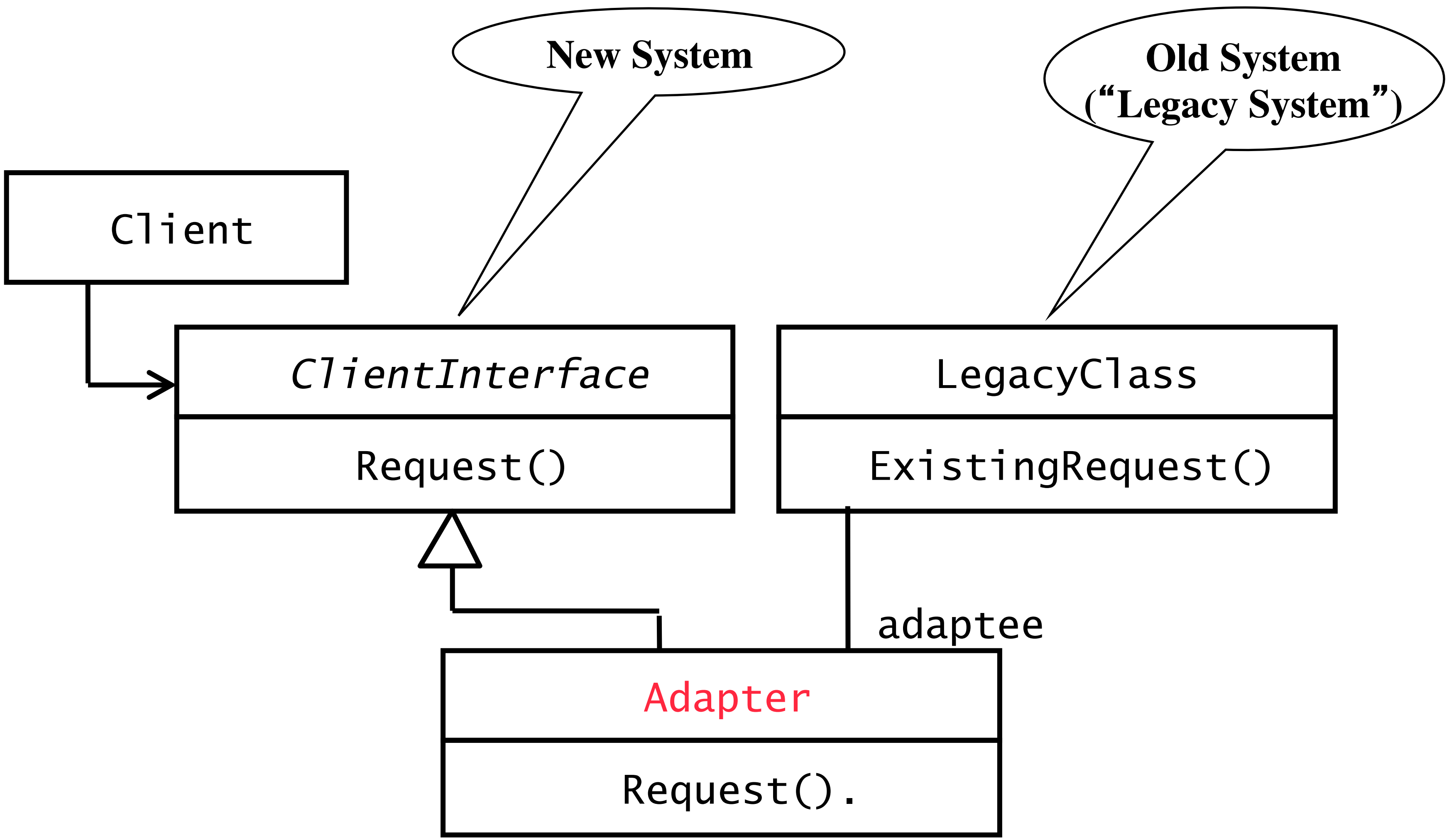
How do design patterns help?

- They are generalized from existing systems
- They provide a shared vocabulary to designers
- They provide examples of modifiable designs
    - Abstract classes
    - Delegation

Adapter Pattern .

- Adapter Pattern: Connects incompatible components
  - It converts the interface of one component into another interface expected by the other (calling) component
  - Used to provide a new interface to existing legacy components (Interface engineering, reengineering)
- Also known as a wrapper.

Adapter Pattern



Where are we?

- ✓ Object Design vs Detailed Design
- ✓ System design vs object design
- ✓ Overview of object design activities
- ✓ Adapter pattern
- ➡ Types of Reuse
  - Code reuse
  - Interface reuse
  - Class reuse
  - Whitebox and blackbox reuse
- Object design leads also to new classes
- Implementation vs Specification Inheritance
- Inheritance vs Delegation
- Class Libraries and Frameworks

Reuse of Code

- I have a list, but the customer wants to have a stack
  - The list offers the operations Insert(), Find(), Delete()
  - The stack needs the operations Push(), Pop() and Top()
  - Can I reuse the existing list?
- I am an employee in a company that builds cars with expensive car stereo systems
  - Can I reuse the existing car software in a home stereo system?

Reuse of interfaces

- I am an off-shore programmer in Hawaii. I have a contract to implement an electronic parts catalog for Daimler
  - How can my contractor make sure that I implement it correctly?
- I would like to develop a window system for Linux that behaves the same way as in Vista
  - How can I make sure that I follow the conventions for Vista and not those of MacOS X?

Reuse of existing classes

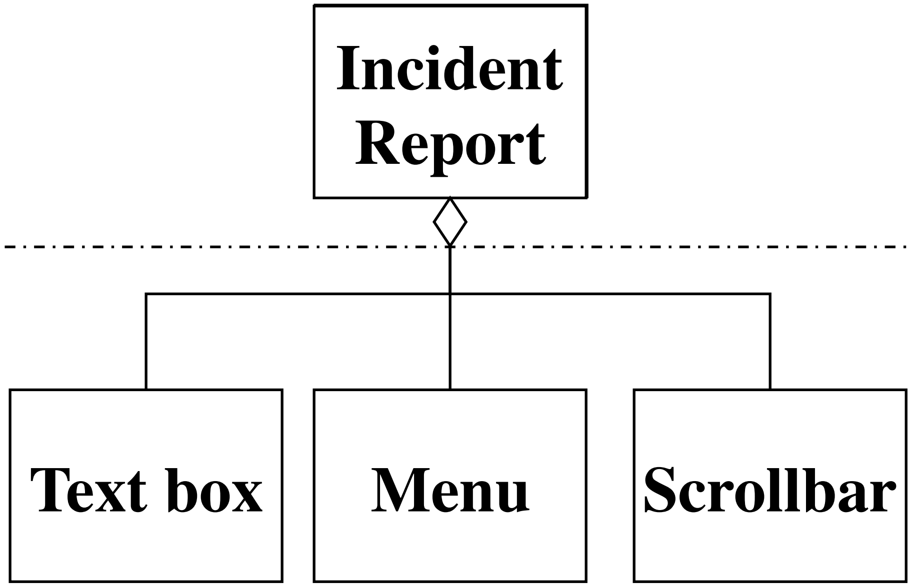
- I have an implementation for a list of elements of Type int
  - Can I reuse this list to build
    - a list of customers
    - a spare parts catalog
    - a flight reservation schedule?
- I have developed a class "Addressbook" in a previous project
  - Can I add it as a subsystem to my e-mail program which I purchased from a vendor (replacing the vendor-supplied address book)?
  - Can I reuse this class in the billing software of my dealer management system?

Customization: Build Custom Objects

- Problem: Closing the object design gap
  - Develop new functionality
- Main goals:
  - Reuse functionality already available
  - Use design knowledge (from previous experience)
- Composition (also called Black Box Reuse)
  - The new functionality is obtained by aggregation
  - The new object with more functionality is an aggregation of existing objects
- Inheritance (also called White-box Reuse)
  - The new functionality is obtained by inheritance.

Example of Composition

Requirements Analysis
(Language of Application
Domain)



Object Design
(Language of Solution
Domain)

Other Reasons for additional Objects in Object Design

- The implementation of algorithms may necessitate objects to hold values
- New low-level operations may be needed during the decomposition of high-level operations
- Example: `EraseArea()` in a drawing program
  - Conceptually very simple
  - Implementation is complicated:
    - `Area` represented by pixels
    - We need a `Repair()` operation to clean up objects partially covered by the erased area
    - We need a `Redraw()` operation to draw objects uncovered by the erasure
    - We need a `Draw()` operation to erase pixels in background color not covered by other objects.

White Box and Black Box Reuse

- White box reuse (inheritance)
  - Access to the development artifacts (analysis model, system design, object design, source code) must be available
- Black box reuse (composition)
  - Access to models and designs is not available, or models do not even exist
    - Worst case: Only executables (binary code) are available
    - Better case: A specification of the system interface is available.

Types of Whitebox Reuse

1. Implementation inheritance
   - Reuse of Implementations
2. Specification Inheritance
   - Reuse of Interfaces

- Programming concepts to achieve reuse
  - ➢ Inheritance
  - Delegation
  - Abstract classes and Method Overriding
  - Interfaces

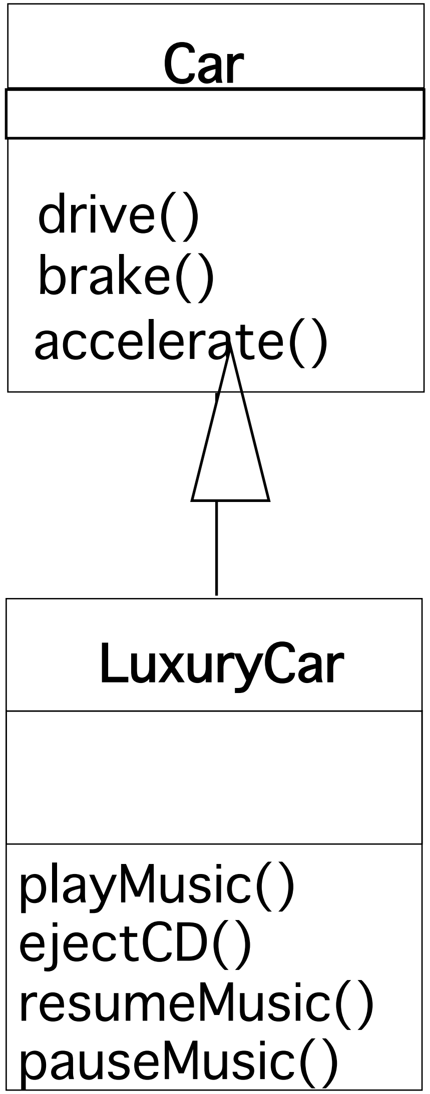
Why Inheritance?

1. Organization (during analysis):
   - Inheritance helps us with the construction of taxonomies to deal with the application domain
     - when talking the customer and application domain experts we usually find already existing taxonomies
2. Reuse (during object design):
   - Inheritance helps us to reuse models and code to deal with the solution domain
     - when talking to developers

The use of Inheritance

- Inheritance is used to achieve two different goals
  - Description of Taxonomies
  - Interface Specification
- Description of Taxonomies
  - Used during *requirements analysis*
  - Activity: identify application domain objects that are hierarchically related
  - Goal: make the analysis model more understandable
- Interface Specification
  - Used during *object design*
  - Activity: identify the signatures of all identified objects
  - Goal: increase reusability, enhance modifiability and extensibility

Example of using Inheritance for Taxonomy



Superclass:

```
public class Car {
   public void drive() {…}
   public void brake() {…}
   public void accelerate() {…}
}
```

Subclass:

```
public class LuxuryCar extends Car
{
   public void playMusic() {…}
   public void ejectCD() {…}
   public void resumeMusic() {…}
   public void pauseMusic() {…}
}
```

Inheritance can be used during Analysis as well as during Design

- Starting point is always the requirements analysis phase:
  - We start with use cases
  - We identify existing objects (“class identification“)
  - We investigate the relationship between these objects; “Discovering associations“:
    - general associations
    - aggregations
    - inheritance associations.

Discovering Inheritance Associations

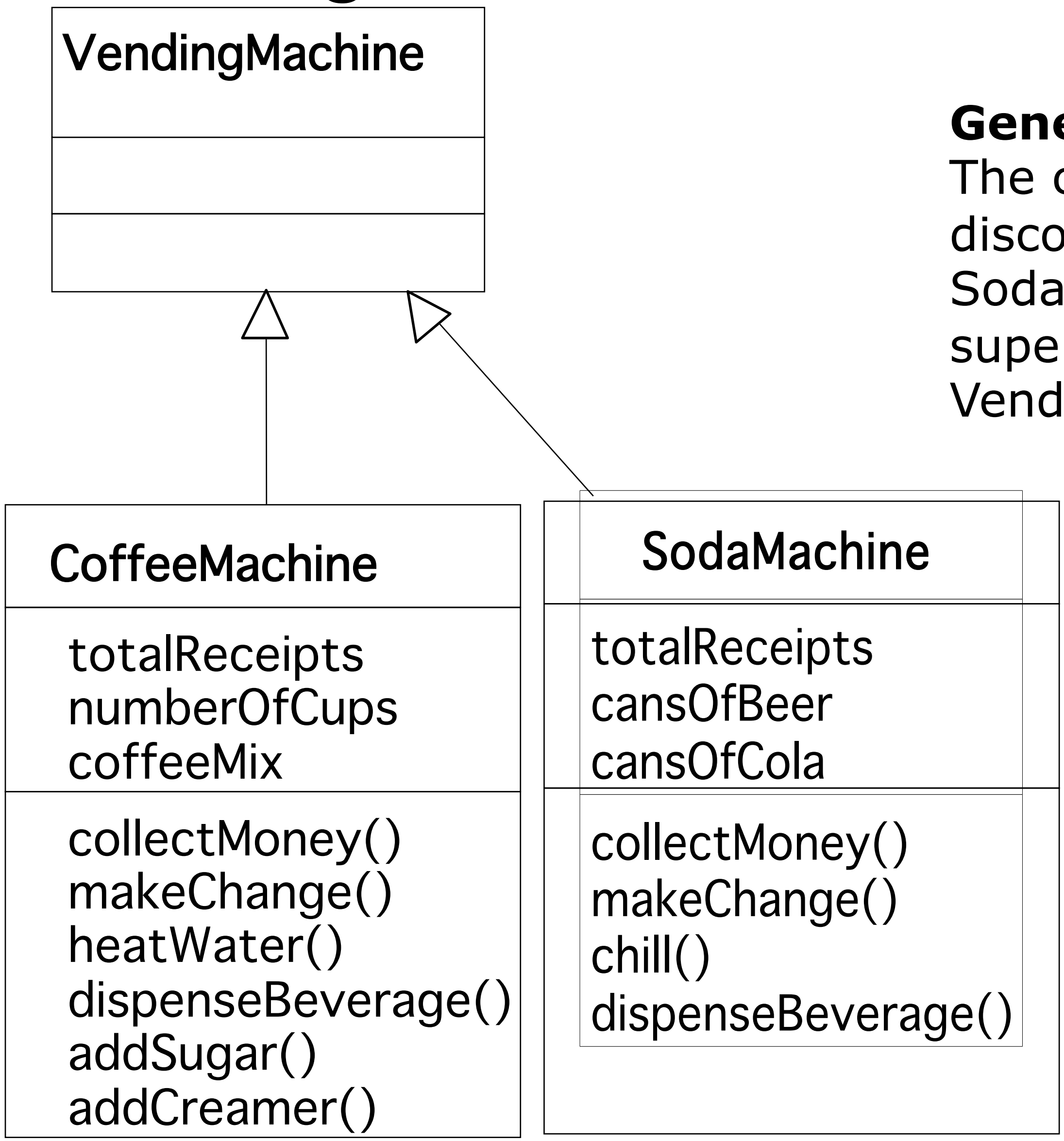
- To “discover“ inheritance associations, we can proceed in two ways, which we call specialization and generalization

- Generalization: the discovery of an inheritance relationship between two classes, where the sub class is discovered first.
- Specialization: the discovery of an inheritance relationship between two classes, where the super class is discovered first.

Generalization

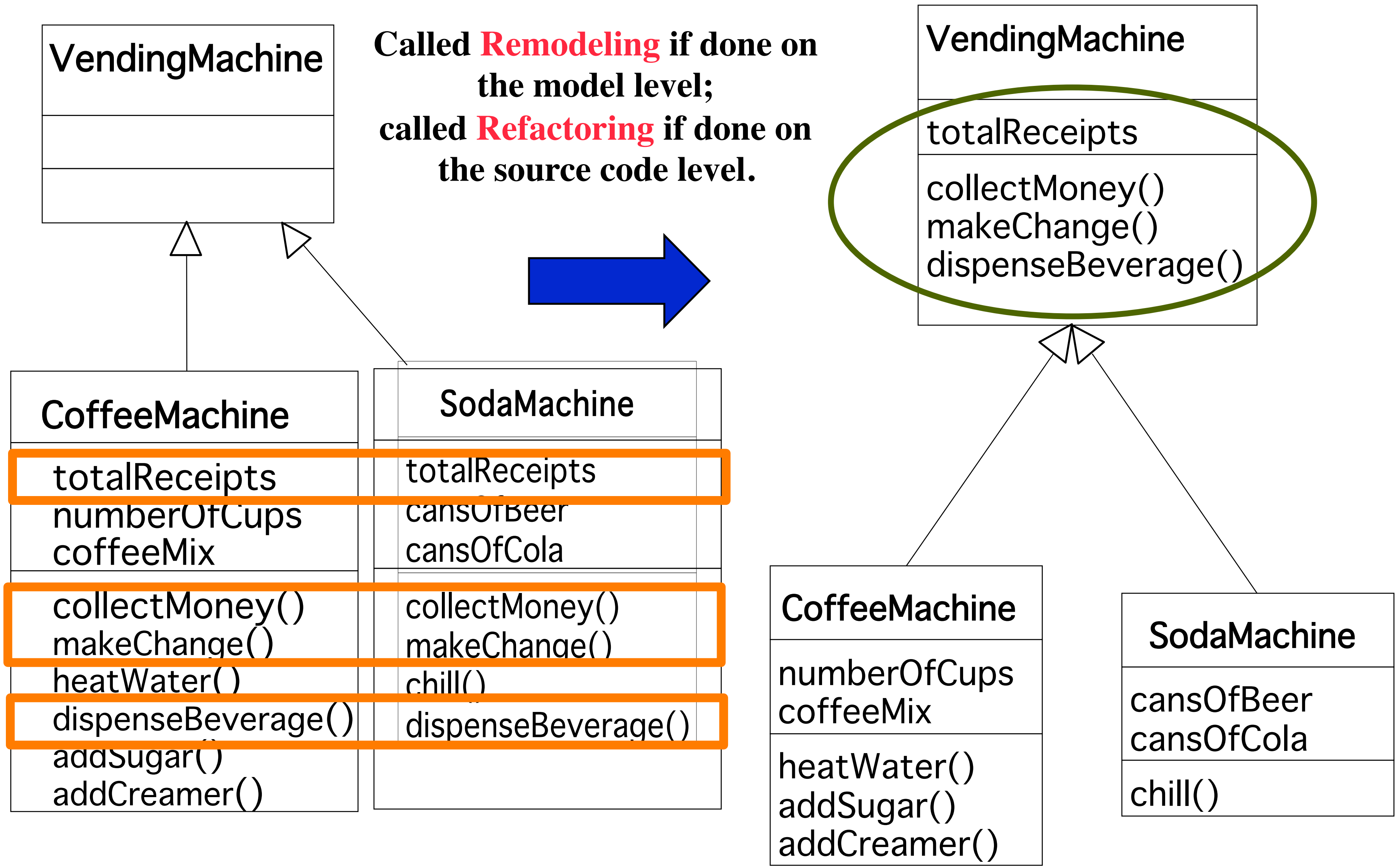
- First we find the subclass, then the super class
- This type of discovery occurs often in science and engineering:
  - **Biology**: First we find individual animals (Elefant, Lion, Tiger), then we discover that these animals have common properties (mammals).
  - **Engineering:** What are the common properties of cars and airplanes?

Generalization Example: Modeling Vending Machines



Generalization:
The class CoffeeMachine is discovered first, then the class SodaMachine, then the superclass VendingMachine

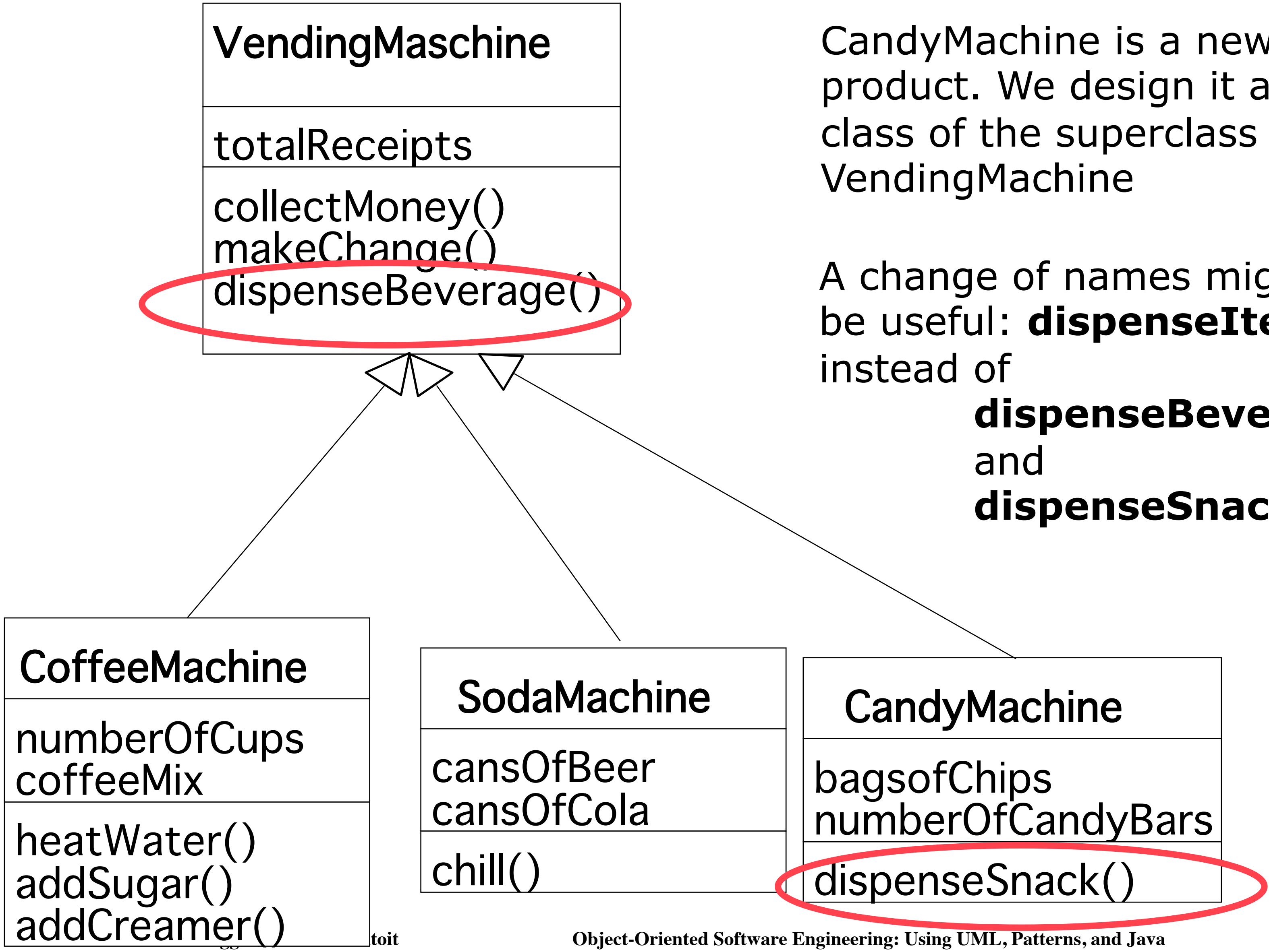
Generalizing often leads to Restructuring



Specialization

- Specialization occurs, when we find a subclass that is very similar to an existing class
  - Example: A theory postulates certain particles and events which we have to find
- New products
  - Last year we finished a project, in which we developed a machine, that delivers coffee and tea with automatic detection of empty containers.
  - In the new project we have to develop the same functionality for a new candy machine.

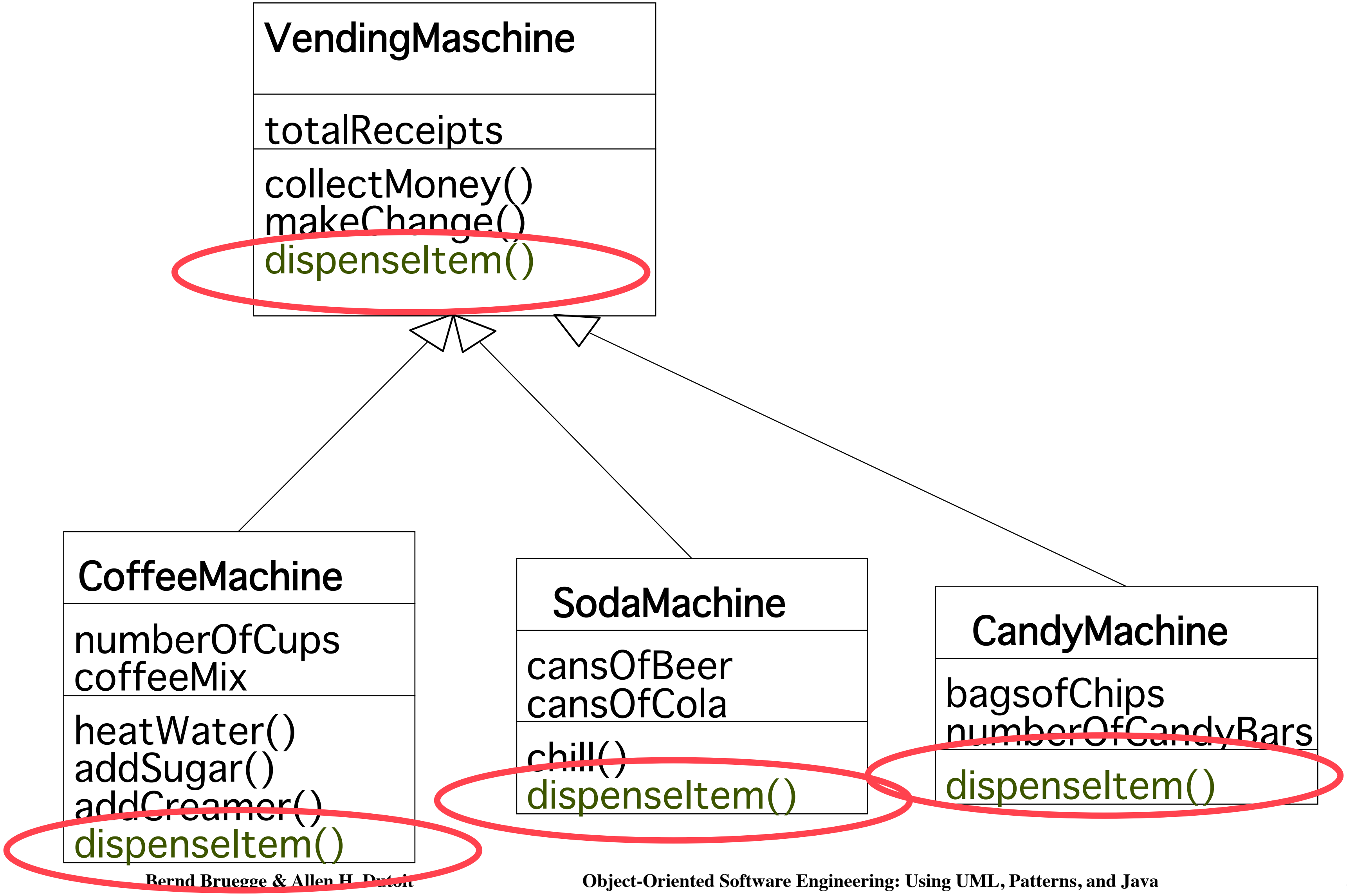
Another Example of a Specialization



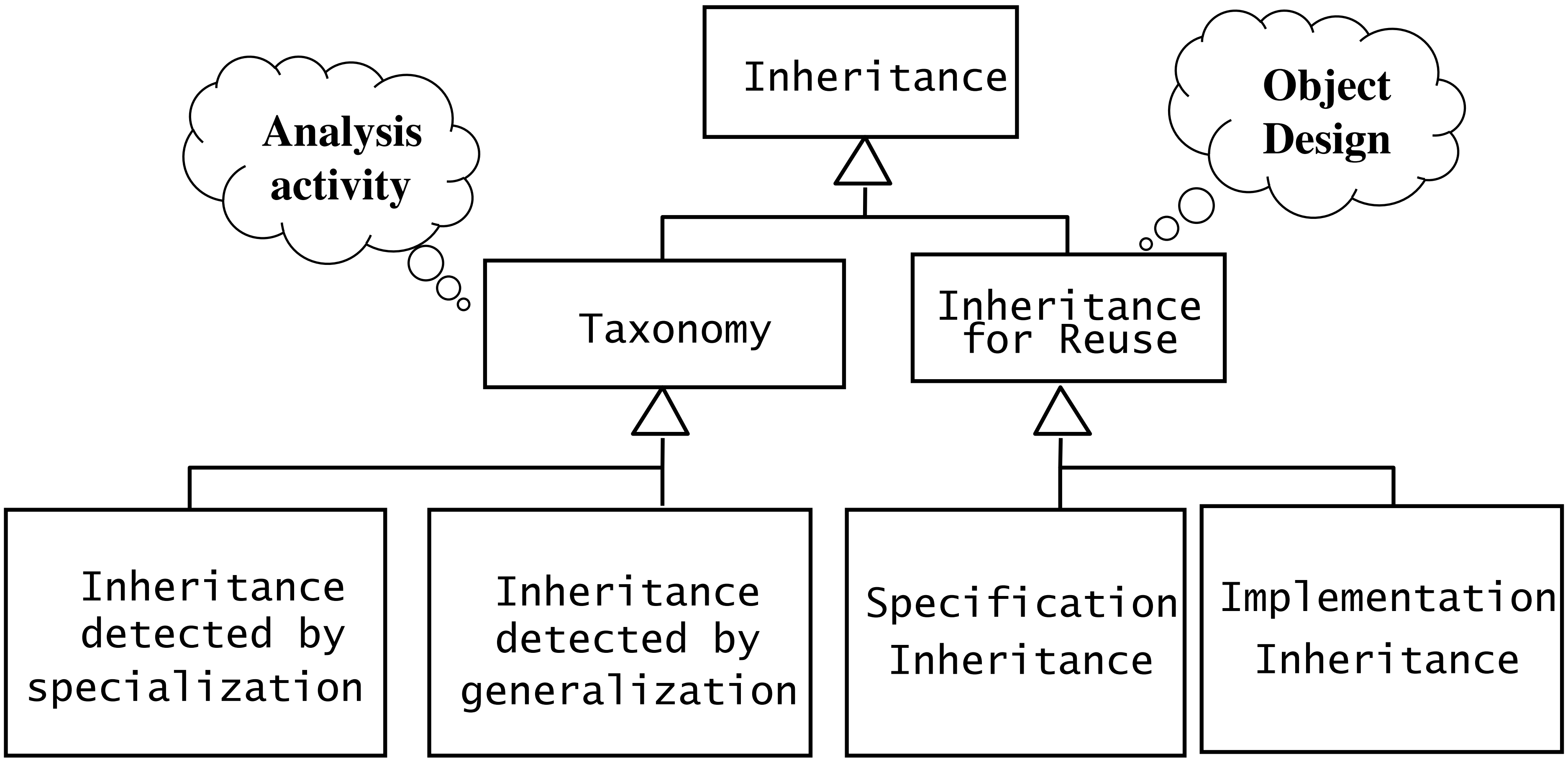
CandyMachine is a new product. We design it as a sub class of the superclass VendingMachine

A change of names might now be useful: **dispenseItem()** instead of
dispenseBeverage()
and
dispenseSnack()

Example of a Specialization (2)



Meta-Model for Inheritance



Implementation Inheritance and Specification Inheritance

- Implementation inheritance
  - Also called class inheritance
  - Goal:
    - Extend an applications' functionality by reusing functionality from the super class
    - Inherit from an existing class with some or all operations already implemented
- Specification Inheritance
  - Also called subtyping
  - Goal:
    - Inherit from a specification
    - The specification is an abstract class with all the operations specified but not yet implemented.

Implementation Inheritance vs. Specification Inheritance

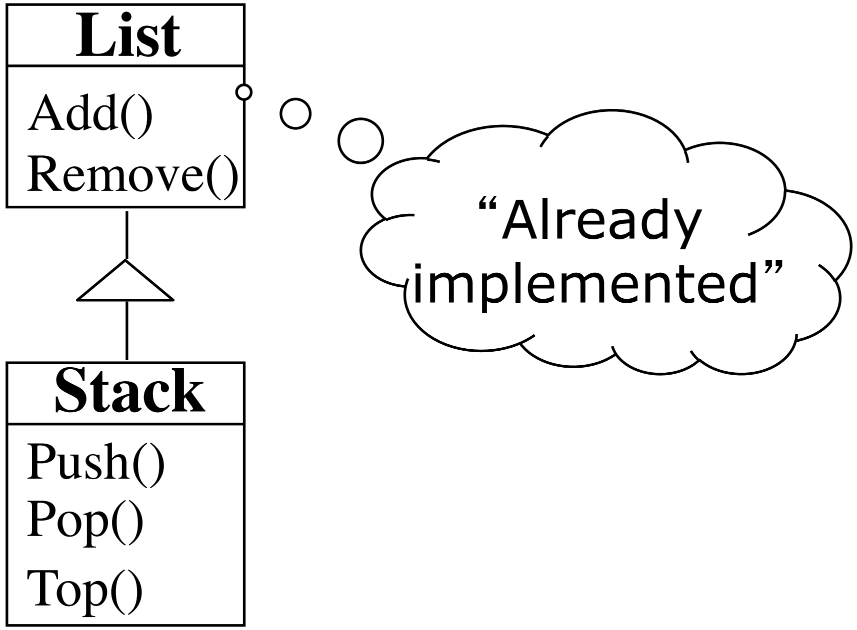
- Implementation inheritance: The combination of inheritance and implementation
  - The interface of the superclass is completely inherited
  - Implementations of methods in the superclass ("Reference implementations") are inherited by any subclass
- Specification inheritance: The combination of inheritance and specification
  - The interface of the superclass is completely inherited
  - Implementations of the superclass (if there are any) are not inherited.

Example for Implementation Inheritance

A class is already implemented that does almost the same as the desired class

Example:

- I have a List, I need a Stack
  - How about subclassing the Stack class from the List class and implementing Push(), Pop(), Top() with Add() and Remove()?

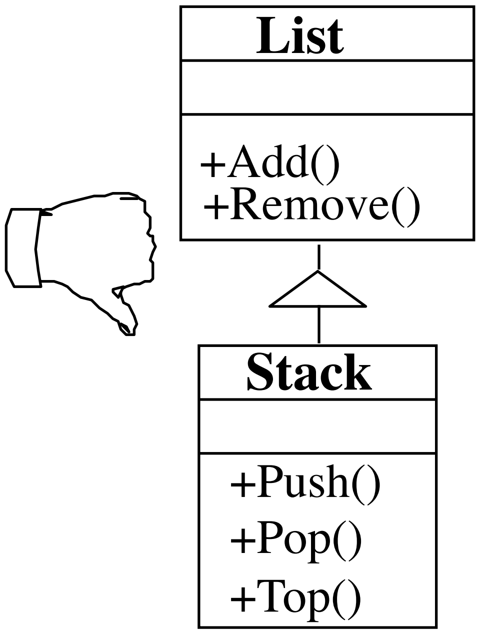


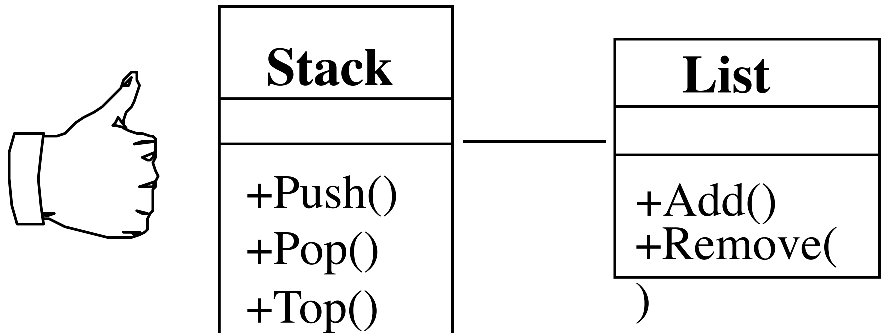
❖ Problem with implementation inheritance:

- The inherited operations might exhibit unwanted behavior
- Example: What happens if the Stack user calls Remove() instead of Pop()?

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?

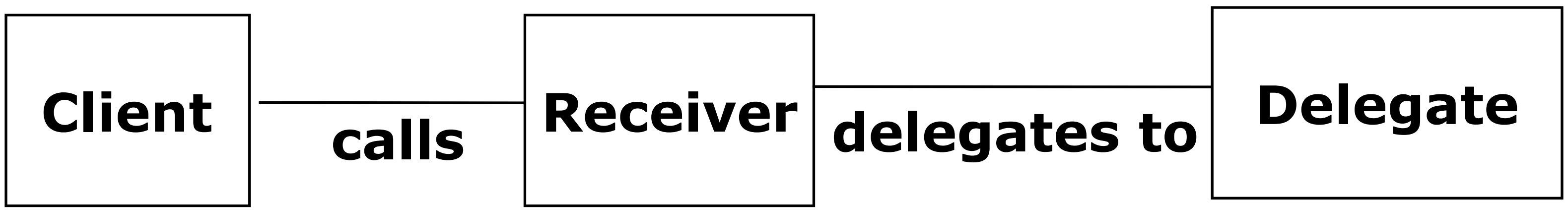




Delegation

* Delegation is a way of making composition as powerful for reuse as inheritance
* In delegation two objects are involved in handling a request from a Client

  * The Receiver object delegates operations to the Delegate object
  * The Receiver object makes sure, that the Client does not misuse the Delegate object.



Comparison: Delegation vs Implementation Inheritance

- Delegation
  - ☺ Flexibility: Any object can be replaced at run time by another one (as long as it has the same type)
  - ☹ Inefficiency: Objects are encapsulated
- Inheritance
  - ☺ Straightforward to use
  - ☺ Supported by many programming languages
  - ☺ Easy to implement new functionality in the subclass
  - ☹ Inheritance exposes a subclass to the details of its parent class
  - ☹ Any change in the parent class implementation forces the subclass to change (which requires recompilation of both).

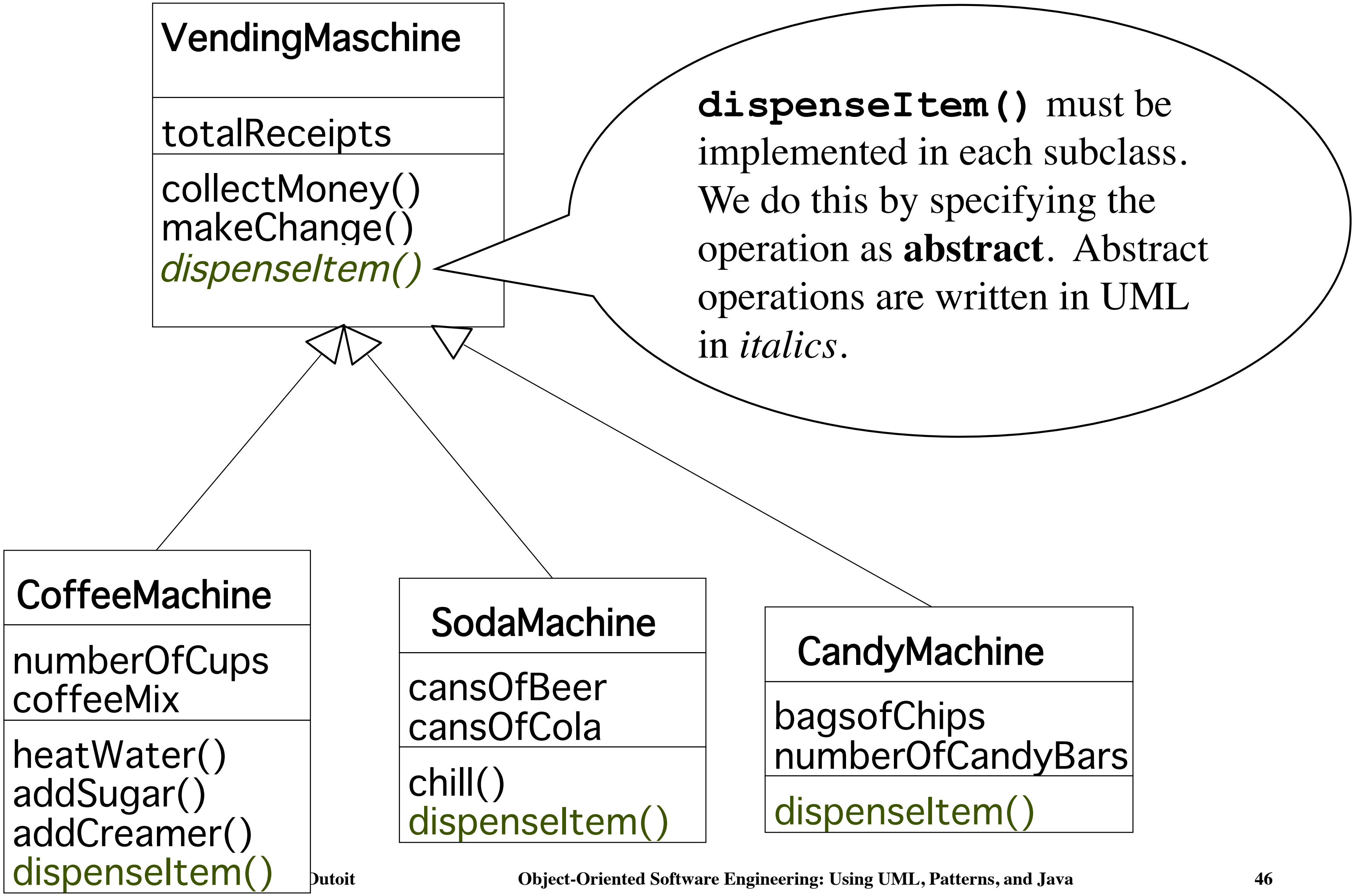
Recall: Implementation Inheritance v. Specification-Inheritance

- Implementation Inheritance: The combination of inheritance and implementation
  - The interface of the super class is completely inherited
  - Implementations of methods in the super class ("Reference implementations") are inherited by any subclass
- Specification Inheritance: The combination of inheritance and specification
  - The interface of the super class is completely inherited
  - Implementations of the super class are not inherited
  - Or the super class is an abstract class.

Abstract Operations and Abstract Classes

- Abstract method:
  - A method with a signature but without an implementation. Also called abstract operation
- Abstract class:
  - A class which contains at least one abstract method is called abstract class
- UML Interface: An abstract class which has only abstract operations
    - An interface is primarily used for the specification of a system or subsystem. The implementation is provided by a subclass or by other mechanisms.

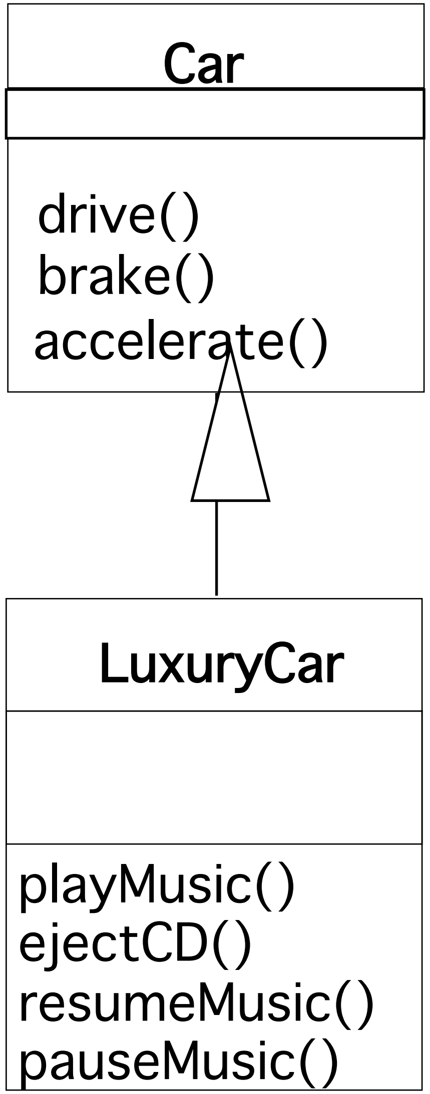
Example of an Abstract Operation



Rewriteable Methods and Strict Inheritance

- Rewriteable Method: A method which allows a reimplementation
  - In Java methods are rewriteable by default, i.e. there is no special keyword
- Strict inheritance
    - The subclass can only add new methods to the superclass, it cannot over write them
  - If a method cannot be overwritten in a Java program, it must be prefixed with the keyword `final`.

Strict Inheritance



Superclass:

```
public class Car {
    public final void drive() {…}
    public final void brake() {…}
    public final void accelerate()
{…}
}
```

Subclass:

```
public class LuxuryCar extends Car
{
    public void playMusic() {…}
    public void ejectCD() {…}
    public void resumeMusic() {…}
    public void pauseMusic() {…}
}
```

Example: Strict Inheritance and Rewriteable Methods

Original Java-Code:

```
class Device {
    int serialnr;
    public final void help() {….}
    public void setSerialNr(int n) {
  serialnr = n;
}
}
class Valve extends Device {
    Position s;
    public void on() {
      ….
    }
}
```

help() not overwritable

setSerialNr() overwritable

Example: Overwriting a Method

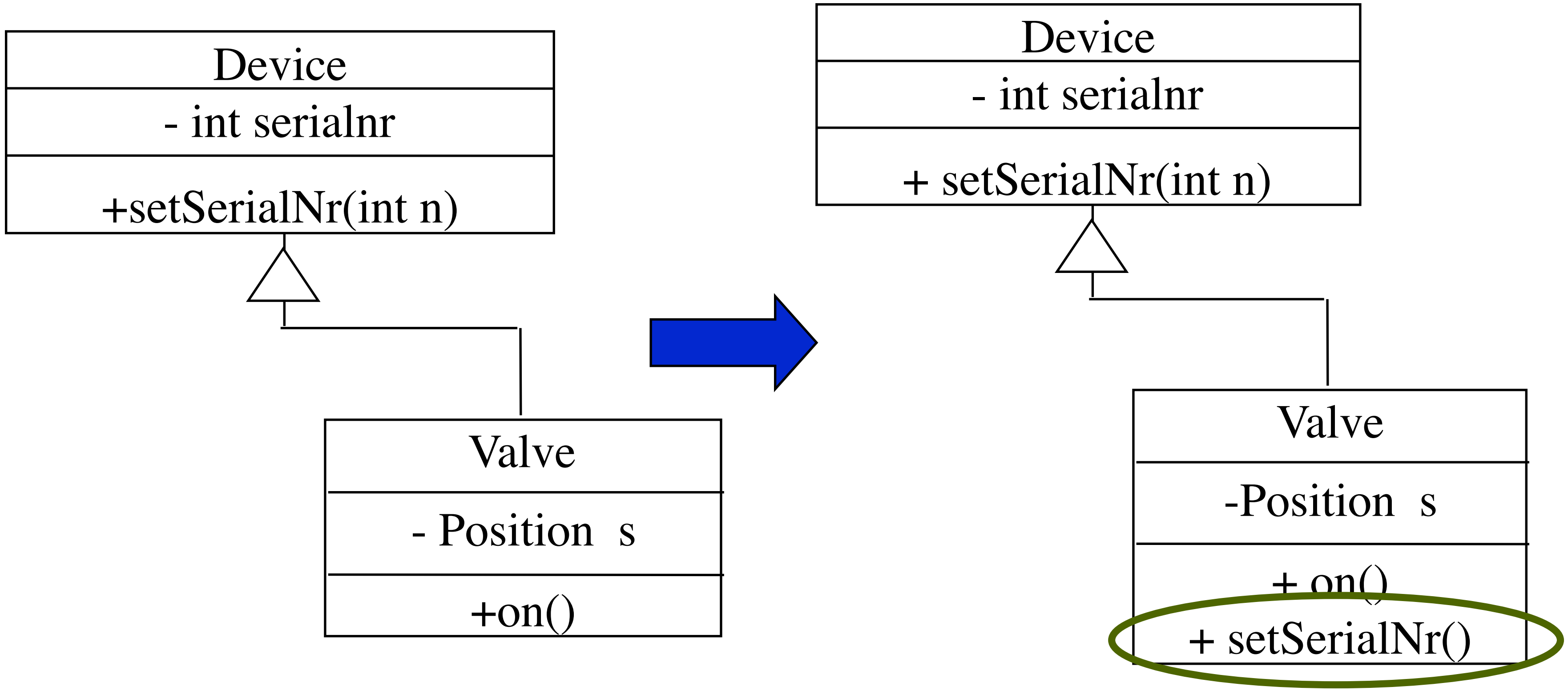
Original Java-Code:

```
class Device {
    int serialnr;
    public final void help() {….}
    public void setSerialNr(int n) {
      serialnr = n;
    }
}
class Valve extends Device {
    Position s;
    public void on() {
      ….
    }
} // class Valve
```

New Java-Code :

```
class Device {
    int serialnr;
    public final void help() {….}
    public void setSerialNr(int n) {
      serialnr = n;
    }
}
class Valve extends Device {
    Position s;
    public void on() {
      …
    }
    public void setSerialNr(int n) {
      serialnr = n + 100;
    }
} // class Valve
```

UML Class Diagram



Overwriteable Methods: Usually implemented with Empty Body

```
class Device {
    int serialnr;
    public void setSerialNr(int n) {}
}
class Valve extends Device {
   Position s;
   public void on() {
      …..
   }
   public void setSerialNr(int n) {
      serialnr = n + 100;
   }
} // class Valve
```

I expect, that the method `setSerialNr()` will be overwritten. I only write an empty body

Overwriting of the method `setSerialNr()` of Class `Device`

Bad Use of Overwriting Methods

One can overwrite the operations of a superclass with completely new meanings

Example:

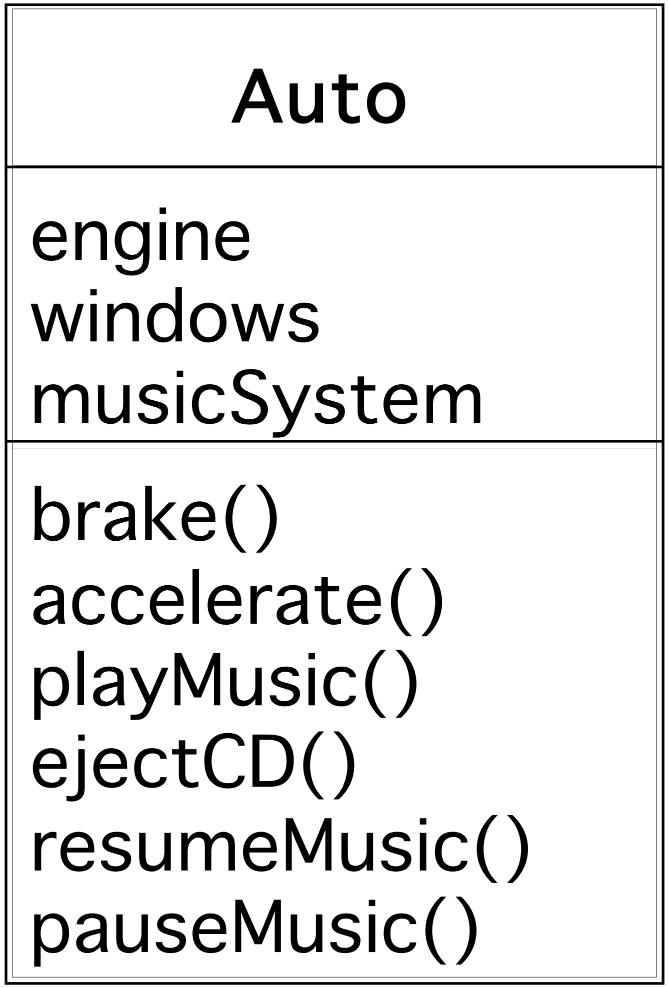
```
Public class SuperClass {
  public int add (int a, int b) { return a+b; }
  public int subtract (int a, int b) { return a-b; }
}
Public class SubClass extends SuperClass {
  public int add (int a, int b) { return a-b; }
  public int subtract (int a, int b) { return a+b; }
}
```

- We have redefined addition as subtraction and subtraction as addition!!

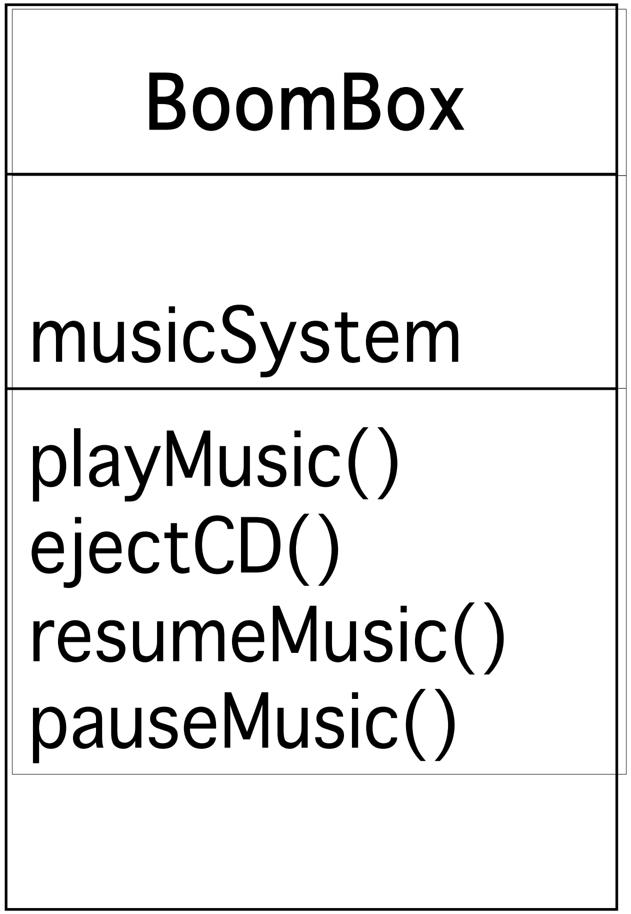
Bad Use of Implementation Inheritance

- We have delivered a car with software that allows to operate an on-board stereo system
  - A customer wants to have software for a cheap stereo system to be sold by a discount store chain
- Dialog between project manager and developer:
  - Project Manager:
    - „Reuse the existing car software. Don ‘t change this software, make sure there are no hidden surprises. There is no additional budget, deliver tomorrow!“
  - Developer:
    - „OK, we can easily create a subclass BoomBox inheriting the operations from the existing Car software“
    - „And we overwrite all method implementations from Car that have nothing to do with playing music with empty bodies!“

What we have and what we want

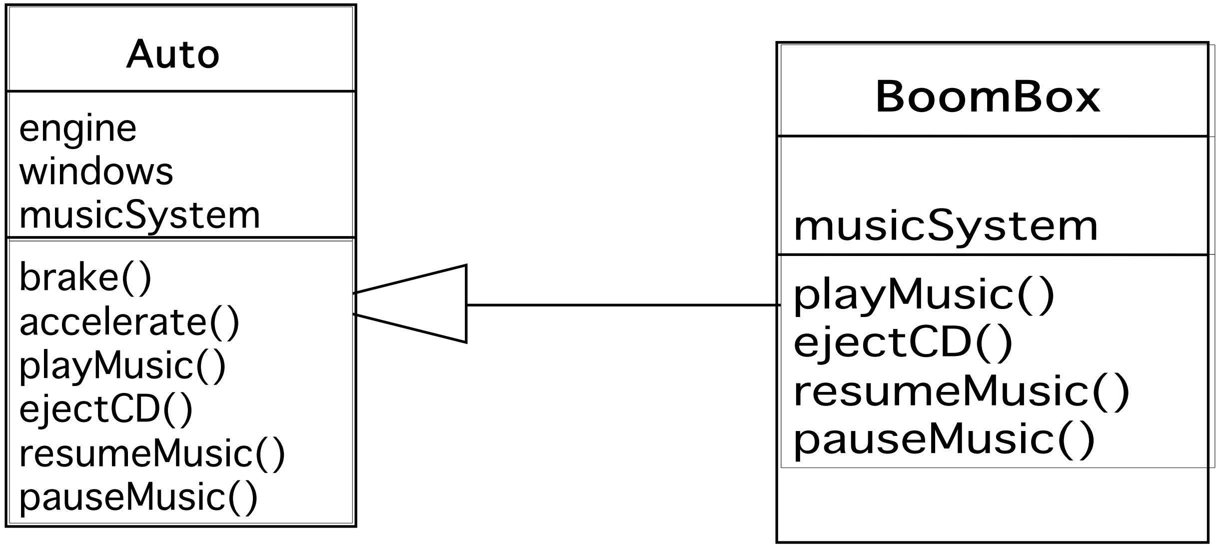


ExistingProduct!



New Product!

What we do to save money and time



Existing Product:

```
public class Auto {
  public void drive() {…}
  public void brake() {…}
  public void accelerate() {…}
  public void playMusic() {…}
  public void ejectCD() {…}
  public void resumeMusic() {…}
  public void pauseMusic() {…}
}
```

New Product:

```
public class Boombox extends
Auto {
  public void drive() {};
  public void brake() {};
  public void accelerate() {};
}
```

Contraction

- **Contraction:** Implementations of methods in the super class are overwritten with empty bodies in the subclass to make the super class operations “invisible“
- Contraction is a special type of inheritance
- It should be avoided at all costs, but is used often.

Contraction should be avoided

A contracted subclass delivers the desired functionality expected by the client, but:

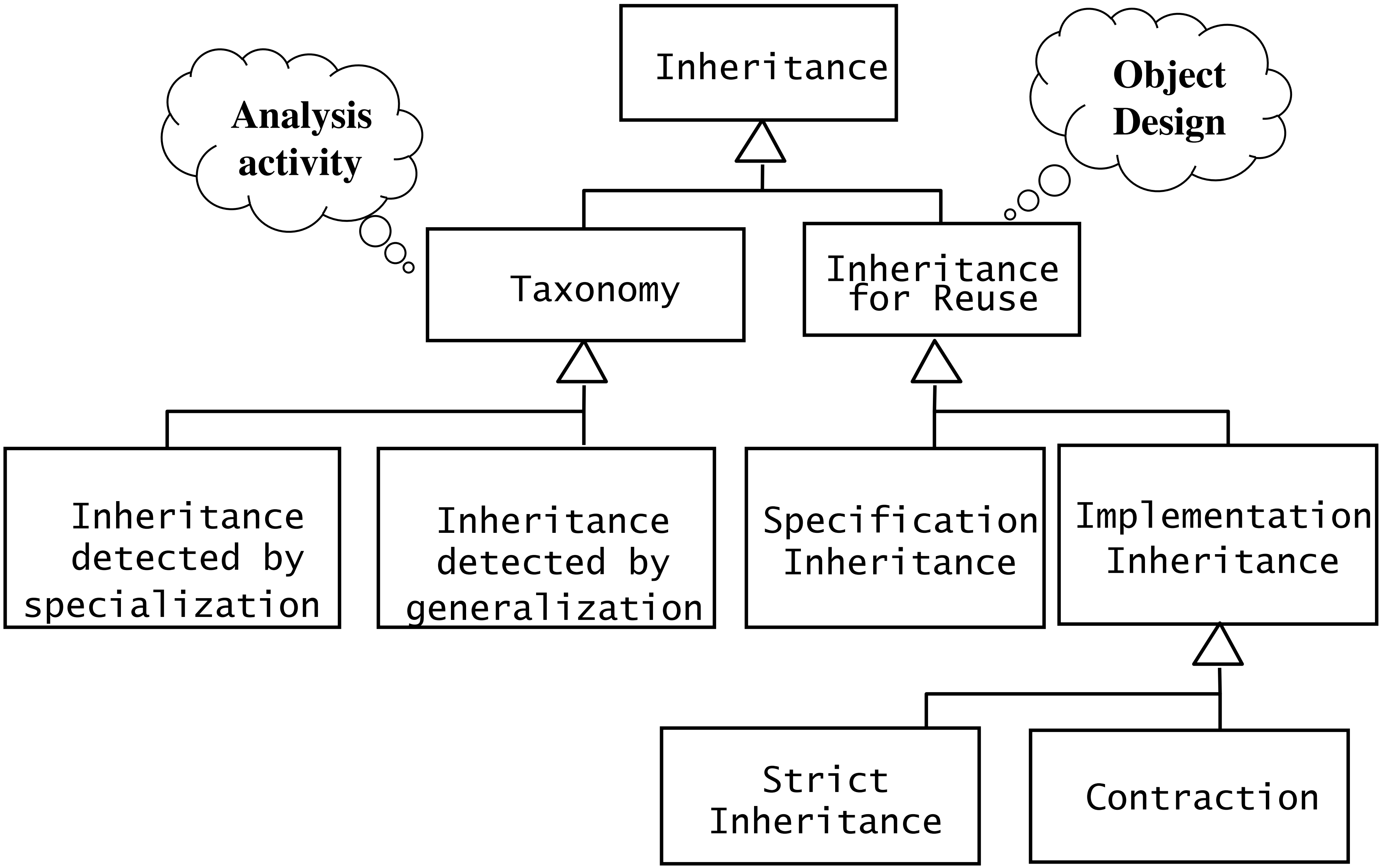
- The interface contains operations that make no sense for this class
- What is the meaning of the operation brake() for a BoomBox?

The subclass does not fit into the taxonomy

A BoomBox ist not a special form of Auto

- The subclass violates Liskov's Substitution Principle:
  - I cannot replace Auto with BoomBox to drive to work.
  - Liskov’s Substitution Principle:
    - If an object of type S can be substituted in all the places where an object of type T is expected, then S is a subtype of T.

Revised Metamodel for Inheritance



Frameworks

- A framework is a partial application that can be *specialized* to produce custom applications
- The key benefits of frameworks are reusability and extensibility:
  - Reusability leverages on the application domain knowledge and prior effort of experienced developers
  - Extensibility is provided by methods which can be overwritten by the application to extend the framework.

Classification of Frameworks

- Frameworks can be classified by their position in the software development process:
  - Infrastructure frameworks
  - Middleware frameworks

- Frameworks can also be classified by the techniques used to extend them:
  - Whitebox frameworks
  - Blackbox frameworks

Frameworks in the Development Process

- Infrastructure frameworks aim to simplify the software development process
  - Used internally, usually not delivered to a client
- Middleware frameworks are used to integrate existing distributed applications
  - Examples: Java RMI, CORBA, WebObjects, WebSphere.
- Enterprise application frameworks are application specific and focus on domains
  - Example of application domains: telecommunications, avionics, environmental modeling, manufacturing, financial engineering, enterprise business activities.

Class libraries vs. Frameworks

- Class Library:
  - Provide a smaller scope of reuse
  - Is less domain specific
  - Class libraries are passive; there is no constraint on the flow of control
- Framework:
  - Classes cooperate for a family of related applications.
  - Frameworks are active; they affect the flow of control.

Components vs. Frameworks

- Components:
  - Self-contained instances of classes
  - Plugged together to form complete applications
  - Can even be reused on the binary code level
    - The advantage is that applications do not have to be recompiled when components change
- Framework:
  - Often used to develop components
  - Components are often plugged into blackbox frameworks.

Documenting the Object Design

- Object design document (ODD)
  = The Requirements Analysis Document (RAD) plus...
    … additions to object, functional and dynamic models (from the solution domain)
    … navigational map for object model
    … Specification for all classes (use Javadoc)

Documenting Object Design: ODD Conventions

- Each subsystem in a system provides a service
  - Describes the set of operations provided by the subsystem
- Specification of the service operations
  - Signature: Name of operation, fully typed parameter list and return type
  - Abstract: Describes the operation
  - Pre: Precondition for calling the operation
  - Post: Postcondition describing important state after the execution of the operation
- Use JavaDoc and Contracts for the specification of service operations
  - Contracts are covered in one of the next lectures.

Package it all up

- Pack up object design into discrete units that can be edited, compiled, linked, reused
- Construct physical modules
  - Ideally use one package for each subsystem
  - But system design might not be good enough for packaging
- Two design principles for packaging
  - Minimize coupling:
    - Classes in client-supplier relationships are usually loosely coupled
    - Avoid large number of parameters in methods to avoid strong coupling (should be less than 4-5)
  - Maximize cohesion: Put classes connected by associations into one package.

Packaging Heuristics

- Each subsystem service is made available by one or more interface objects within the package
- Start with one interface object for each subsystem service
  - Try to limit the number of interface operations (7+-2)
- If an interface object has too many operations, reconsider the number of interface objects
- If you have too many interface objects, reconsider the number of subsystems
- Interface objects vs Java interface:
  - Interface object: Used during requirements analysis, system design, object design. Denotes a service or API
  - Java interface: Used during implementation in Java (May or may not implement an interface object).

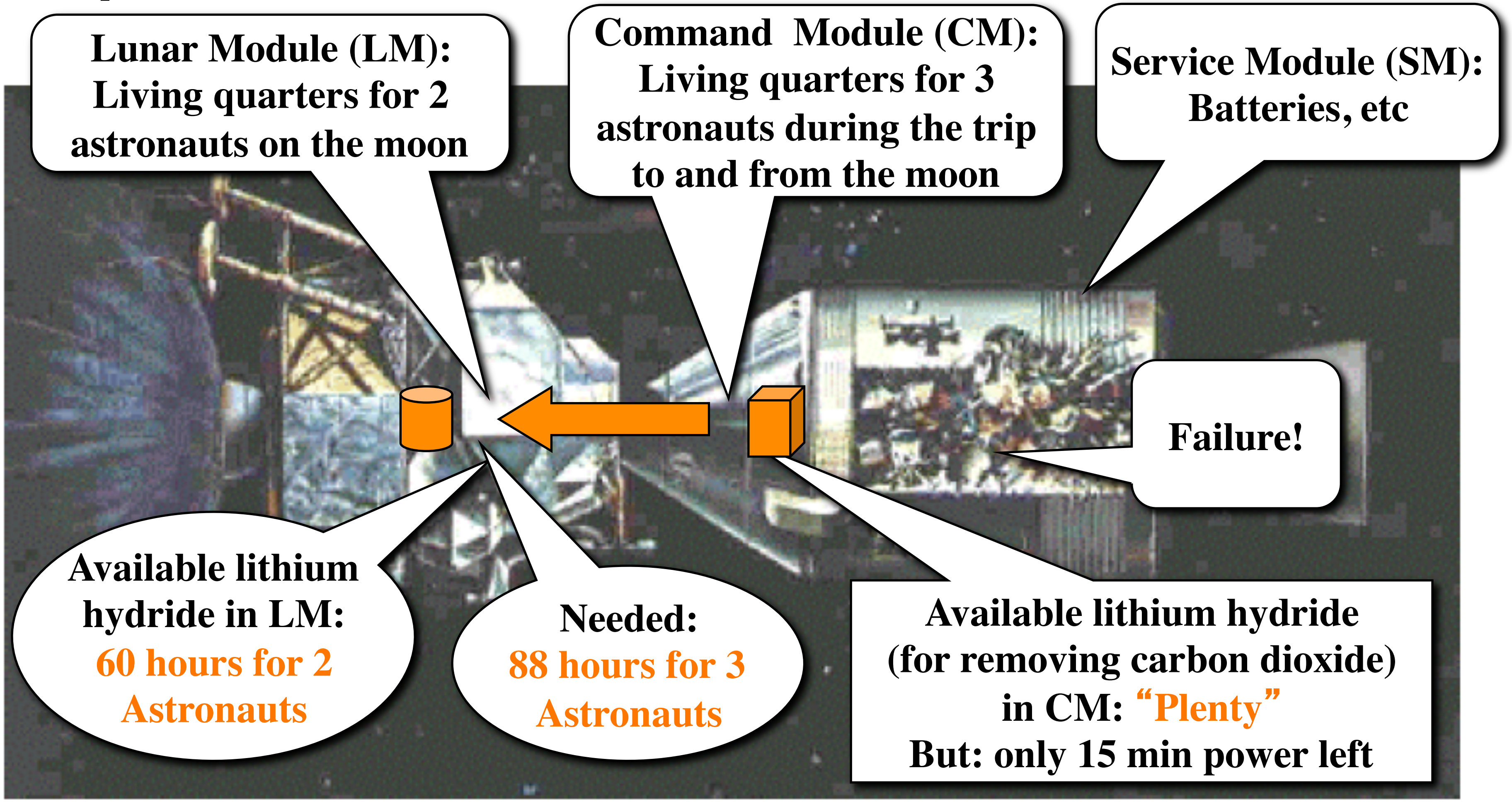
Summary

- Object design closes the gap between the requirements and the machine
- Object design adds details to the requirements analysis and makes implementation decisions
- Object design activities include:
  - ✓ Identification of Reuse
  - ✓ Identification of Inheritance and Delegation opportunities
  - ✓ Component selection
  - Interface specification (Next lecture)
  - Object model restructuring
  - Object model optimization

  Lectures on Mapping Models to Code
- Object design is documented in the Object Design Document (ODD).

Backup Slides

Apollo 13: “Houston, we’ve had a Problem!”



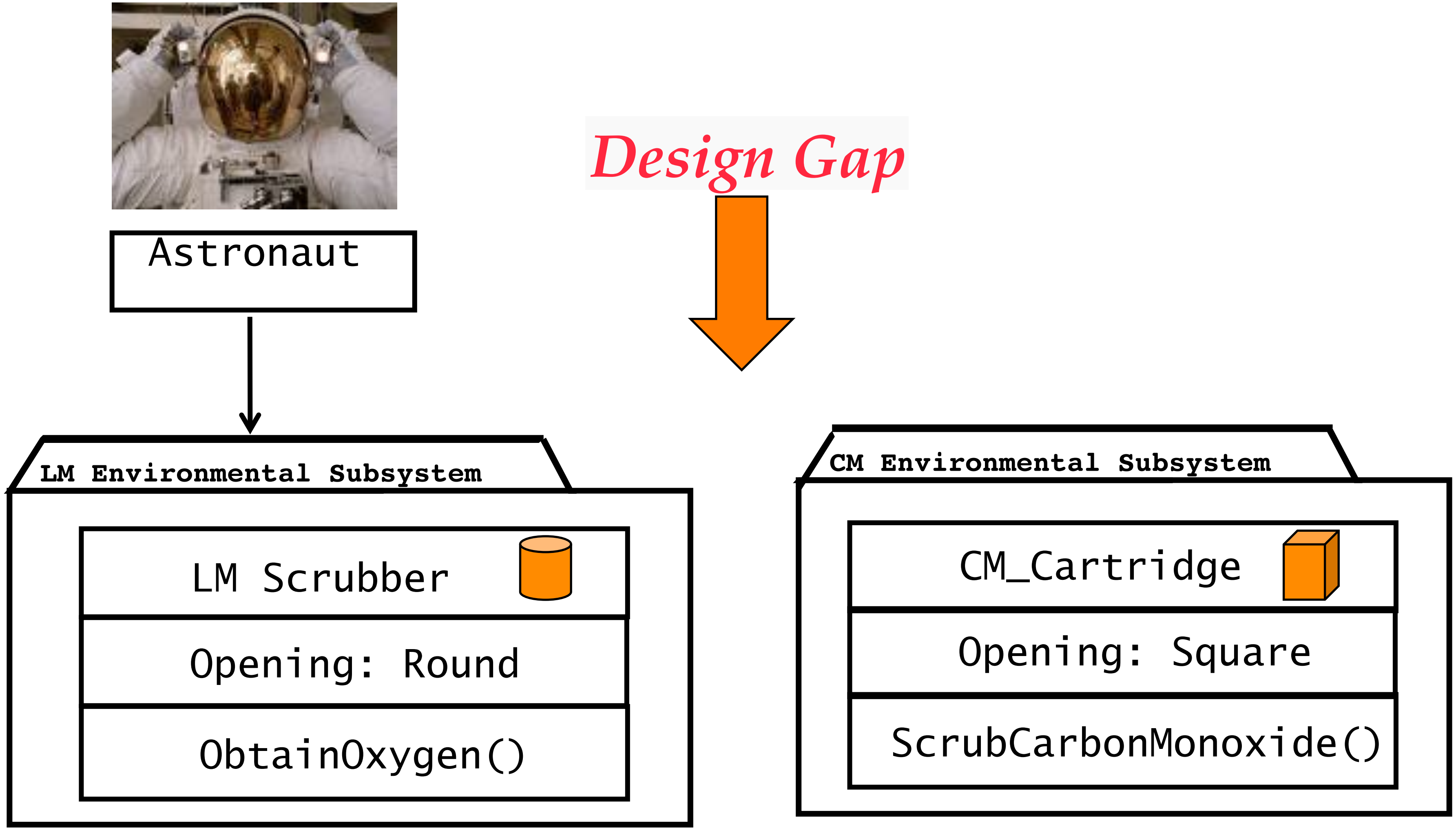
The LM was designed for 2 astronauts staying 2 days on the moon (4 man-days)

Redesign challenge: Can the LM be used for 12 man-days (2 1/2 days until reentry into Earth)?

Proposal: Use the lithium hydride cartridges from the CM to extend life in LM

Problem: Lithium hydride openings in CM were incompatible with scrubber in LM!

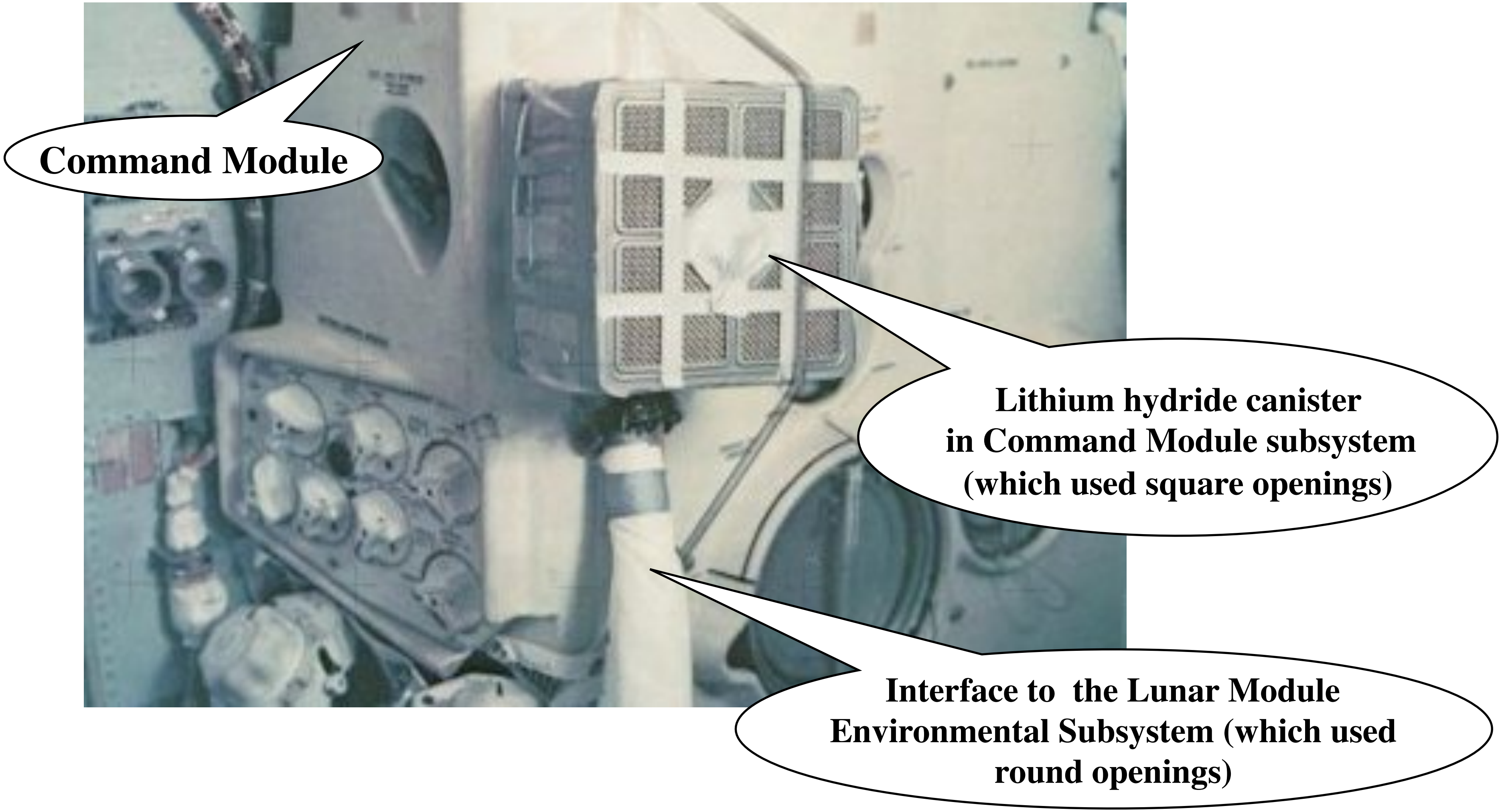
Incompatibility between LM Scrubber and CM Cartridge System



Apollo 13: "Fitting a square peg in a round hole"

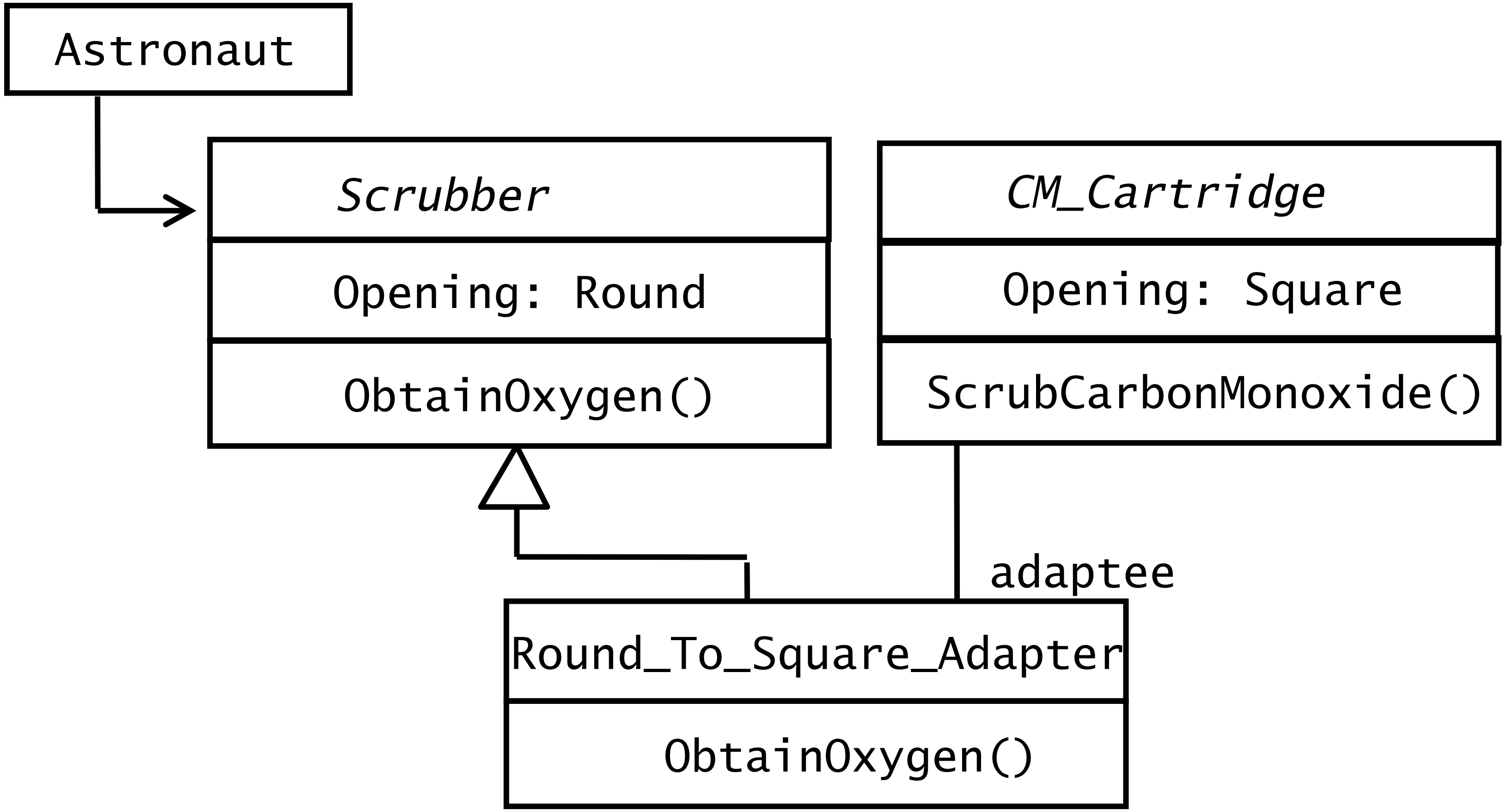
Source: http://www.hq.nasa.gov/office/pao/History/SP-350/ch-13-4.html

A Typical Object Design Challenge: Connecting Incompatible Components



Source: http://www.hq.nasa.gov/office/pao/History/SP-350/ch-13-4.html

Adapter for Scrubber in Lunar Module

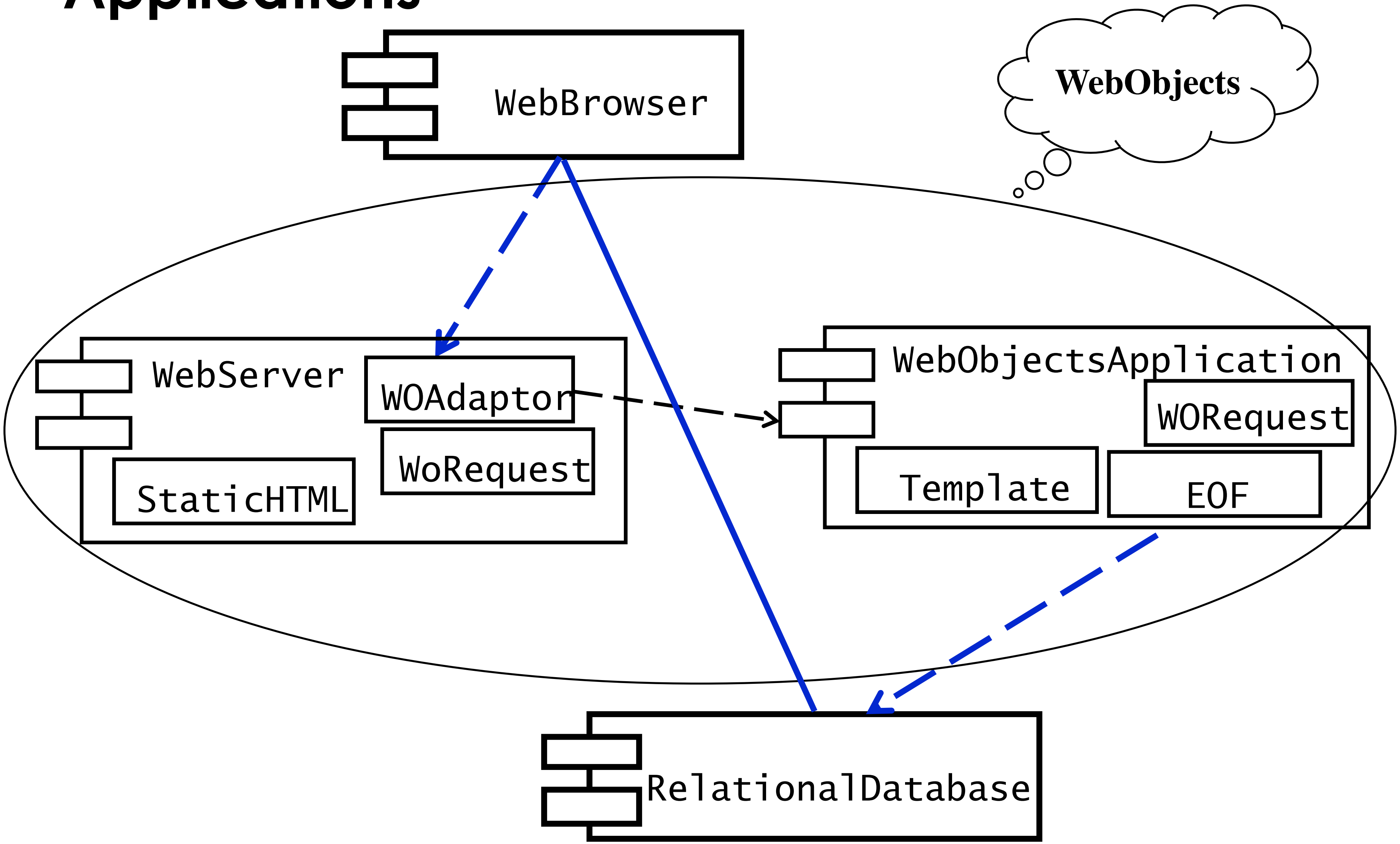


- Using a carbon monoxide scrubber (round opening) in the lunar module with square cartridges from the command module (square opening).

Reuse

- Main goal:
  - Reuse knowledge from previous experience to current problem
  - Reuse functionality already available
- Composition (also called Black Box Reuse)
  - New functionality is obtained by aggregation
  - The new object with more functionality is an aggregation of existing components
- Inheritance (also called White-box Reuse)
  - New functionality is obtained by inheritance.
- Three ways to get new functionality:
  - Implementation inheritance
  - Interface inheritance
  - Delegation

Example: Framework for Building Web Applications

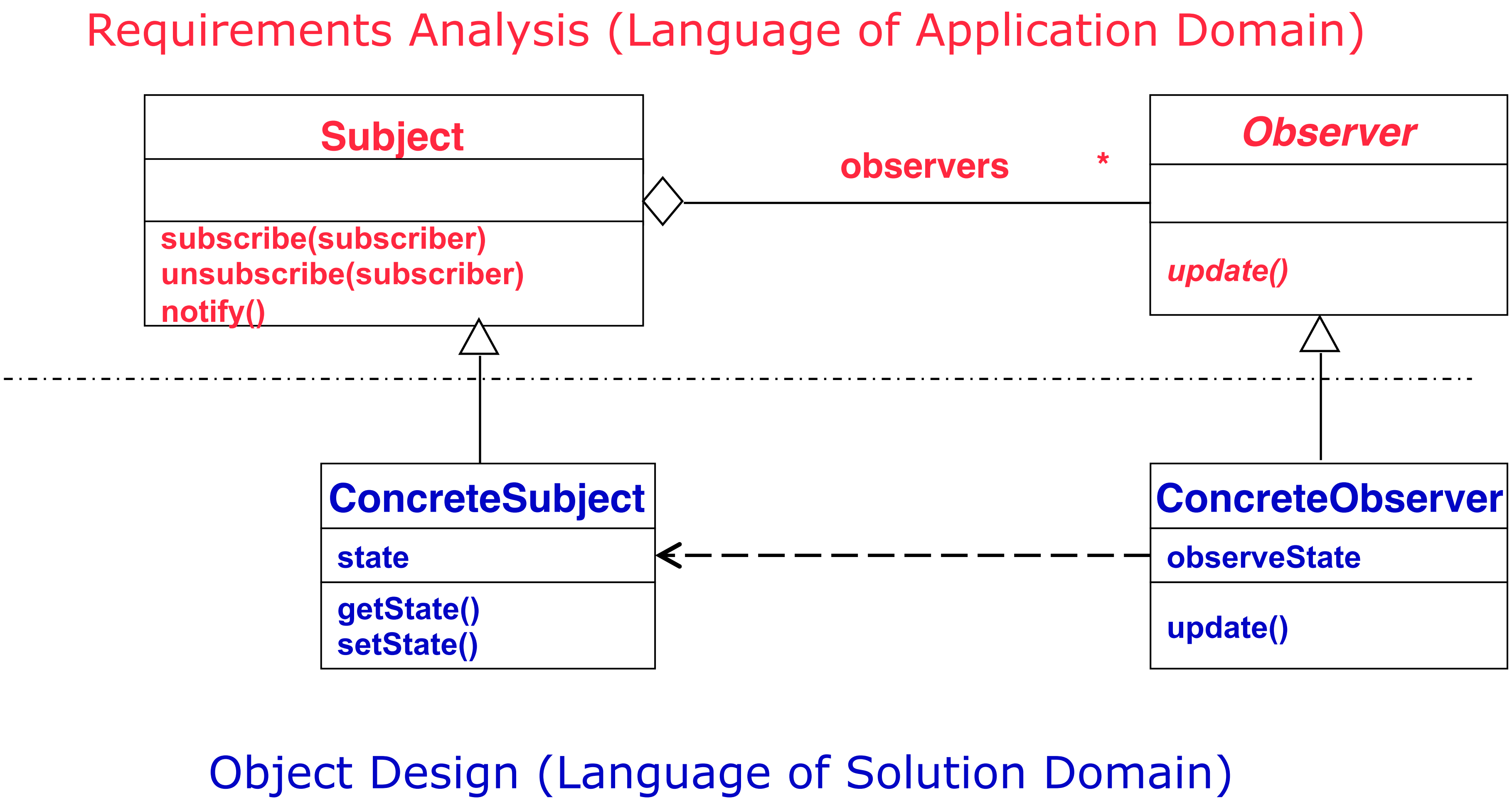


Customization Projects are like Advanced Jigsaw Puzzles

Design Patterns!

Source http://www.puzzlehouse.com/_

Application Domain vs Solution Domain Objects



White-box and Black-box Frameworks

- White-box frameworks:
  - Extensibility is achieved through *inheritance* and dynamic binding
  - Existing functionality is extended by subclassing framework base classes and overwriting specific methods designed to be overwritten(so-called hook methods)
- Black-box frameworks:
  - Extensibility is achieved by defining interfaces for components that can be plugged into the framework.
  - Existing functionality is reused by defining components that conform to a particular interface
  - These components are integrated with the framework via *delegation*.