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

- 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

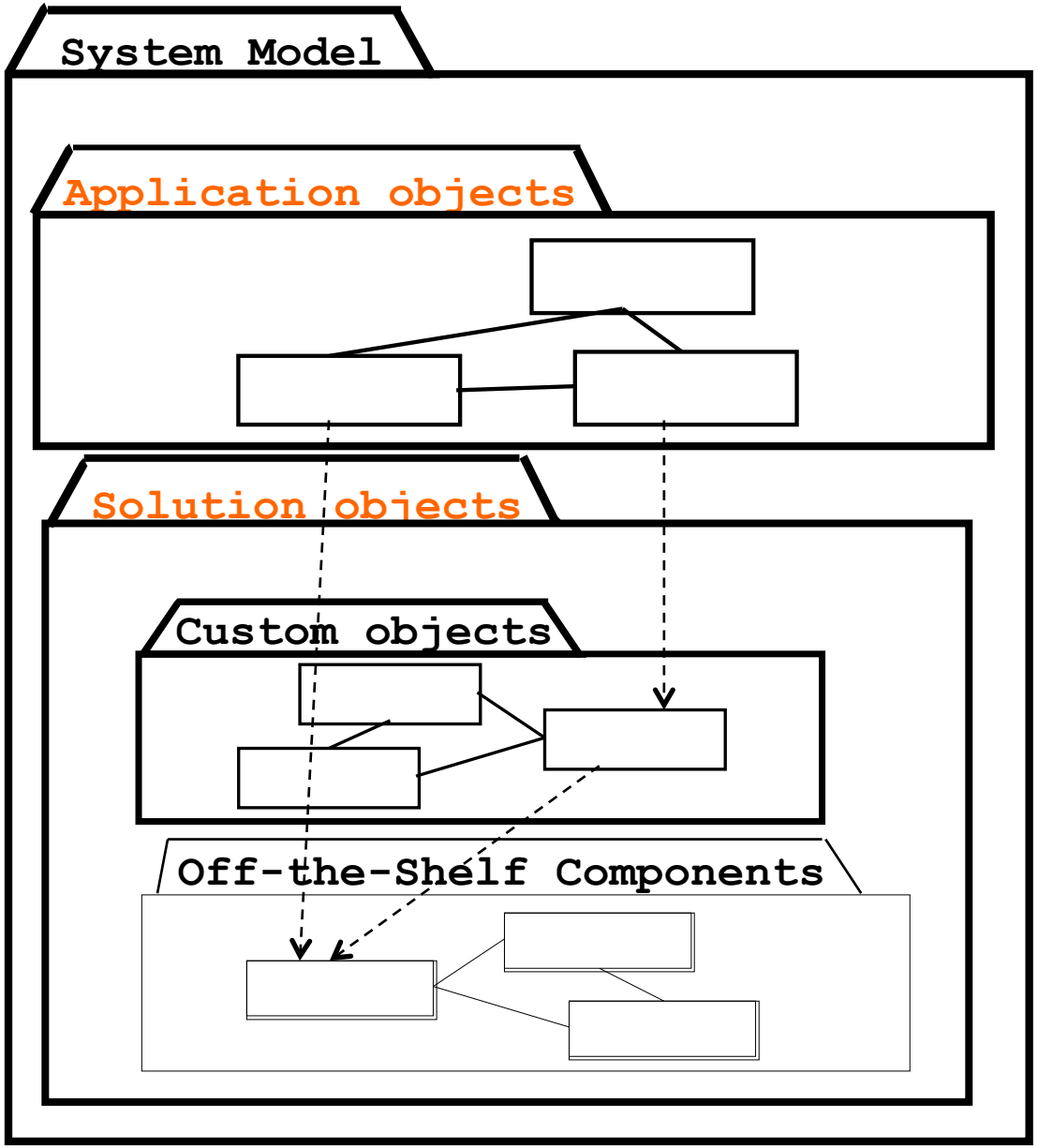
Object Design

- 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



Problem

Analysis

Design

Object Design

System Design

Existing Machine

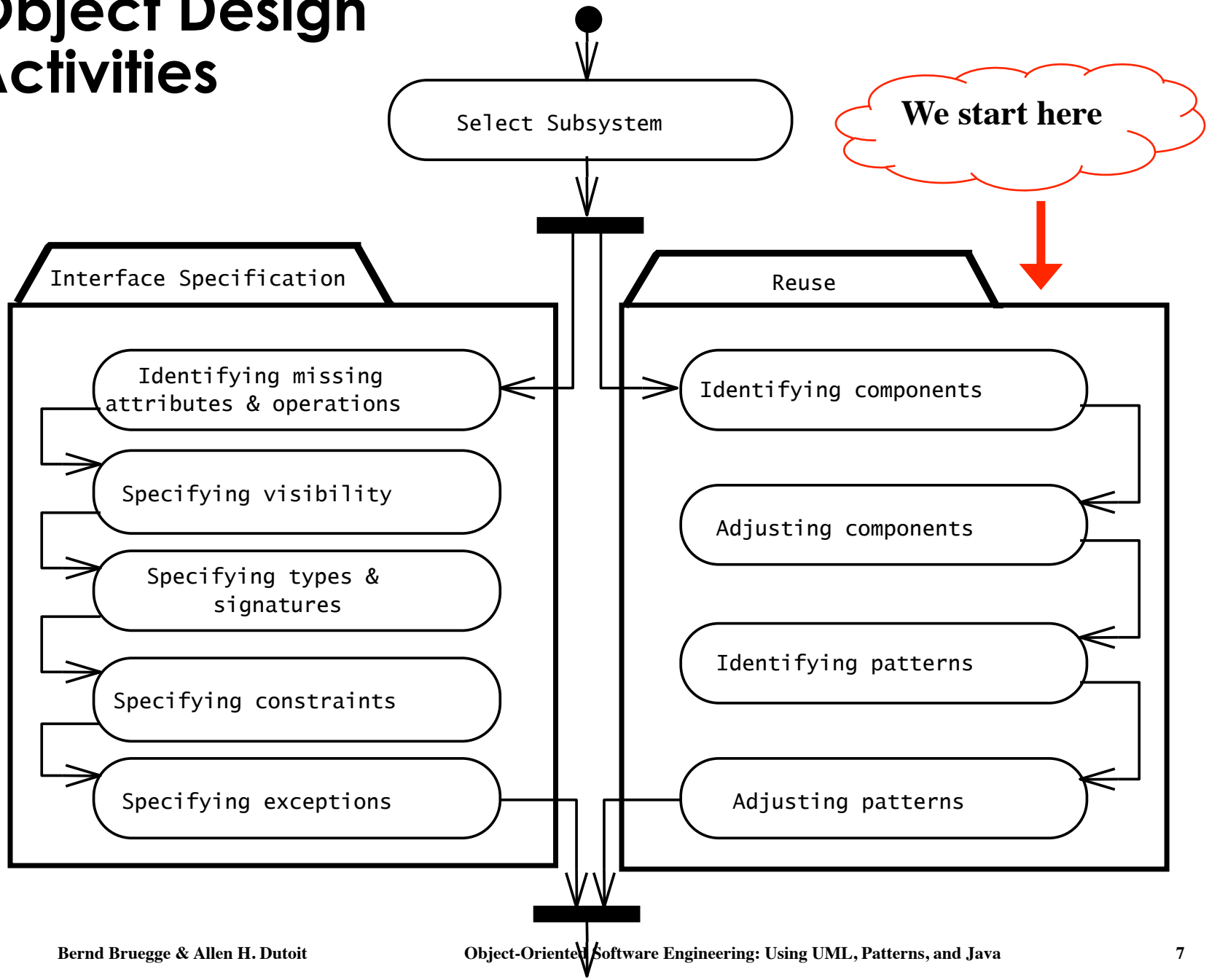
Object Design Activities consists of 4 Activities

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.

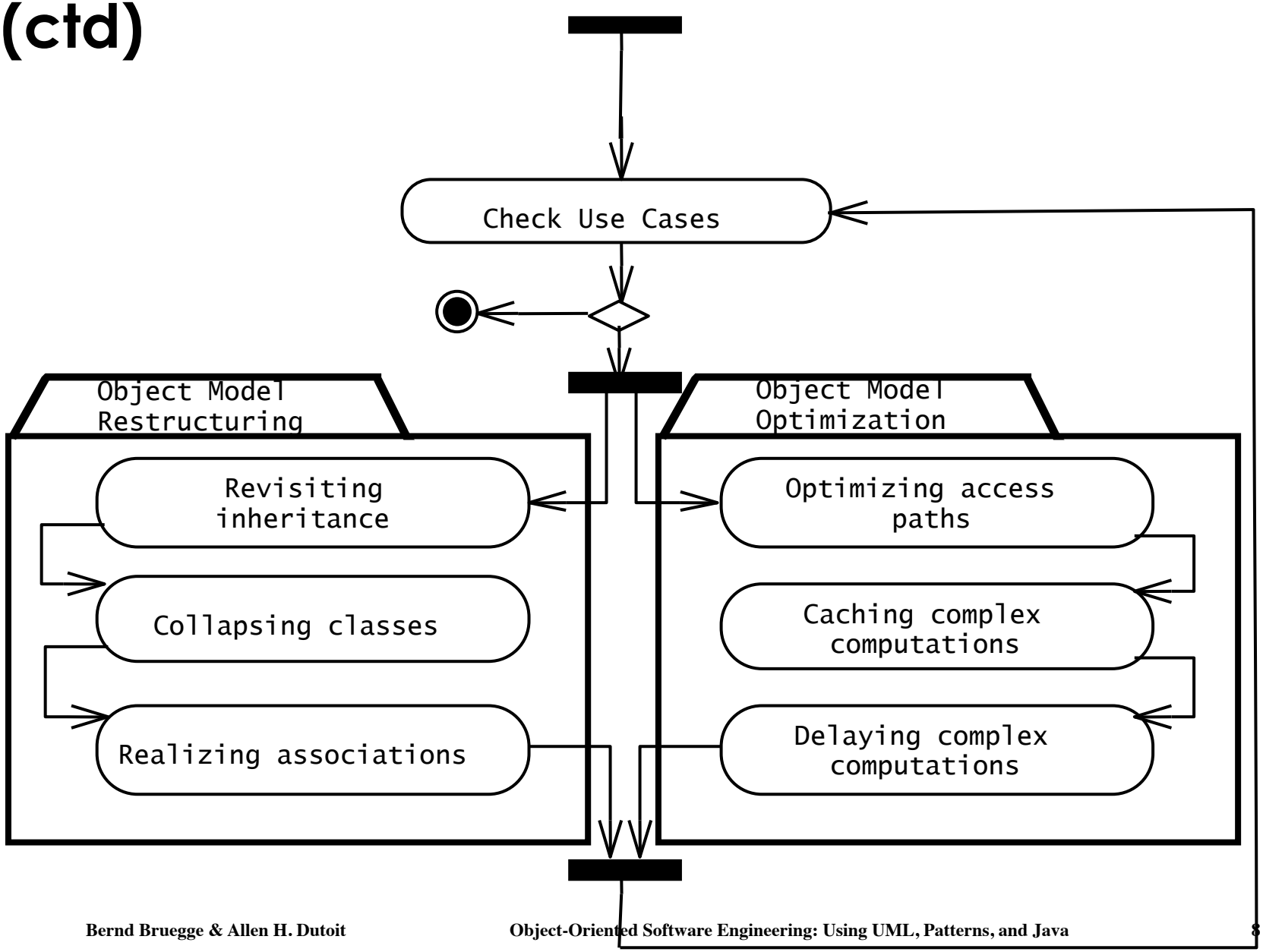
Focus on Reuse and Specification

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.

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*

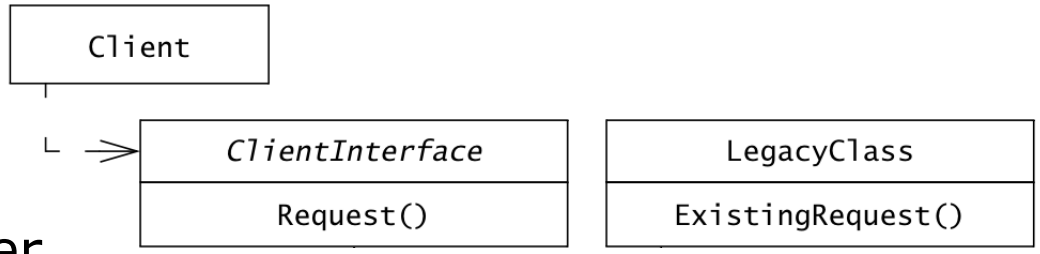
- Learning how to design starts by studying other designs.

Example: 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.

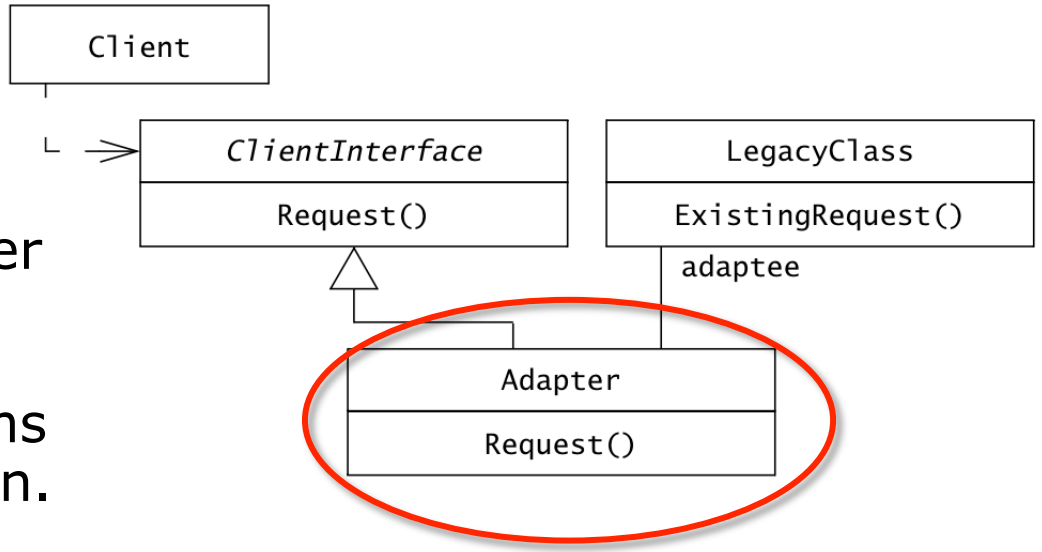
Example of design pattern

- Name: **Adapter Design Pattern**
- Problem Description
  - Convert the interface of a legacy class into a different interface expected by the client, so that the client and the 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.



Example of design pattern

- Name: **Adapter Design Pattern**
- Problem Description
  - Convert the interface of a legacy class into a different interface expected by the client, so that the client and the 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.



What makes a design modifiable?

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

Where are we?

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

Types of Reuse

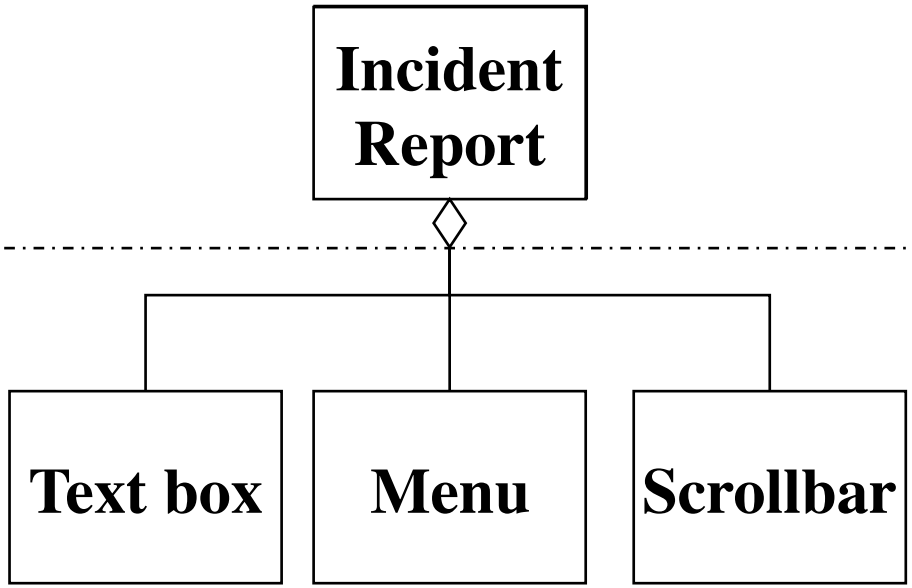
Whitebox and blackbox reuse

Types of reuse: white/black box reuse

- Main goals:
  - Reuse functionality already available
  - Reuse 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 (black box reuse)

Requirements Analysis
(Language of Application Domain)



Object Design
(Language of Solution Domain)

White Box and Black Box Reuse

- What is needed for white/black box reuse

- Black box reuse (composition)
  - models and designs 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.
- White box reuse (inheritance)
  - development artifacts must be available
    - (analysis model, system design, object design, source code)

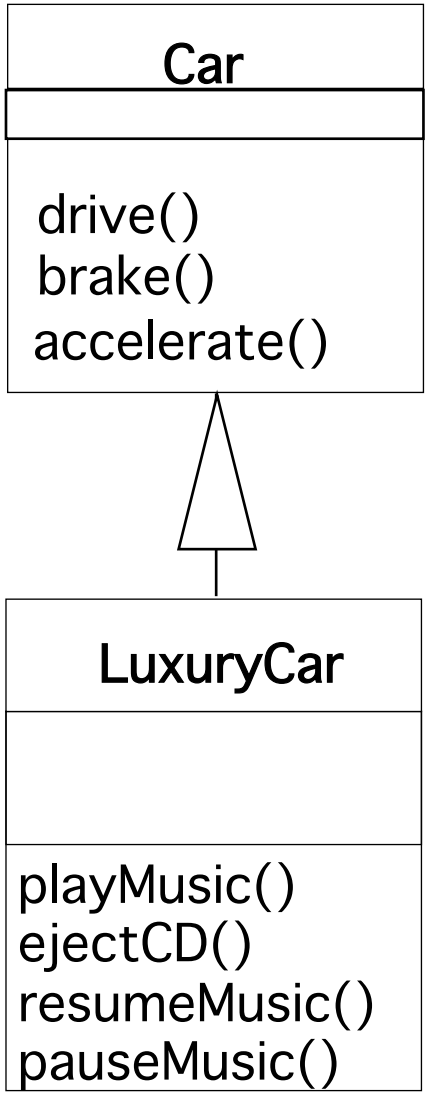
Types of Whitebox Reuse

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

The use of Inheritance

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

Example of Inheritance for Taxonomy



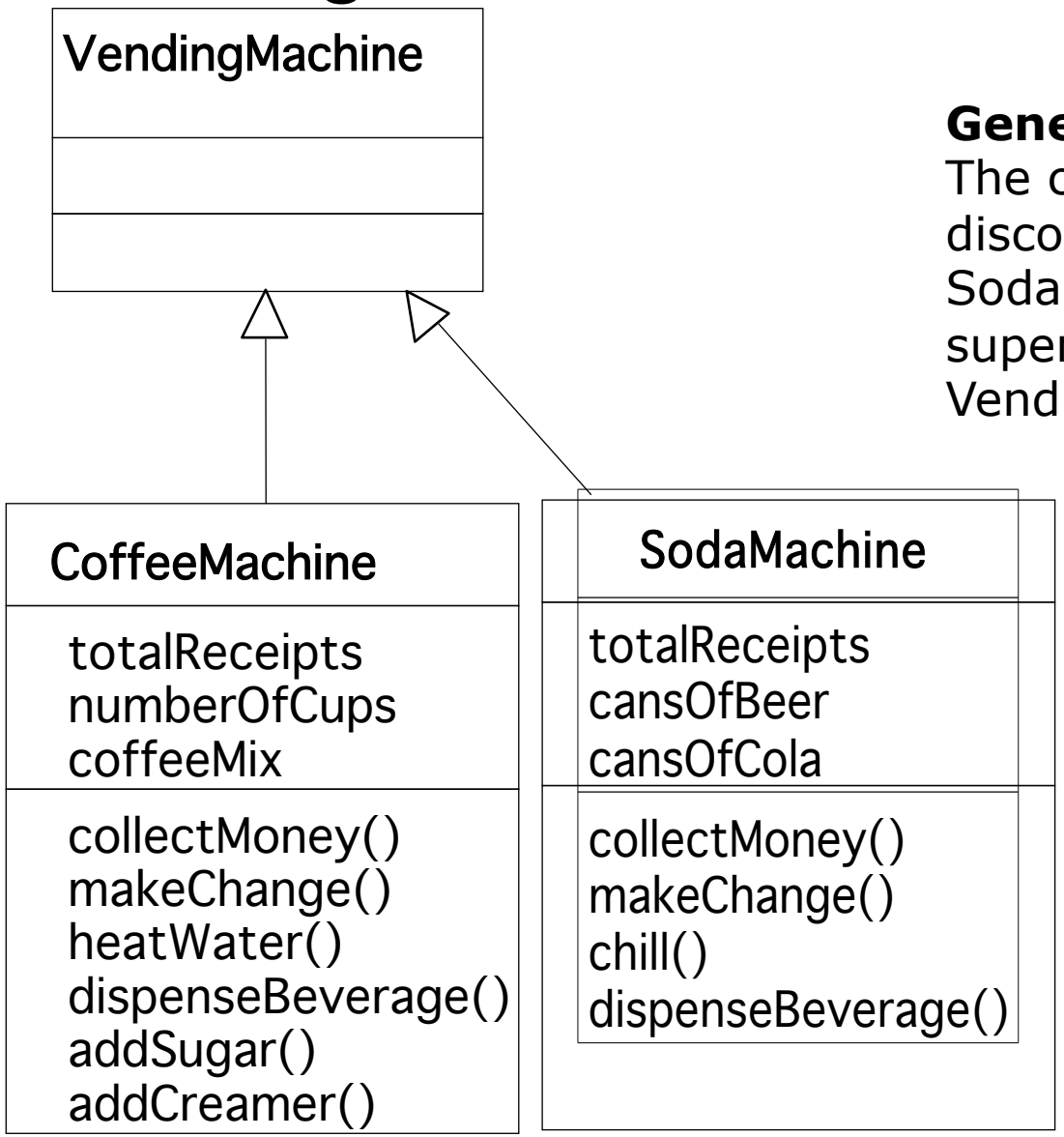
Superclass:

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

Subclass:

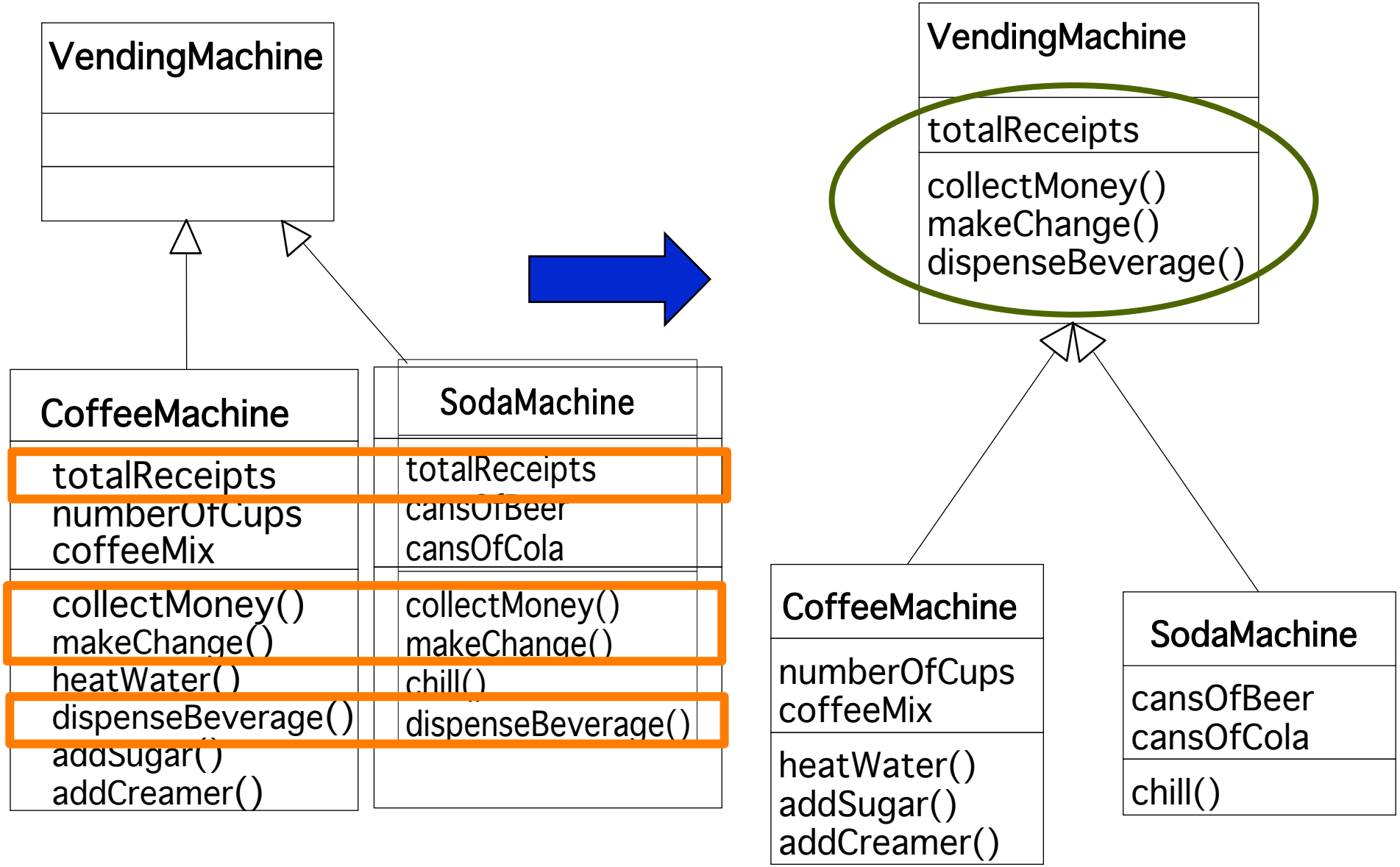
```
public class LuxuryCar extends Car
{
    public void playMusic() {…}
    public void ejectCD() {…}
    public void resumeMusic() {…}
    public void pauseMusic() {…}
}
```

Generalization Example: Modeling Vending Machines



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

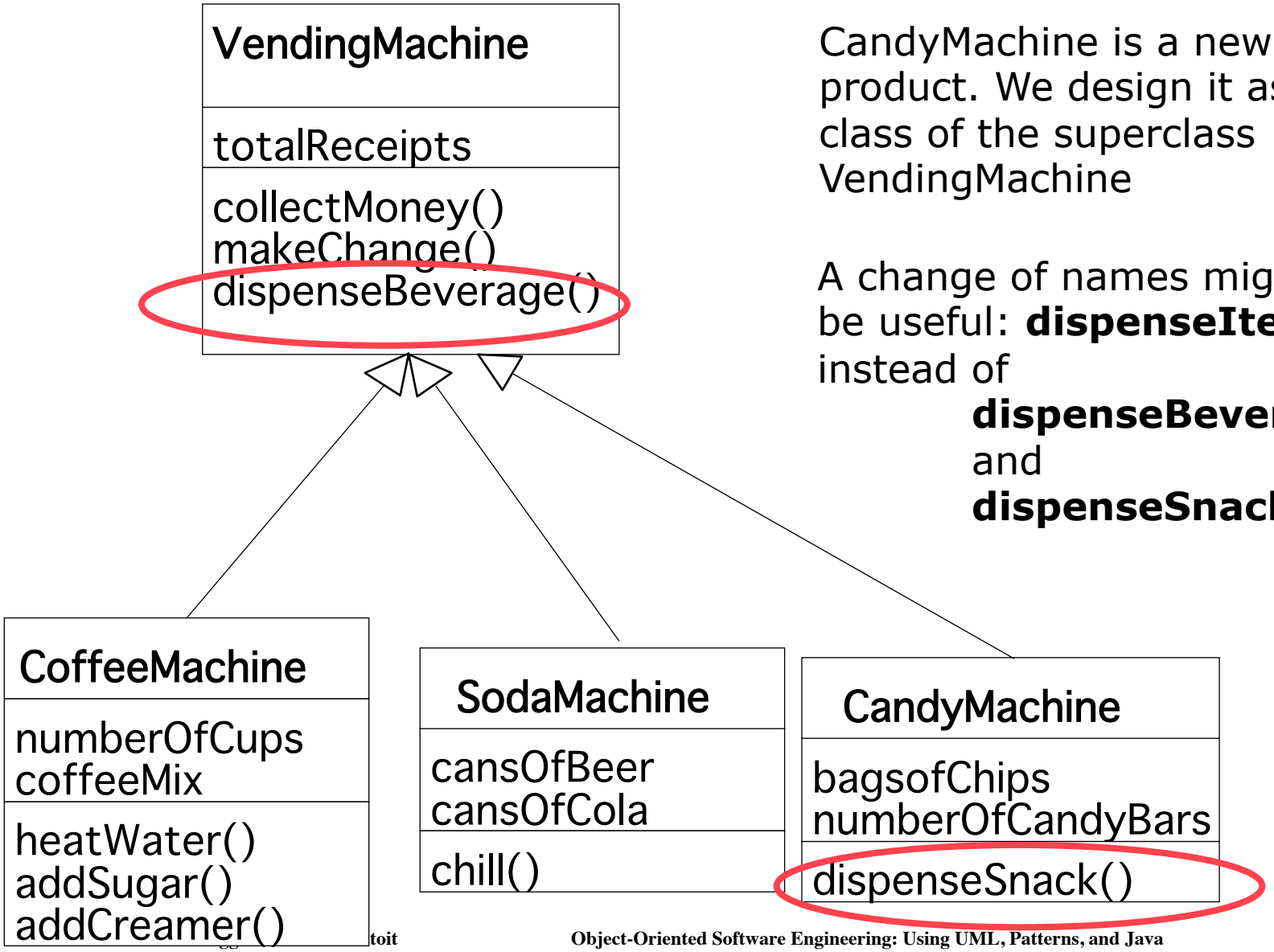
Generalizing often leads to Restructuring



Specialization

- Specialization occurs, when we find a **subclass** that is similar to an existing (mother) class
- 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.

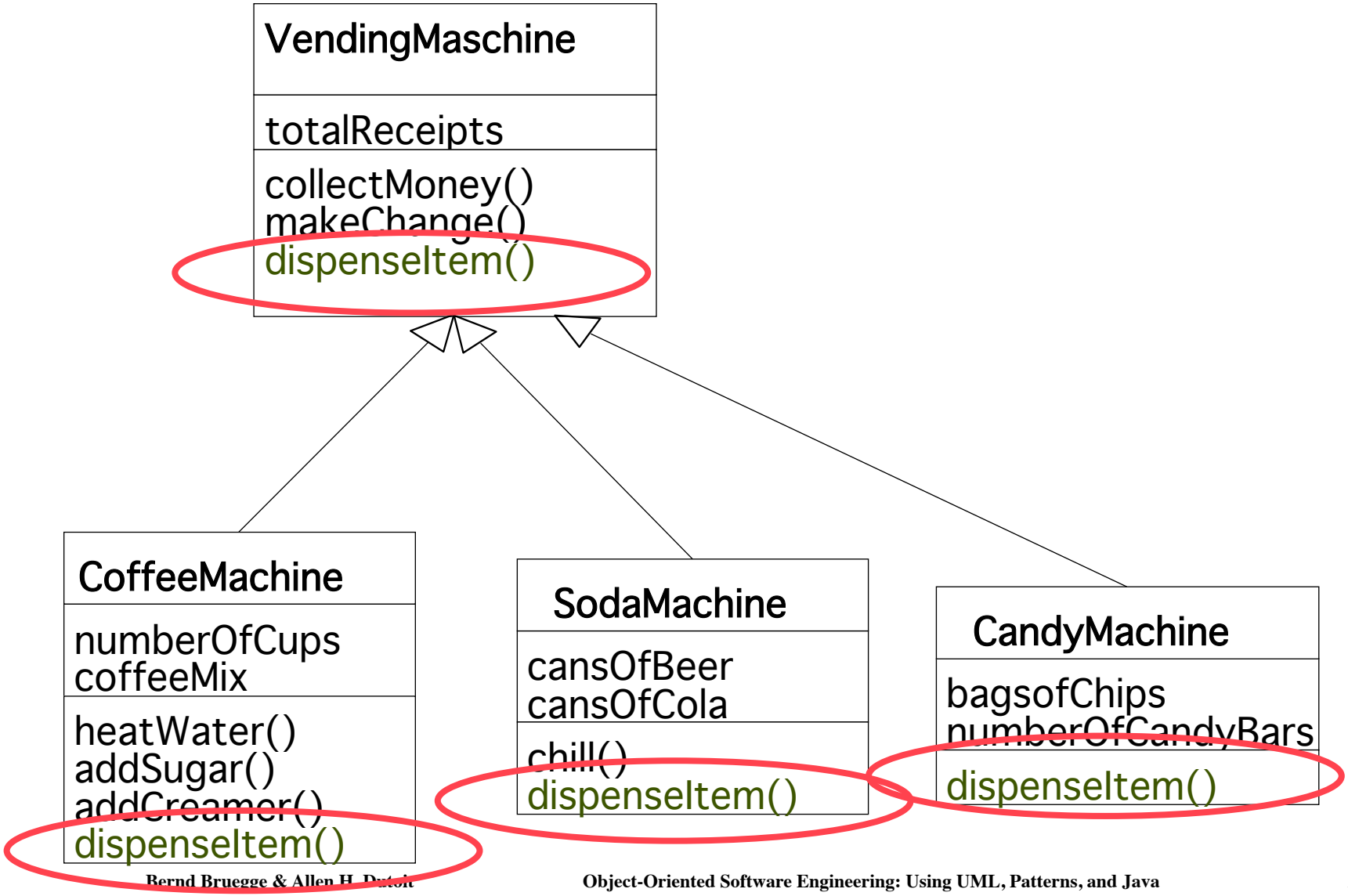
Another Example of a Specialization



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)



Implementation Inheritance and Specification Inheritance

There are two different types of inheritance:

- Implementation inheritance
  - Also called class inheritance
  - Goal:
    - Extend an applications' functionality by reusing functionality from the super class
    - Inherit from an existing (concrete) 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

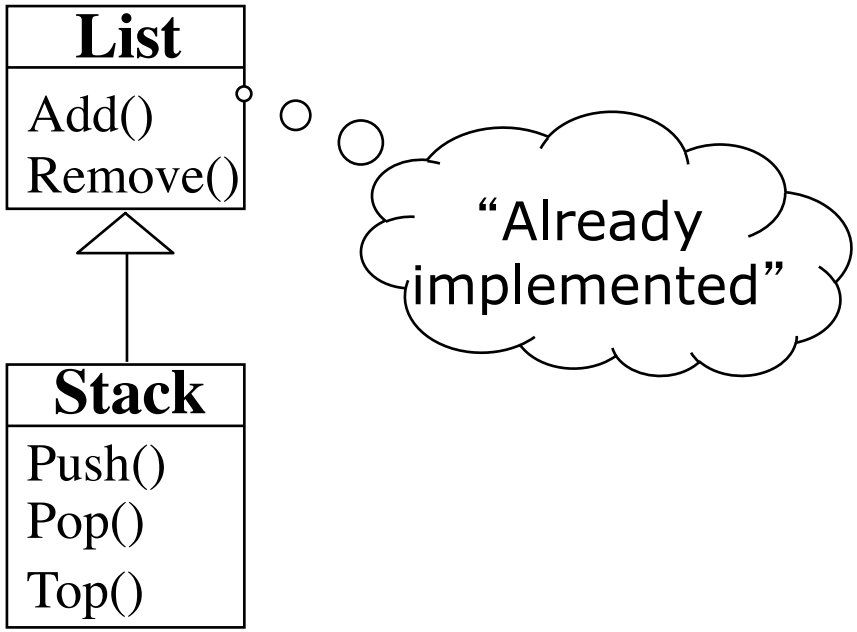
	Interface (of superclass)	Implementations of methods (of superclass)
Implementation Inheritance	Inherited	Inherited
Specification Inheritance	Inherited	NOT inherited

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
- I can define the Stack class as a subclass of the List class and implement 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

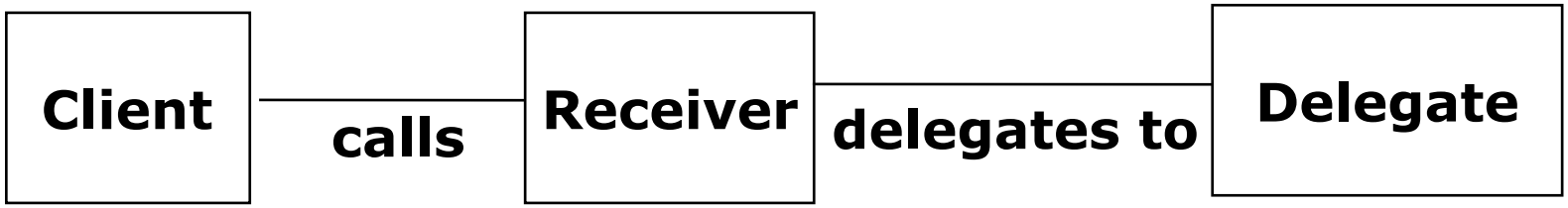
- Inheritance: Extending a Base class by a new operation or overwriting an operation

- Delegation: Catching an operation and sending it to another object

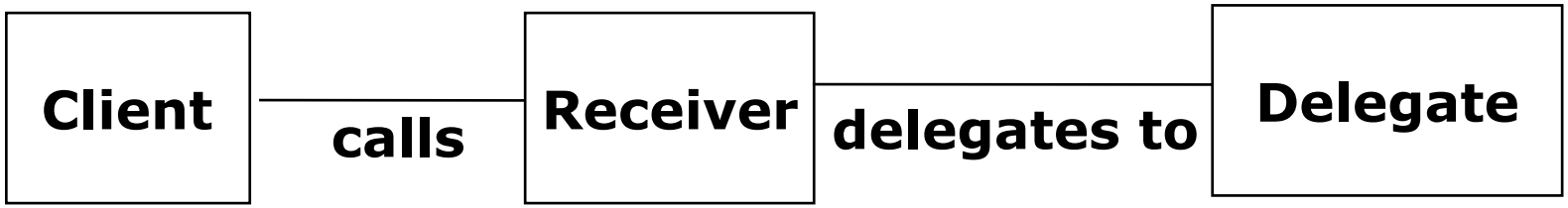
Delegation

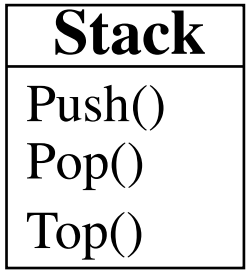
- Delegation is a way of making composition as powerful for reuse as inheritance
- In delegation two objects are involved in handling a request from a Client

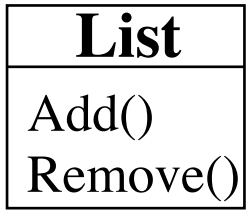
  •The Receiver object delegates operations to the Delegate object
  •The Receiver object makes sure, that the Client does not misuse the Delegate object.



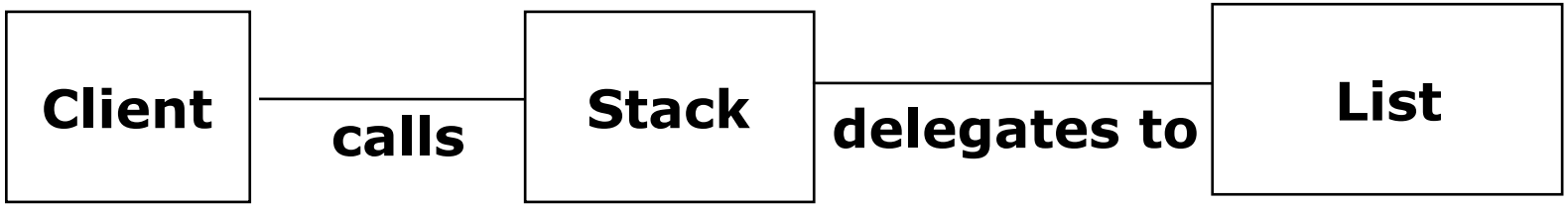
Solution for the stack class





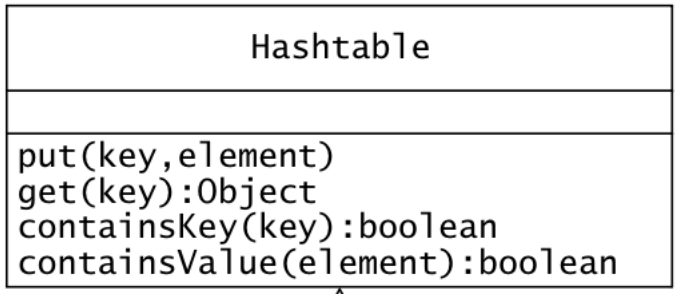


Push(element) -> Add(0, element)



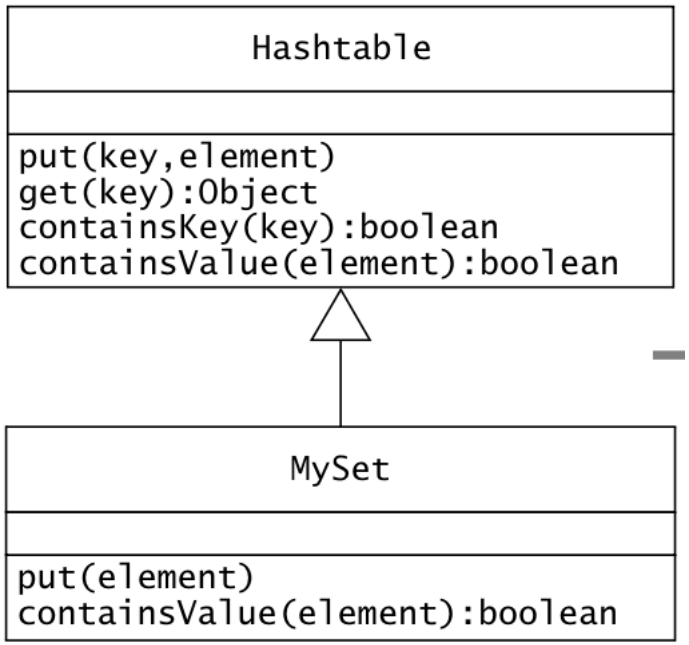
Exercise: define a new Set class

- Suppose Java has not the Set class
- we can define that by reusing the Hashtable class



- Hashtable class from Java Oracle documentation:
  - *This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value.*

Solution with implementation inheritance



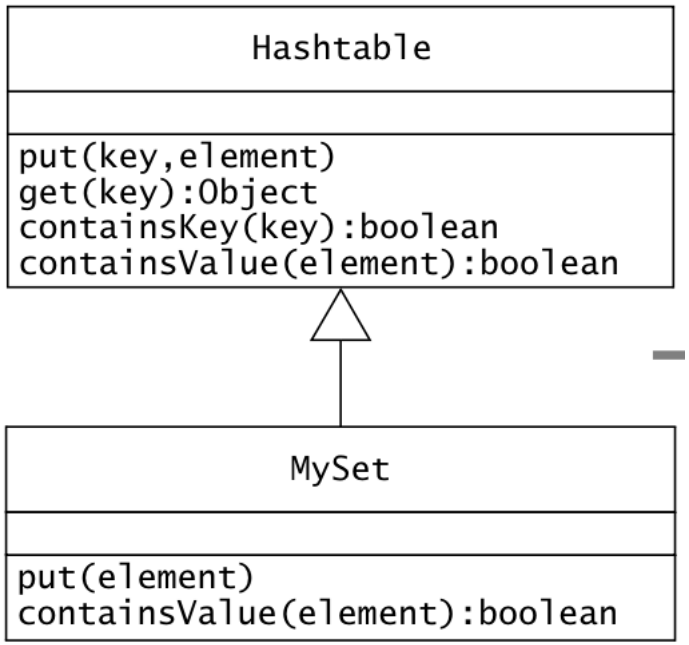

```
/* Implementation of MySet using
inheritance */
class MySet extends Hashtable {
    /* Constructor omitted */
    MySet() {
    }

    void put(Object element) {
        if (!containsKey(element)){
            put(element, this);
        }
    }
    boolean containsValue(Object
            element){
        return containsKey(element);
    }
    /* Other methods omitted */
}
```

In the Hashtable class:

- boolean containsKey(Object key)
  - Tests if the specified object is a key in this hashtable.
- boolean containsValue(Object value)
  - Returns true if this hashtable maps one or more keys to this value.

Solution with implementation inheritance



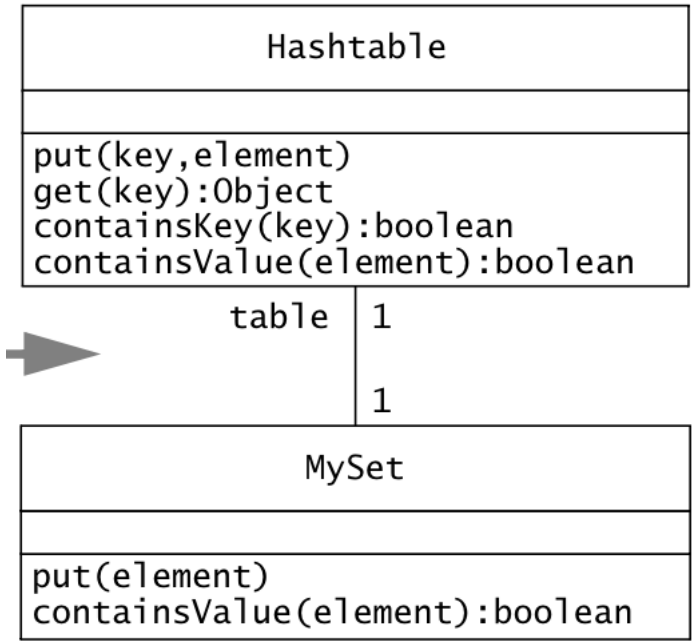

```
/* Implementation of MySet using
inheritance */
class MySet extends Hashtable {
    /* Constructor omitted */
    MySet() {
    }

    void put(Object element) {
        if (!containsKey(element)){
            put(element, this);
        }
    }
    boolean containsValue(Object
            element){
        return containsKey(element);
    }
    /* Other methods omitted */
}
```

- Problems:
  - *containsValue* (overriden in the subclass) provides the same behavior as *containsKey*

    - That is not intuitive!
    - A programmer may use both and make it difficult to maintain the class

Solution with delegation




```
/* Implementation of MySet using
delegation */
class MySet {
    private Hashtable table;
    MySet() {
        table = Hashtable();
    }
    void put(Object element) {
        if (!containsValue(element)){
            table.put(element,this);
        }
    }
    boolean containsValue(Object
            element) {
        return
        (table.containsKey(element));
    }
    /* Other methods omitted */
}
```

- The only significant change is the private field table and its initialization in the MySet() constructor.
- This addresses both problems we mentioned before

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).

The Liskov Substitution Principle for specification inheritance

- The Liskov Substitution Principle [Liskov, 1988] provides a formal definition for **specification inheritance**.
- It essentially states that, if a client code uses the methods provided by a superclass, then developers should be able to add new subclasses without having to change the client code.

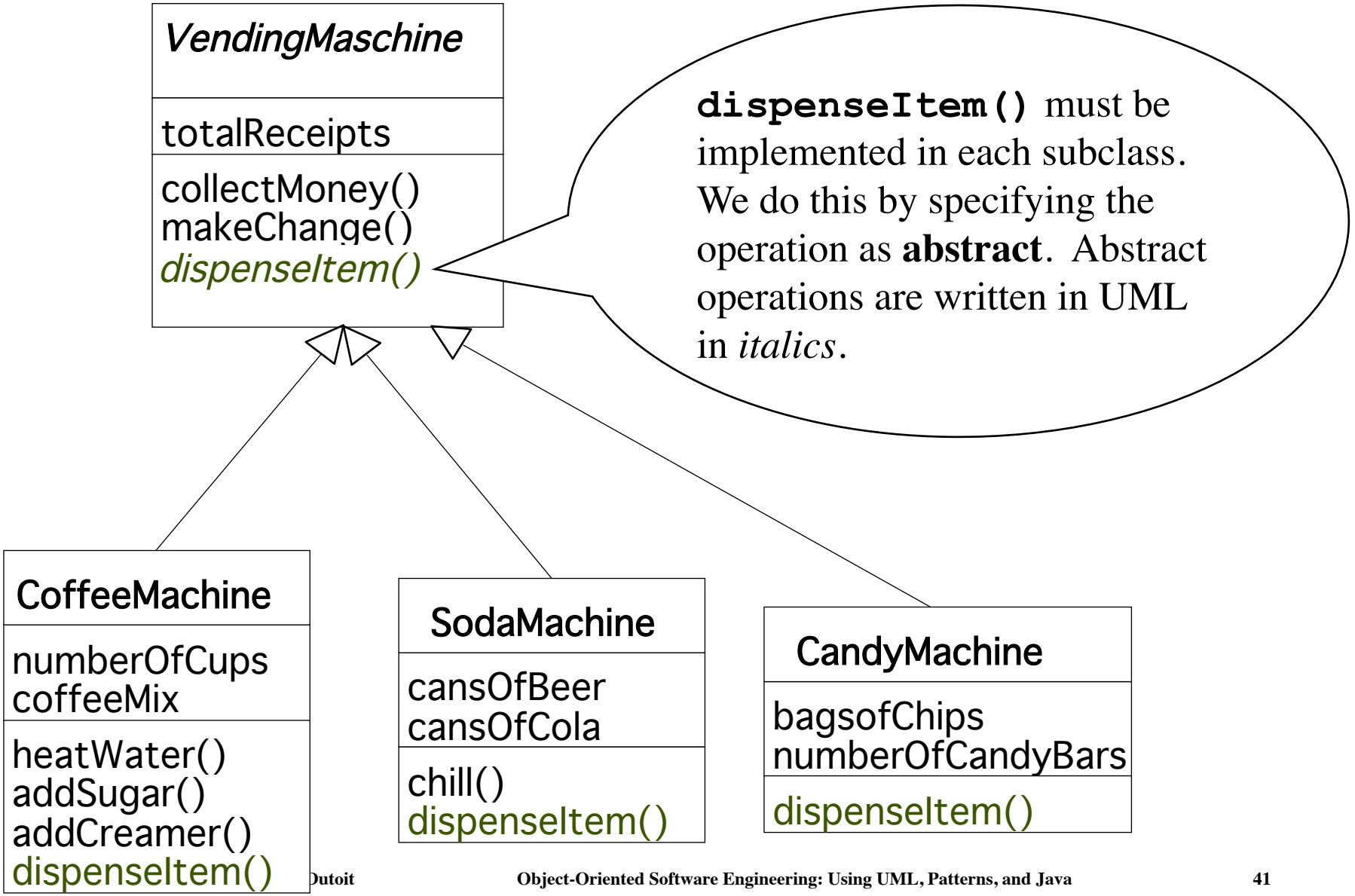
- Liskov Substitution Principle
  - If an object of type S can be substituted in all the places where an object of type T is expected, then S is a subtype of T.

- Interpretation
  - In other words, a method written in terms of a superclass T must be able to use instances of any subclass of T without knowing whether the instances are of a subclass.
- An inheritance relationship that complies with the Liskov Substitution Principle is called **strict inheritance**.

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.

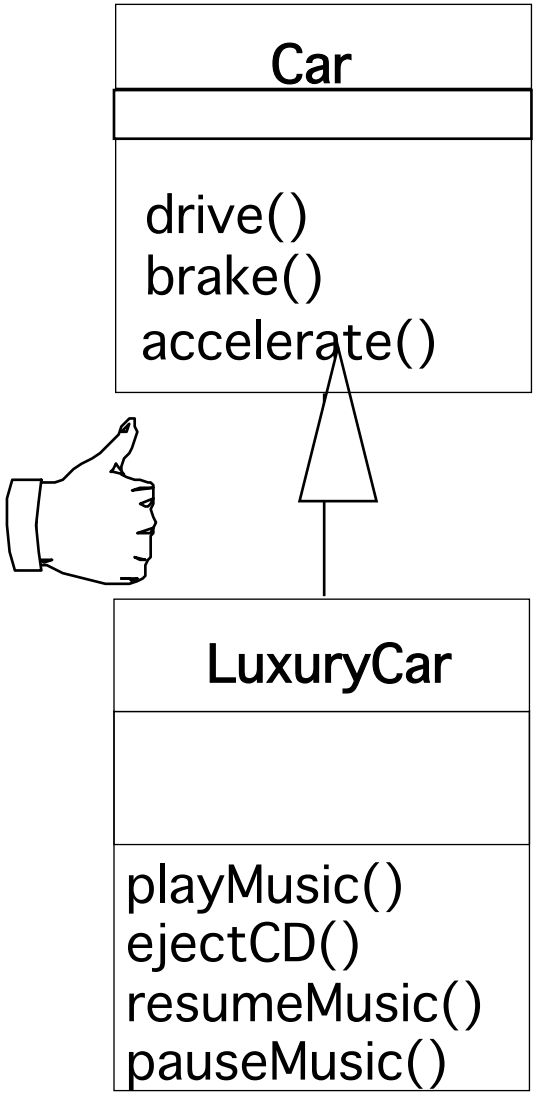
Example of an Abstract Operation



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()
{…}
}
```

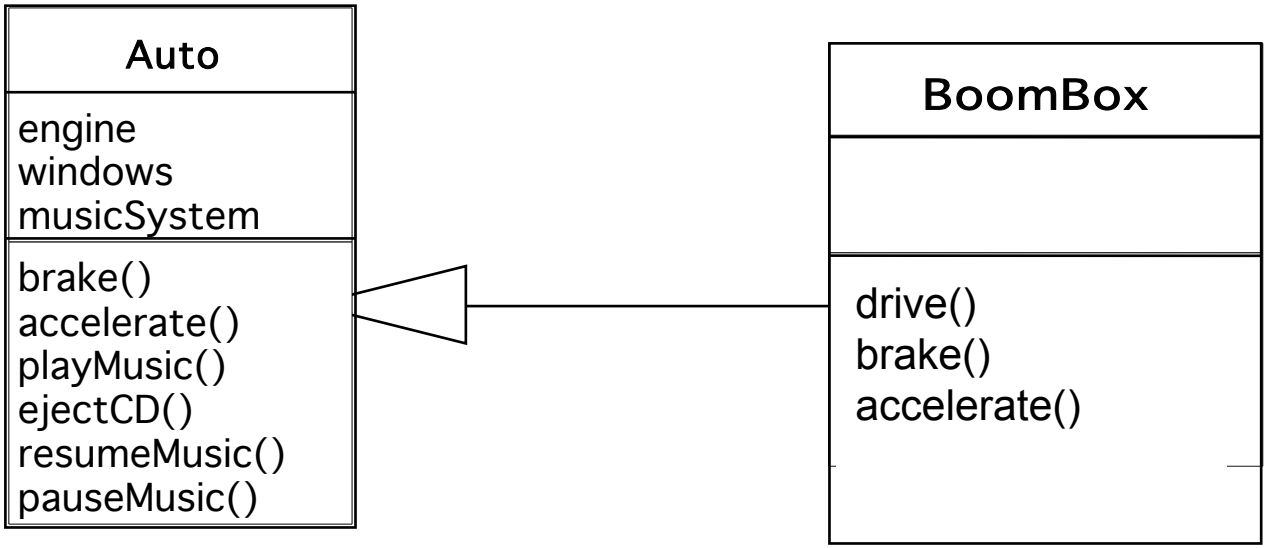
Subclass:

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

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 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 it 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 is 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.

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)

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).

Modeling of the Real World

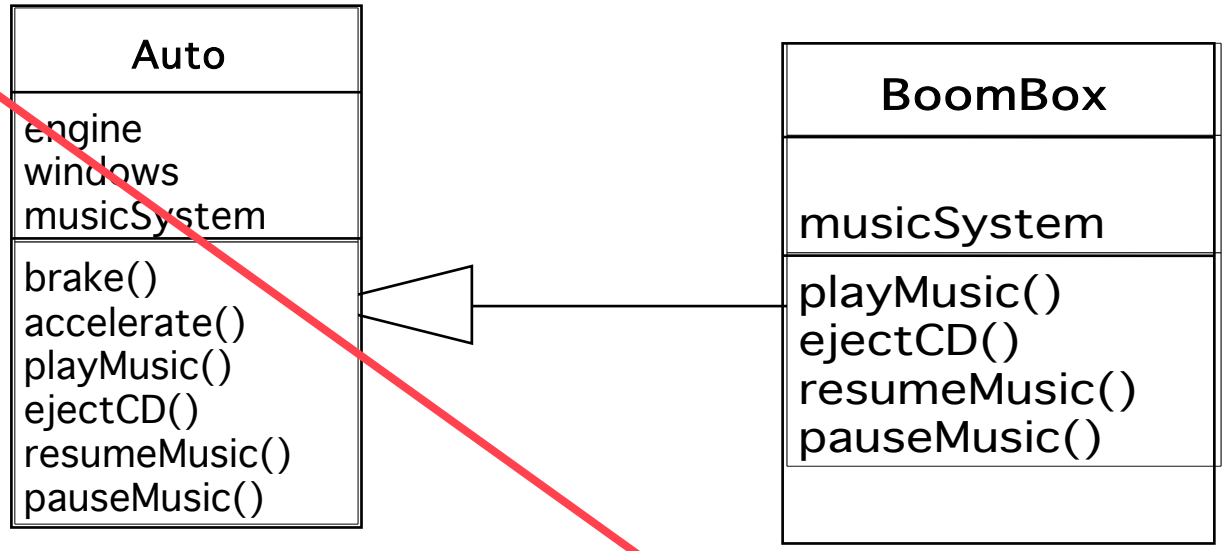
- Design knowledge such as the adapter pattern complements application domain knowledge and solution domain knowledge
- Modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's
- There is a need for reusable and extendable ("flexible") designs.

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

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() {};
}
```

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.