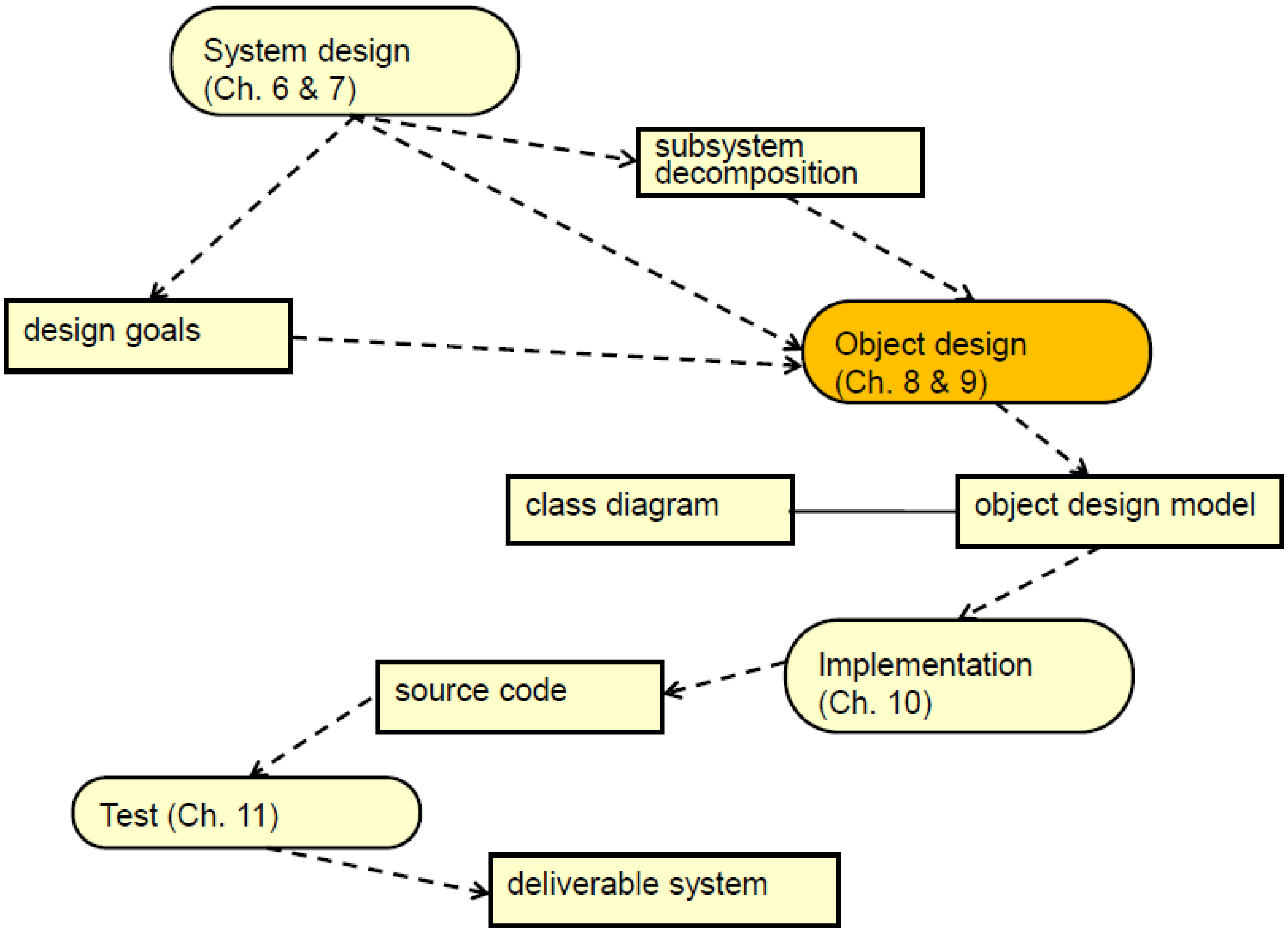
Chapter 8, Object Design: Reuse and Patterns

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.

Object design Activities



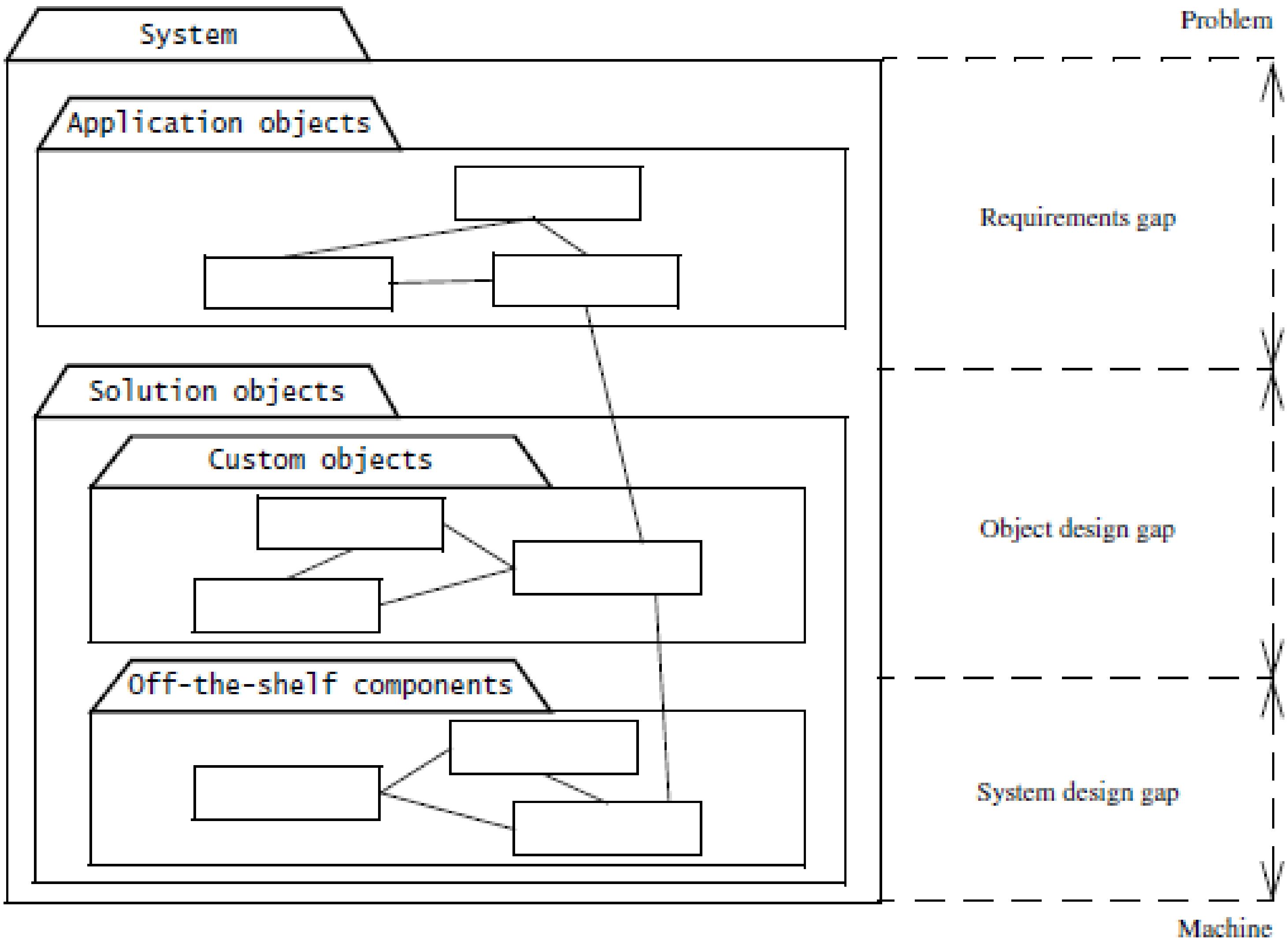
Terminology: Naming of Design Activities

Methodology: Object-oriented software engineering (OOSE)

- *System Design*
  - Decomposition into subsystems, etc

- *Object Design*
  - Data structures and algorithms chosen
- *Implementation*
  - Implementation language is chosen

Design means “Closing the Gap”



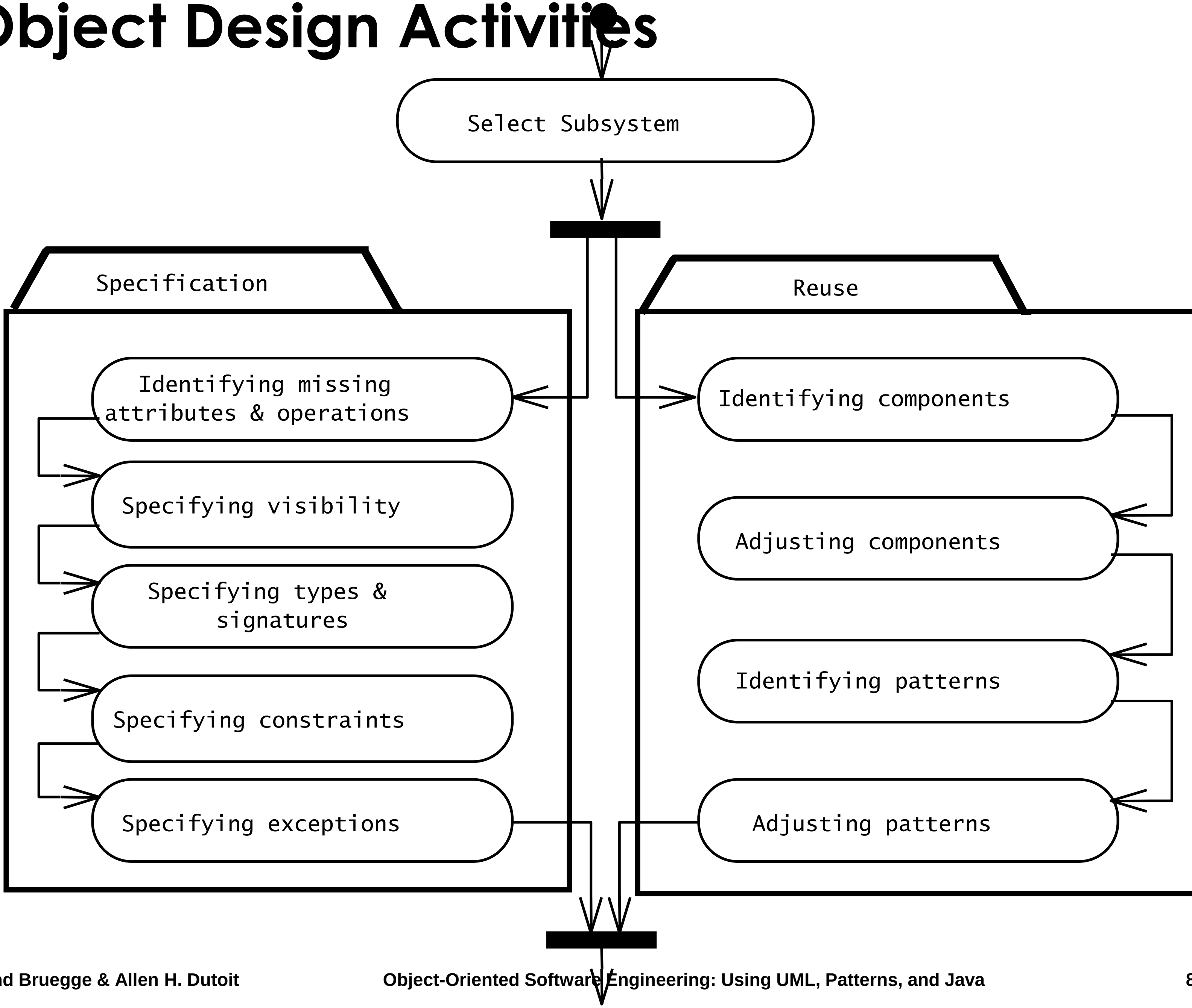
Typical Activities

- Full definition of associations
- Full definition of classes
- Choosing algorithms and data structures
- Identifying possibilities of reuse
- Optimization
- Increase of inheritance
- Decision on control
- Packaging

Object Design consists of 4 Activities

1. Reuse: Identification of existing solutions
   - Use of inheritance
   - Selecting Off-the-shelf components and additional solution objects
   - Design patterns, class libraries and framework
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 Activities



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:

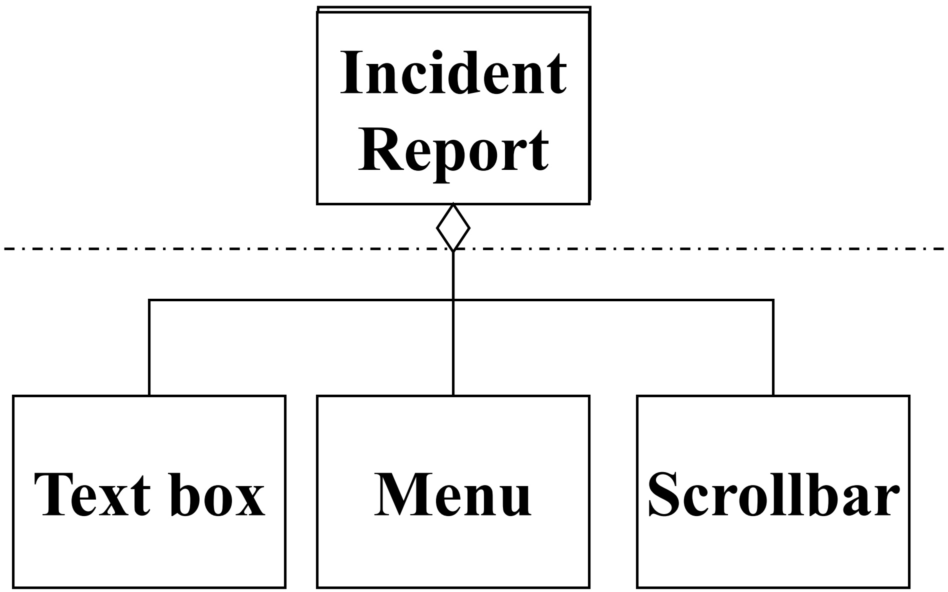
The design gap is filled with available components ("0 % coding").

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

=> Design with standard components.

Identification of new Objects during Object Design

Requirements Analysis
(Language of Application
Domain)



Object Design
(Language of Solution
Domain)

Modeling of the Real World

- 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 flexible designs

Reuse of Code

- I have a list, but my customer would like 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 stero system?

Reuse of existing classes

- I have an implementation for a list of elements of Typ 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 another project
  - Can I add it as a subsystem to my e-mail program which I purchased from a vendor (replacing the vendor-supplied addressbook)?
  - Can I reuse this class in the billing software of my dealer management system?

Customization: Build Custom Objects

- Problem: Close the object design gap
  - Develop new functionality
- Main goal:
  - Reuse knowledge from previous experience
  - 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 objects
- Inheritance (also called White-box Reuse)
  - New functionality is obtained by inheritance

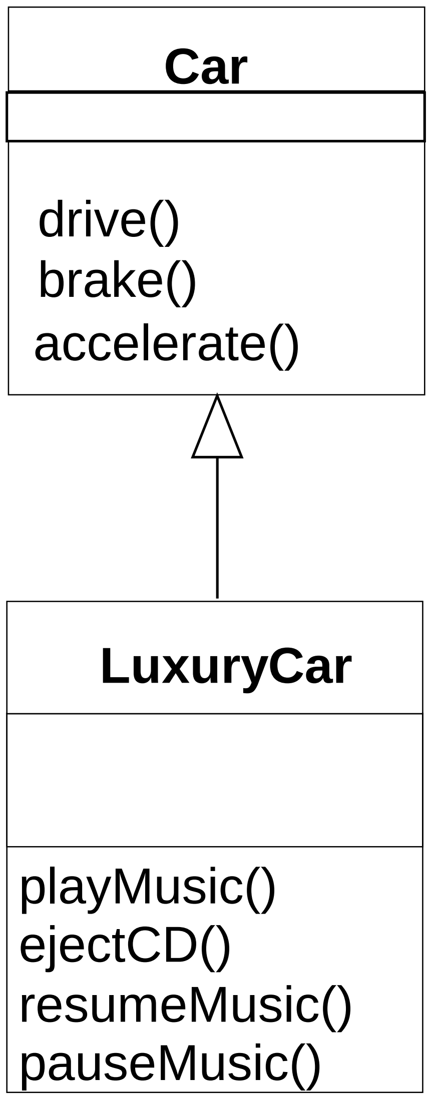
Why Inheritance?

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

Inheritance can be used during Modeling as well as during Implementation

- 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; “Identification of associations“:
    - general associations
    - aggregations
    - inheritance associations.

Example of Inheritance



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 comes in many Flavors

Inheritance is used in four ways:

- Specialization
- Generalization
- Specification Inheritance
- Implementation Inheritance.

Discovering Inheritance

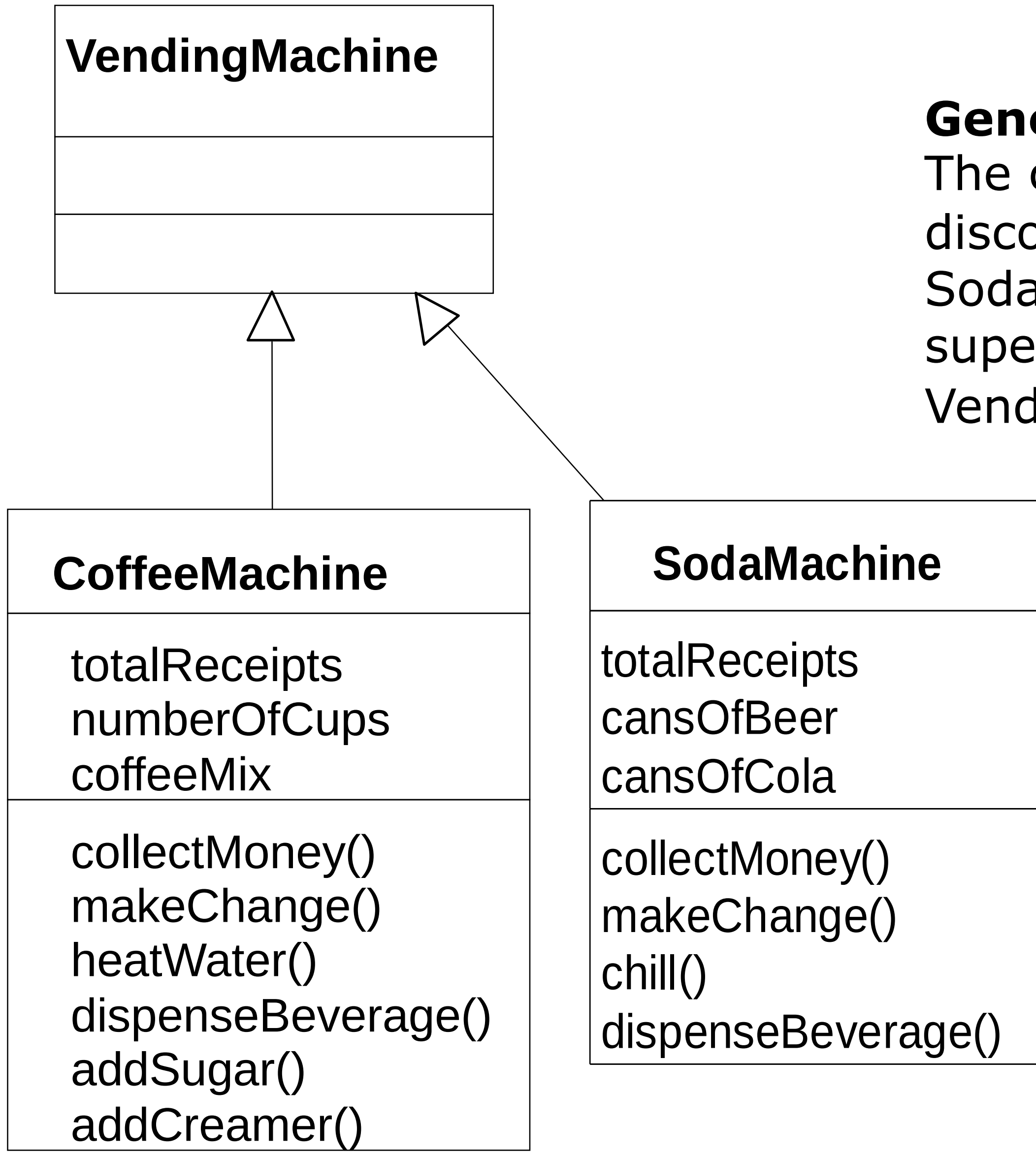
- To “discover“ inheritance associations, we can proceed in two ways, which we call specialization and generalization

- 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 (Elephant, Lion, Tiger), then we discover that these animals have common properties (mammals).
  - **Engineering:** What are the common properties of cars and airplanes?

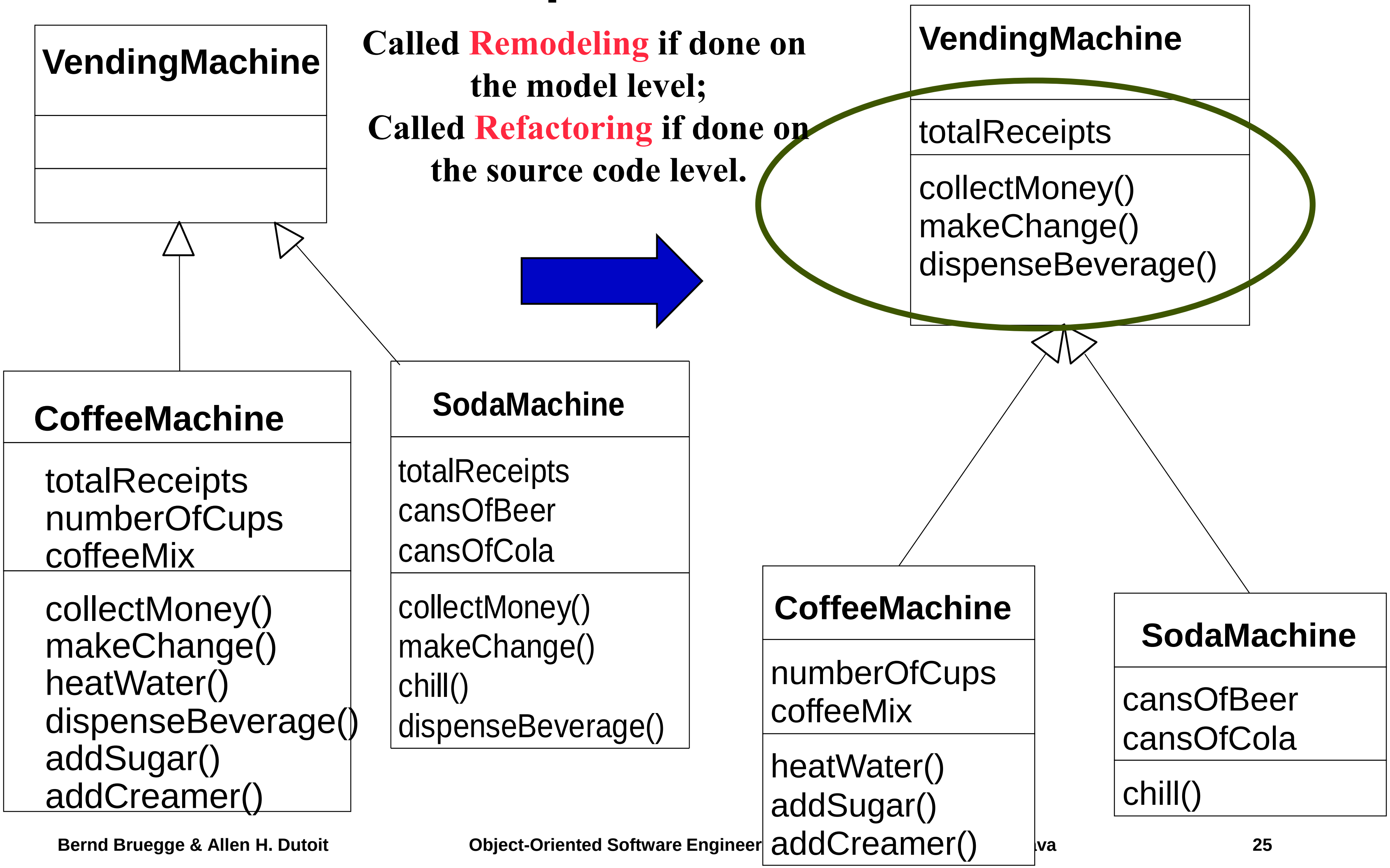
Generalization Example: Modeling a Coffee Machine



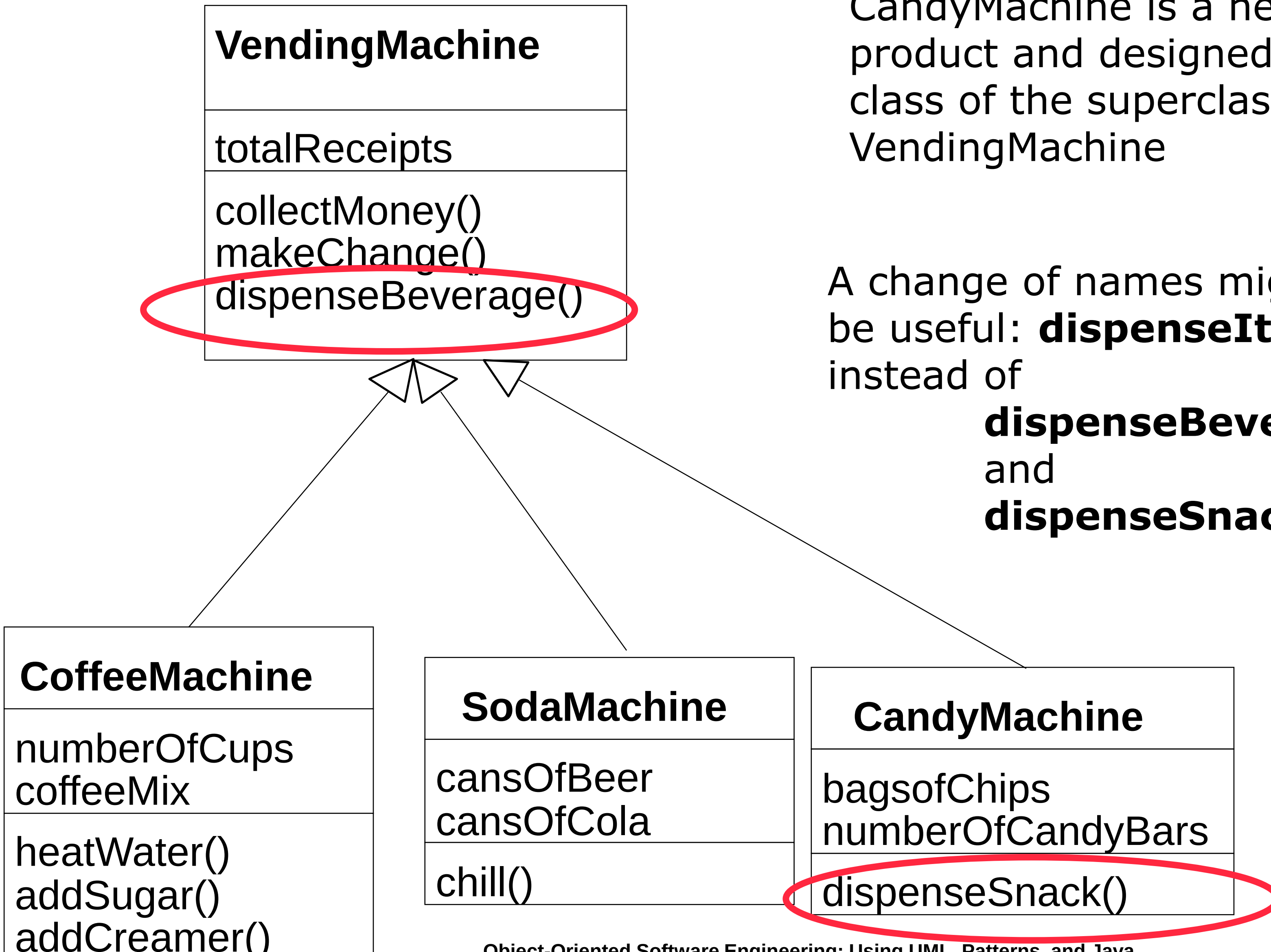
Generalization:
The class CoffeeMachine is discovered first, then the class SodaMachine, then the superclass VendingMachine

Restructuring of Attributes and Operations is often a Consequence of Generalization

Called Remodeling if done on the model level;
Called Refactoring if done on the source code level.



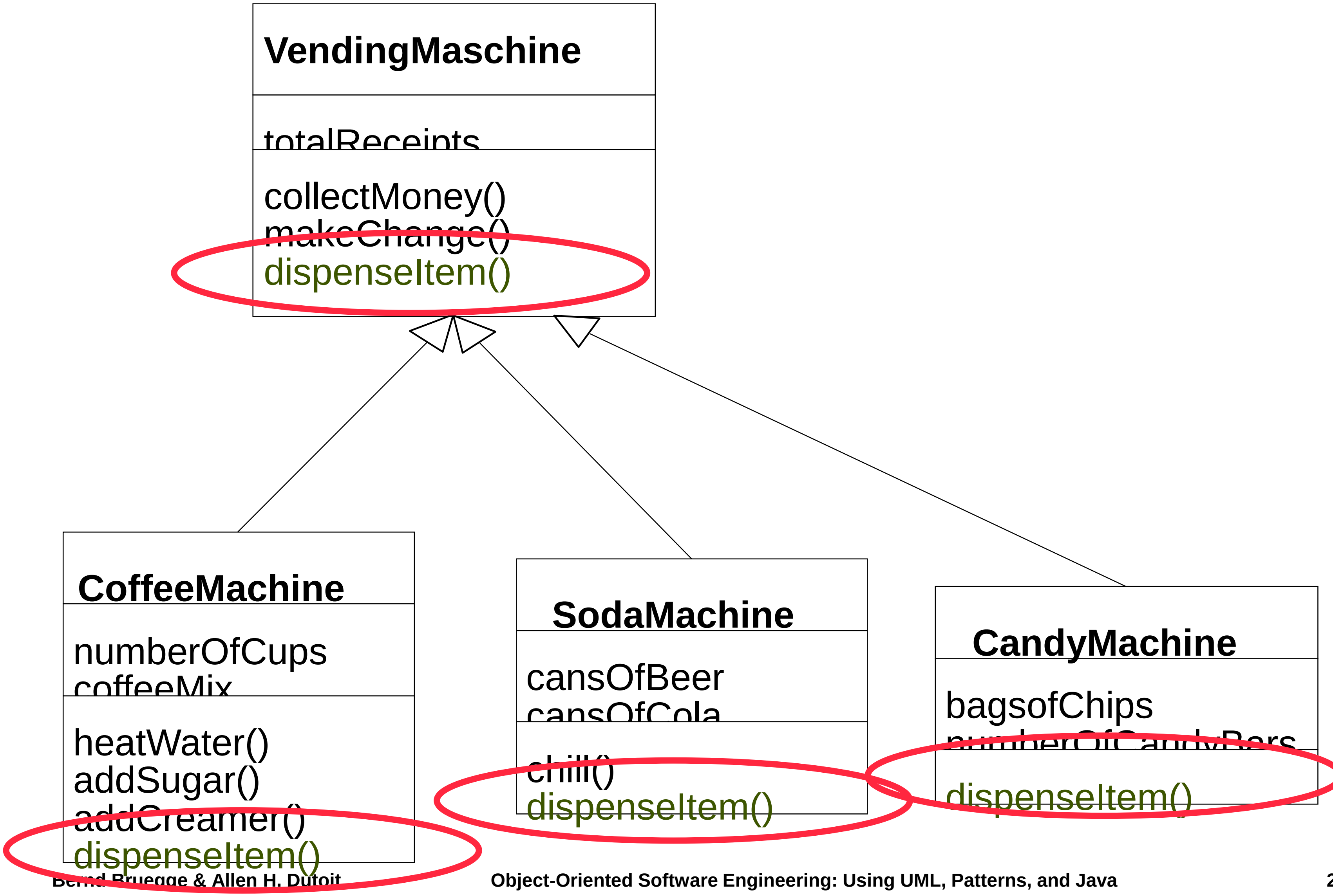
An Example of a Specialization



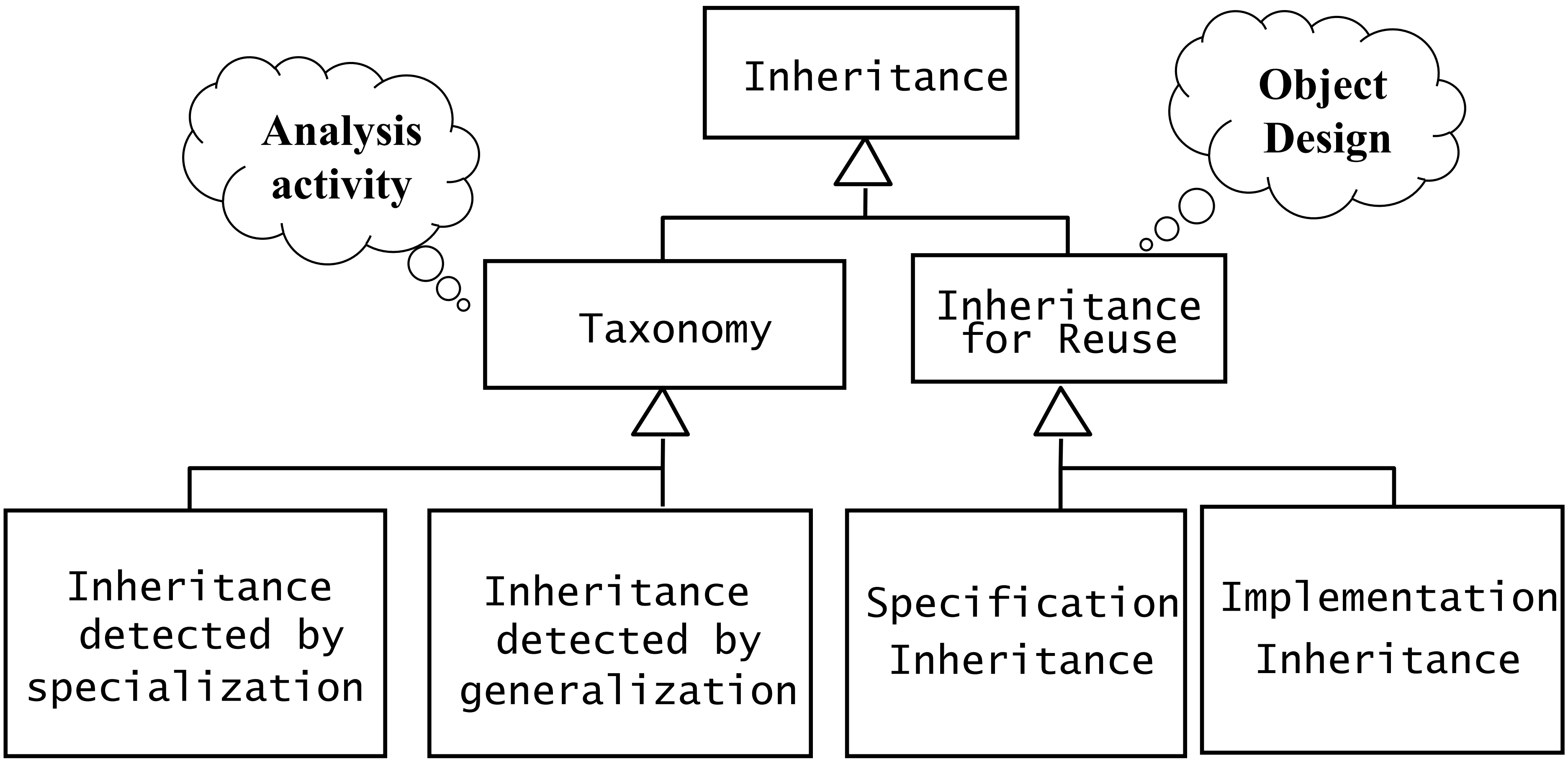
CandyMachine is a new product and designed as a sub class of the superclass VendingMachine

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

Example of a Specialization (2)



Meta-Model for Inheritance

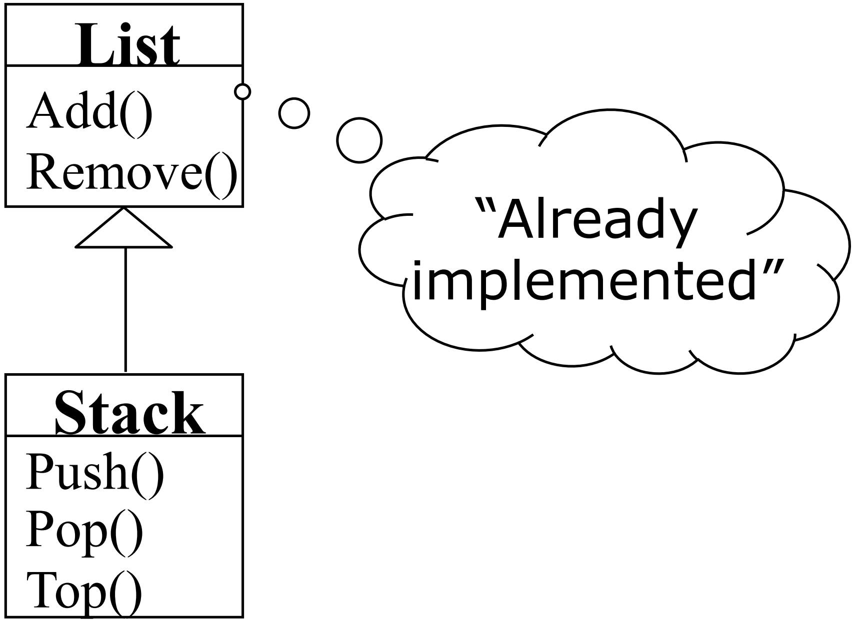


For Reuse: Implementation Inheritance and Specification Inheritance

- Implementation inheritance
  - Also called class inheritance
  - The use of inheritance for the sole purpose of reusing code is called **implementation 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, the classification of concepts into type
  - hierarchies is called specification inheritance
  - Goal:
    - Inherit from a specification
    - The specification is an abstract class with all operations specified, but not yet implemented.

Example for Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation



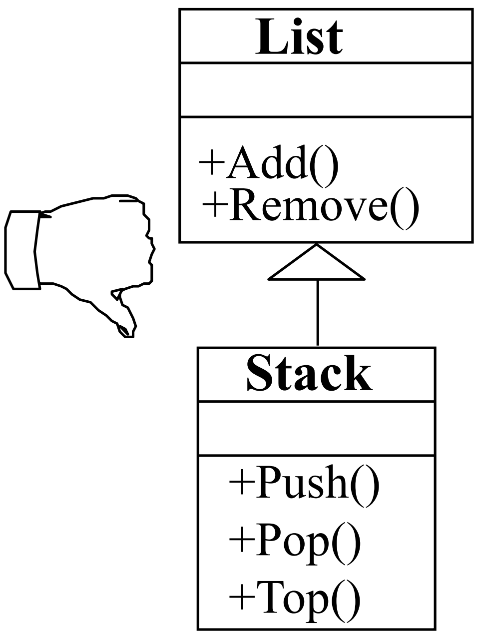
Example:

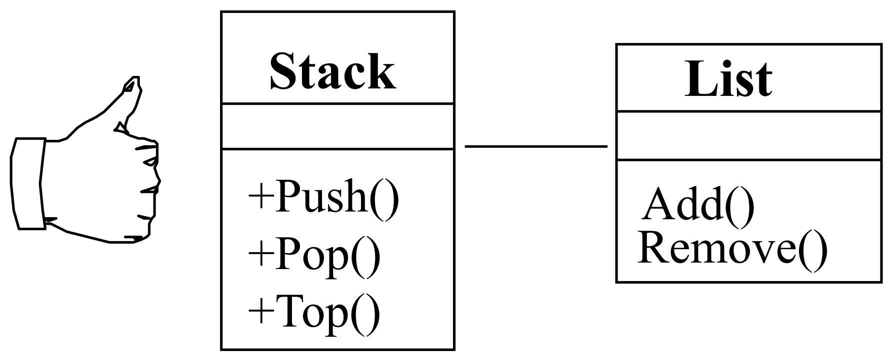
- I have a **List** class, I need a **Stack** class
  - How about subclassing the **Stack** class from the **List** class and implementing **Push()**, **Pop(), Top()** with **Add()** and **Remove()**?

- 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

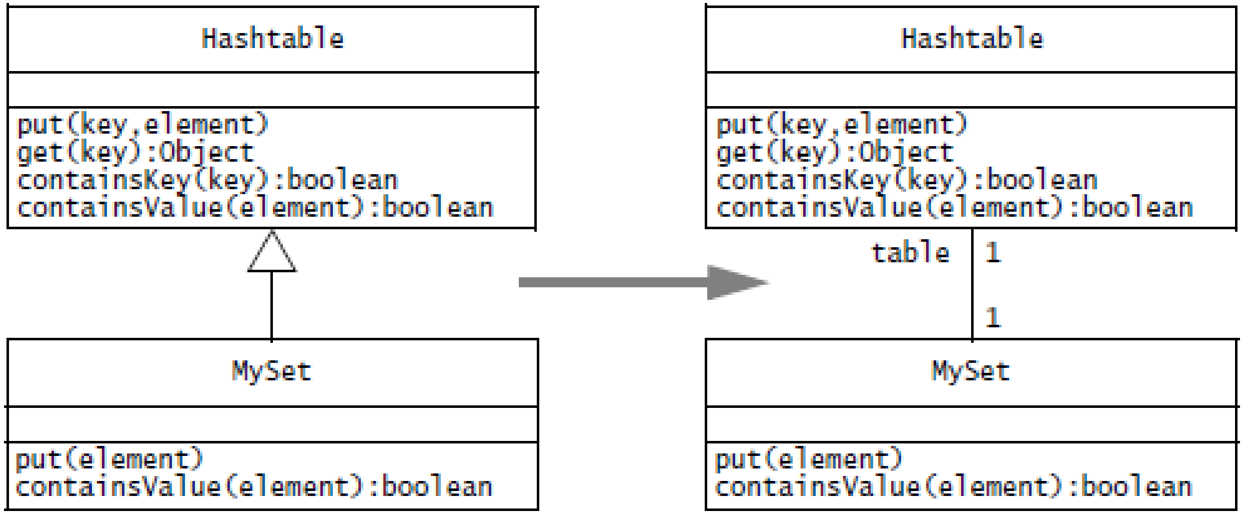
- Inheritance: Extending a Base class by a new operation or overriding an operation.
- Delegation: Catching an operation and sending it to another object.
- Which of the following models is better?





Object design model before transformation

Object design model after transformation



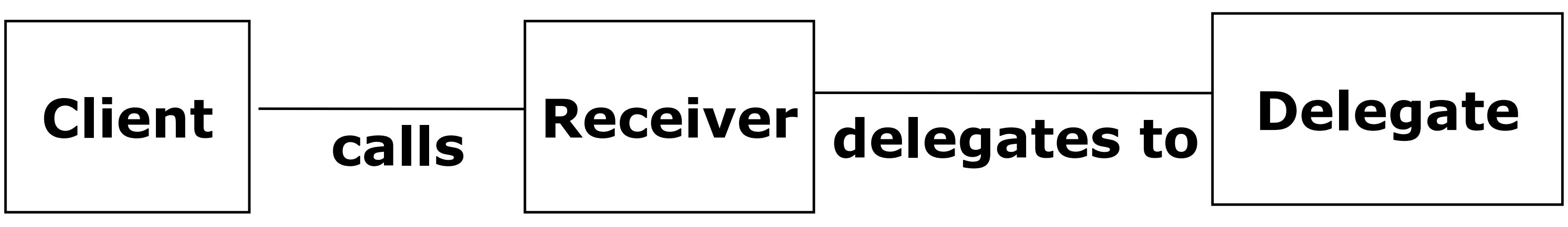

```
/* Implementation of MySet using
inheritance */
class MySet extends Hashtable {
    /* Constructor omitted */
    MySet() {
    }

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

```
/* 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 */
}
```

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
  - ☹ 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)

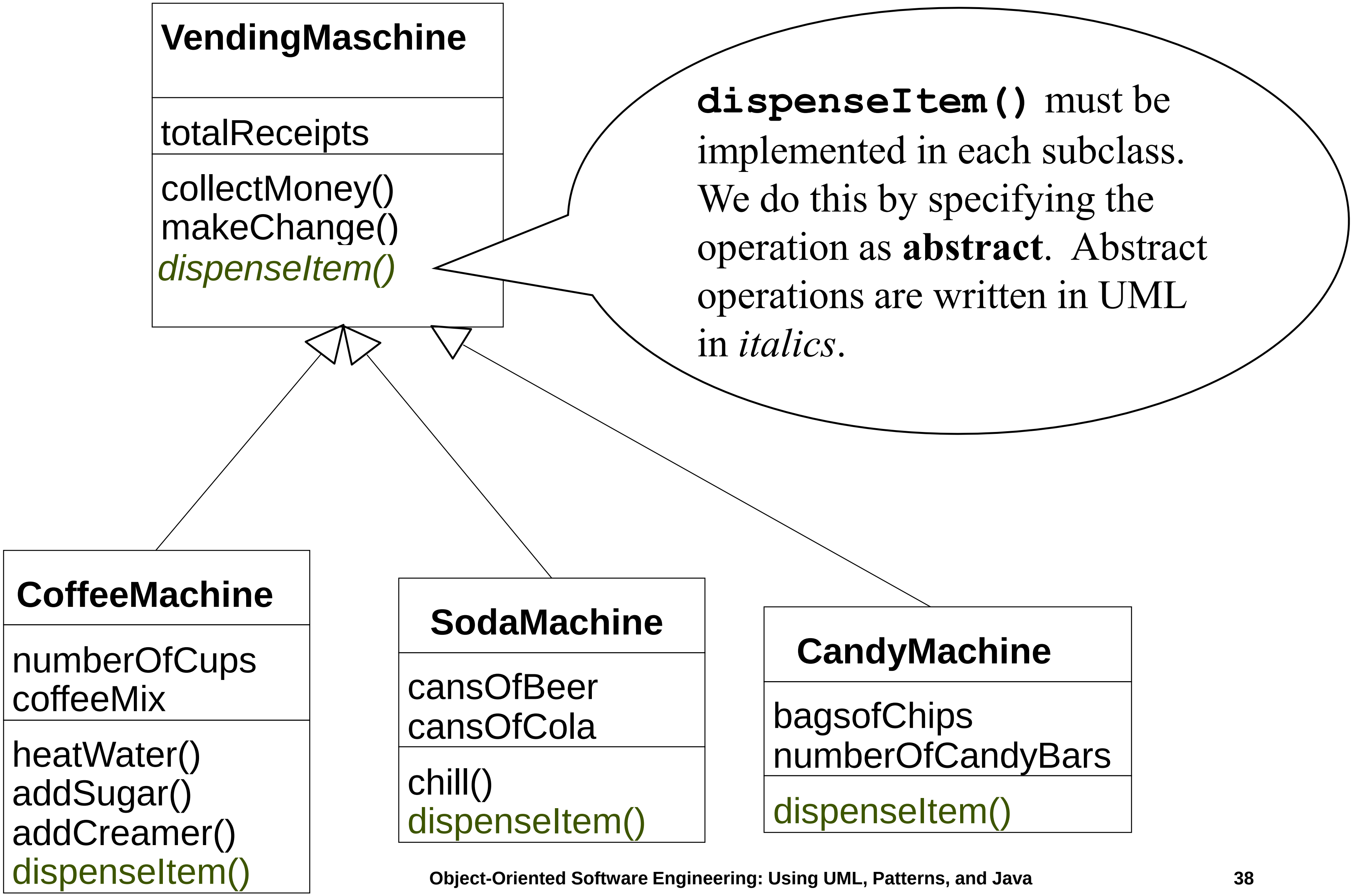
Comparison: Delegation v. Inheritance

- Code-Reuse can be done by delegation as well as inheritance
- Delegation
  - Flexibility: Any object can be replaced at run time by another one
  - Inefficiency: Objects are encapsulated
- Inheritance
  - Straightforward to use
  - Supported by many programming languages
  - Easy to implement new functionality
  - Exposes a subclass to details of its super class
  - Change in the parent class requires recompilation of the subclass.

Abstract Methods 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
- Interface: An abstract class which has only abstract methods
    - An interface is primarily used for the specification of a system or subsystem. The implementation is provided by a subclass or by other mechanisms.

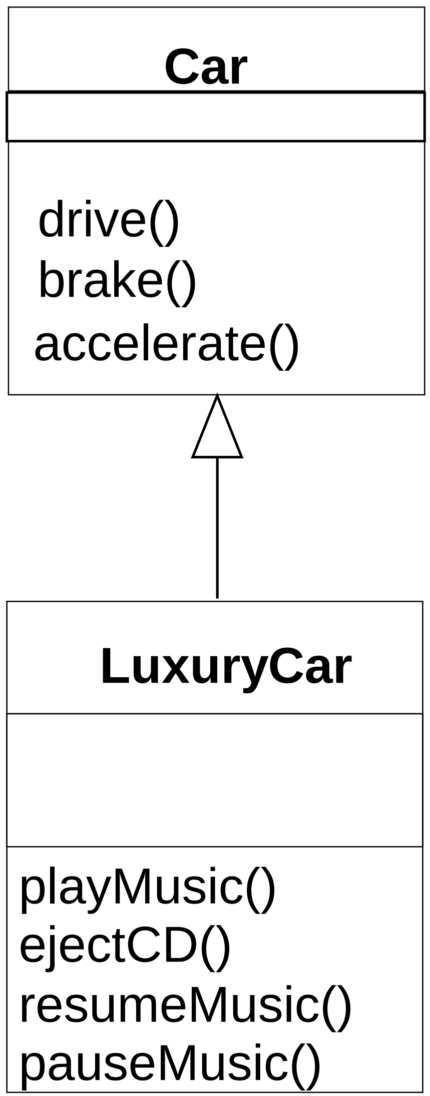
Example of an Abstract Method



Rewriteable Methods and Strict Inheritance

- Rewriteable Method: A method which allow 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 Overwriting   Methods

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

Example:

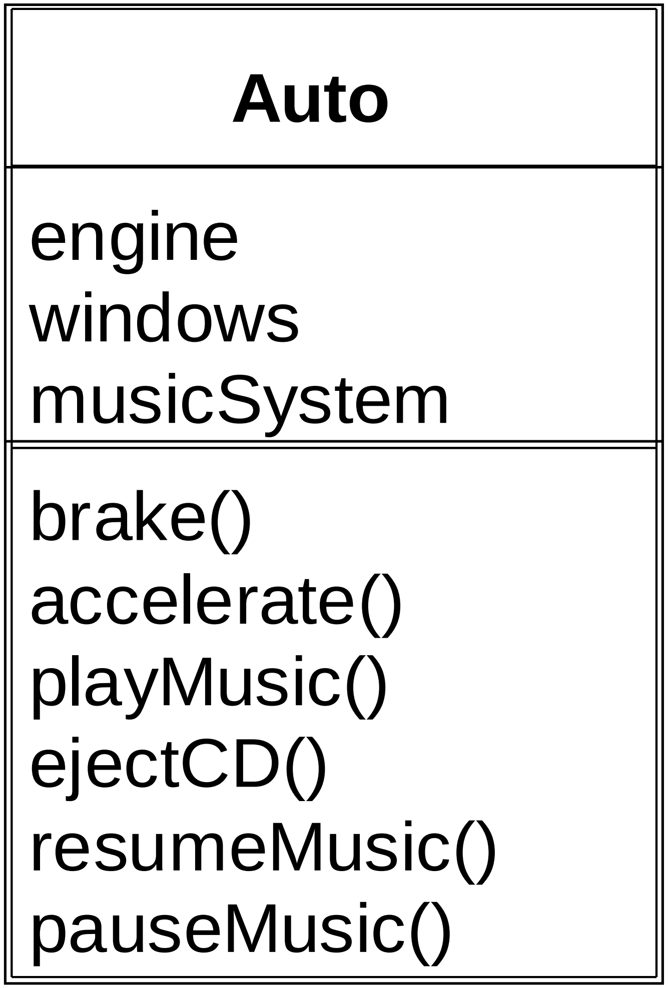
```
Public class SuperClass {
  public int add (int a, int b) { return a+b; }
  public int subtract (int a, int b) { return a-b; }
}
Public class SubClass extends SuperClass {
  public int add (int a, int b) { return a-b; }
  public int subtract (int a, int b) { return a+b; }
}
```

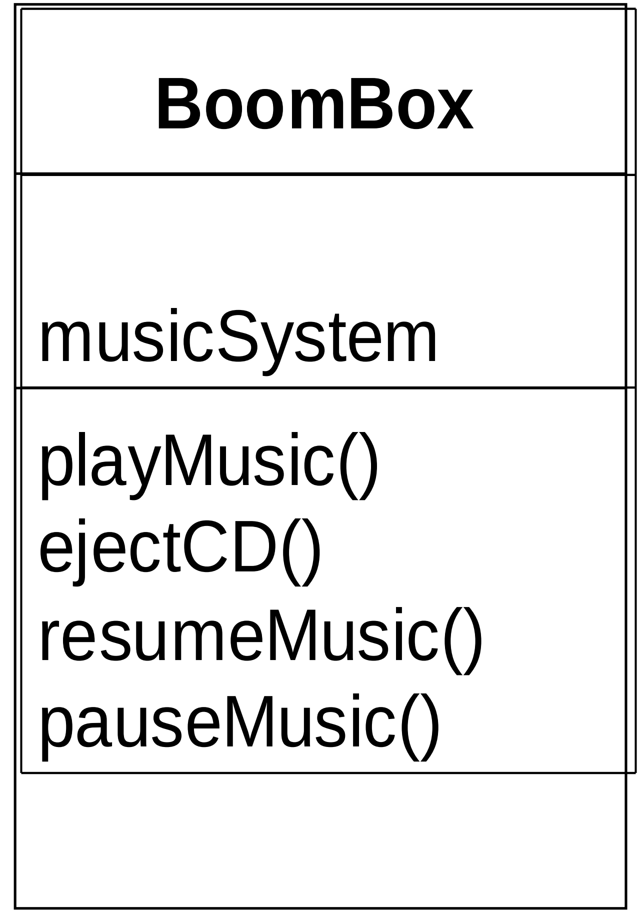
- We have redefined addition as subtraction and subtraction as addition!!

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!“

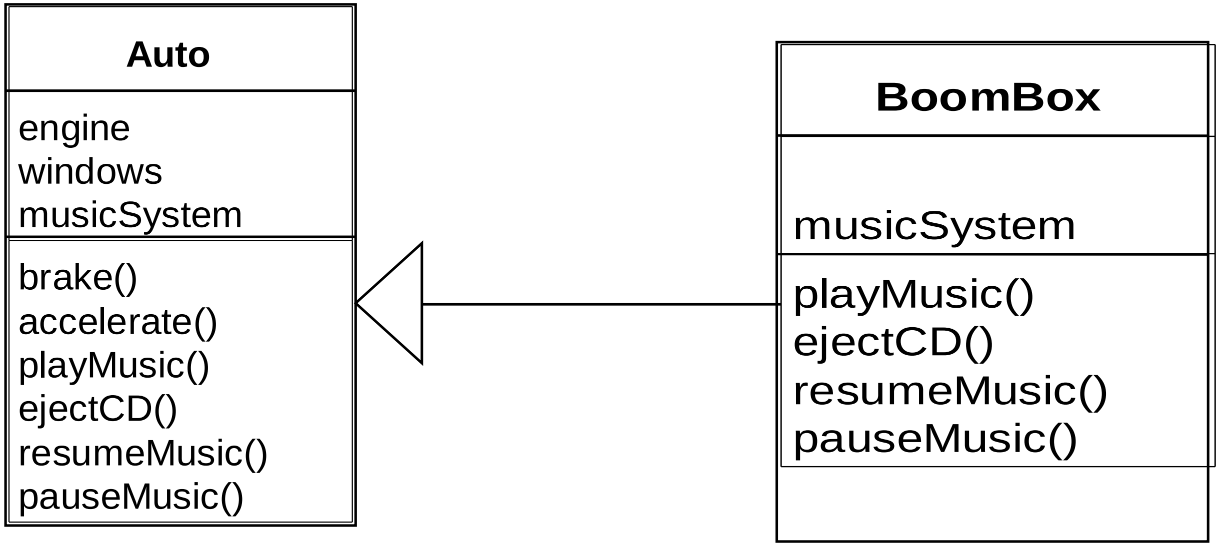
What we have and what we want





New Abstraction!

What we do to save money and time



Existing Class:

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

Boombox:

```
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 must be avoided by all Means

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?

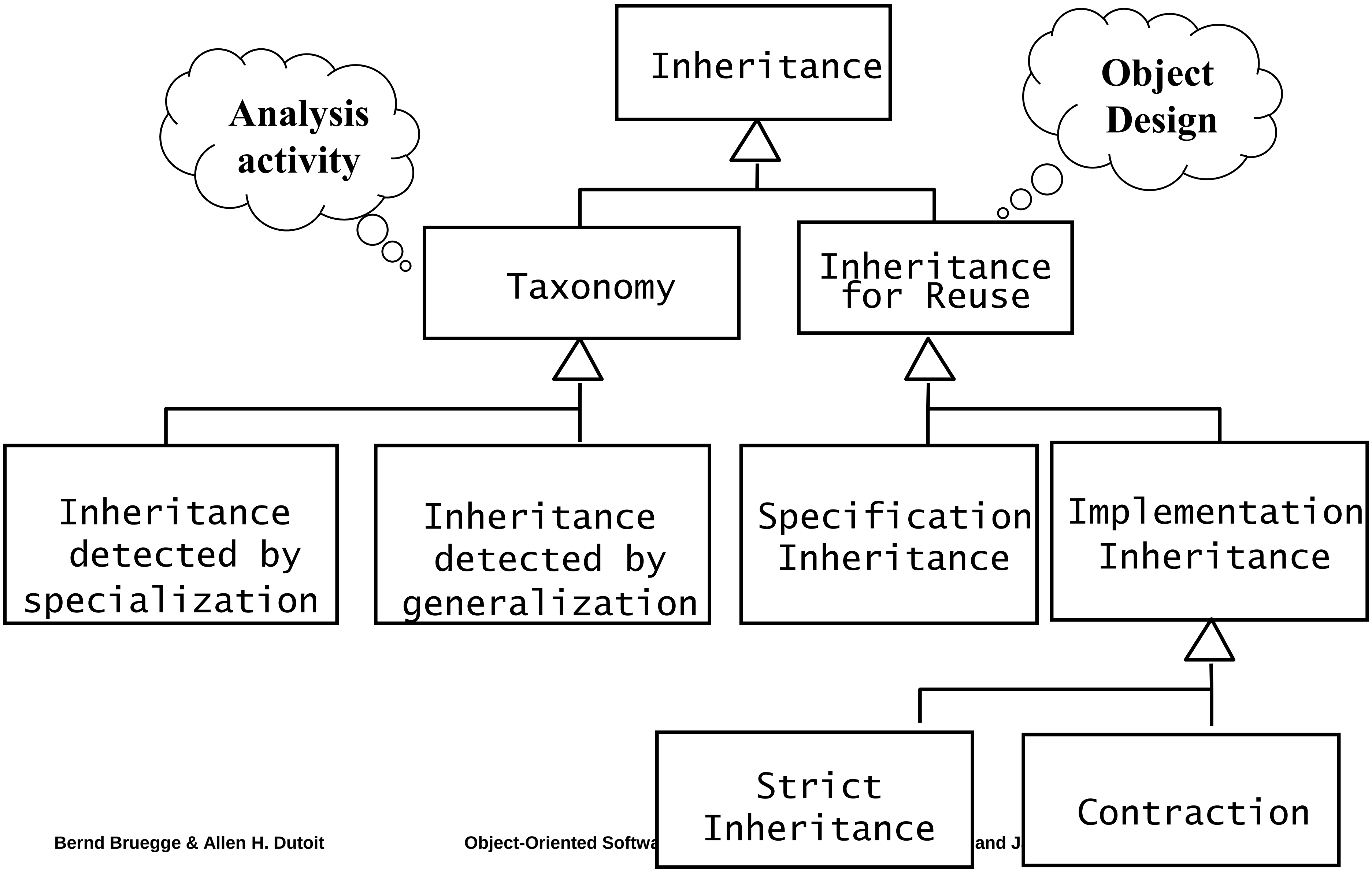
The subclass does not fit into the taxonomy

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.

- LSP: 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



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).