Design Patterns

Dr. Serdar ARSLAN

CENG 522

What are Design Patterns?

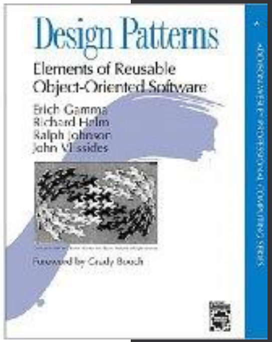
- What Are Design Patterns?
  - Wikipedia definition
    - “a design pattern is a general repeatable solution to a commonly occurring problem in software design”
  - Quote from Christopher Alexander
    - “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” (GoF,1995)

The Beginning of Patterns

- Christopher Alexander, architect
  - A Pattern Language--Towns, Buildings, Construction
  - Timeless Way of Building (1979)
  - "Each pattern describes a *problem* which occurs over and over again in our environment, and then describes the core of the *solution* to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
- Other patterns: novels (tragic, romantic, crime), movies genres (drama, comedy, documentary)

“Gang of Four” (GoF) Book

- Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1994
- Written by this "gang of four"
  - Dr. Erich Gamma, then Software Engineer, Taligent, Inc.
  - Dr. Richard Helm, then Senior Technology Consultant, DMR Group
  - Dr. Ralph Johnson, then and now at University of Illinois, Computer Science Department
  - Dr. John Vlissides, then a researcher at IBM
    - Thomas J. Watson Research Center
    - See John's WikiWiki tribute page http://c2.com/cgi/wiki?JohnVlissides



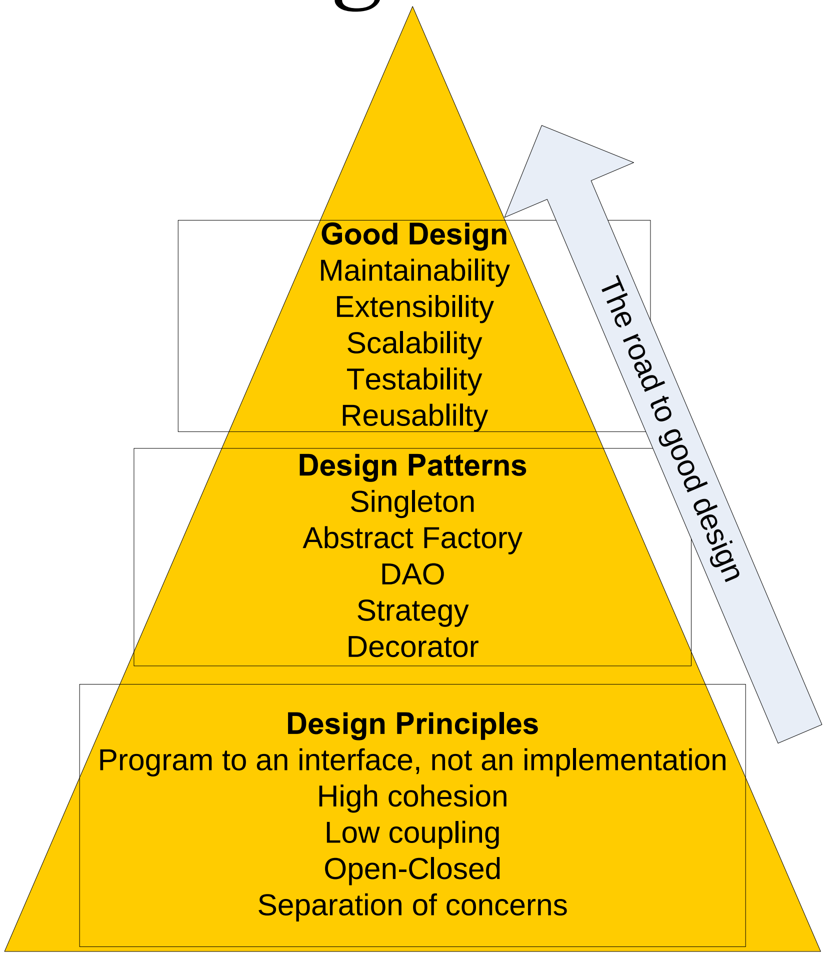
Object-Oriented Design Patterns

- This book defined 23 patterns in three categories
  - *Creational patterns* deal with the process of object creation
  - *Structural patterns*, deal primarily with the static composition and structure of classes and objects
  - *Behavioral patterns*, which deal primarily with dynamic interaction among classes and objects

Documenting Discovered Patterns

- Many other patterns have been introduced documented
  - For example, the book **Data Access Patterns** by Clifton Nock introduces 4 decoupling patterns, 5 resource patterns, 5 I/O patterns, 7 cache patterns, and 4 concurrency patterns.
  - Other pattern languages include telecommunications patterns, pedagogical patterns, analysis patterns
  - Patterns are mined at places like Patterns Conferences

Why use Design Patterns?



Why use Design Patterns?

- Design Objectives
  - Good Design (the “ilities”)
    - High readability and maintainability
    - High extensibility
    - High scalability
    - High testability
    - High reusability

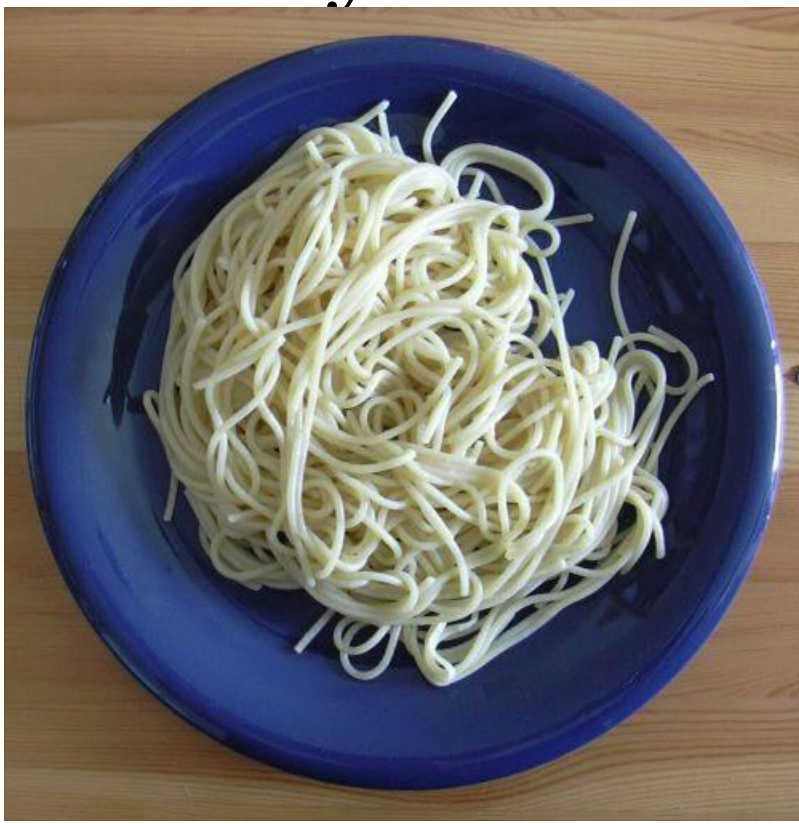
Why Study Patterns?

- Reuse tried, proven solutions
  - Provides a head start
  - Avoids gotchas later (unanticipated things)
  - No need to reinvent the wheel
- Establish common terminology
  - Design patterns provide a common point of reference
  - Easier to say, "We could use Strategy here."
- Provide a higher level prospective
  - Frees us from dealing with the details too early

Other advantages

- Most design patterns make software more modifiable, less brittle
  - we are using time tested solutions
- Using design patterns makes software systems easier to change—more maintainable
- Helps increase the understanding of basic object-oriented design principles
  - encapsulation, inheritance, interfaces, polymorphism

Why use Design 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.

Terminology: Naming of Design Activities

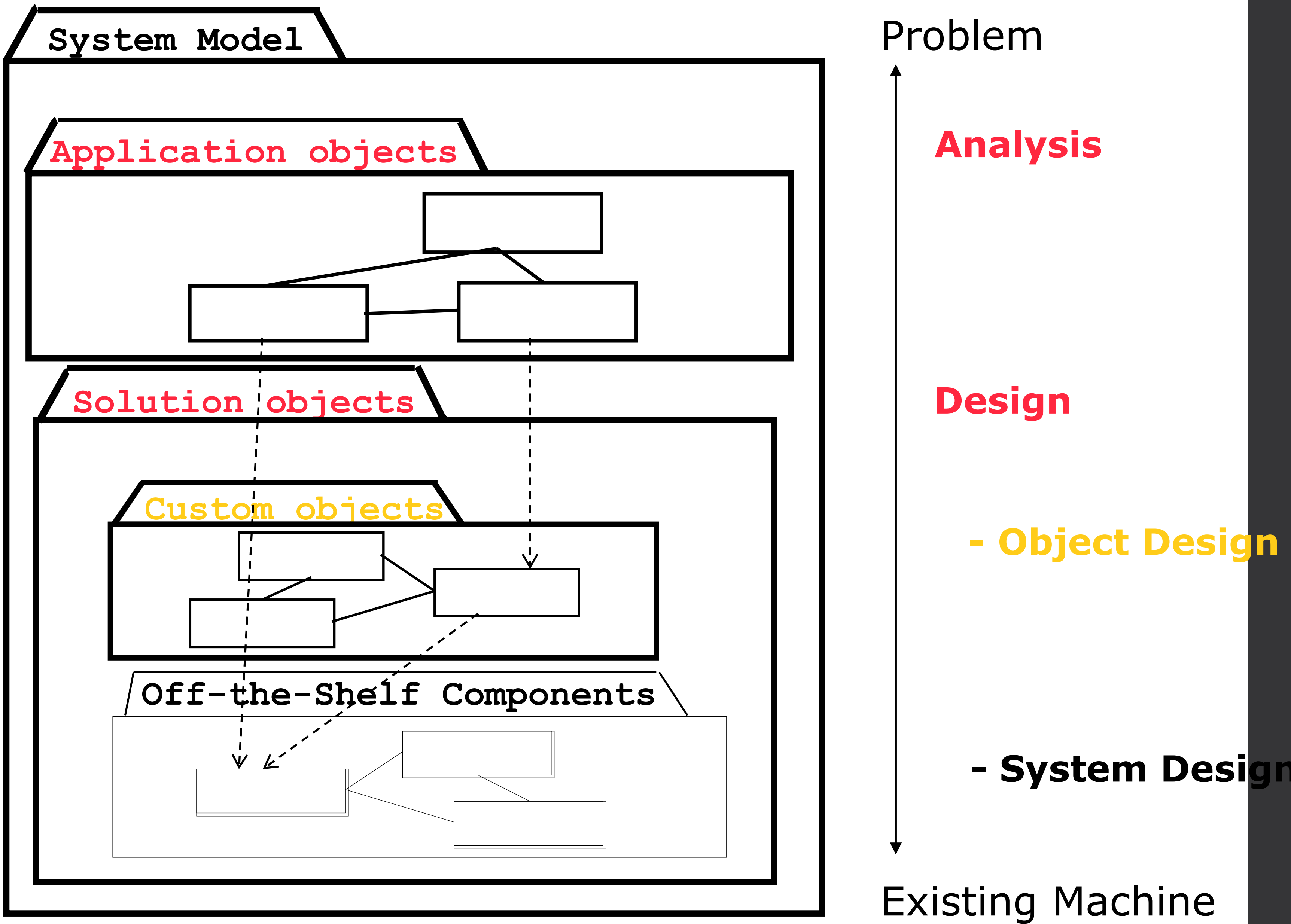
Methodology: Object-oriented software engineering (OOSE)

- *System Design*
  - Decomposition into subsystems, etc
- *Object Design*
  - Data structures and algorithms chosen
- *Implementation*
  - Implementation language is chosen

Methodology: Structured analysis/structured design (SA/SD)

- *Preliminary Design*
  - Decomposition into subsystems, etc
  - Data structures are chosen
- *Detailed Design*
  - Algorithms are chosen
  - Data structures are refined
  - Implementation language is chosen.

System Development as a Set of Activities



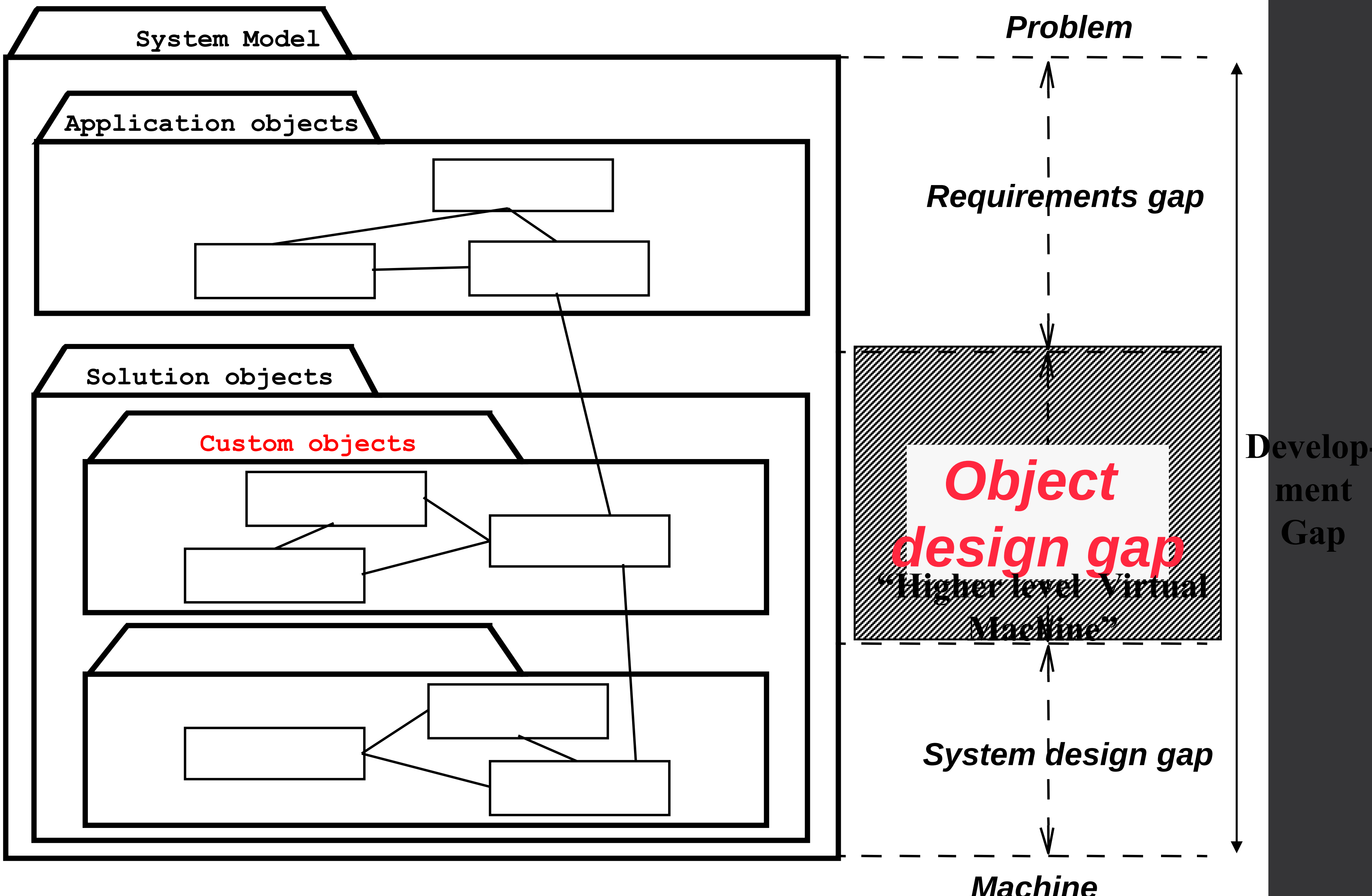
“Subsystem 1”: Rock material from the Southern Sierra Nevada mountains (moving north)

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”



Object Design consists of 4 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.

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:

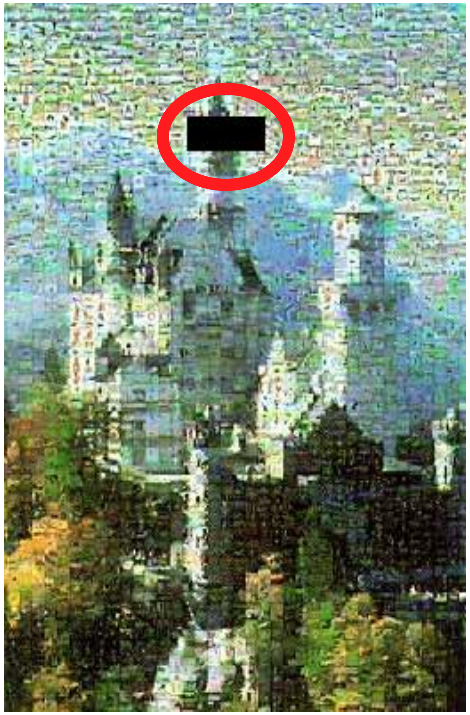
The design gap is filled with available components (“0 % coding”).

- 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.

Design with Standard Components is simil
solving a Jigsaw Puzzle

Standard Puzzles:
„Corner pieces have two straight edges“



What do we do if that is not true?“

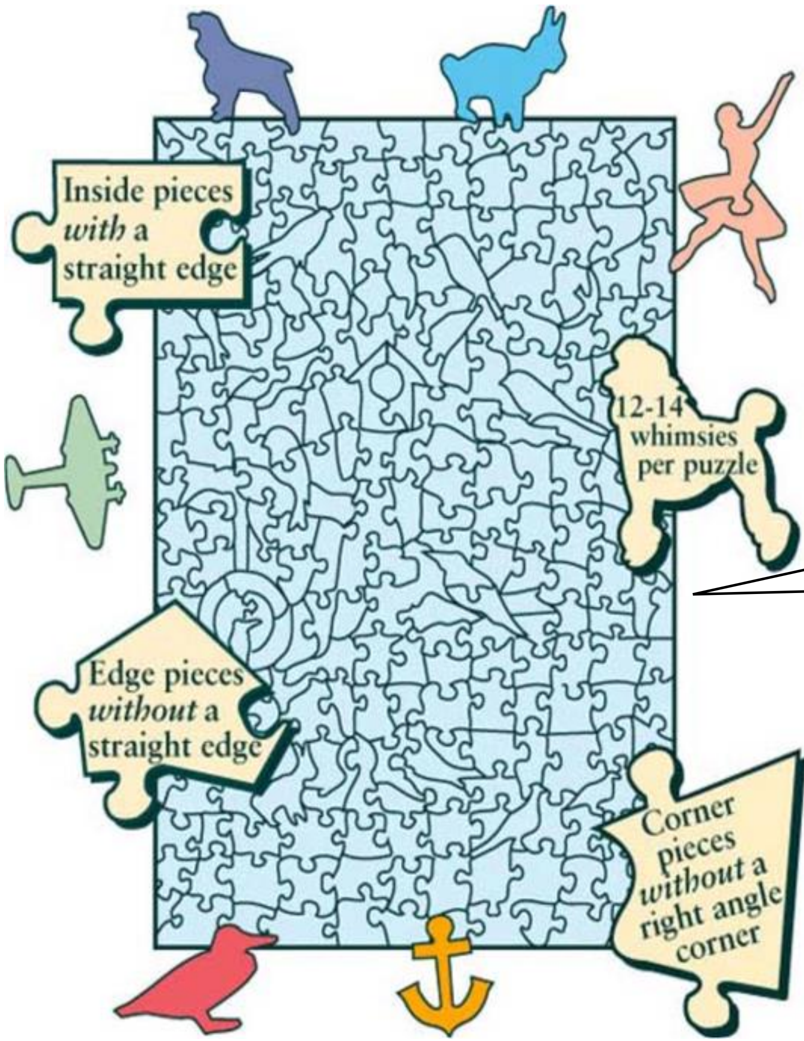
Puzzle Piece („component")

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?

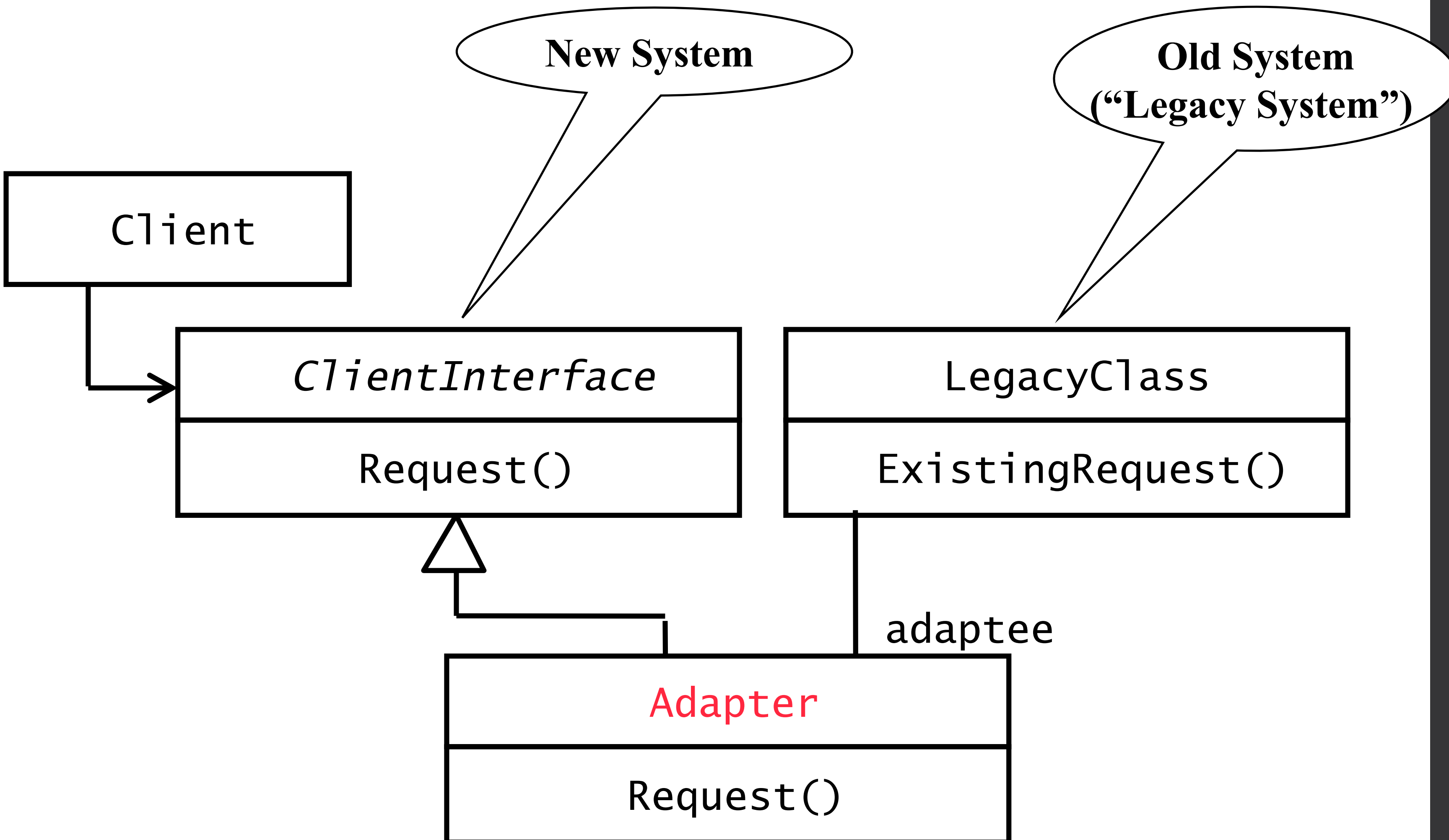


Advanced Jigsaw Puzzles

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



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
- Design knowledge such as the adapter pattern complements application domain knowledge and solution domain knowledge.

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 DaimlerChrysler
  - How can I and my contractor be 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 windows and not those of MacOS X?
- I have to develop a new service for cars, that automatically call a help center when the car is used the wrong way.
  - 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.

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

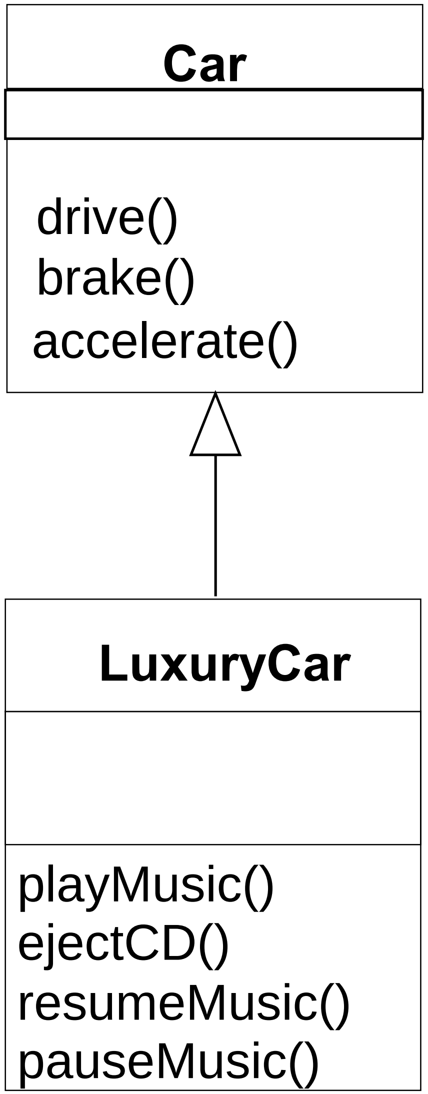
The use of Inheritance

- 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

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

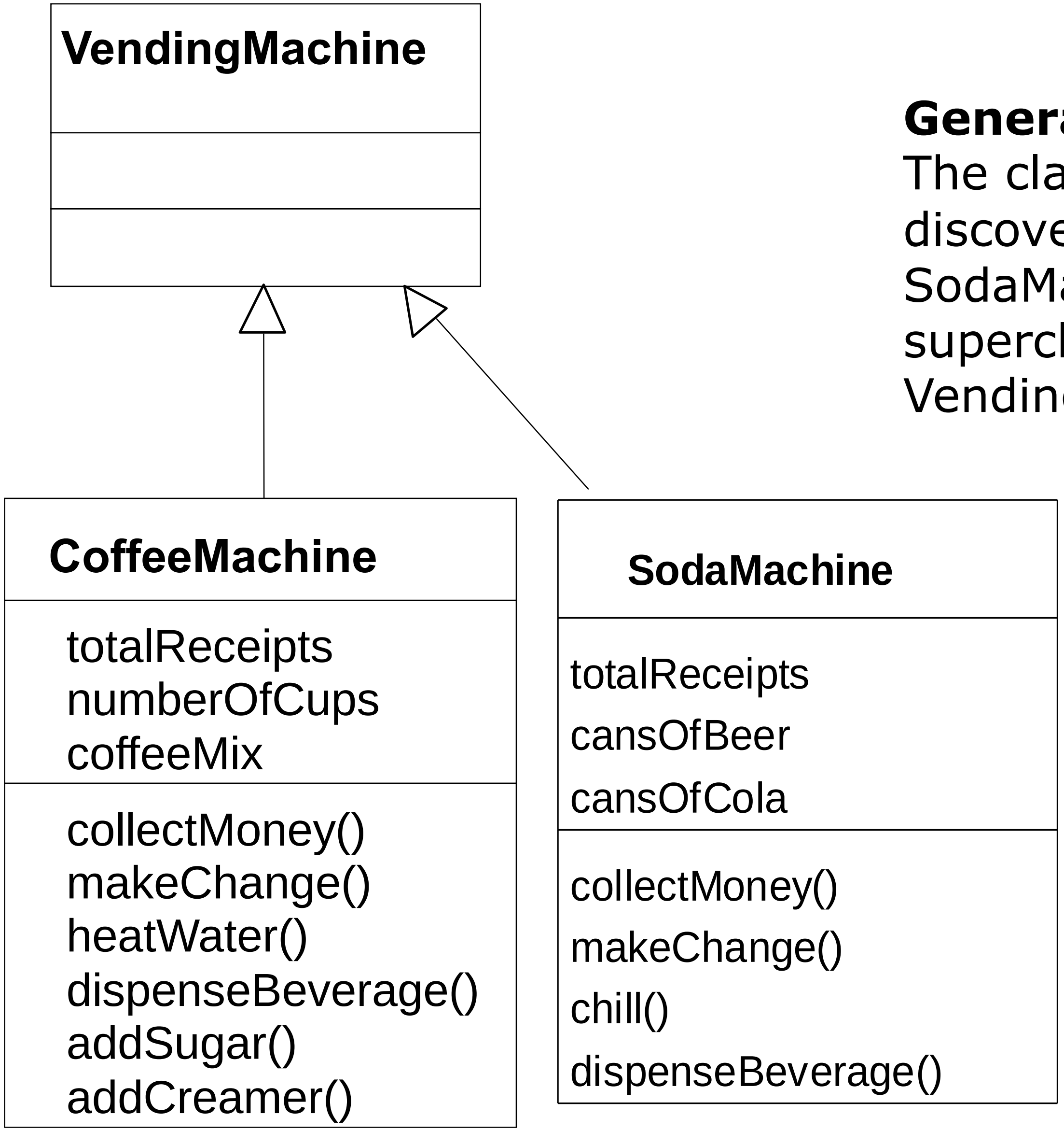
- Generalization: the discovery of an inheritance relationship between two classes, where the sub class is discovered first.

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

Generalization

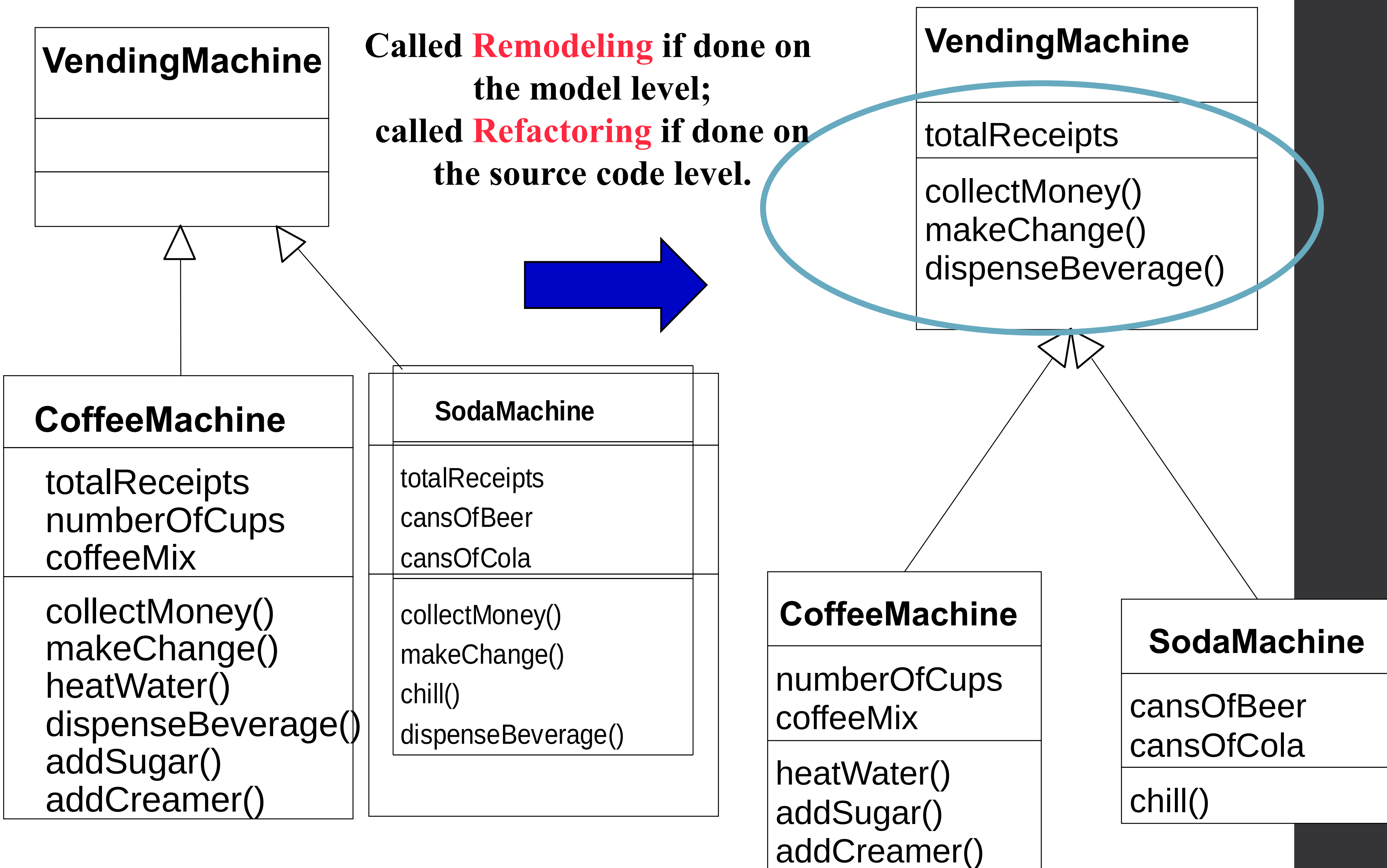
- First we find the subclass, then the super class
- This type of discovery occurs often in science and engineering:
  - **Biology**: First we find individual animals (Elefant, Lion, Tiger), then we discover that these animals have common properties (mammals).
  - **Engineering:** What are the common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

Generalization Example: Modeling a Coffee Machine



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

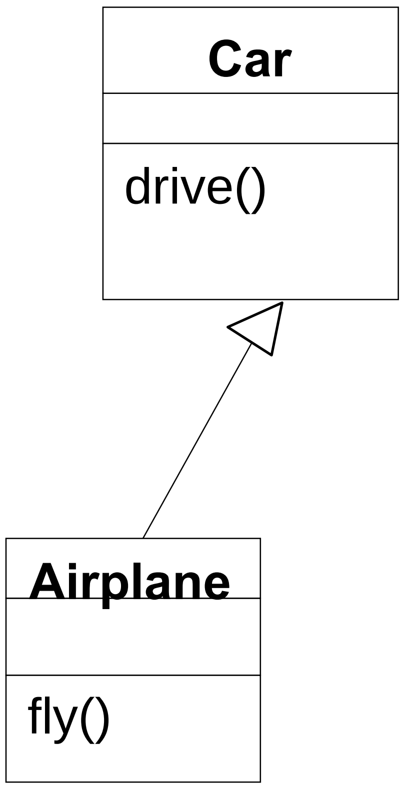
Restructuring of Attributes and Operations is often a Consequence of Generalization

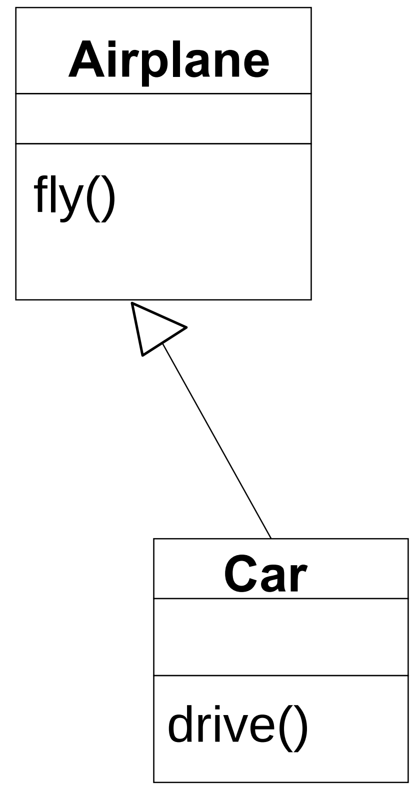


Specialization

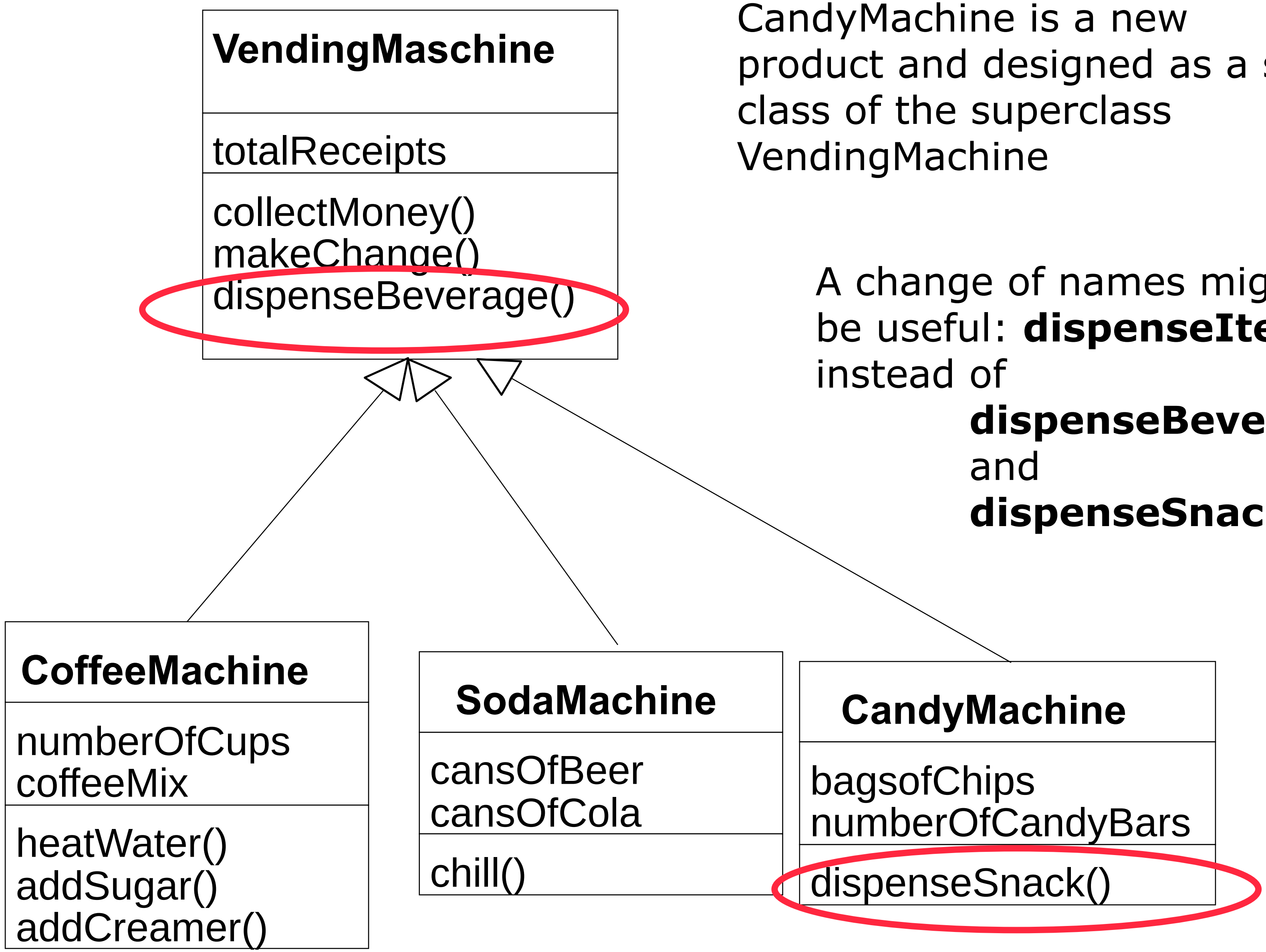
- Specialization occurs, when we find a subclass that is very similar to an existing class.
  - Example: A theory postulates certain particles and events which we have to find.
- Specialization can also occur unintentionally:

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





Another Example of a Specialization

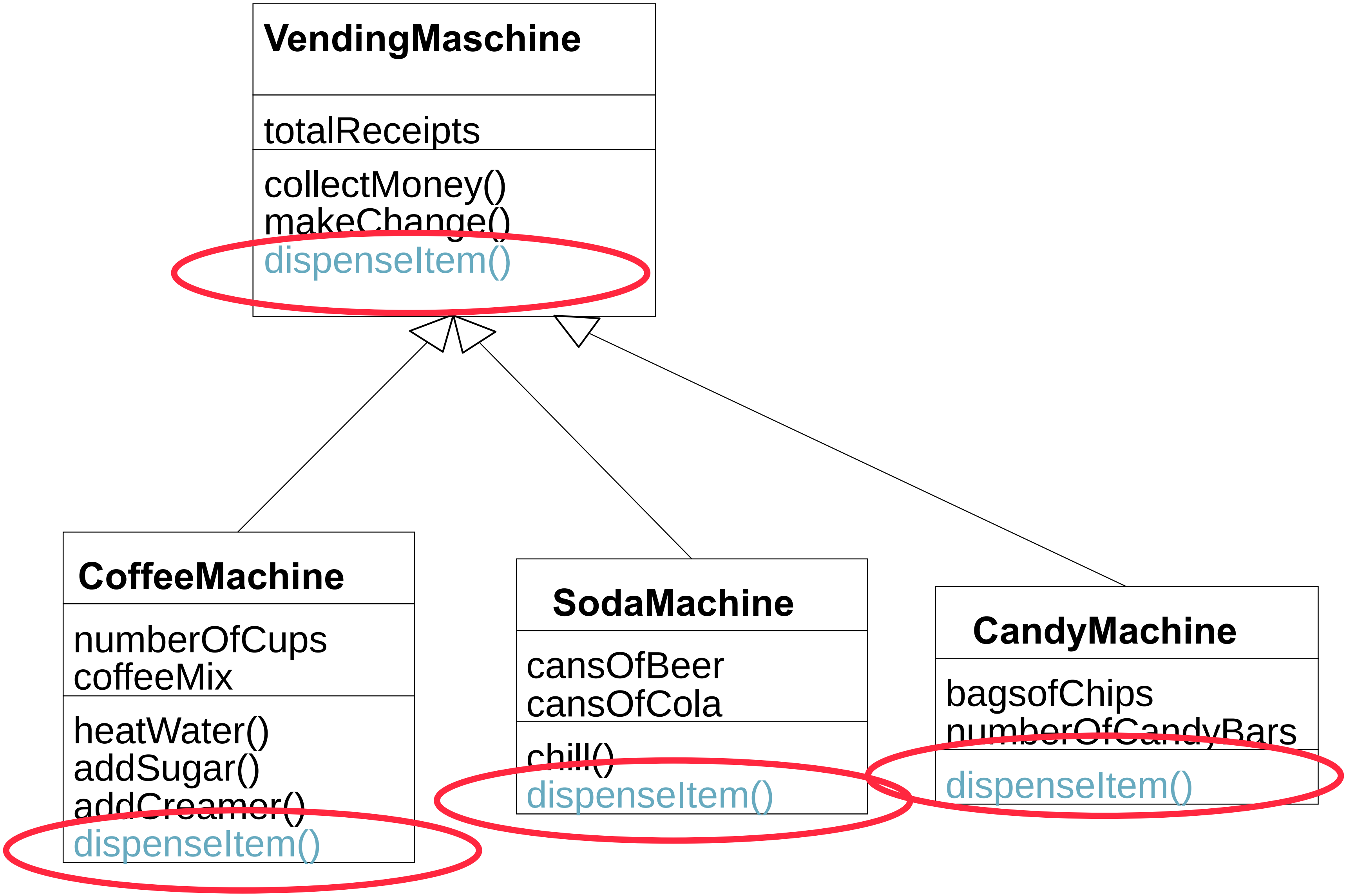


CandyMachine is a new product and designed as a sub class of the superclass VendingMachine

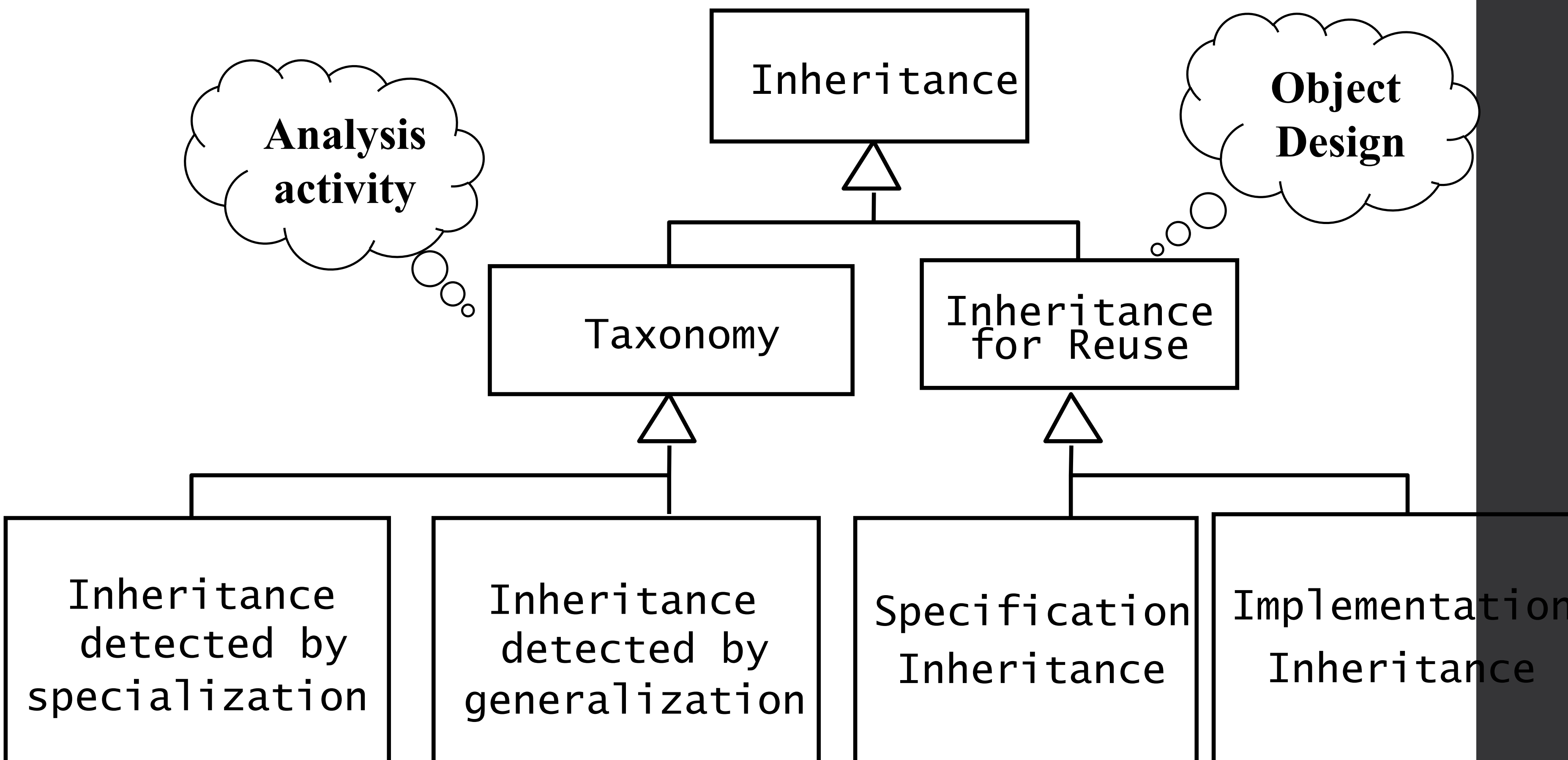
A change of names might now be useful: **dispenseItem()** instead of

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

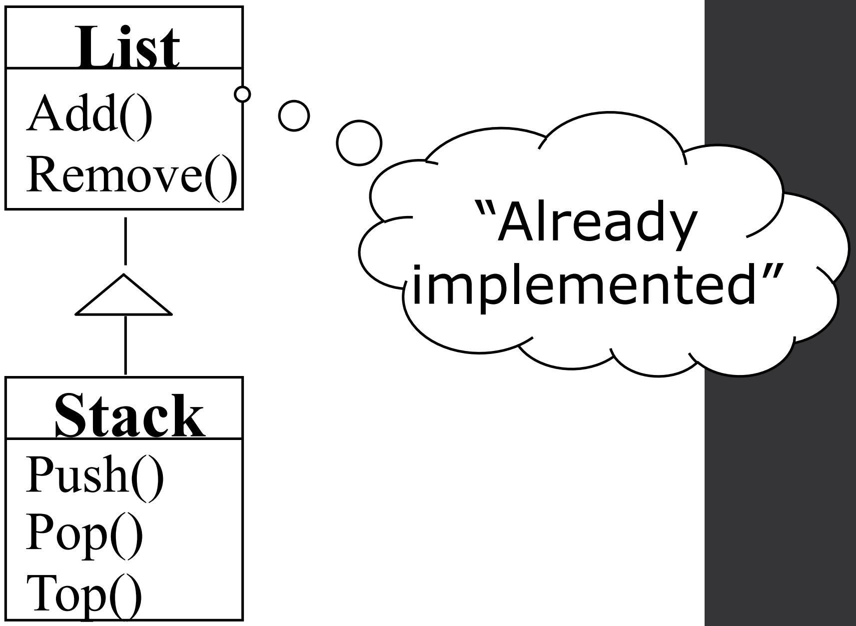
- Implementation Inheritance: The combination of inheritance and implementation
  - The Interface of the superclass is completely inherited
  - Implementations of methods in the superclass ("Reference implementations") are inherited by any subclass
- Specification Inheritance: The combination of inheritance 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()**?

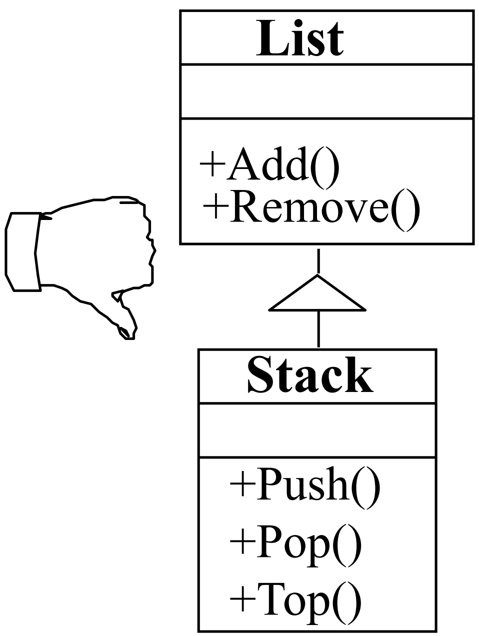


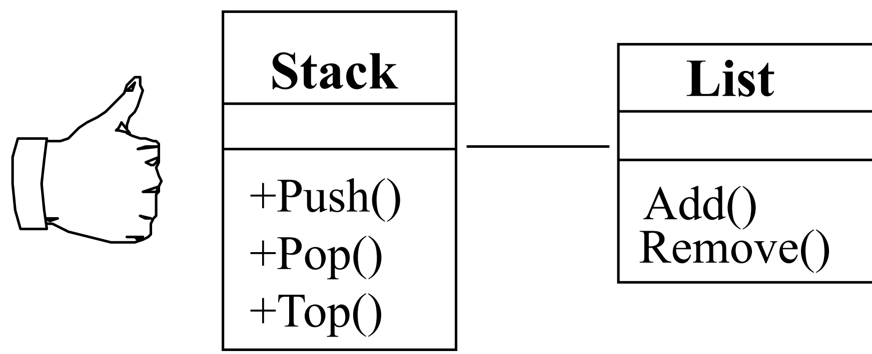
❖ Problem with implementation inheritance:

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

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?

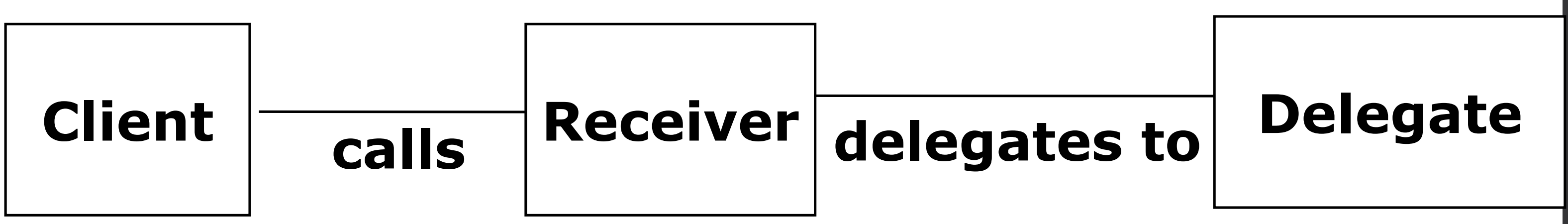




Delegation

- 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
  - ☹ 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)

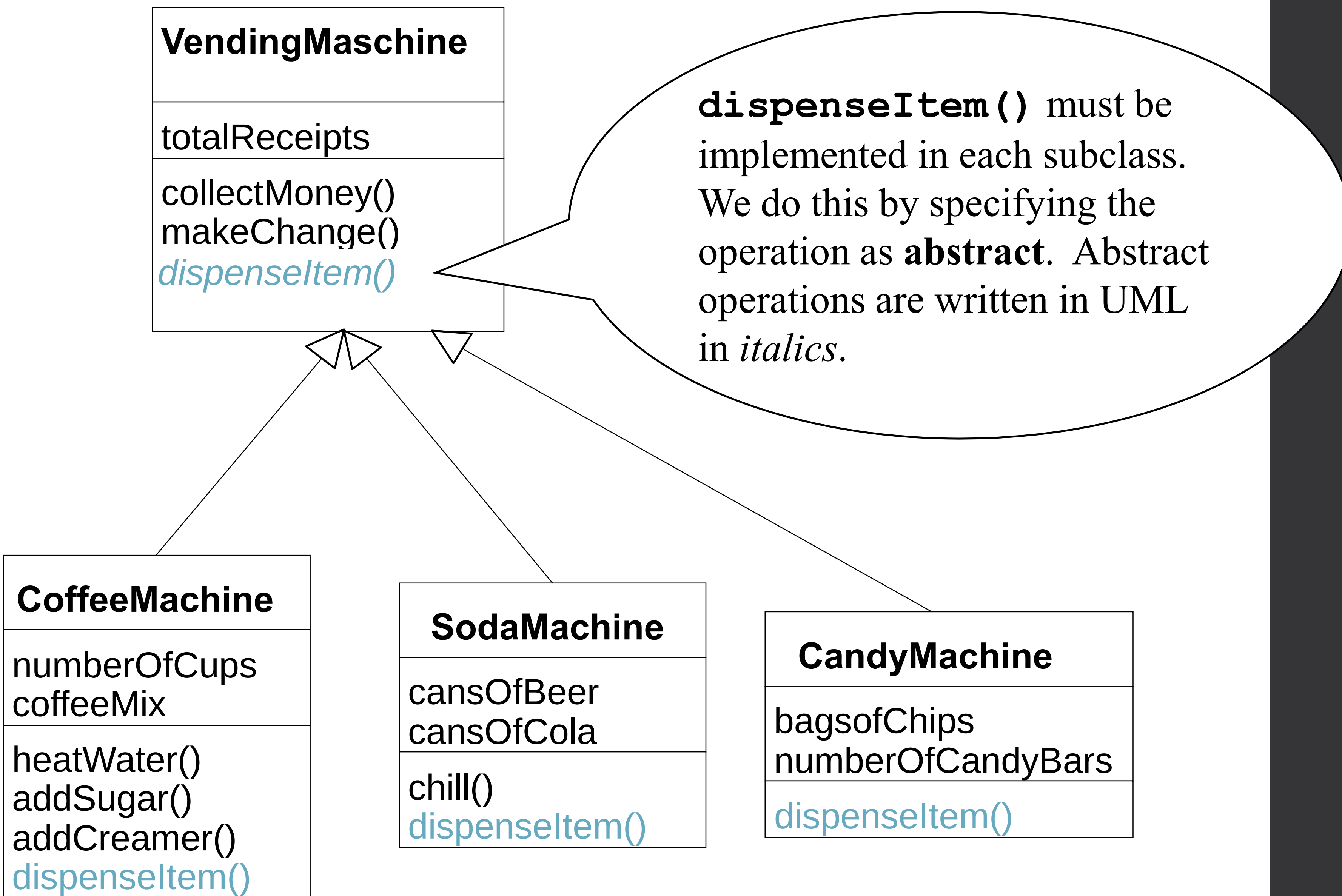
Comparison: Delegation v. Inheritance

- 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.

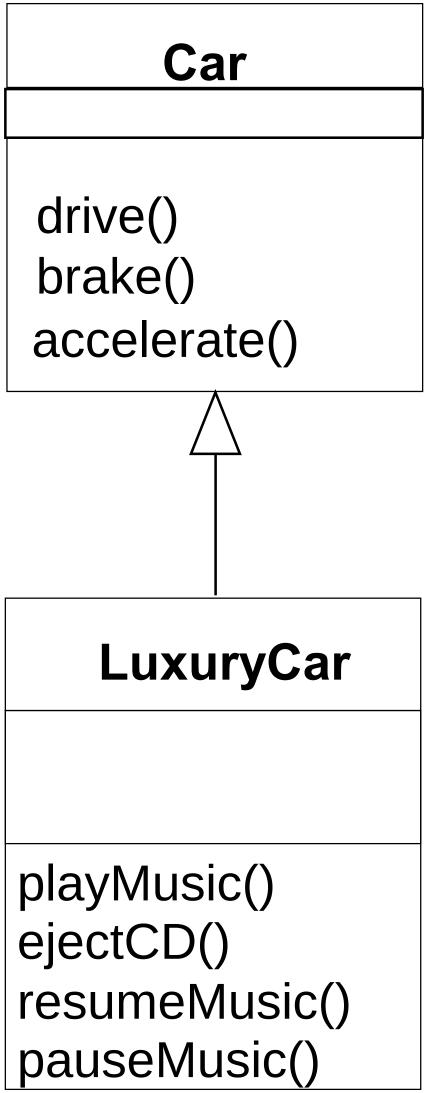
Example of an Abstract Method



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() {…}
}
```

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:

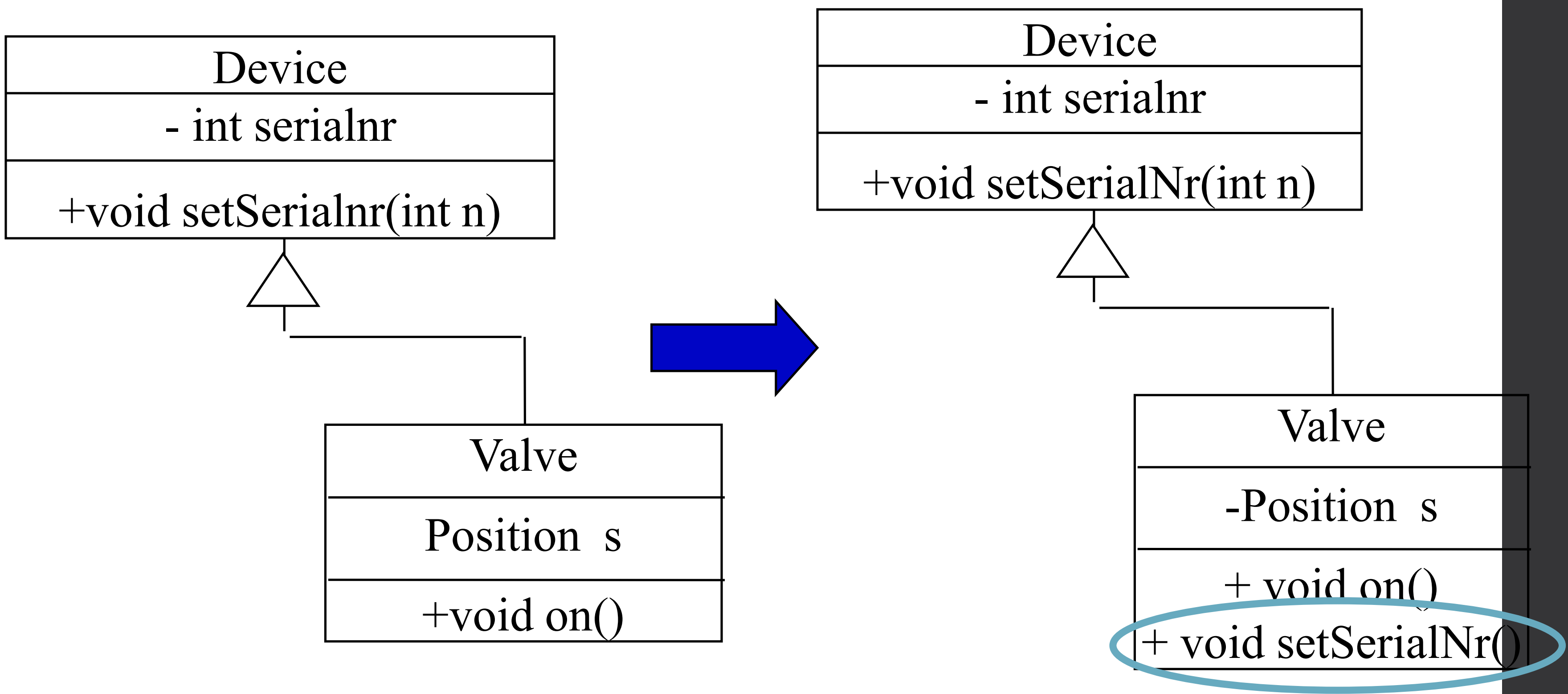
```
class Device {
    int serialnr;
    public final void help() {….}
    public void setSerialNr(int n) {
  serialnr = n;
}
}
class Valve extends Device {
    Position s;
    public void on() {
      ….
    }
}
```

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!“

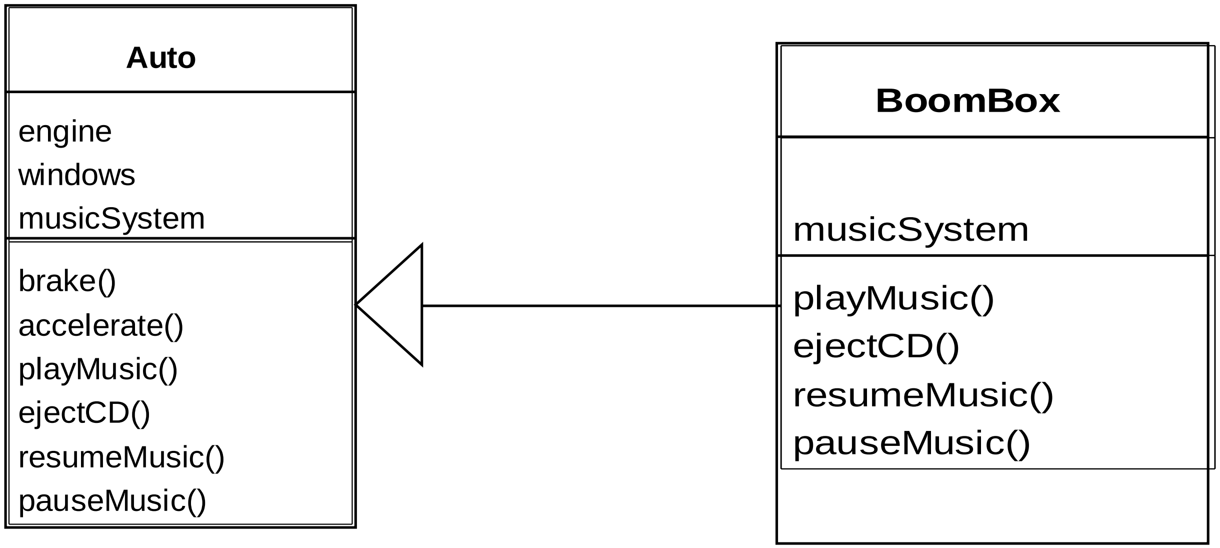
What we have and what we want

Auto
engine windows musicSystem
brake() accelerate() playMusic() ejectCD() resumeMusic() pauseMusic()

BoomBox
musicSystem
playMusic() ejectCD() resumeMusic() pauseMusic()

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:

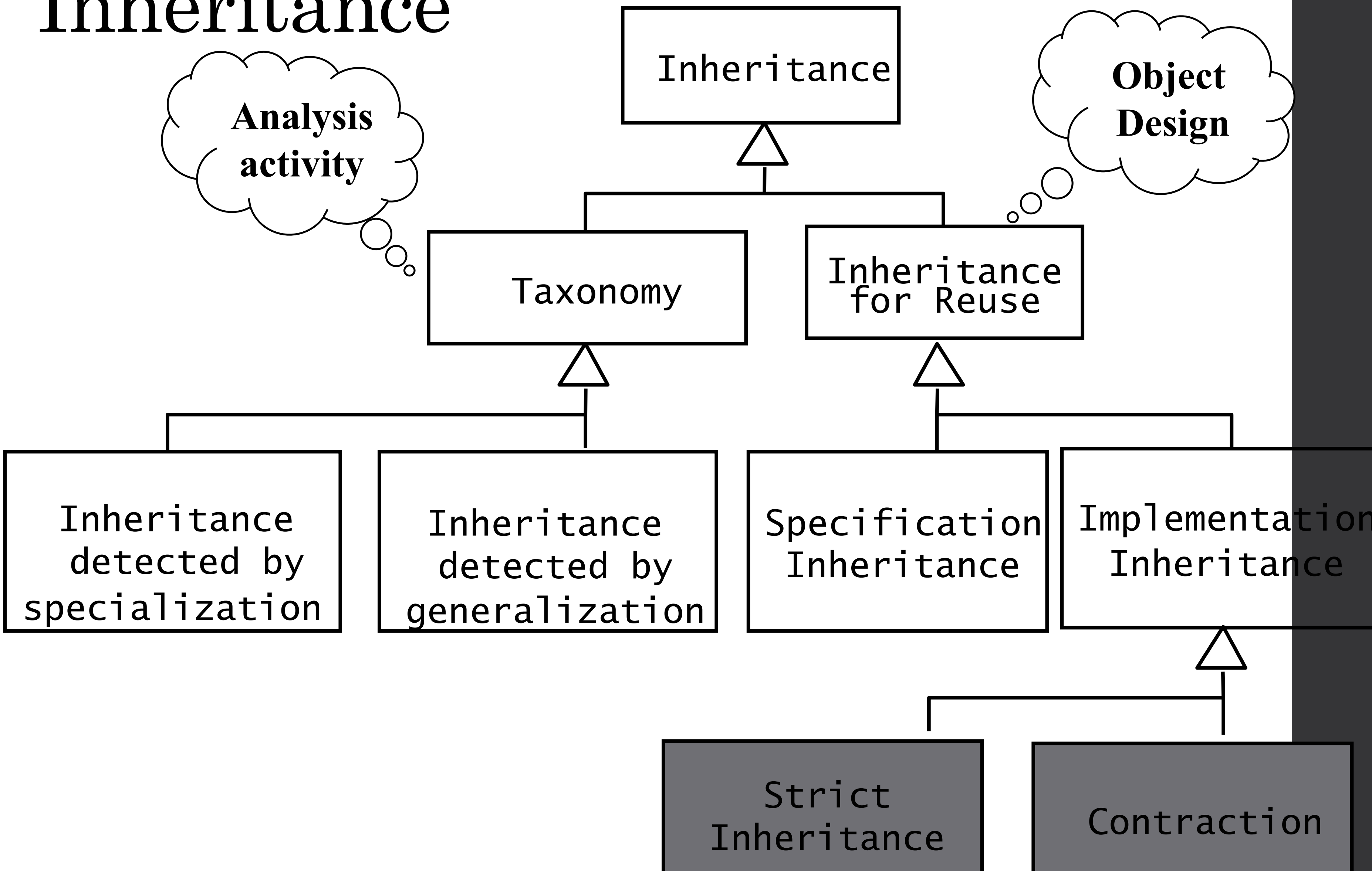
- The interface contains operations that make no sense for this class
- What is the meaning of the operation brake() for a BoomBox?

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.

Design Patterns

Again ☺

Elements of a Design Pattern

- A pattern has four essential elements (GoF)
  - Name
    - Describes the pattern
    - Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)
  - Problem
    - Describes when to apply the pattern
    - Answers - What is the pattern trying to solve?

Elements of a Design Pattern (cont.)

- Solution
  - Describes elements, relationships, responsibilities, and collaborations which make up the design
- Consequences
  - Results of applying the pattern
  - Benefits and Costs
  - Subjective depending on concrete scenarios

Design Patterns Classification

- Creational
  - Factory Pattern
  - Abstract Factory Pattern
  - Singleton Pattern
  - Prototype Pattern
  - Builder Pattern.
- Structural
  - Adapter Pattern
  - Bridge Pattern
  - Composite Pattern
  - Decorator Pattern
  - Facade Pattern
  - Flyweight Pattern
  - Proxy Pattern
- Behavioral
  - Chain Of Responsibility Pattern
  - Command Pattern
  - Interpreter Pattern
  - Iterator Pattern
  - Mediator Pattern
  - Memento Pattern
  - Observer Pattern
  - State Pattern
  - Strategy Pattern
  - Template Pattern
  - Visitor Pattern

Pros/Cons of Design Patterns

- Pros
  - Add **consistency** to designs by solving similar problems the same way, independent of language
  - Add **clarity** to design and design communication by enabling a common vocabulary
  - Improve **time** to solution by providing templates which serve as foundations for good design
  - Improve **reuse** through composition

Pros/Cons of Design Patterns

- Cons
  - Some patterns come with negative consequences (i.e. object proliferation, performance hits, additional layers)
  - Consequences are subjective depending on concrete scenarios
  - Patterns are subject to different interpretations, misinterpretations, and philosophies
  - Patterns can be overused and abused → Anti-Patterns

Popular Design Patterns

- We will look at following patterns;
  - Factory
  - Abstract Factory
  - Singleton
  - Decorator
  - Façade
  - Adapter
  - And more….

SOLID Principles

SOLID

- The **S**ingle Responsibility Principle
- The **O**pen-Closed Principle
- The **L**iskov Substitution Principle
- The **I**nterface Segregation Principle
- The **D**ependency Inversion Principle

SOLID

- Why?
  - "To create understandable, readable, and testable code that many developers can collaboratively work on.“

S — Single Responsibility

- A class should have a single responsibility
- This principle aims to separate behaviours so that if bugs arise as a result of your change, it won't affect other unrelated behaviours.

O — Open-Closed

- Classes should be open for extension, but closed for modification
- This principle aims to extend a Class’s behaviour without changing the existing behaviour of that Class. This is to avoid causing bugs wherever the Class is being used.

L — Liskov Substitution

- If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.
- This principle aims to enforce consistency so that the parent Class or its child Class can be used in the same way without any errors.

I — Interface Segregation

- Clients should not be forced to depend on methods that they do not use.
- This principle aims at splitting a set of actions into smaller sets so that a Class executes ONLY the set of actions it requires.

D — Dependency Inversion

- High-level modules should not depend on low-level modules. Both should depend on the abstraction.
- Abstractions should not depend on details. Details should depend on abstractions.
- This principle aims at reducing the dependency of a high-level Class on the low-level Class by introducing an interface.

Questions?

Thank You!