Object Design: Reuse, Part 2

Software Engineering I
Lecture 12

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen

Outline of Today

- Abstract Methods and Abstract Classes
- Overwriting of Methods
- Simple Inheritance
- Contraction

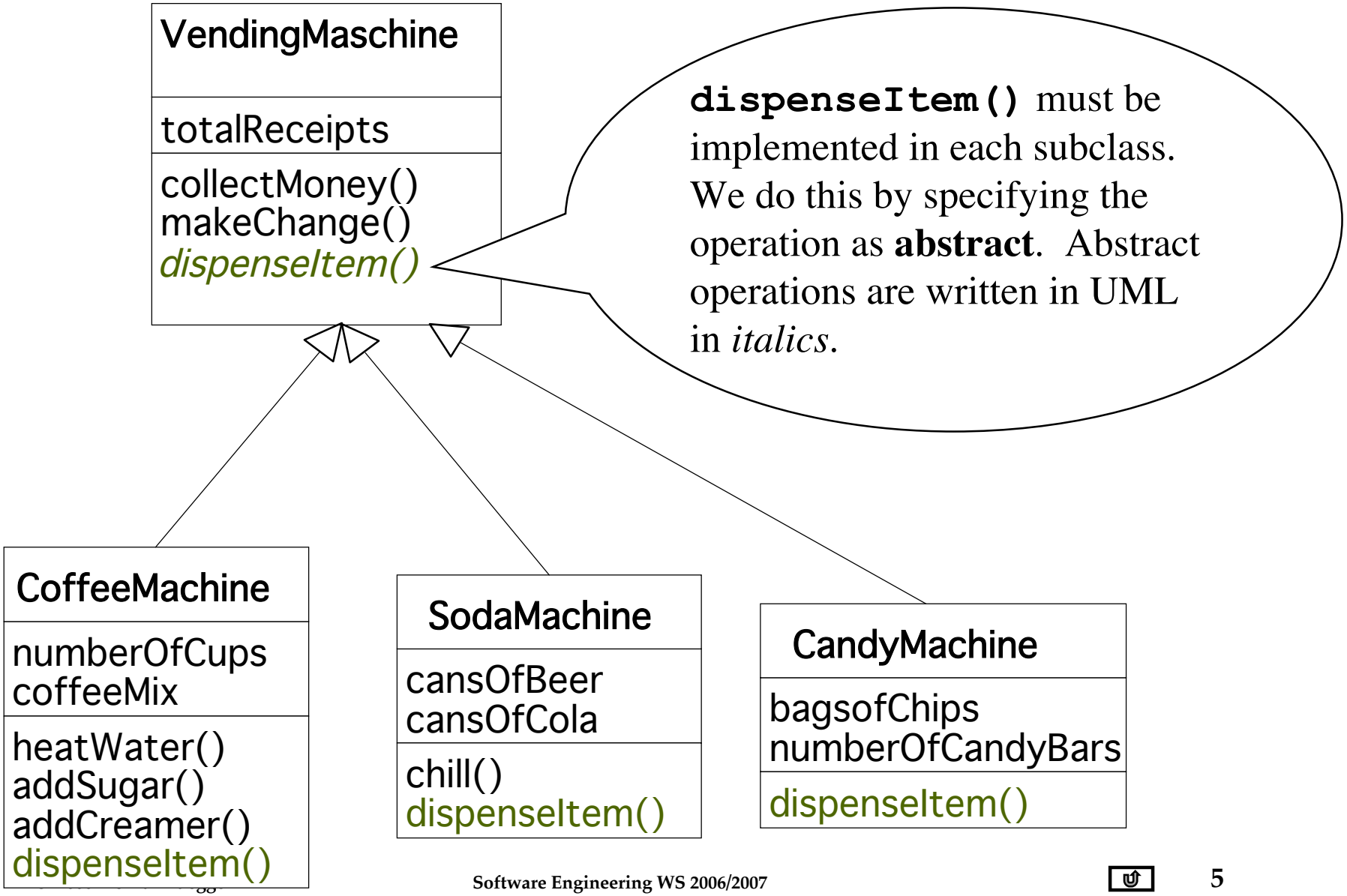
Recall: Implementation Inheritance v. Specification-Inheritance

- 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

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.

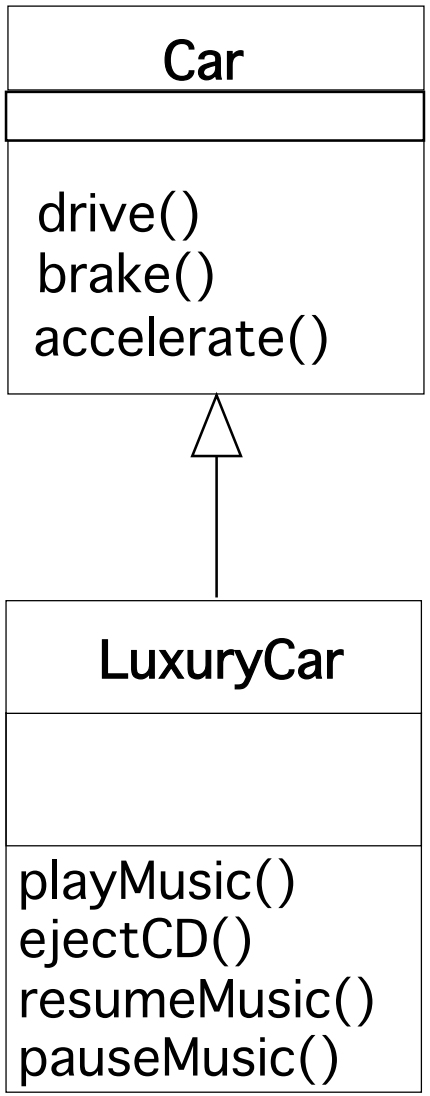
Example of an Abstract Method



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()
{…}
}
```

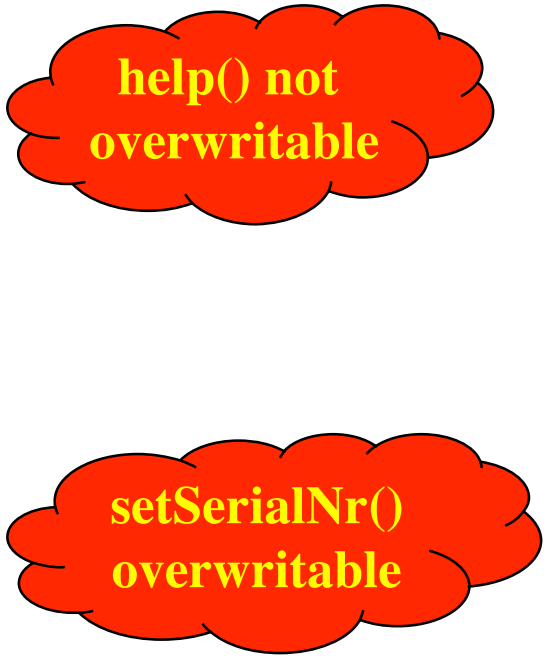
Subclass:

```
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:

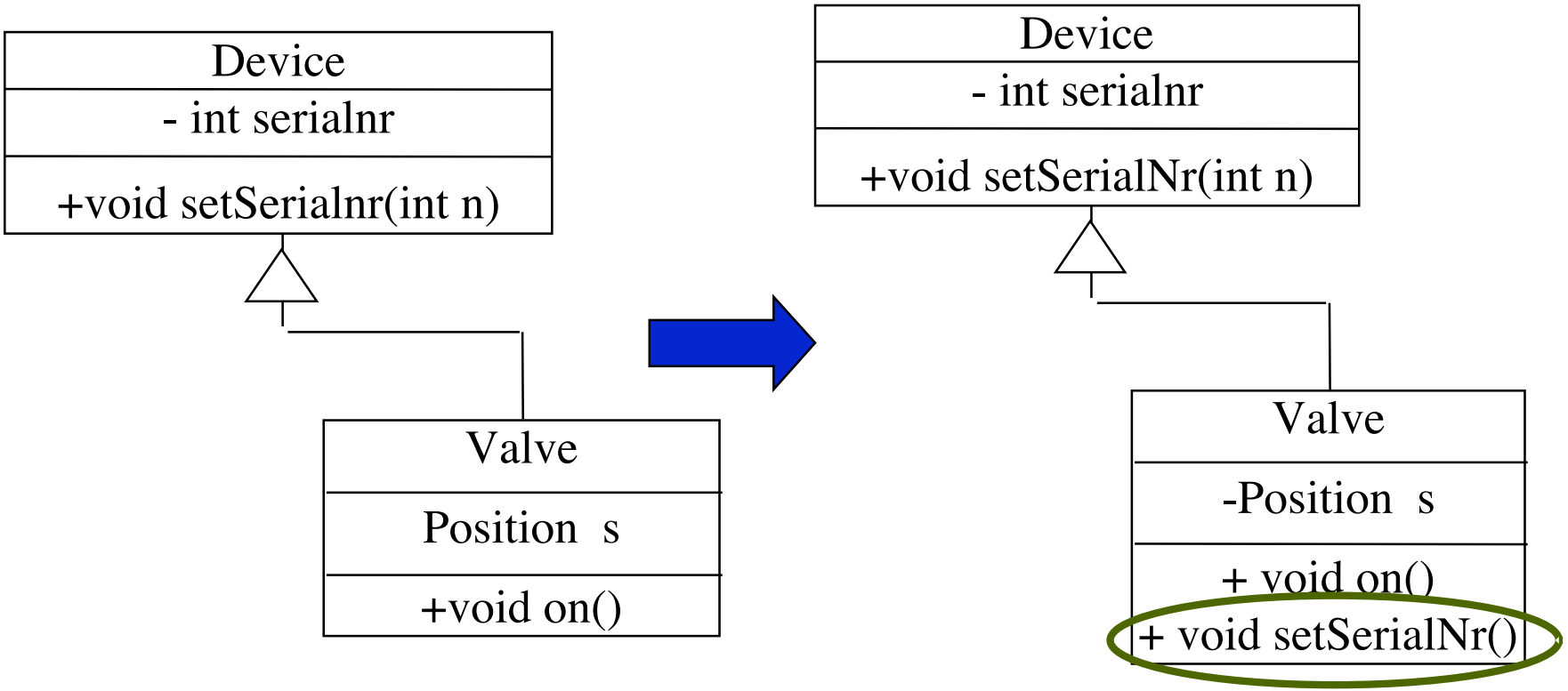
```
class Device {
    int serialnr;
    public final void help() {….}
    public void setSerialNr(int n) {
   serialnr = n;
}
}
class Valve extends Device {
    Position s;
    public void on() {
      ….
    }
}
```

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:

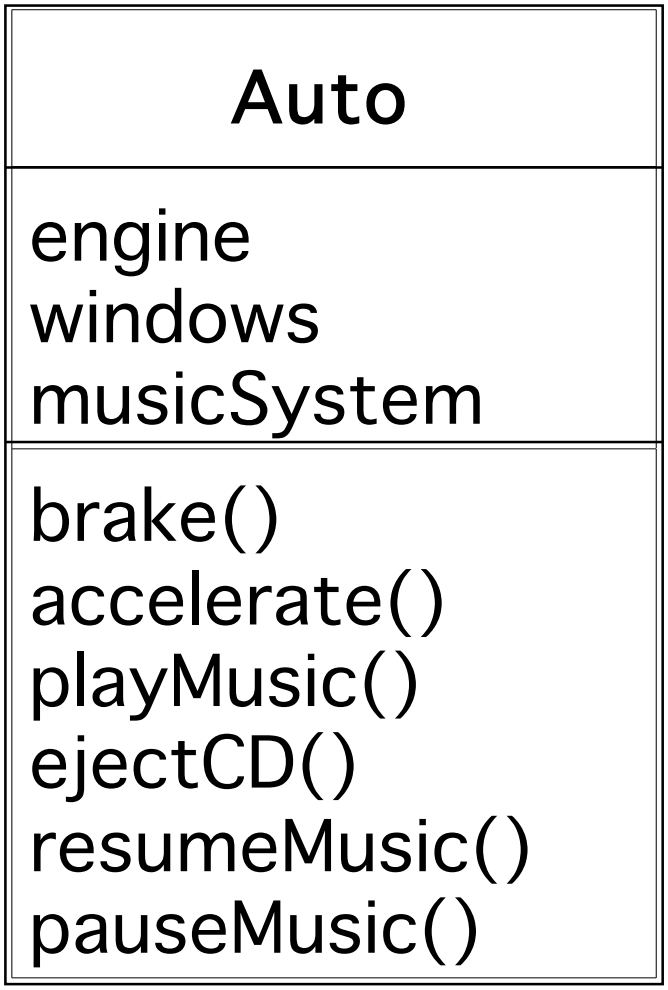
```
Public class SuperClass {
  public int add (int a, int b) { return a+b; }
  public int subtract (int a, int b) { return a-b; }
}
Public class SubClass extends SuperClass {
  public int add (int a, int b) { return a-b; }
  public int subtract (int a, int b) { return a+b; }
}
```

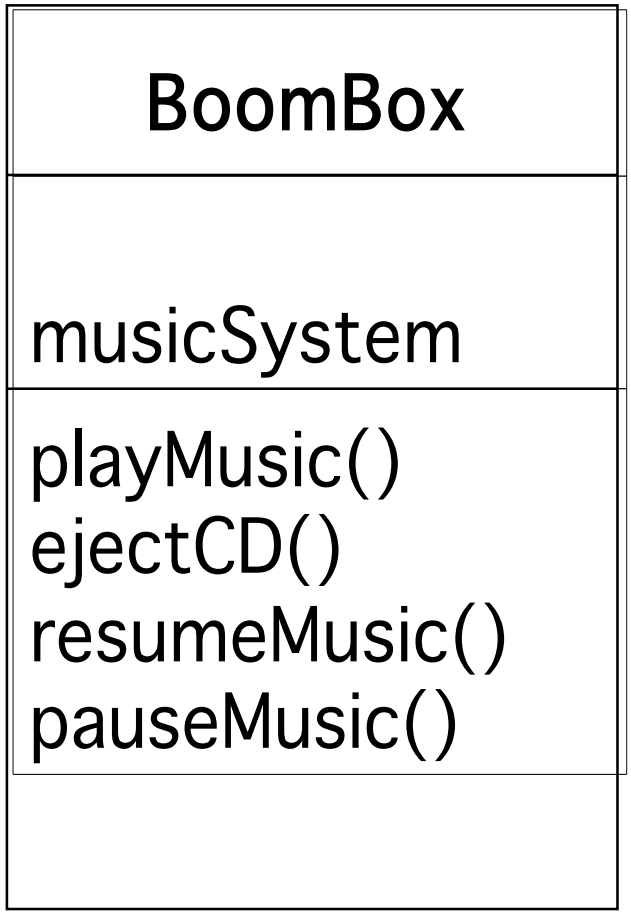
- We have redefined addition as subtraction and subtraction as addition!!

Bad Use of Implementation Inheritance

- We have delivered a car with software that allows to operate a 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 all the operations from the existing Car software“
    - „All method implementations from Car that have nothing to do with playing music will be overwritten with empty bodies!“

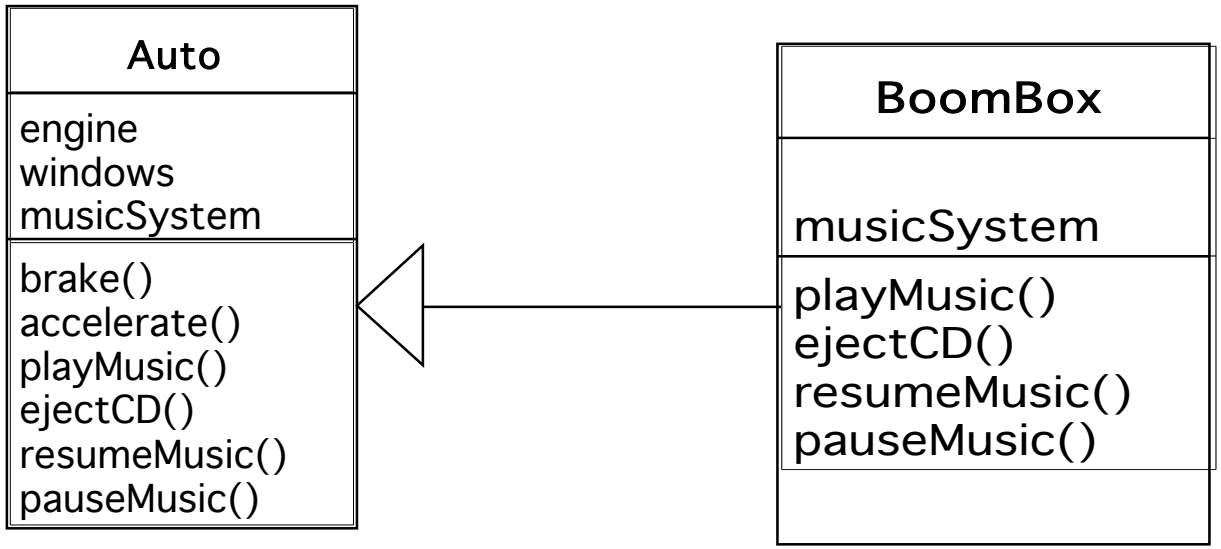
What we have and what we want





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:

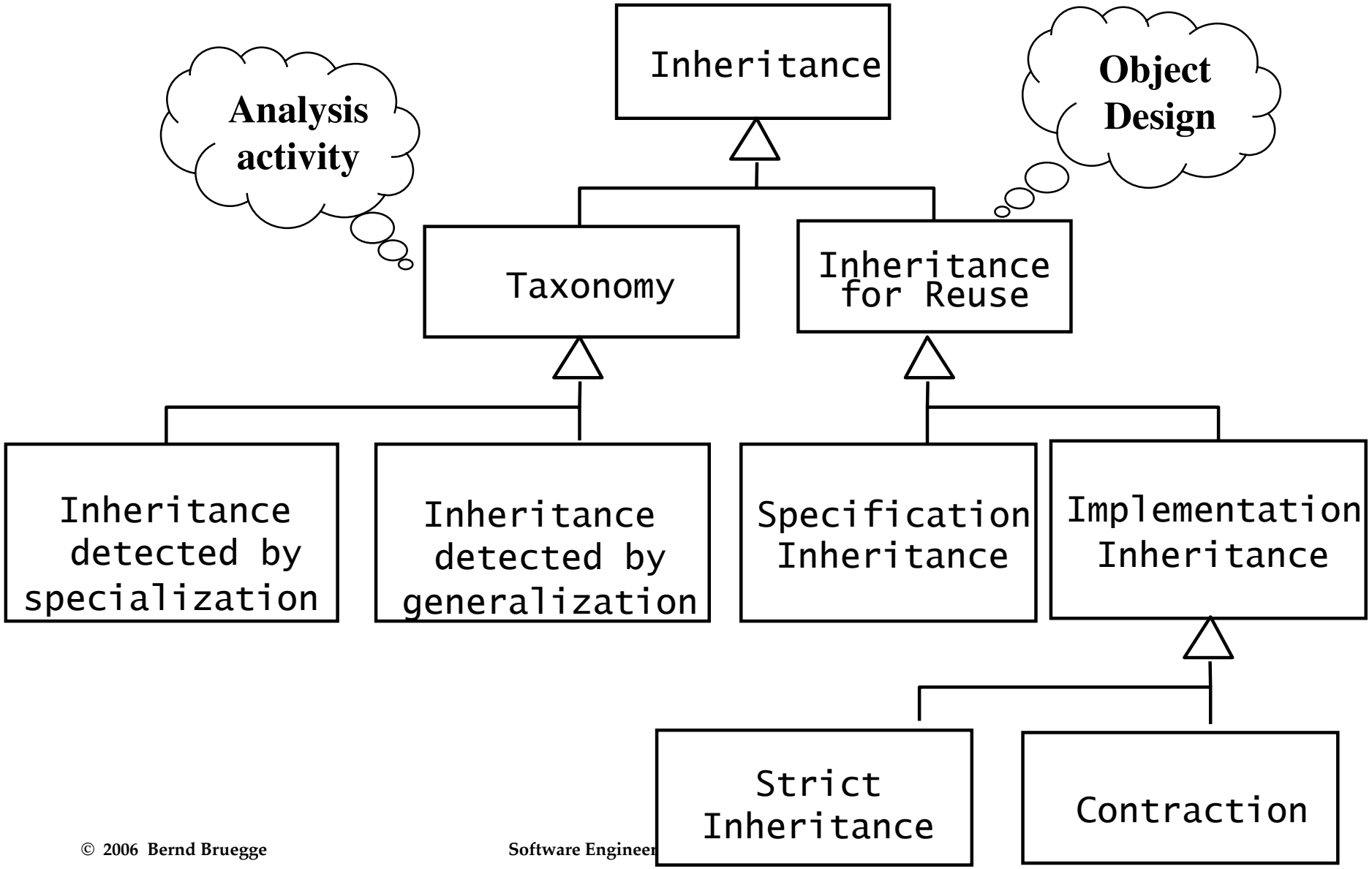
- The interface contains operations that make no sense for this class.
- What is the meaning of the operation brake() for a BoomBox?

The subclass does not fit into the taxonomy

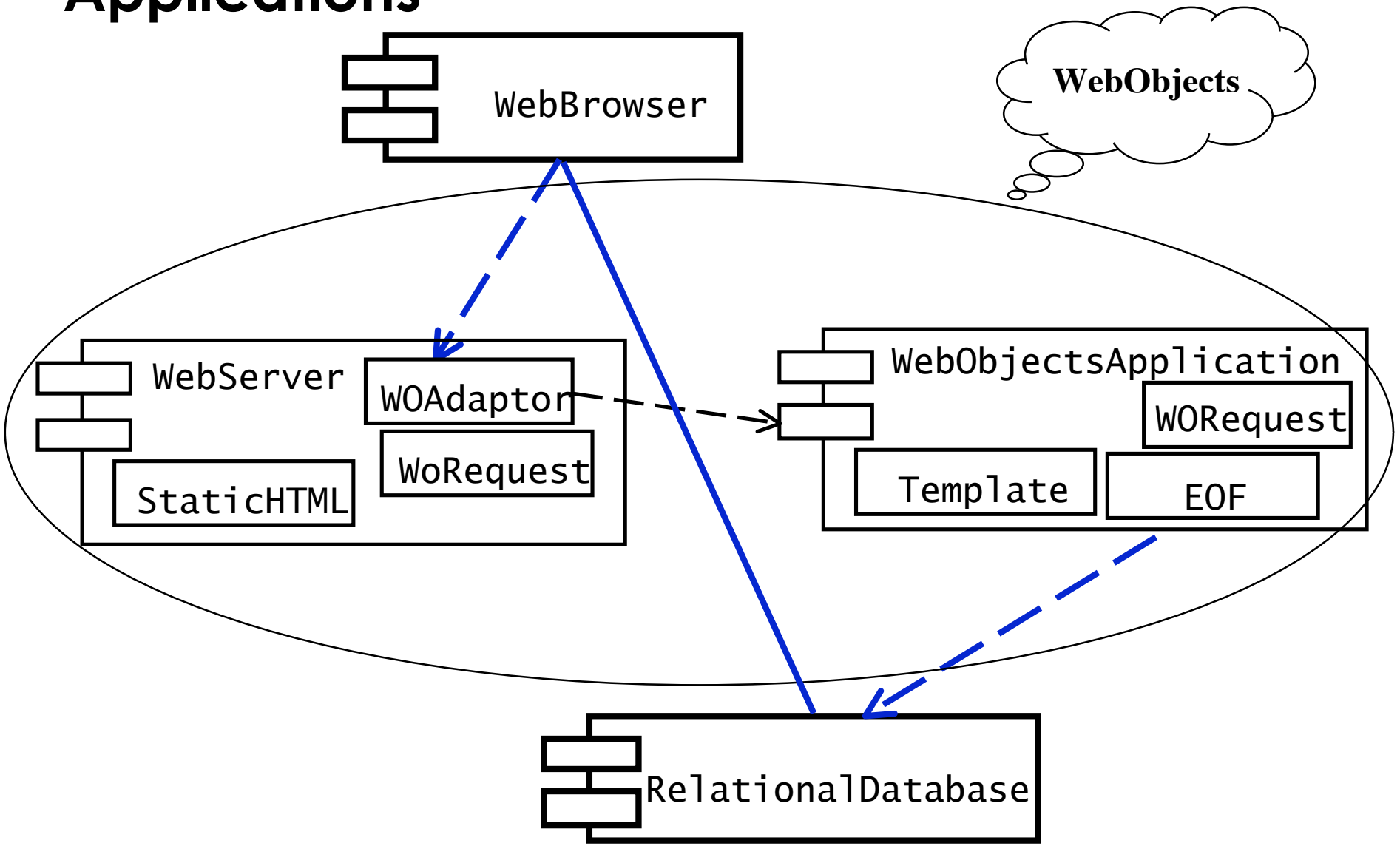
A BoomBox ist not a special form of Auto

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

Revised Metamodel for Inheritance



Example: Framework for Building Web Applications



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 and 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 and 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
  - … Javadoc documentation for all classes

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.

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