Object Design: Reuse

Introduction into Software Engineering
Lecture 10

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen

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)
  - Design Patterns (Chapter 8 and Appendix A)
- Today:
  - Object Design (Chapter 8)
- Next week (Wednesday 30 May 2007)
  - System Design (Chapter 6)
- Saturday 2 June 2007:
  - Mid-Term.

Details for the Mid-Term:

- Coverage:
  - Lecture 1 - lecture 10 (this lecture)
  - Textbook: Chapter 1 - 8 (Chapter 6 - 7 are not covered)
- Date: Saturday 2 June 2007, Location: MW 0001
- Closed book exam
  - 13:00 to 14:30 am: 90 min
  - Format: Paper-based, handwritten notes
  - Questions about definitions and modeling activities
- Questions in English
    - Answers in English or German
    - Dictionaries are allowed
- For additional information, check the lecture portal
  - http://wwwbruegge.in.tum.de/twiki/bin/view/Lehrstuhl/SoftwareEngineeringSoSe2007#GradingCriteria

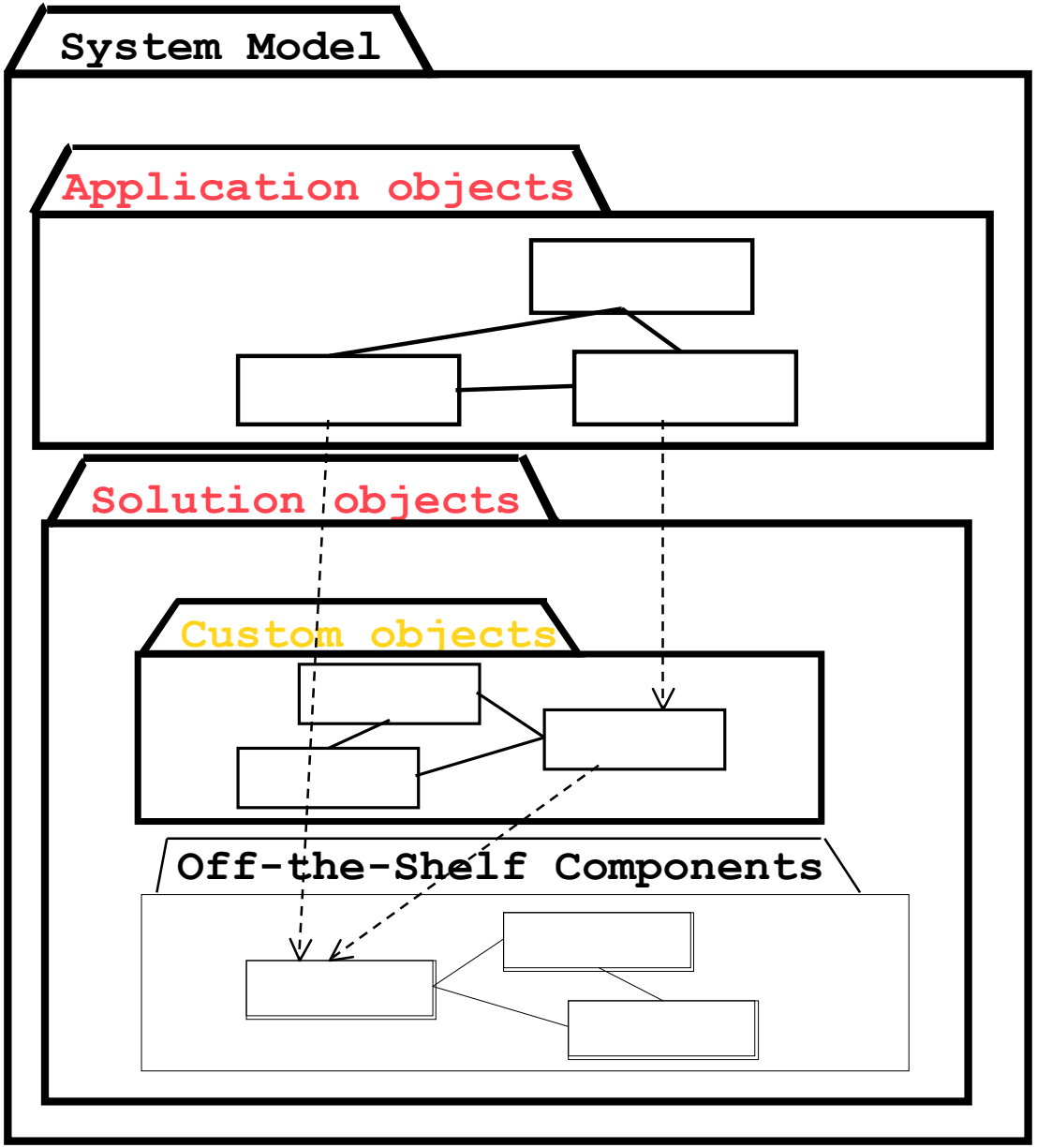
Outline of Today

- Object Design
- Reuse examples
  - Reuse of code, interfaces and existing classes
- White box and black box reuse
- The use of inheritance
- Implementation vs. specification inheritance
- Delegation vs. Inheritance
- Abstract classes and abstract methods
- Contraction: Bad example of inheritance
- Meta model for inheritance
- Frameworks and components
- Documenting the object design.

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

Design means "Closing the Gap"

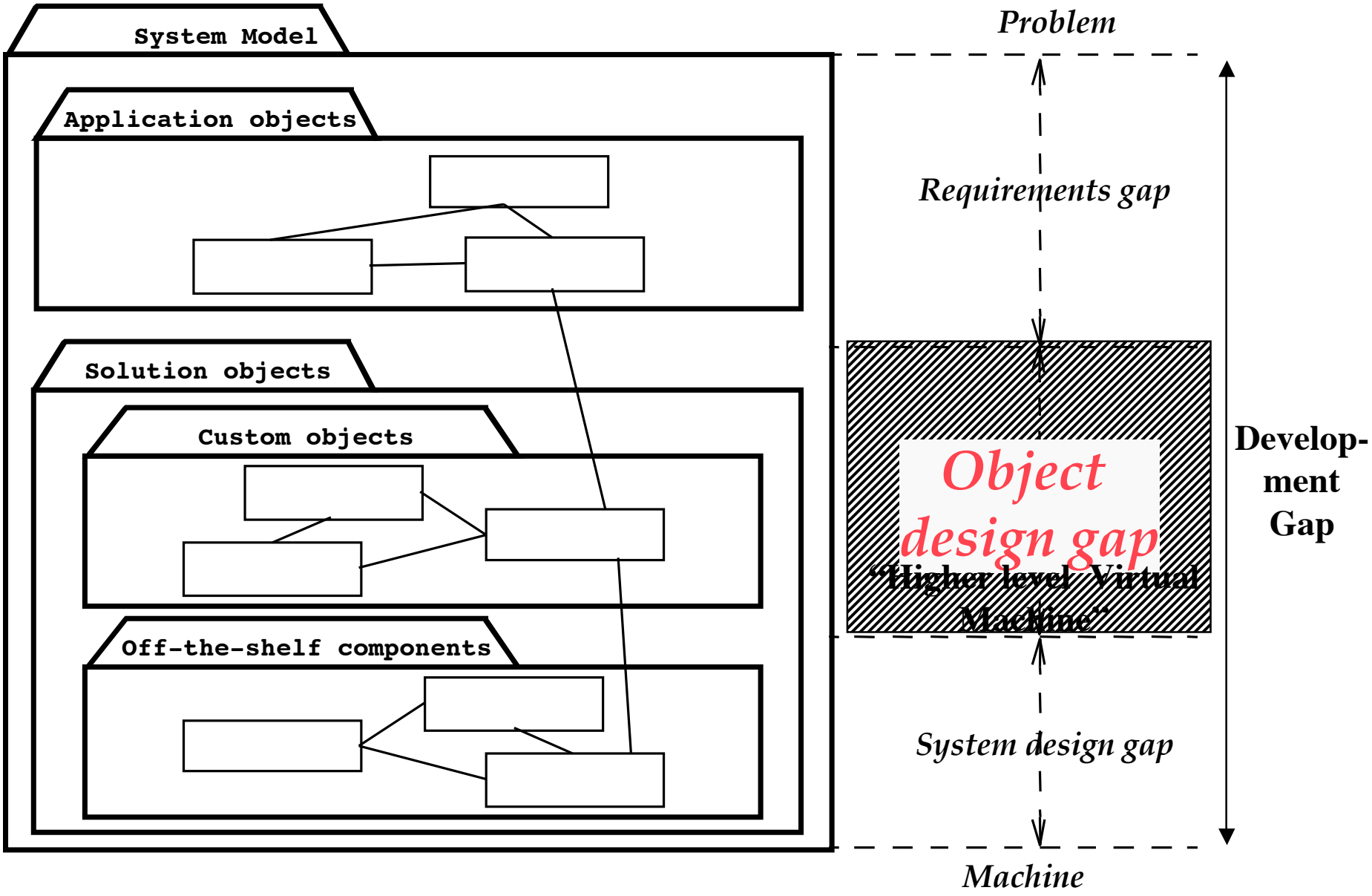
"Subsystem 1": Rock material from the Southern Sierra Nevada mountains (moving north)

Example of a Gap: San Andreas Fault

"Subsystem 3" closes the Gap: San Andreas Lake

"Subsystem 2": San Francisco Bay Area

Design means "Closing the Gap"



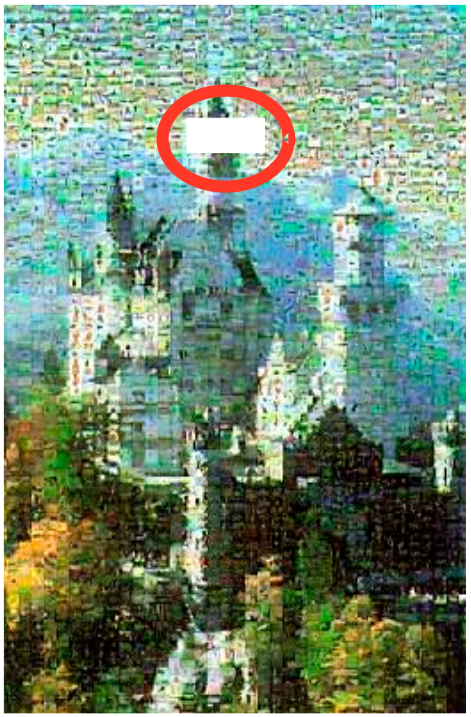
One Way to do System Design

- Component-Based Software Engineering
  1. Identify the missing components
  2. Make a build or buy decision to get the missing component

- Special Case: COTS-Development
  - COTS: Commercial-off-the-Shelf
  - Every gap is filled with a commercial-off-the-shelf-component.

  => Design with standard components

Design with Standard Components is s
solving a Jigsaw Puzzle



What do we do if that is not true?“

Standard Puzzles: „Corner pieces have two straight edges“

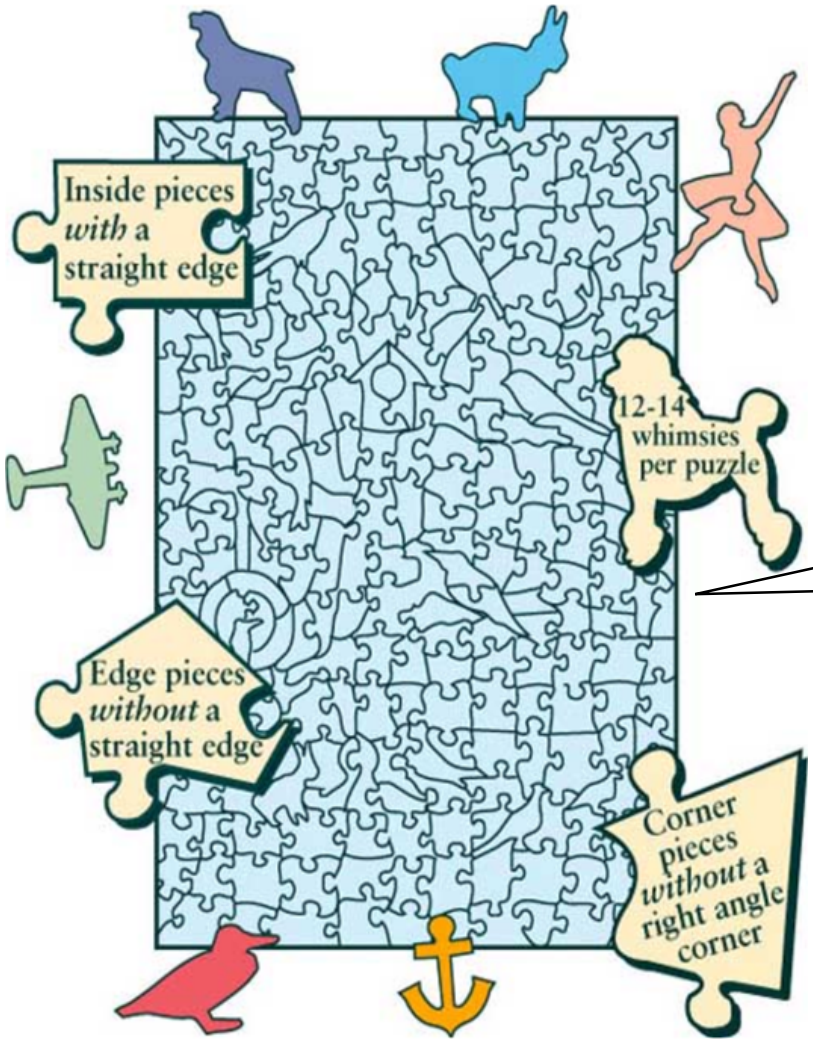
Puzzle Piece (“component”)

Next week‘s Lecture (Chapter 6)

Design Activities:

1. Start with the architecture (subsystem decomposition)
2. Identify the missing component
3. Make a build or buy decision for the component
4. Add the component to the system (finalizing the design).

What do we do if we have non-Standard Components?



Advanced Jigsaw Puzzles

Customization Projects are like Advanced Jigsaw Puzzles

Design Patterns!

http://www.puzzlehouse.com/_

Outline of Today

- Object Design
- Reuse examples
  - ➡ Reuse of code, interfaces and existing classes
- White box and black box reuse
- The use of inheritance
- Implementation vs. specification inheritance
- Delegation vs. Inheritance
- Abstract classes and abstract methods
- Contraction: Bad example of inheritance
- Meta model for inheritance
- Frameworks and components
- Documenting the object design.

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 interfaces

- I am an off-shore programmer in Hawaii. I have a contract to implement an electronic parts catalog for an automotive company
  - How can I be sure that I access the company's database 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 for MacOS X?
- I want to develop a new service for cars, that automatically calls a help center when the car is involved in a crash
  - Can I reuse the help desk software that I developed for a company in the telecommuniction industry?

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

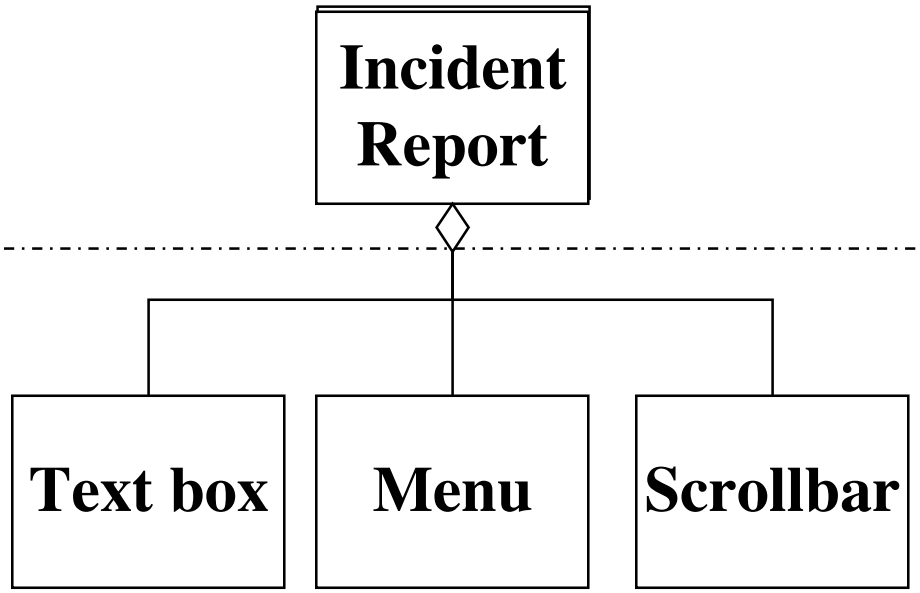
White Box and Black Box Reuse

- White box reuse
  - Access to the development products (models, system design, object design, source code) must be available

- Black box reuse
  - Access to models and designs is not available, or models do not exist
    - Worst case: Only executables (binary code) are available
    - Better case: A specification of the system interface is available.

Identification of new Objects during Object Design

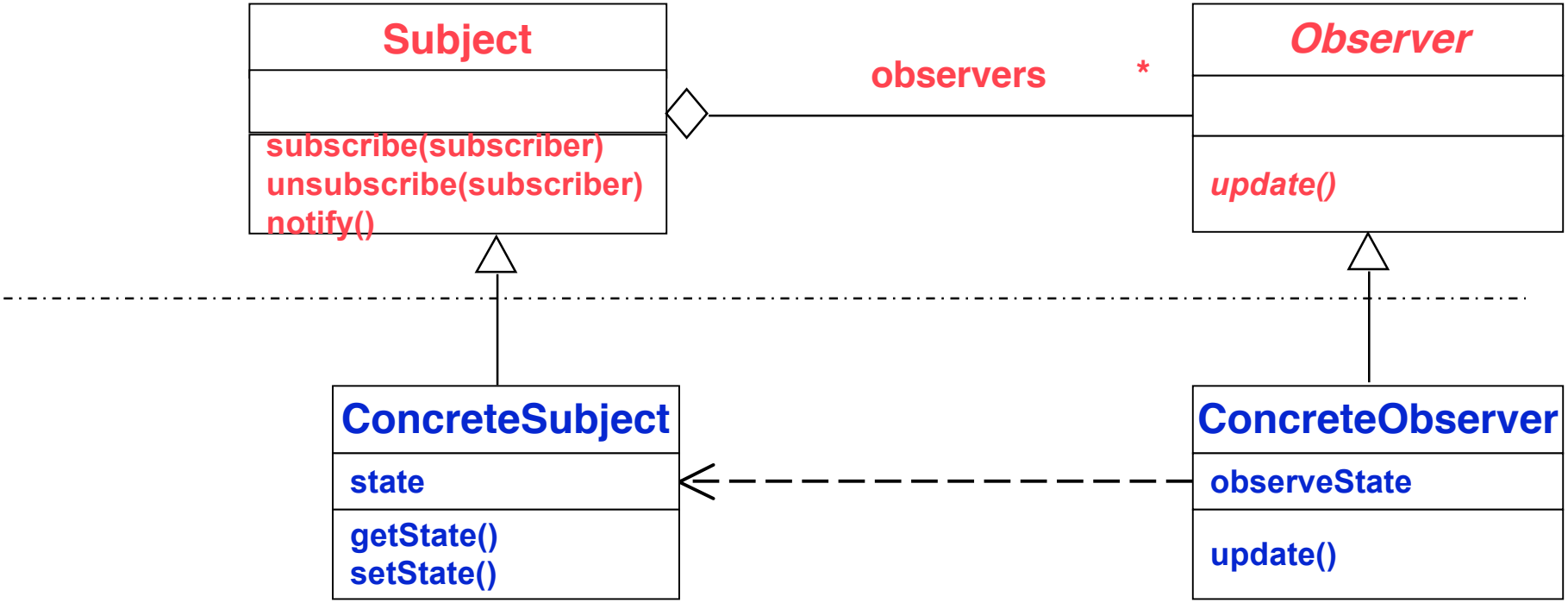
Requirements Analysis (Language of Application Domain)



Object Design (Language of Solution Domain)

Application Domain vs Solution Domain Objects

Requirements Analysis (Language of Application Domain)



Object Design (Language of Solution Domain)

Other Reasons for new Objects

- 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.

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

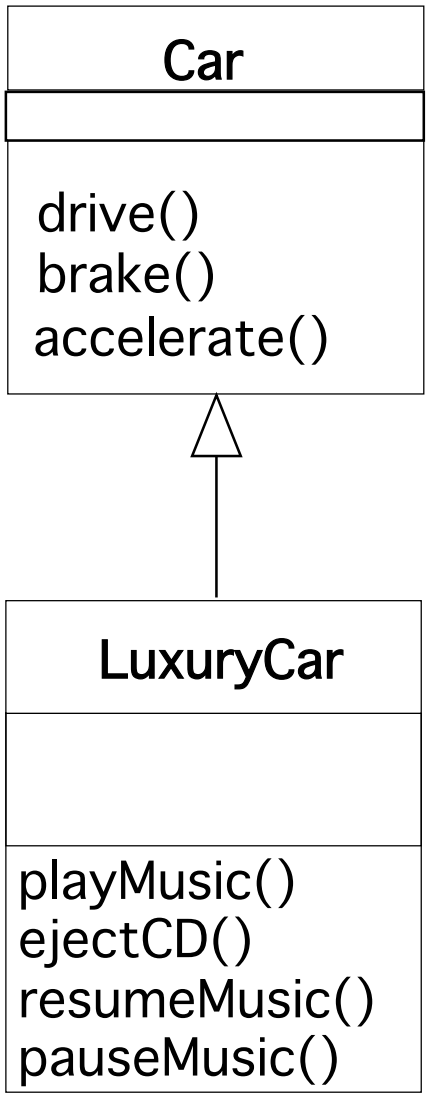
The use of Inheritance

- 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

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

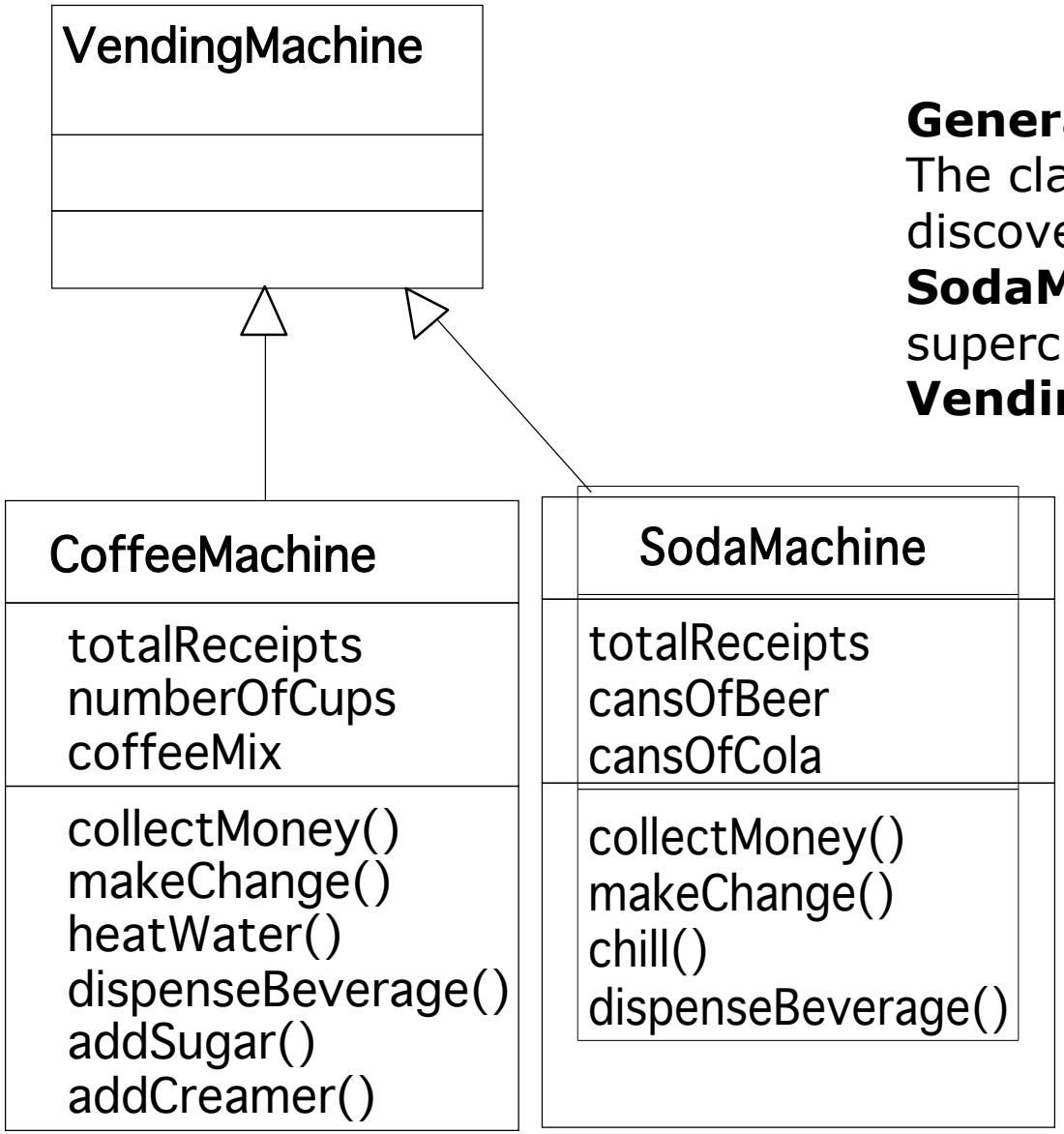
- To "discover" inheritance associations, we can proceed in two ways, which we call specialization and generalization

- 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

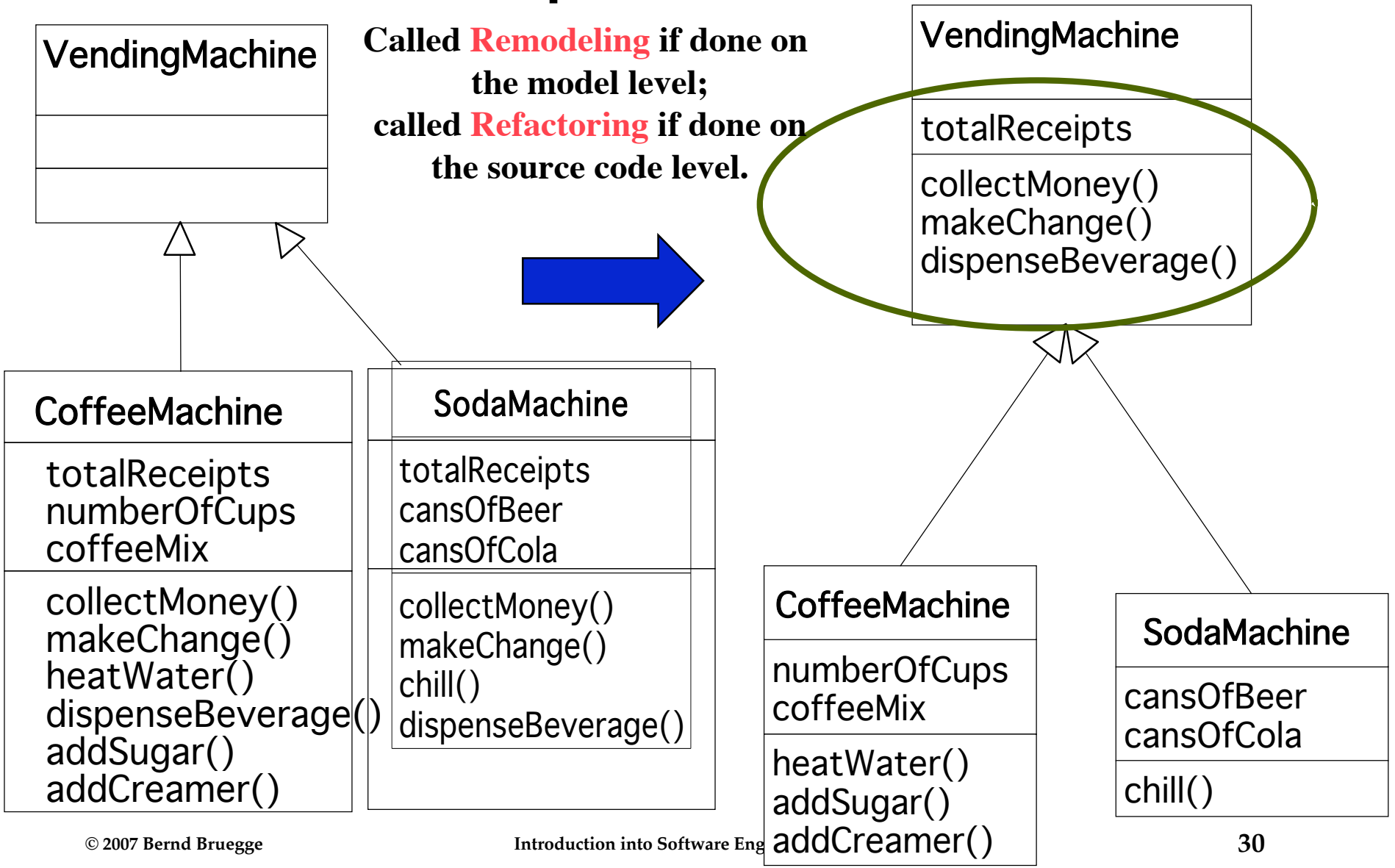
- First we find the subclass, then the super class
- This type of discovery occurs often in science

Generalization Example: Modeling a Coffee Machine



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

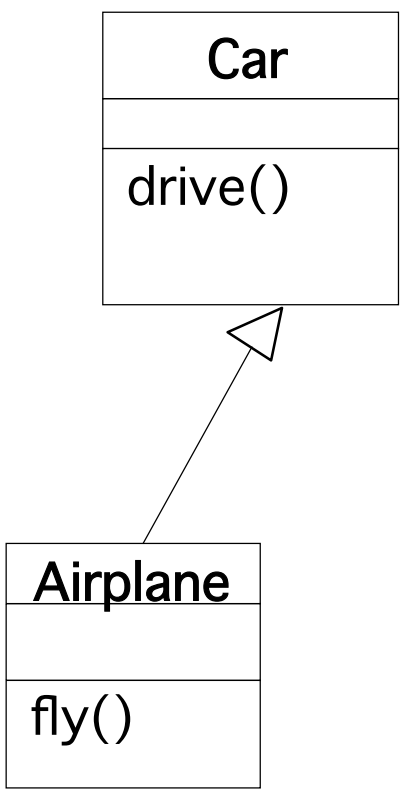
Restructuring of Attributes and Operations is often a Consequence of Generalization

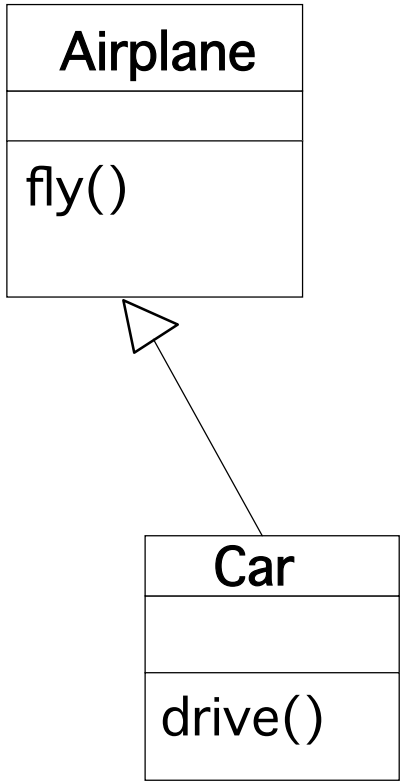


Specialization

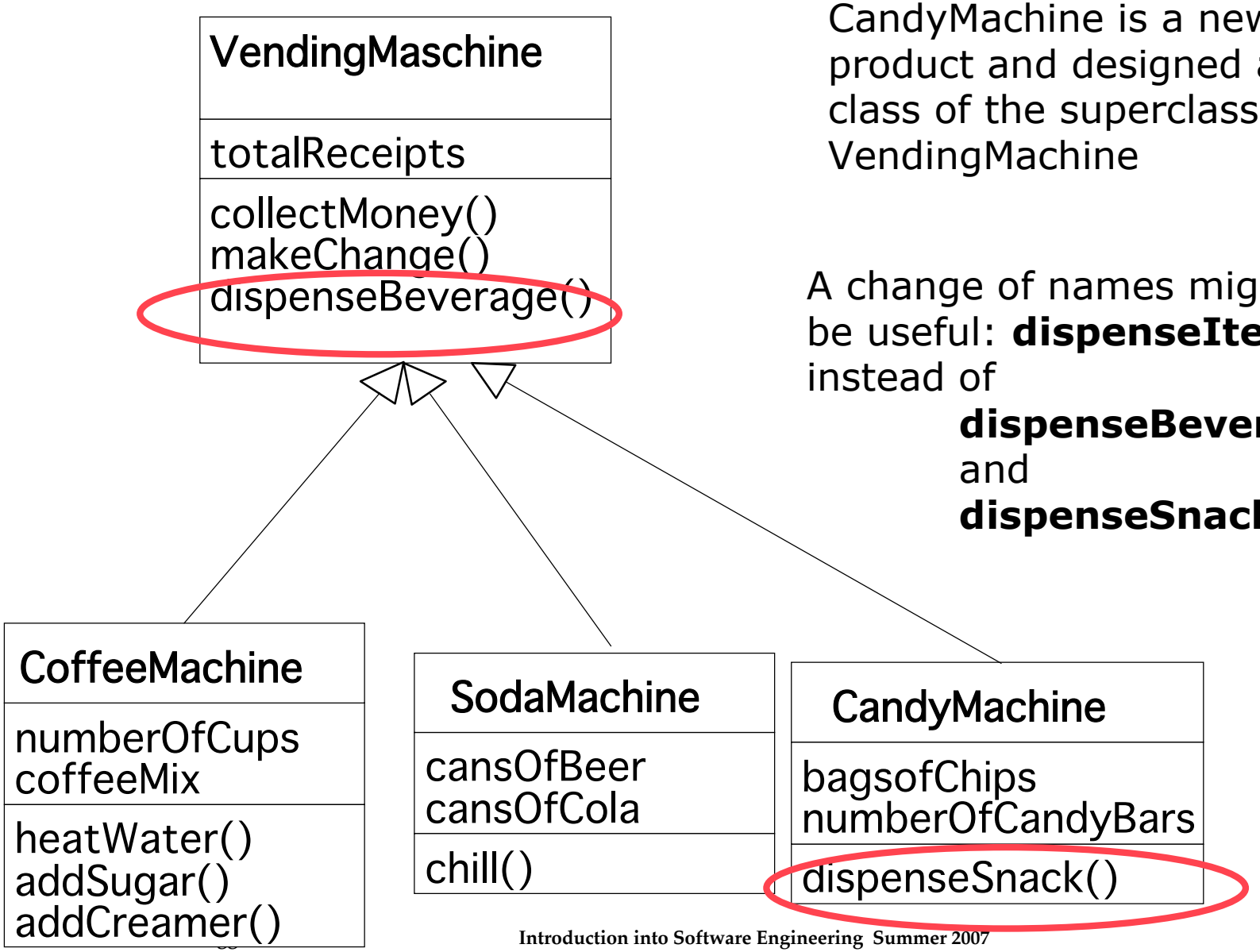
- Specialization occurs, when we find a subclass that is very similar to an existing class.
  - Example: A theory postulates certain particles and events which we have to find.
- Specialization can also occur unintentionally:

Which Taxonomy is correct for the Example in the previous Slide?





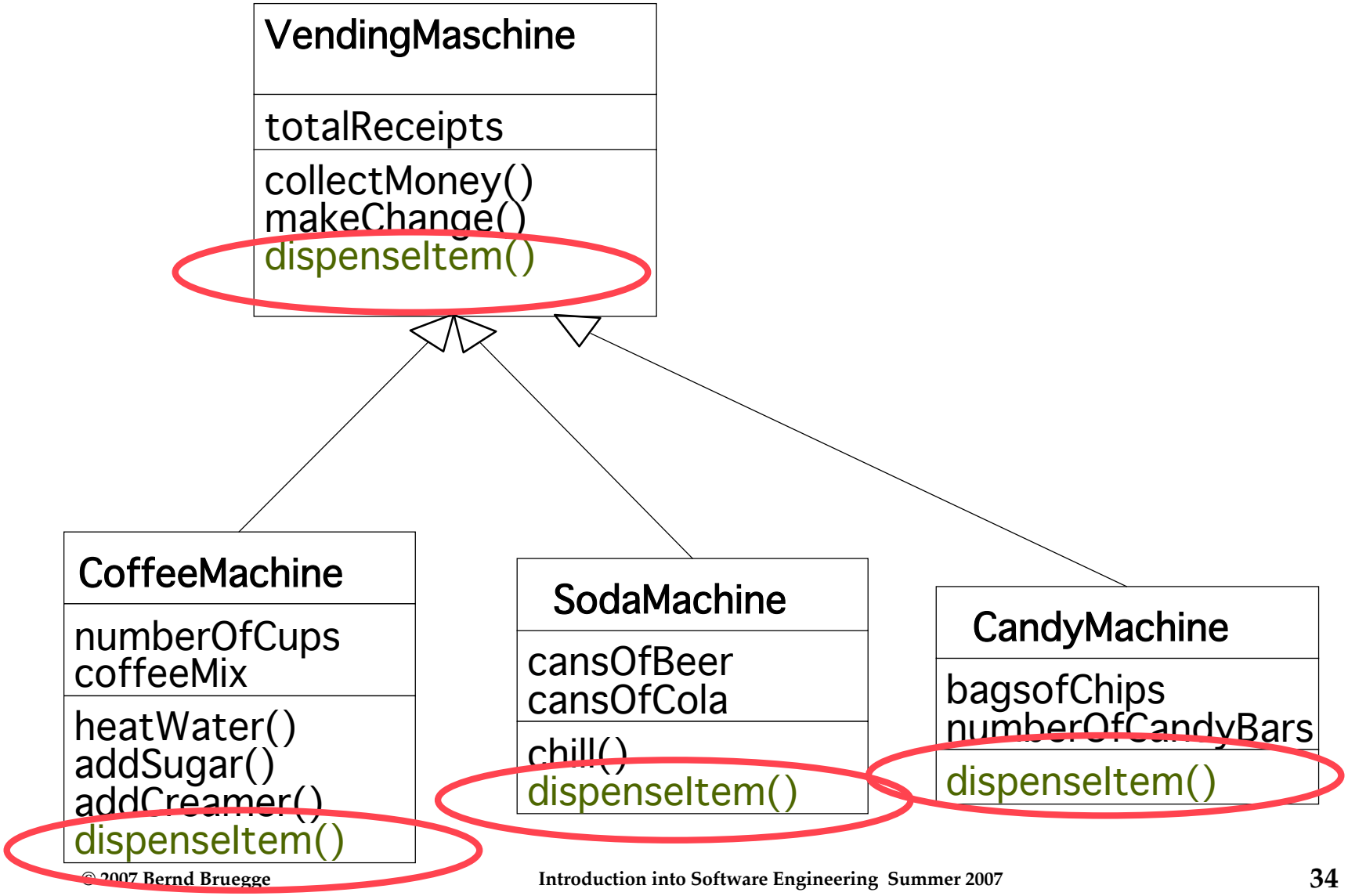
Another Example of a Specialization



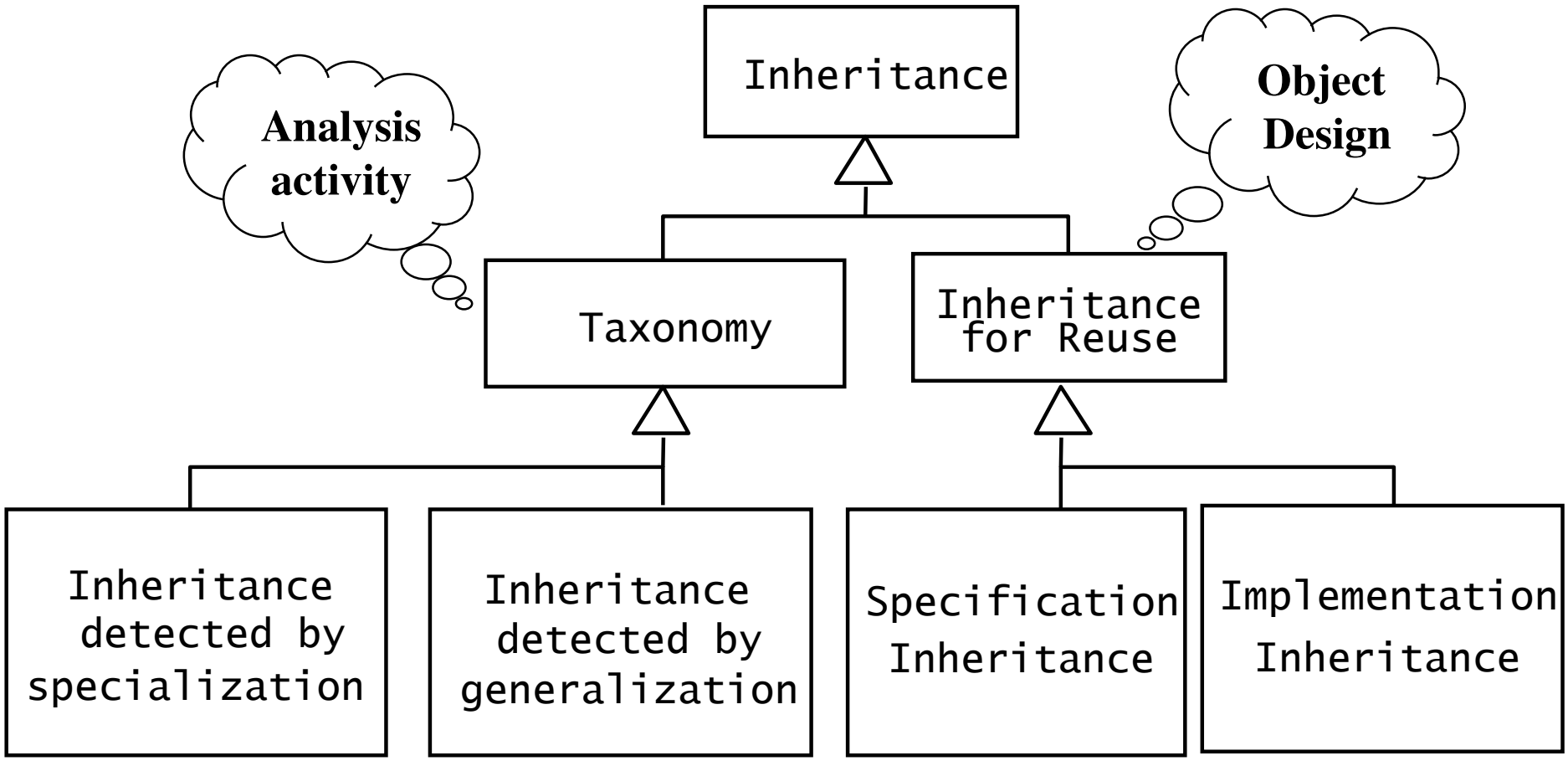
CandyMachine is a new product and designed as a sub class of the superclass VendingMachine

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

Example of a Specialization (2)



Meta-Model for Inheritance



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 operations specified, but not yet implemented.

Implementation Inheritance vs. Specification Inheritance

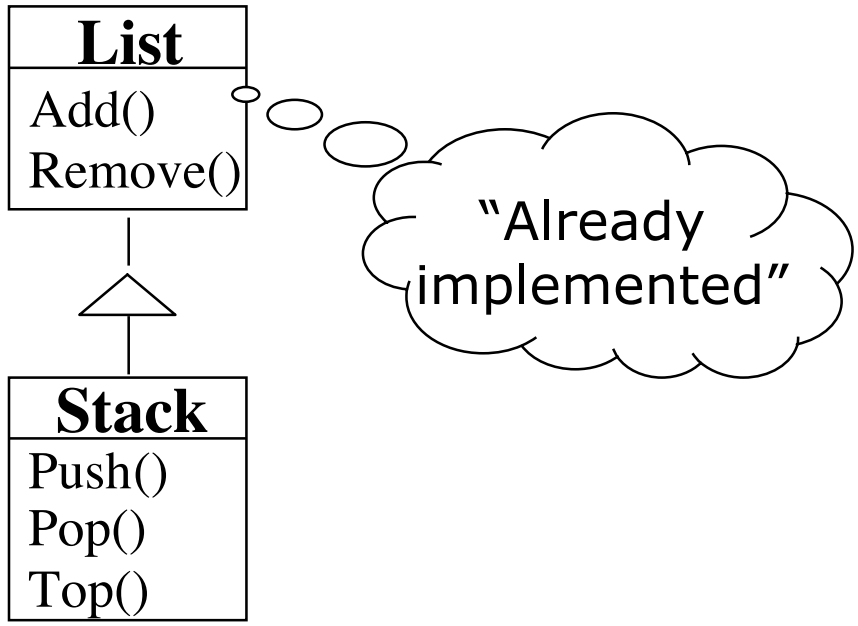
- Implementation Inheritance: The combination of inheritance and implementation
  - The Interface of the superclass is completely inherited
  - Implementations of methods in the superclass ("Reference implementations") are inherited by any subclass
- Specification Inheritance: The combination of inheritance 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 very similar class is already implemented that does almost the same as the desired class implementation

Example:

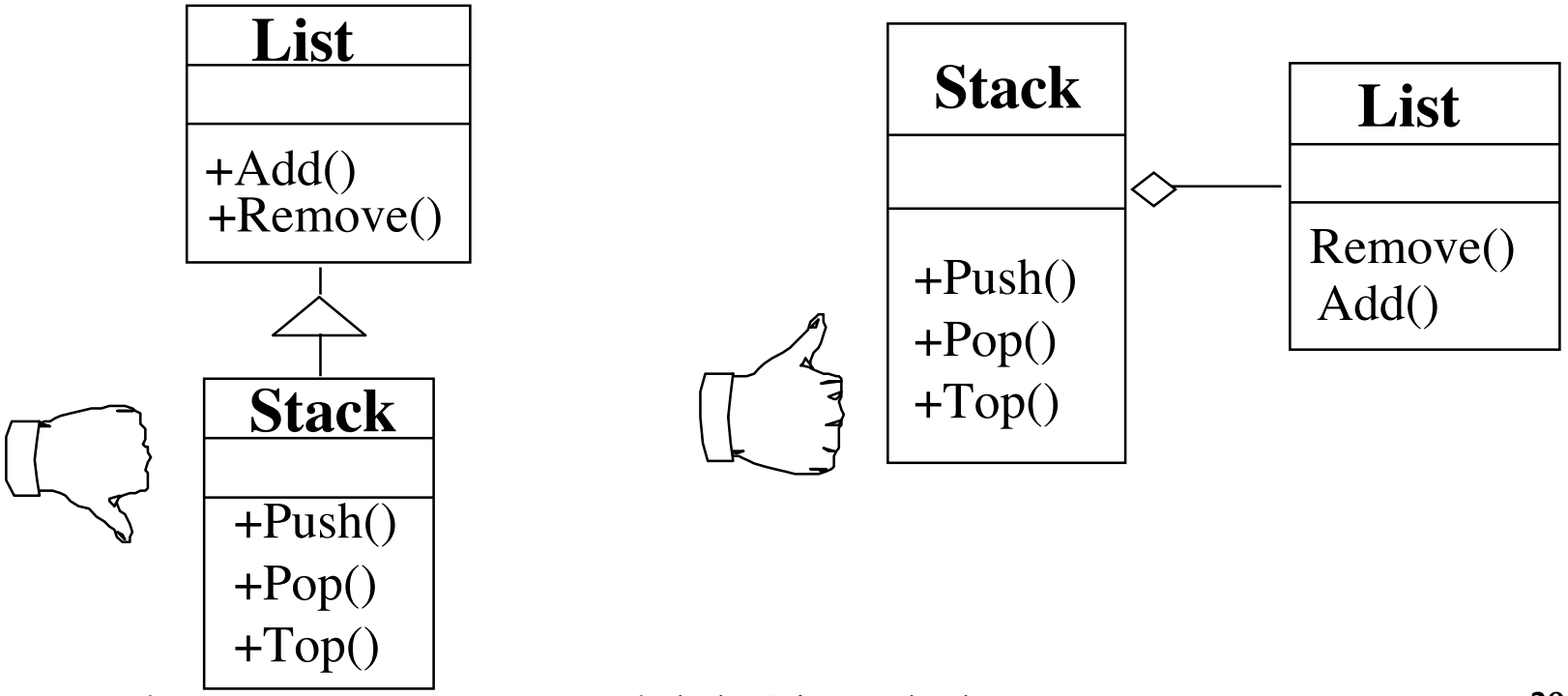
- I have a **List** class, I need a **Stack** class
- How about subclassing the **Stack** class from the **List** class and implementing **Push()**, **Pop(), Top()** with **Add()** and **Remove()**?



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

Better Code Reuse: Delegation

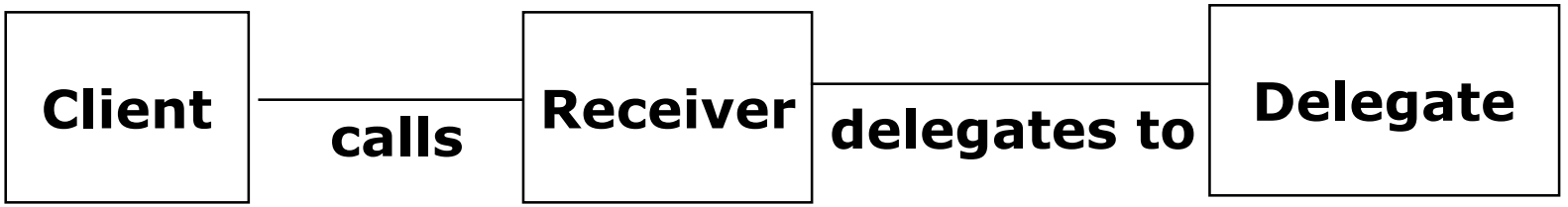
- Implementation-Inheritance: Using the implementation of super class operations
- Delegation: Catching an operation and sending it to another object that implements the operation



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 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.

Object Design Activities

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

Mapping Models to Code

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 super class is an abstract class
    - Implementations of the super class (if there are any) are not inherited
  - The Interface of the super class is completely inherited

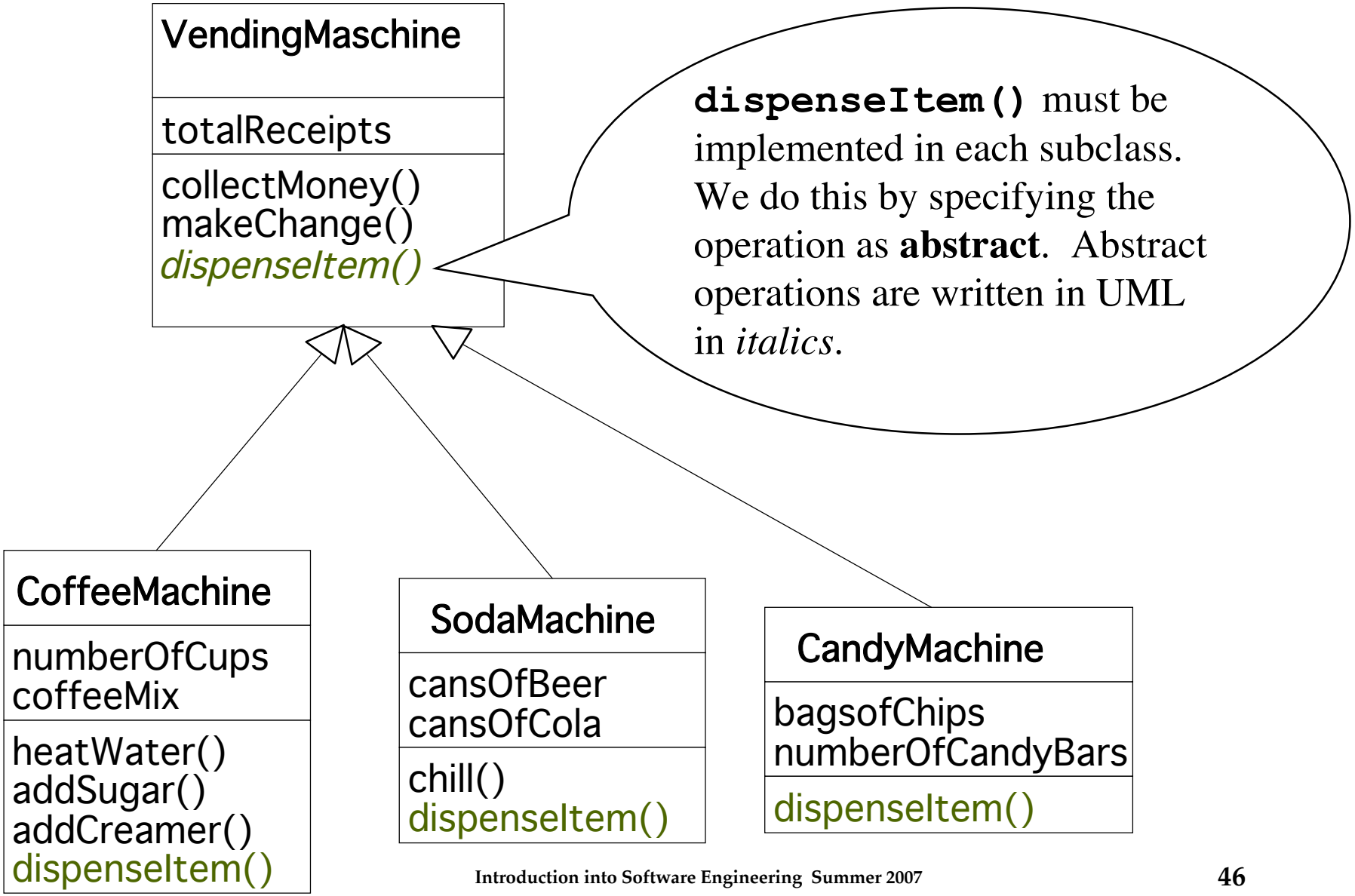
Outline of Today

- ✓ Reuse examples
  - ✓ Reuse of code, interfaces and existing classes
- ✓ White box and black box reuse
- ✓ Object design leads to new classes
- ✓ The use of inheritance
- ✓ Implementation vs. specification inheritance
- ✓ Delegation vs. Inheritance
- ⇨ Abstract classes and abstract methods
  - Overwriting methods
- Contraction: Bad example of inheritance
- Meta model for inheritance
- Frameworks and components
- Documenting the object design.

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.

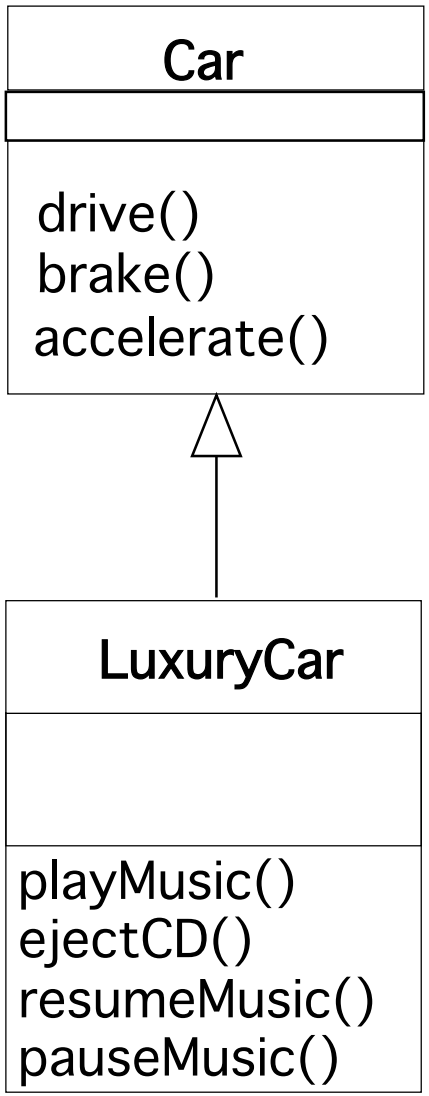
Example of an Abstract Method



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()
{…}
}
```

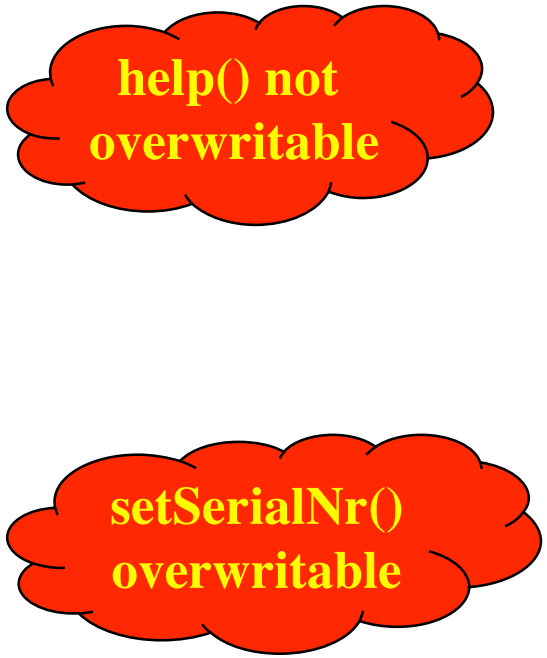
Subclass:

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

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() {
       ….
    }
}
```



Example: Overwriting a Method

Original Java-Code:

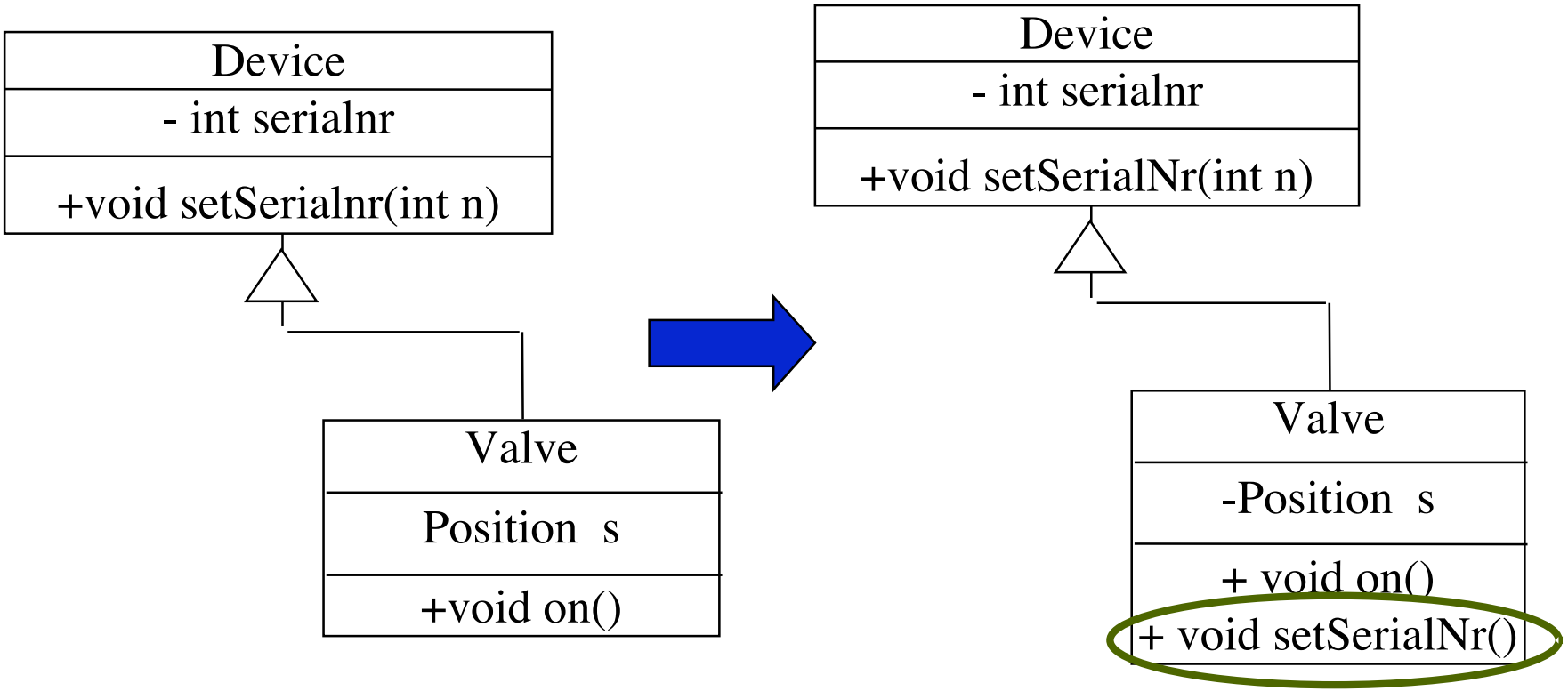
```
class Device {
    int serialnr;
    public final void help() {….}
    public void setSerialNr(int n) {
  serialnr = n;
}
}
class Valve extends Device {
    Position s;
    public void on() {
      ….
    }
}
```

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 + s.serialnr;
    }
} // class Valve
```

UML Class Diagram



Rewriteable 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) {
    seriennr = n + s.serialnr;
  }
} // 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:

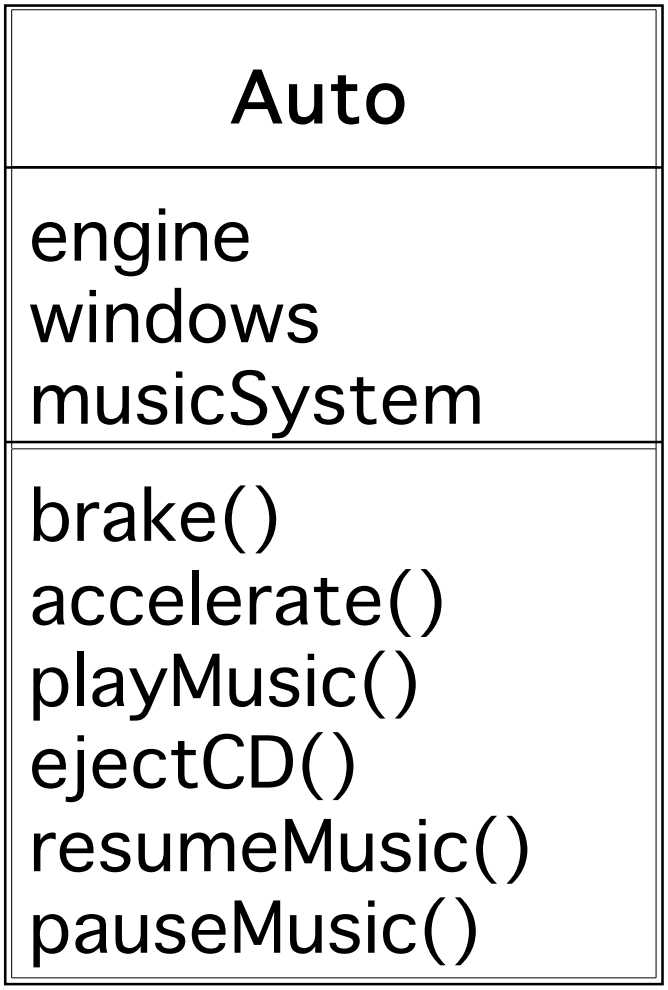
```
Public class SuperClass {
  public int add (int a, int b) { return a+b; }
  public int subtract (int a, int b) { return a-b; }
}
Public class SubClass extends SuperClass {
  public int add (int a, int b) { return a-b; }
  public int subtract (int a, int b) { return a+b; }
}
```

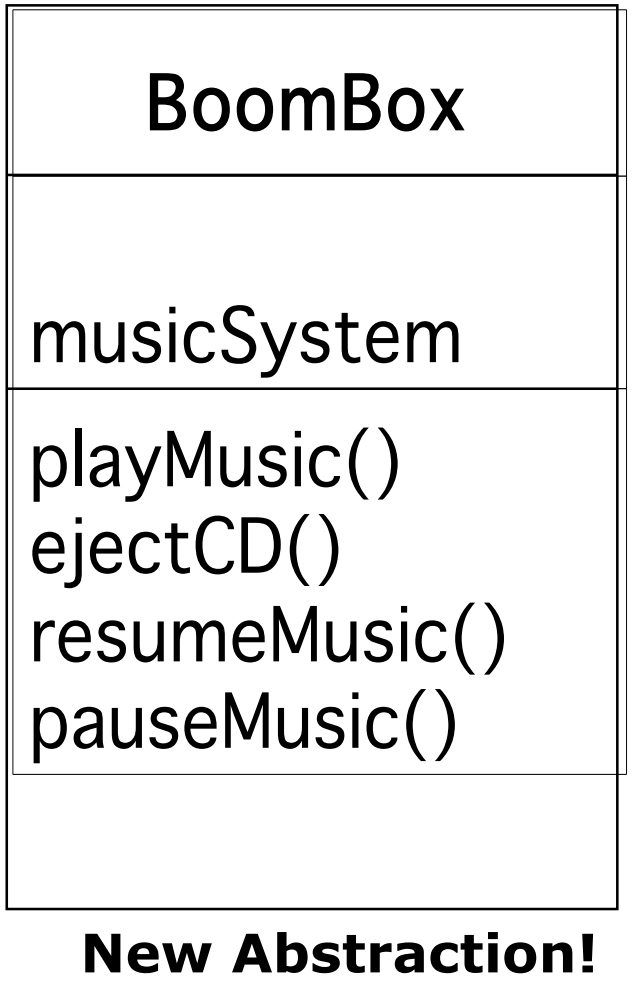
- We have redefined addition as subtraction and subtraction as addition!!

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!“

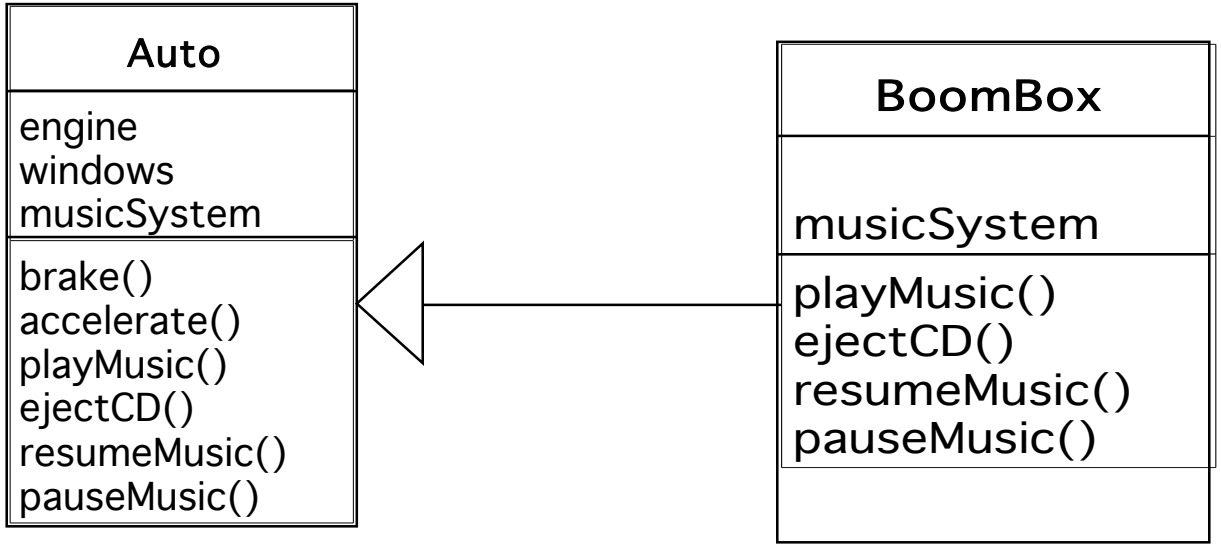
What we have and what we want





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:

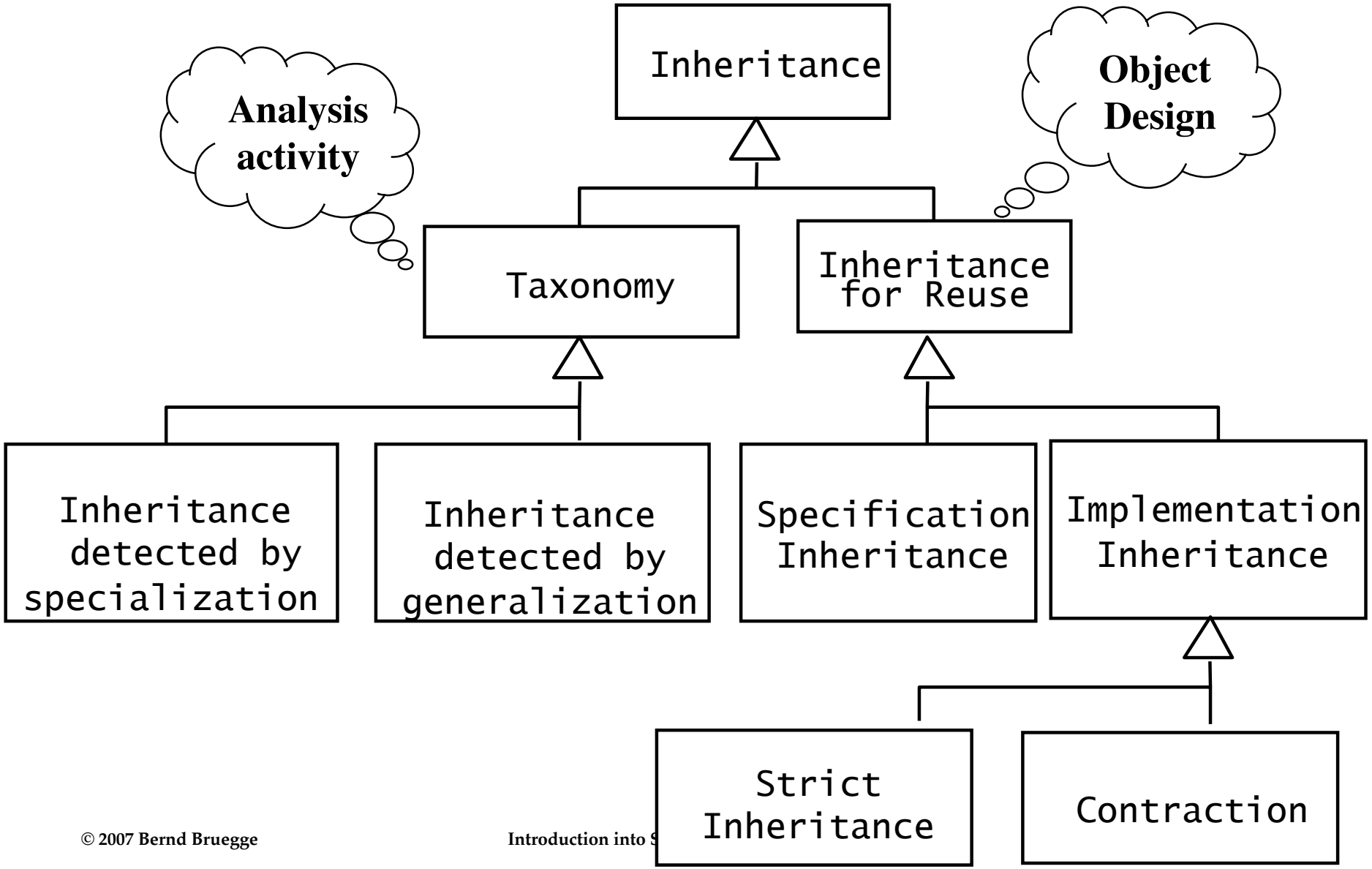
- The interface contains operations that make no sense for this class
- What is the meaning of the operation brake() for a BoomBox?

The subclass does not fit into the taxonomy

A BoomBox ist not a special form of Auto

- The subclass violates Liskov's Substitution Principle:
  - I cannot replace Auto with BoomBox to drive to work.

Revised Metamodel for Inheritance



Frameworks

- A framework is a reusable partial application that can be specialized to produce custom applications.
- The key benefits of frameworks are reusability and extensibility:
  - Reusability leverages of the application domain knowledge and prior effort of experienced developers
  - Extensibility is provided by hook methods, which are 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: MFC, DCOM, Java RMI, WebObjects, WebSphere, WebLogic Enterprise Application [BEA].
- Enterprise application frameworks are application specific and focus on domains
  - Example of application domains: telecommunications, avionics, environmental modeling, manufacturing, financial engineering, enterprise business activities.

White-box and Black-box Frameworks

- White-box frameworks:
  - Extensibility achieved through *inheritance* and dynamic binding.
  - Existing functionality is extended by subclassing framework base classes and overriding specific methods (so-called hook methods)
- Black-box frameworks:
  - Extensibility 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*.

Class libraries vs. Frameworks

- Class Library:
  - Provide a smaller scope of reuse
  - Less domain specific
  - Class libraries are passive; 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 the next lecture.

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 interface and implementation inheritance
  - Identification of opportunities for delegation
  - Abstract operations and overwriting of methods.

Example: Framework for Building Web Applications