Object Design: Reuse

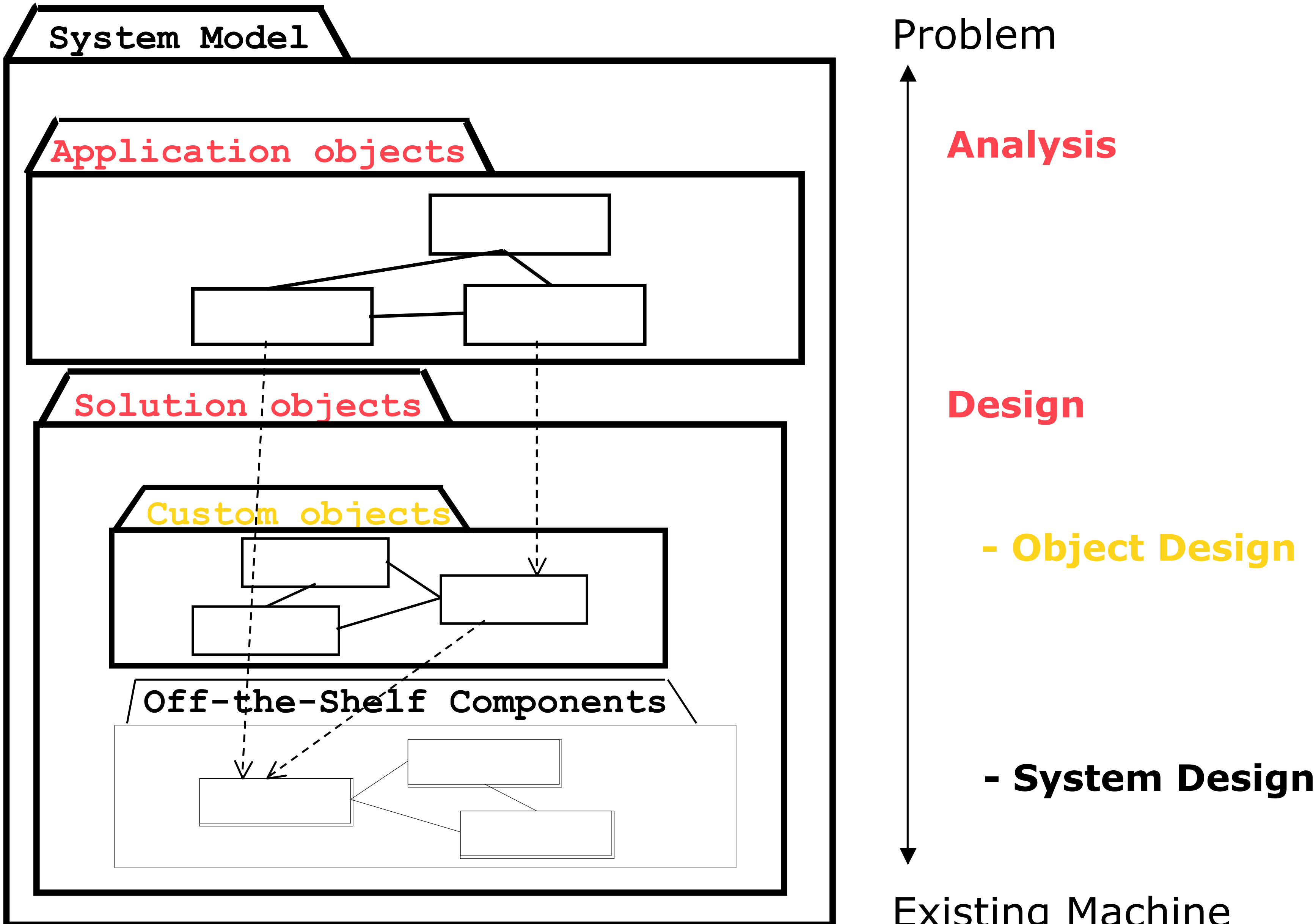
Software Engineering I
Lecture 11

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen

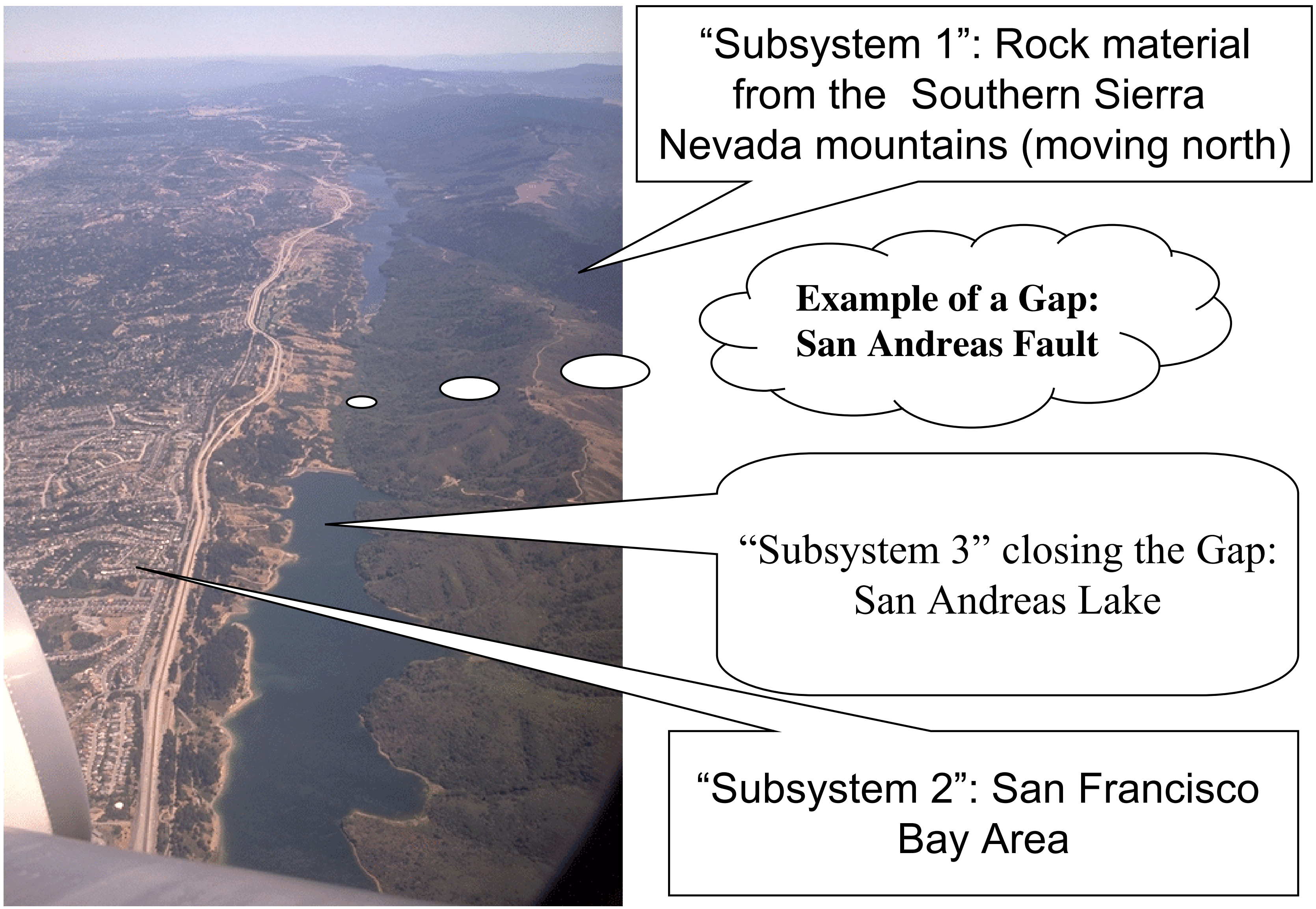
Object Design

- Purpose of object design:
  - Prepare for the implementation of the analysis model based on system design decisions
  - Transform analysis and system design models
- Investigate alternative ways to implement the analysis model
  - Use design goals: minimize execution time, memory and other measures of cost.
- Object Design serves as the basis of implementation

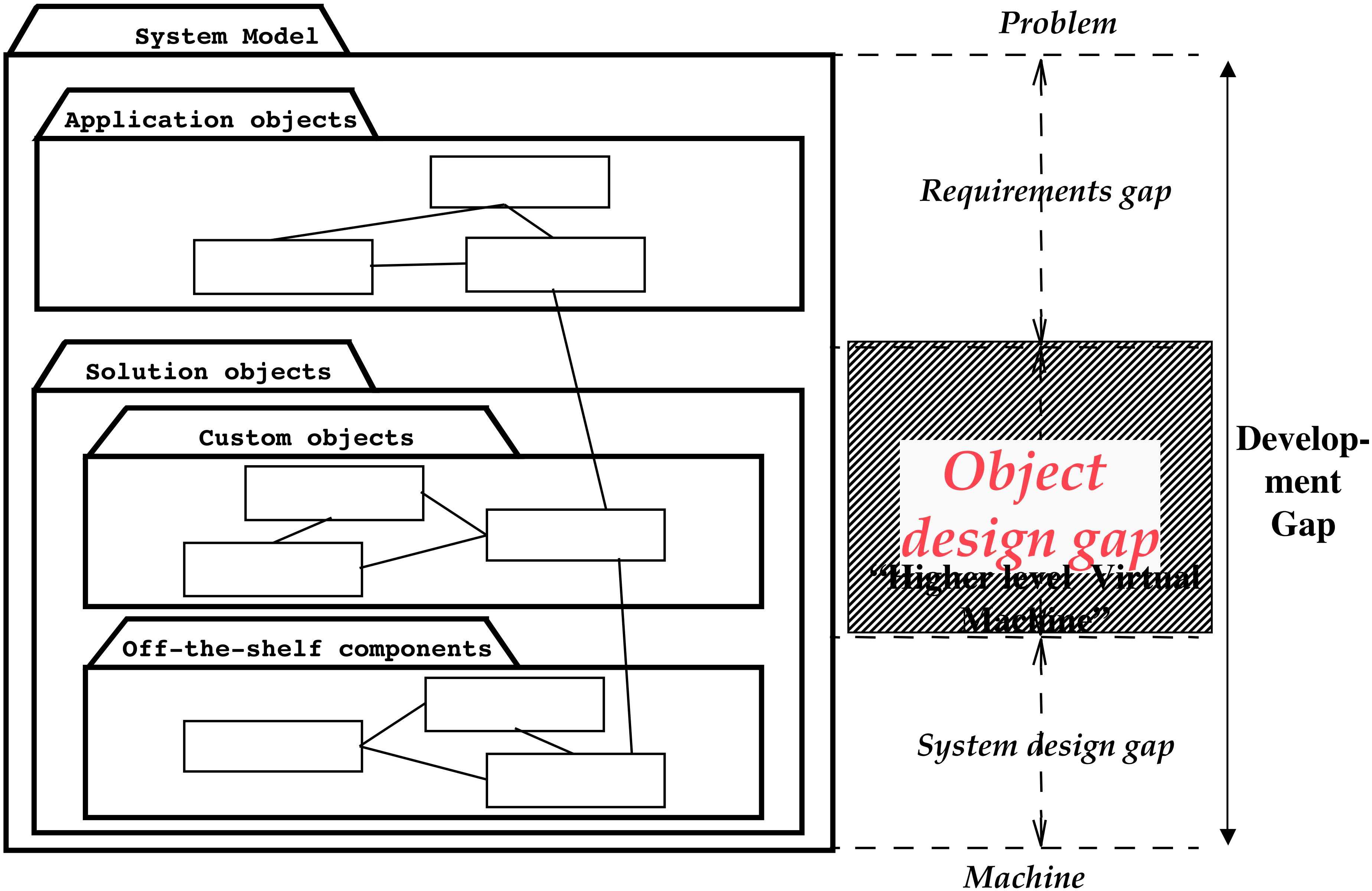
System Development as a Set of Activities



Design means "Closing the Gap"



Design means “Closing the Gap”



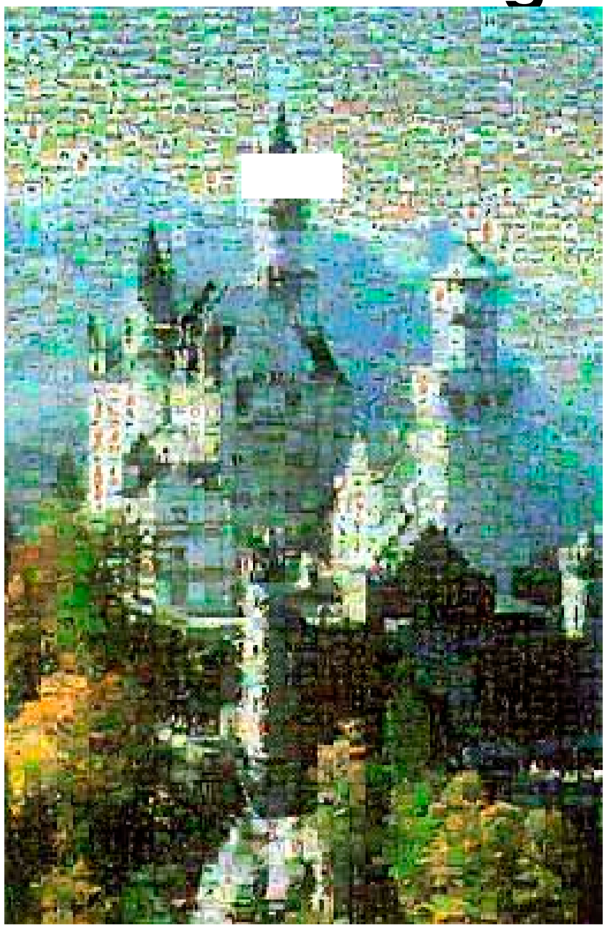
One Way to do System Design

- 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 like solving a Traditional Jigsaw Puzzle

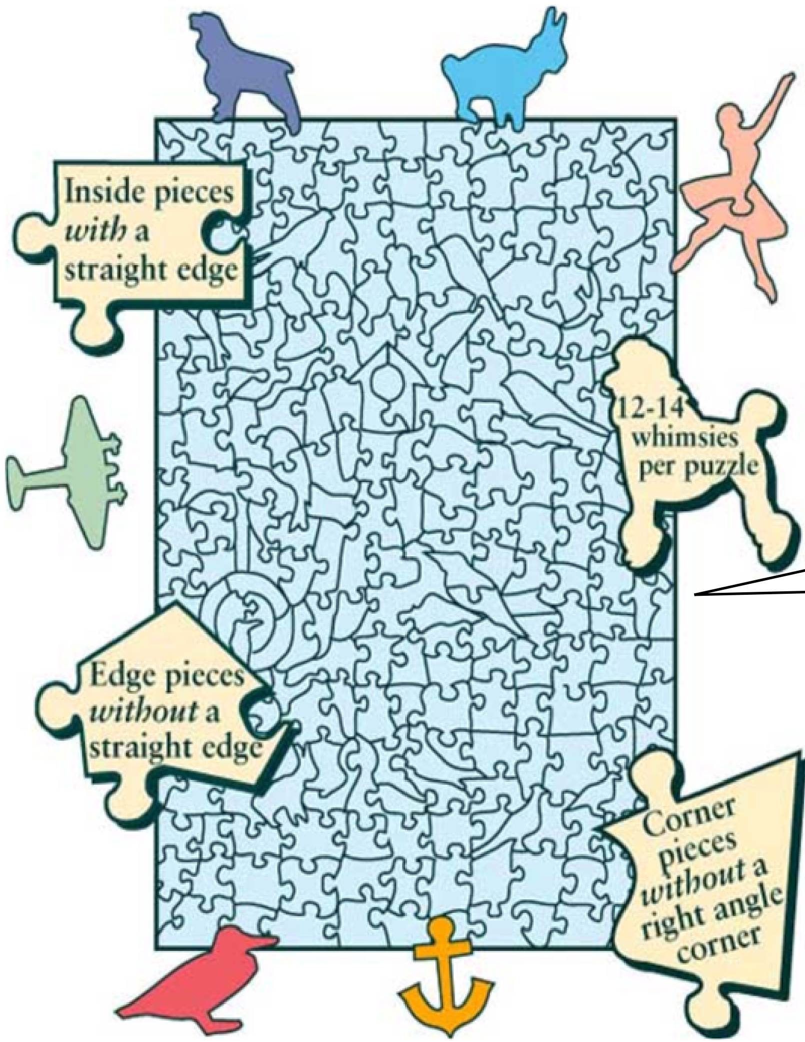


Remaining puzzle piece ("component")

Design Activities:

1. Identify the missing components
2. Make a build or buy decision to get the missing component.

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

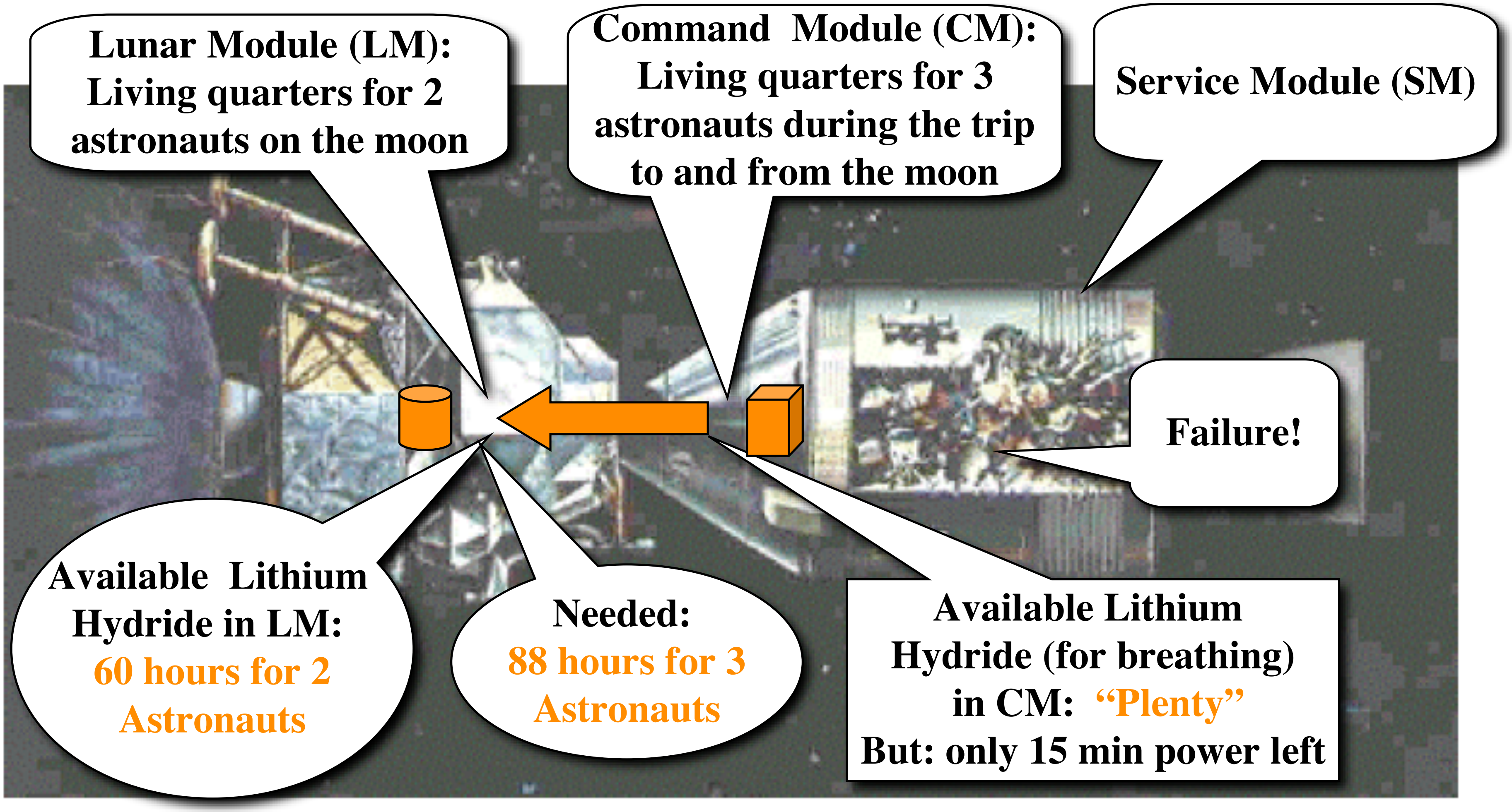


Advanced Jigsaw Puzzles

Adapter Pattern

- Adapter Pattern: Converts the interface of a component into another interface expected by the calling component
- Used to provide a new interface to existing legacy components (Interface engineering, reengineering)
- Also known as a wrapper
- Two adapter patterns:
  - Class adapter:
    - Uses multiple inheritance to adapt one interface to another
  - Object adapter:
    - Uses single inheritance and delegation.

Apollo 13: “Houston, we’ve had a Problem!”

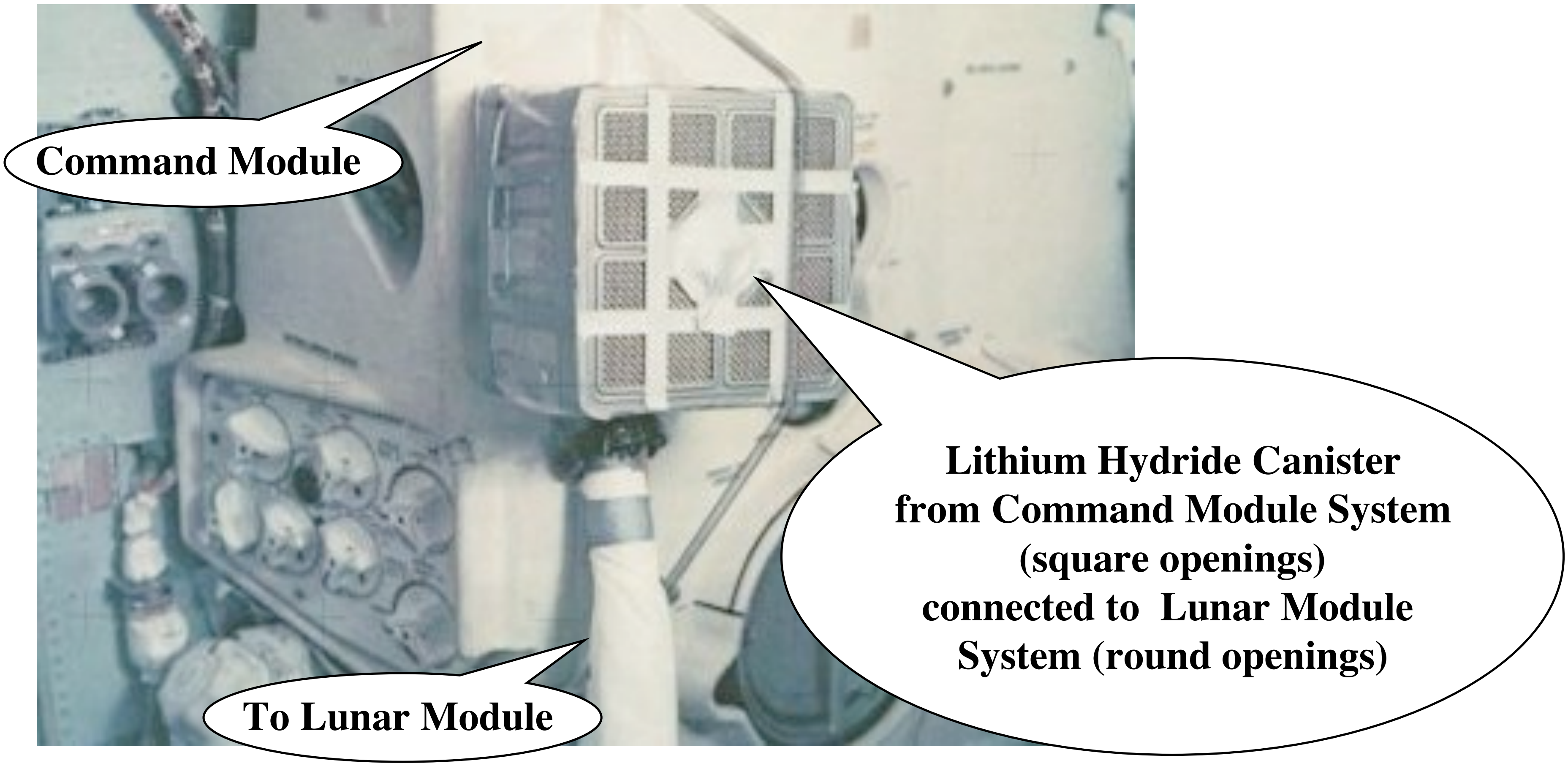


The LM was designed for 60 hours for 2 astronauts (2 days on the moon)
Could its resources be used for 12 man-days (2 1/2 days until reentry)?

Source: http://www1.jsc.nasa.gov/er/seh/apollo13.pdf

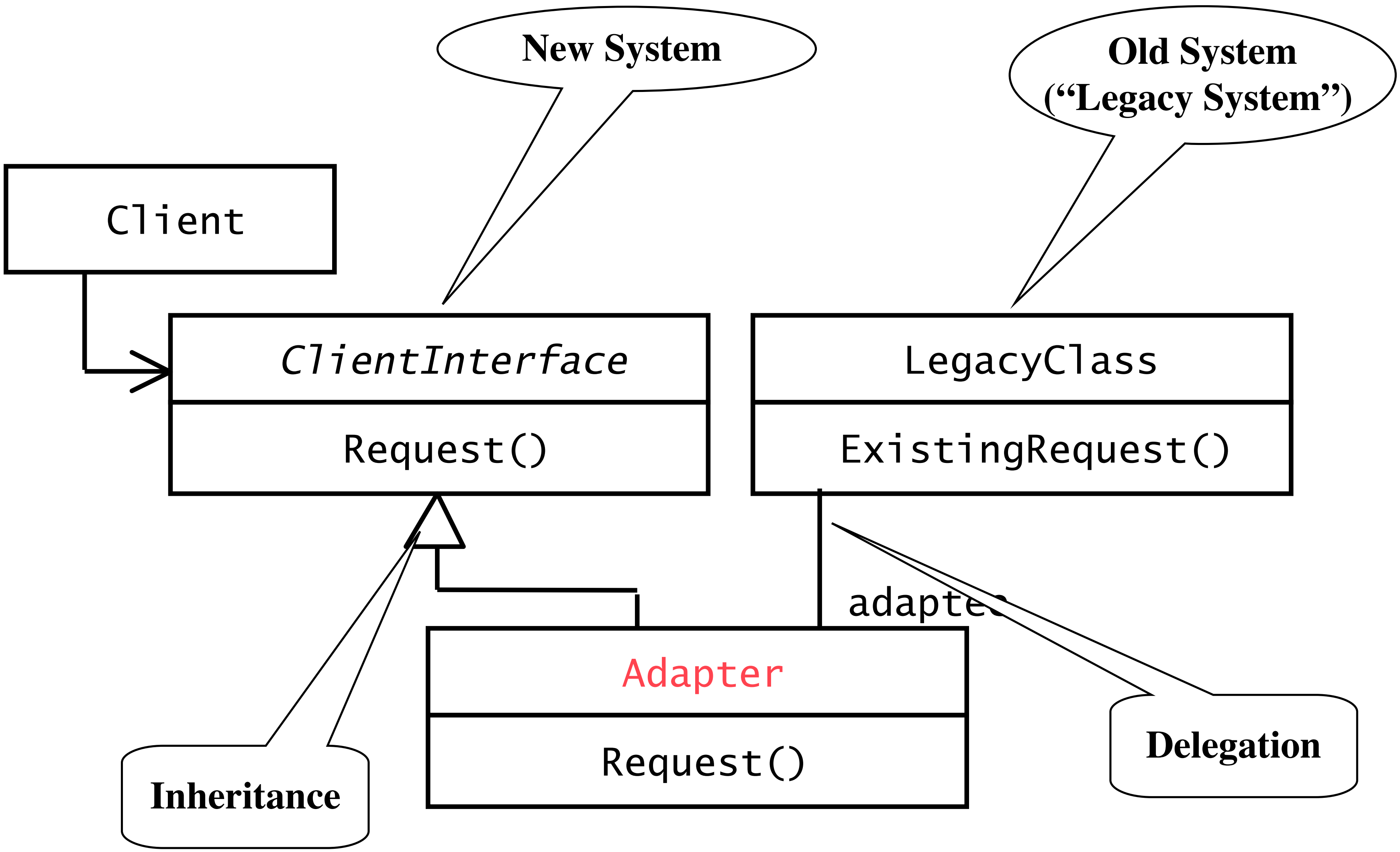
Apollo 13: "Fitting a square peg in a round hole"

A Typical Object Design Challenge: Connecting Incompatible Components

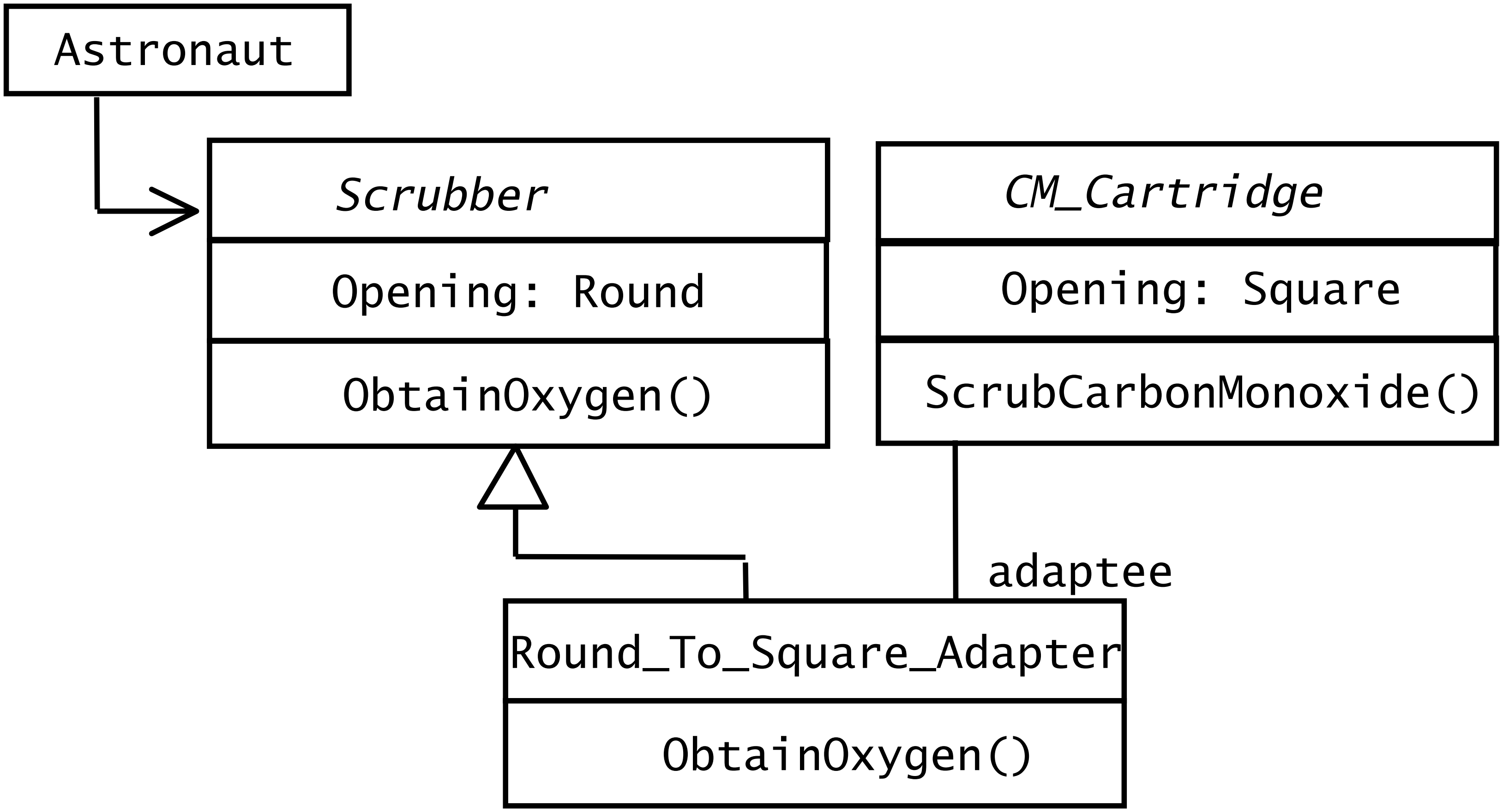


Source: http://www.hq.nasa.gov/office/pao/History/SP-350/ch-13-4.html

Adapter Pattern



Adapter for Scrubber in Lunar Module



- Using a carbon monoxide scrubber (round opening) in the lunar module with square cartridges from the command module (square opening)

Outline of Today

- Reuse examples
  - Reuse of code, interfaces and existing classes
- Whitebox and Blackbox Reuse
- Object design leads also to new classes
- The use of inheritance
- Implementation vs Specification Inheritance
- Delegation
- Components
- Class Libraries and Frameworks
- Study yourself:
  - Documenting the Object Design
  - JavaDoc

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 Windows
  - How can I make sure that I follow the conventions for Windows XP 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 vom Typ int
- How can I reuse this list without major effort to build a list of customers, or a spare parts catalog or a flight reservation schedule?
- Can I reuse a class “Addressbook”, which I have developed in another project, as a subsystem in my commercially obtained proprietary e-mail program?
  - Can I reuse this class also 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

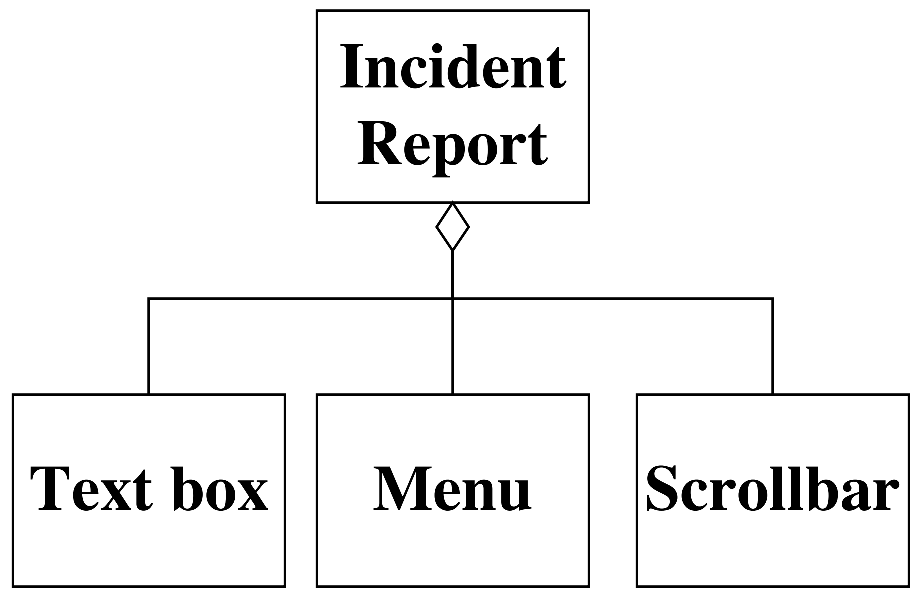
White Box and Black Box Reuse

- 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 avaliable, 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)



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

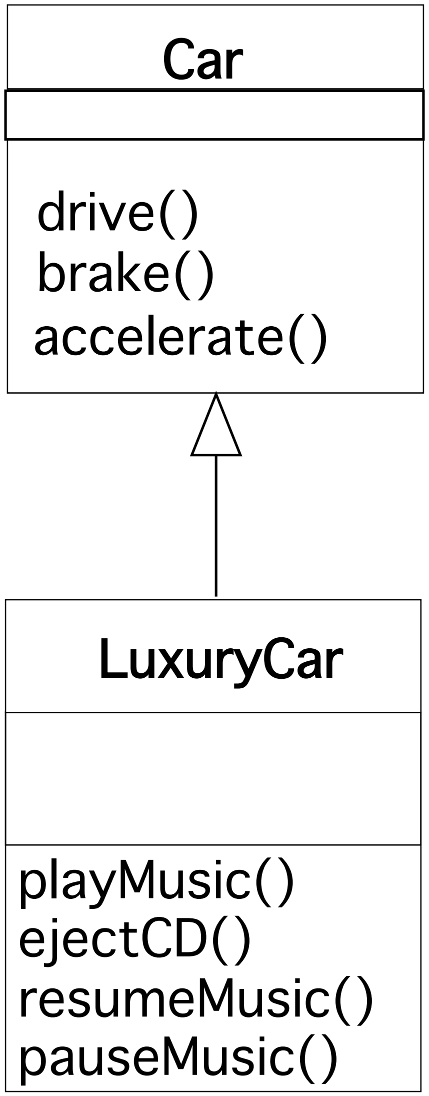
The use of Inheritance

- 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

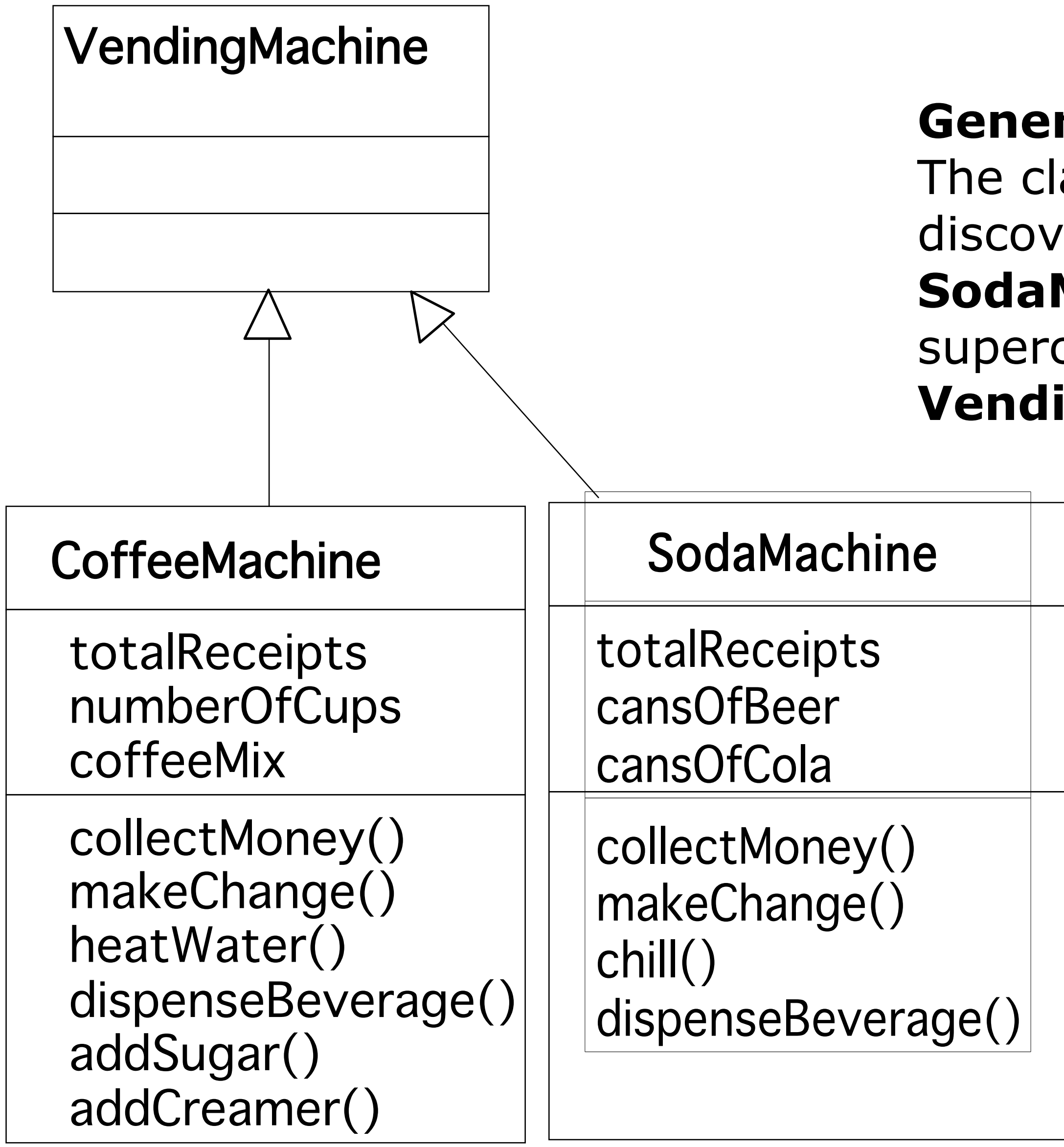
- To “discover“ inheritance associations, we can proceed in two ways, which we call specialization and generalization

- 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

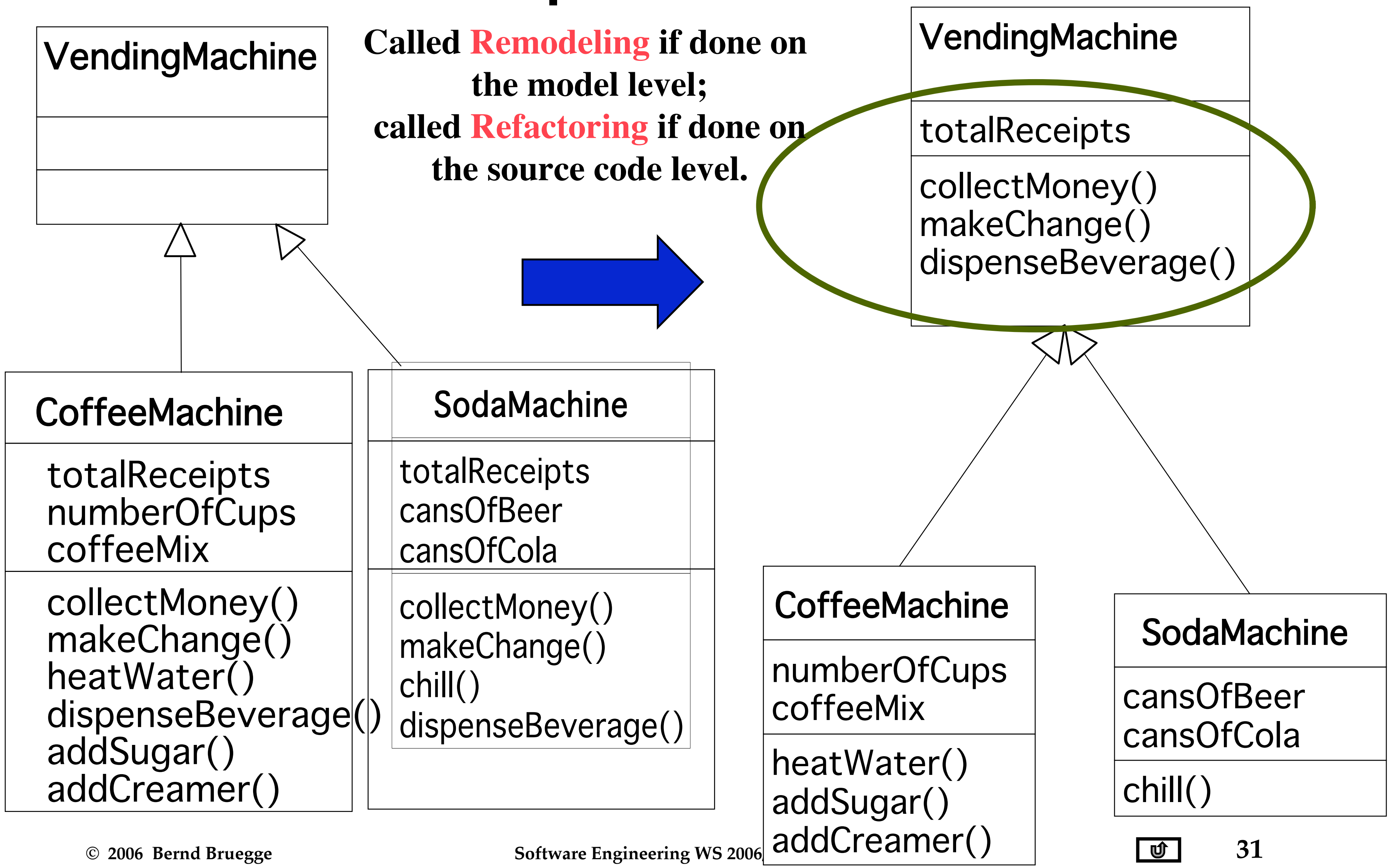
Generalization Example: Modeling a Coffee Machine



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

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



Details for the Mid-Term: Alternative 1

- Coverage: Lecture 1 - Lecture 11 (this lecture)

- Alternative 1: Closed book exam
  - Duration 9:00 to 10:00 am
    - 45 min (15 min extra if you appear at 9am)
  - Format: Paper-based, handwritten notes
  - Questions about definitions and/or modeling activities from material covered in lecture 1 to lecture 11.

  - Questions in English
  - Answers in English or German

Details for the Midterm (2): Alternative 2: Project exam

- If you cannot take the closed book exam, send a request to bruegge@in.tum.de (preferred: via a TUM e-mail) at the latest by Wed 10:30 am
    - Subject: SE 1 midterm request, <Your First Name and Family Name and MatrikelNr>
- You will then get access to a problem statement in PDF format by 12:00 o'clock
    - Tasks: Read the problem statement, describe the steps for the solution, using everything you learned so far.
    - Requirements elicitation, analysis, design and object design to demonstrate a solution to the problem
    - Send e-mail with PDF attachments to bruegge@in.tum.de by Thursday 12:00 noon (Timestamp of sender!)
        - Subject: SE 1 midterm solution, <Your First Name and Family Name>

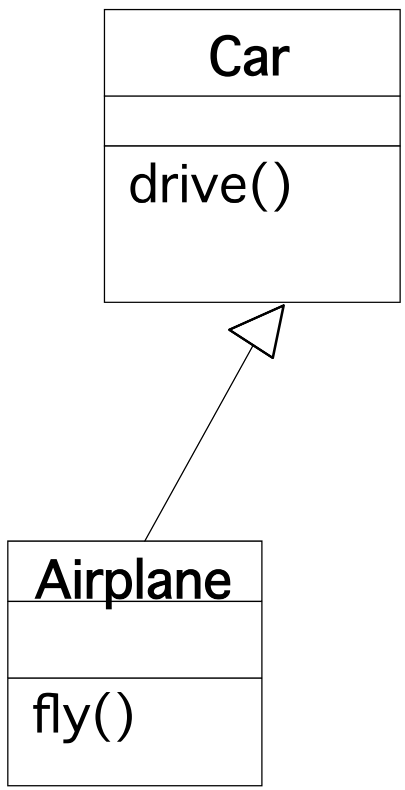
More Details for Alternative 2: Project exam

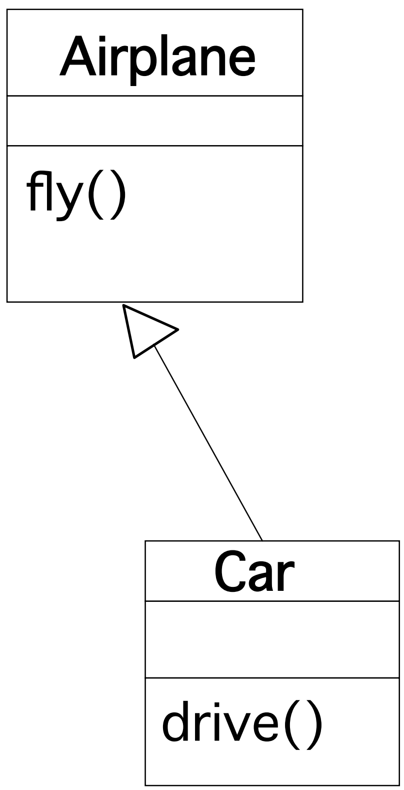
- You are expected to work alone
- No reuse of solutions from other students
- No cheating, submit your own solution!
- Use any kind of tools you have access to
  - Handwritten text, hand-drawings, scetches, UML CASE tools
- Format for submission:
  - One (1) file in PDF format
  - If you have more than one document, make sure to put all the documents together in one file.
  - If you need to compress: Use the Zip format.
- If you don't want to use e-mail:
  - Drop your solution (with your first and family name!) at Room 01.07.52 Secretary Monika Markl. Deadline: Thursday 12:00 noon.

Specialization 12 13 2006

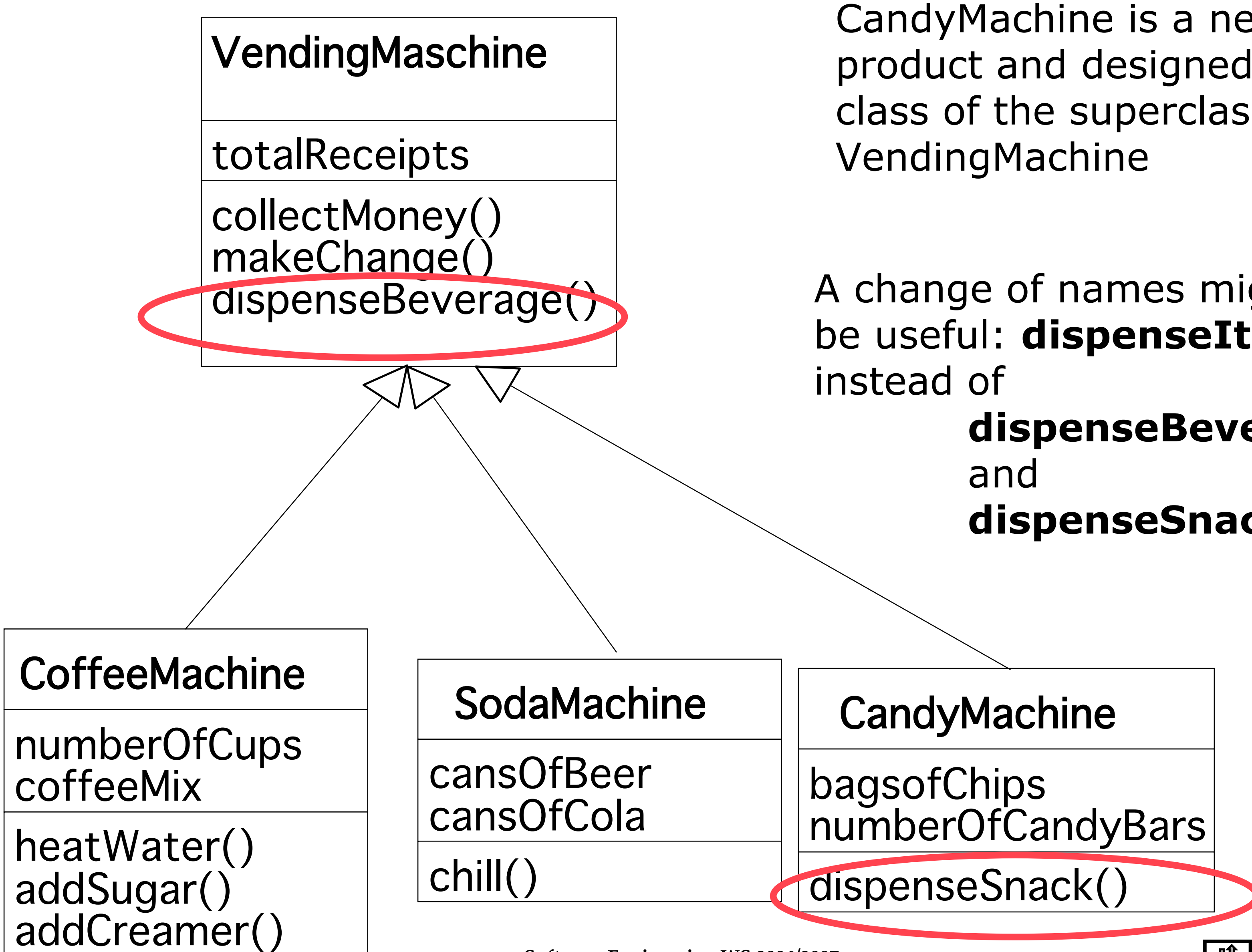
- Specialization occurs, when we find a subclass that is very similar to an existing class.
  - Example: A theory postulates certain particles and events which we have to find.
- Specialization can also occur unintentionally:

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





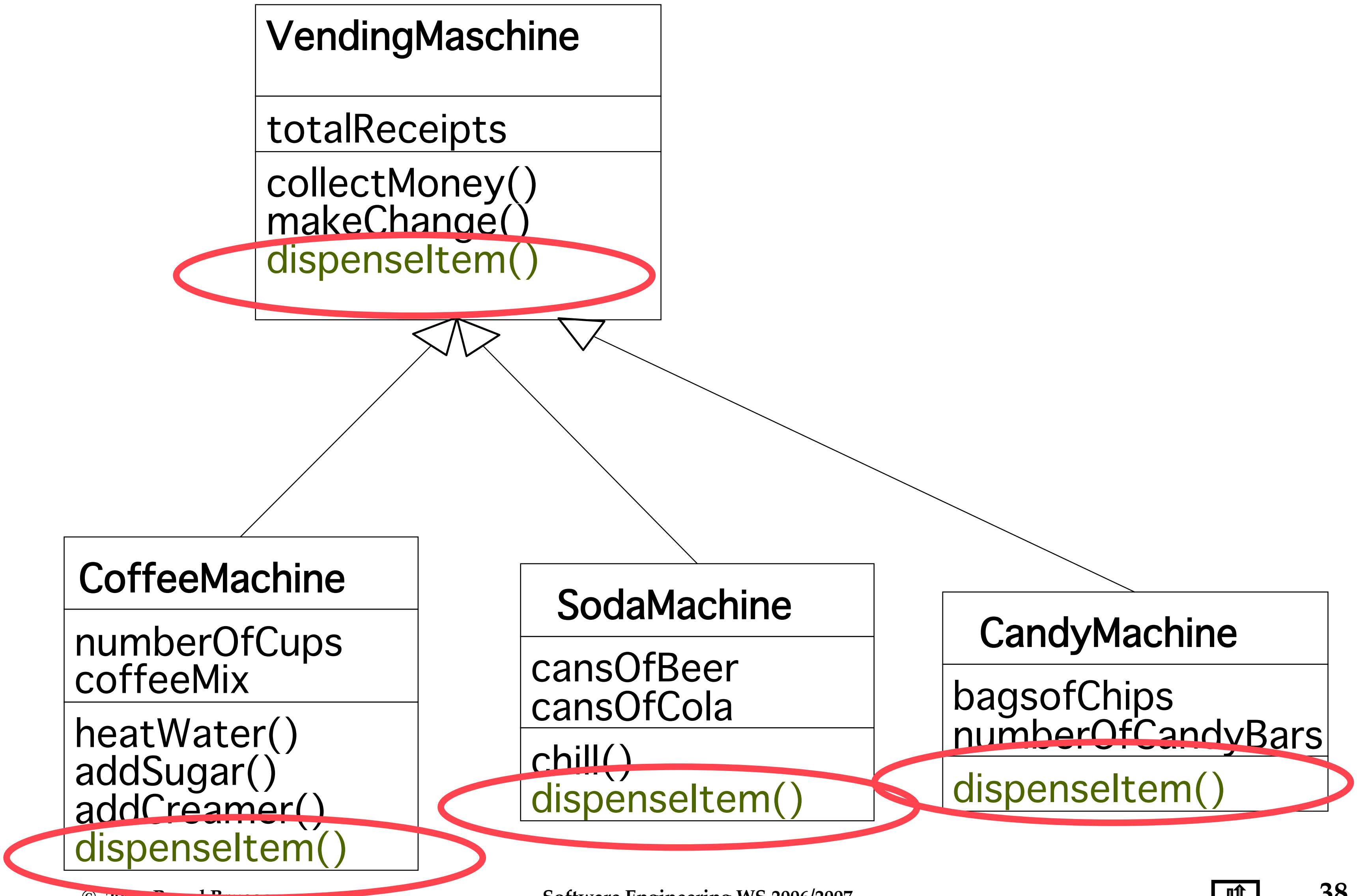
Another Example of a Specialization



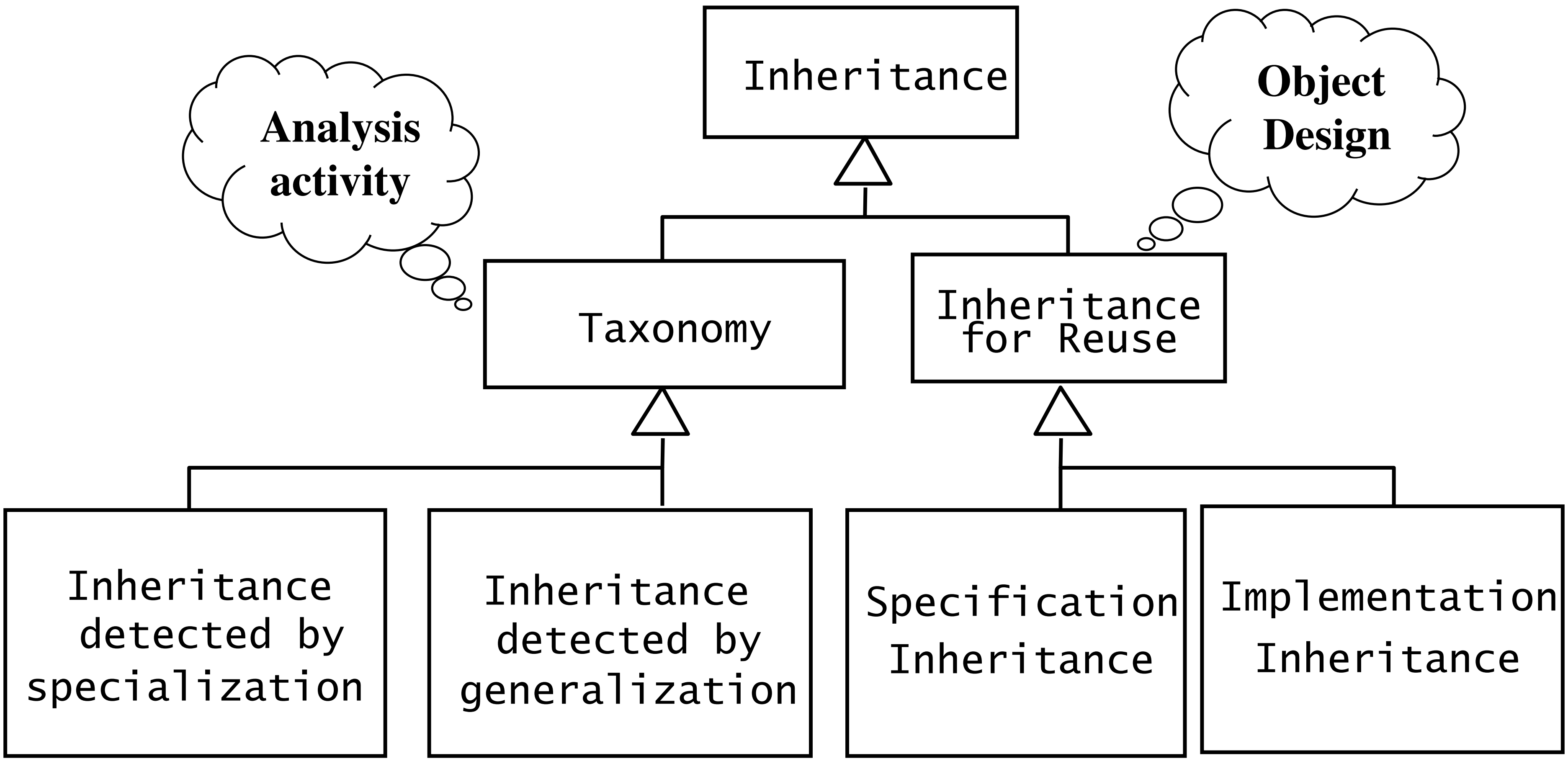
CandyMachine is a new product and designed as a sub class of the superclass VendingMachine

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

Example of a Specialization (2)



Meta-Model for Inheritance

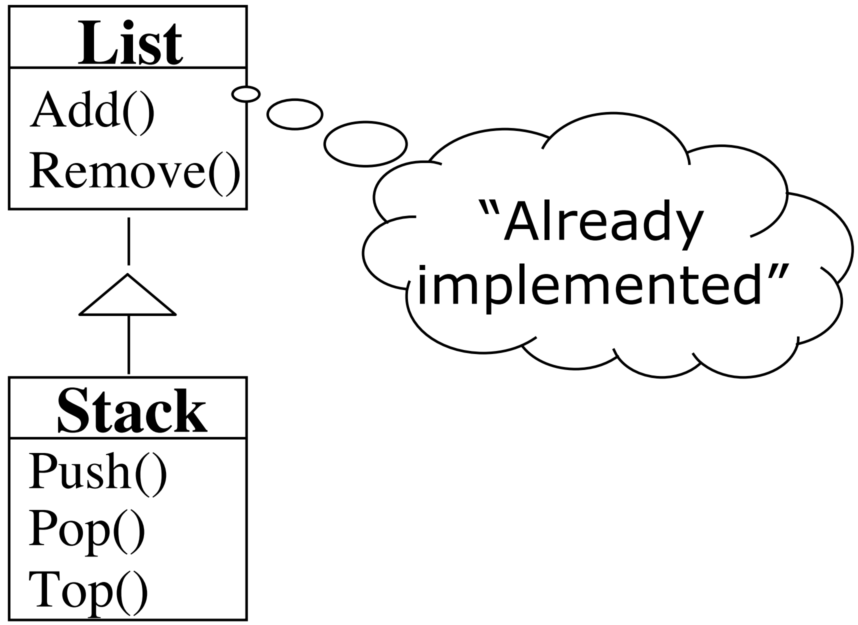


Implementation Inheritance and Specification Inheritance

- 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.

Example for Implementation Inheritance

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



Example:

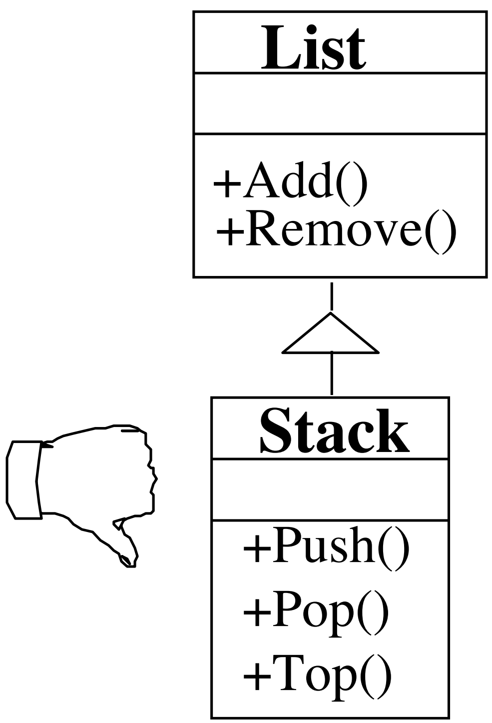
- I have a **List** class, I need a **Stack** class
  - How about subclassing the **Stack** class from the **List** class and implementing **Push()**, **Pop(), Top()** with **Add()** and **Remove()**?

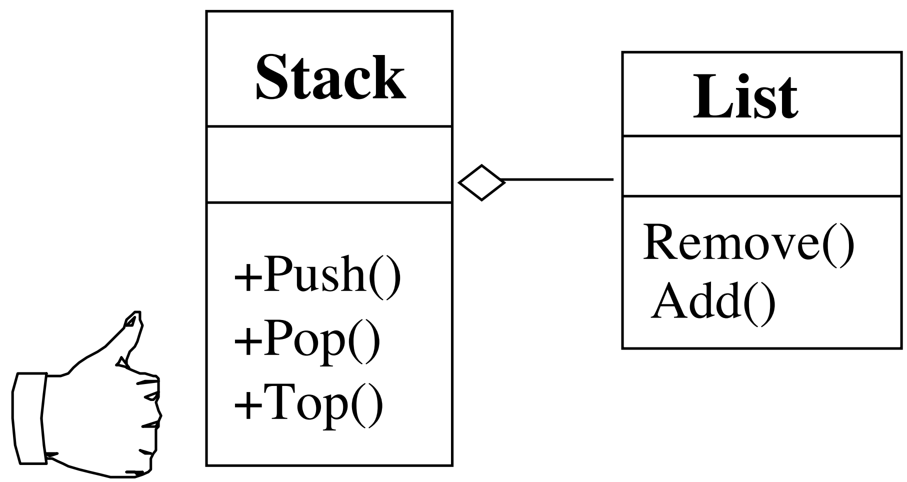
❖ Problem with implementation inheritance:

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

Better Code Reuse: Delegation

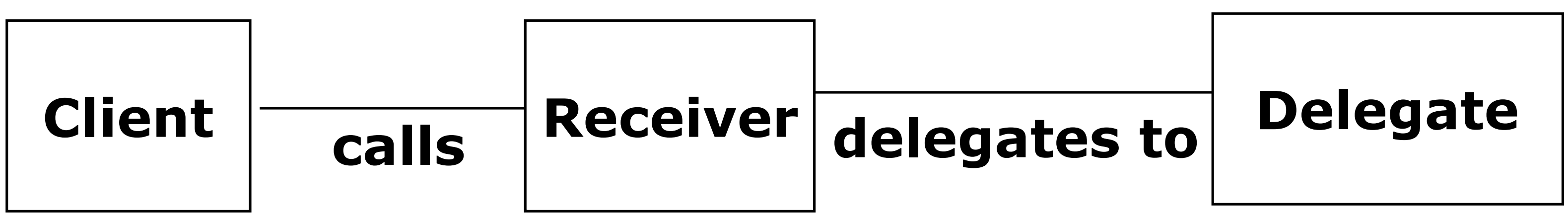
- Implementation-Inheritance: Using the implementation of super class operations
- Delegation: Catching an operation and sending it to another object that implements the operation





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.

Implementation Inheritance v. 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.

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.

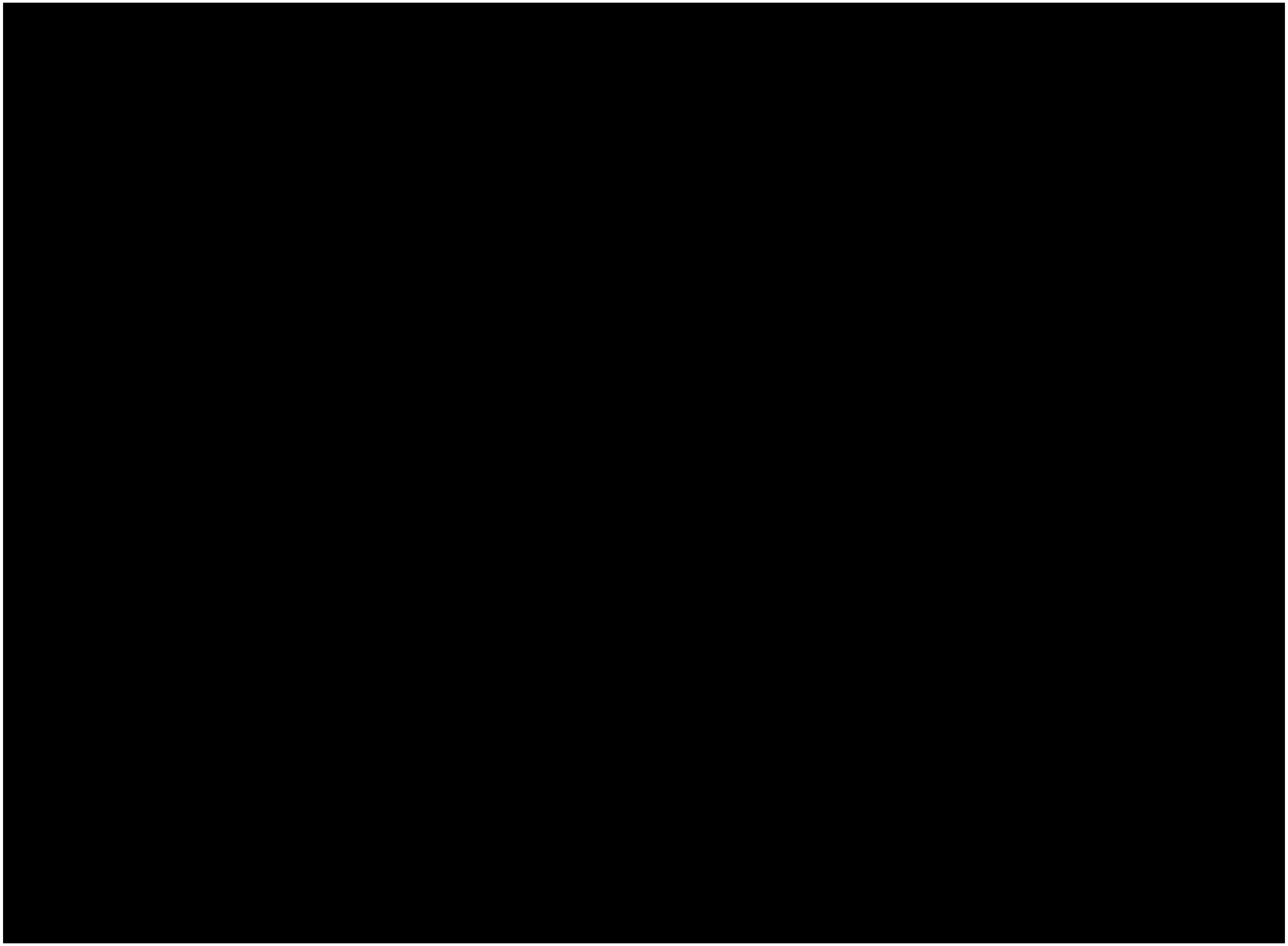
Object Design (activities 1–2)

Mapping Models to Code (activities 3–4)

Additional Readings

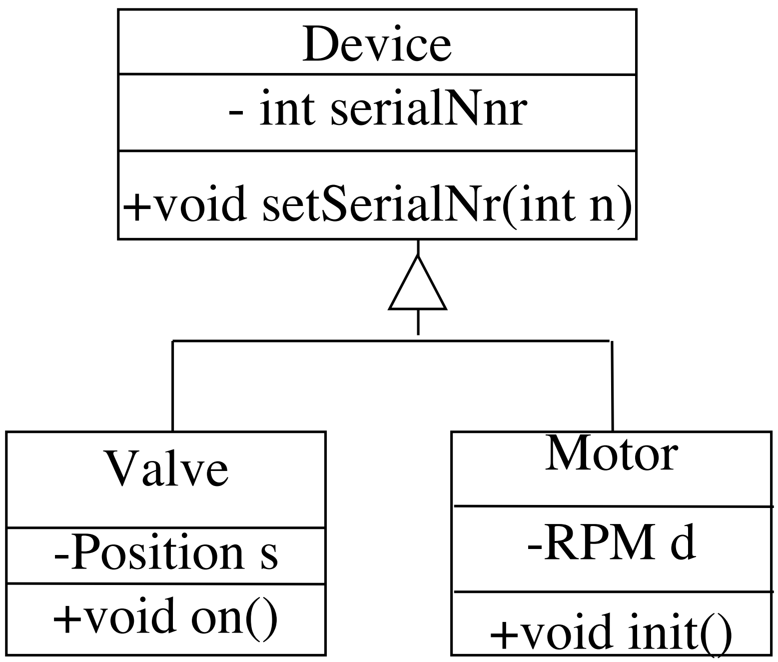
Summary

- 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



Another Example for Inheritance

Model:

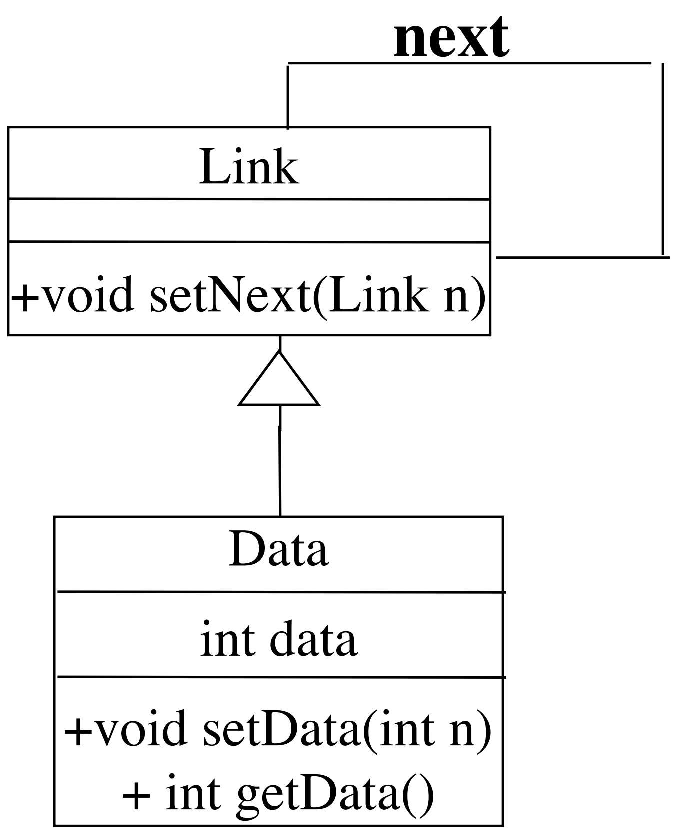


Java Code:

```
class Device {
    private int serialNr;
    public void setSerialNr(int n) {
        serialNr = n;
    }
}
    class Valve extends Device {
        private Position s;
        public void on() {
            ….;
        }
    }
    class Motor extends Device {
        private RPM d;
        public void init () {
            …;
        }
    }
```

Another Example (Customization)

Model:



Java Code:

```
class Link {
   Link next;
    public void setNext(Link n) {
        next = n;
    }
}
class Data extends Link {
   int data;
    public void setData(int d) {
       data = d;
    }
}
.....
Link l = new  Data();
l.setData(5000);
.....
```

`Data` extends `Link` with a new field `data` and two new methods `setData()` and `getData(`, which can be called on objects of Typ `Data`.

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 complements application domain knowledge and solution domain knowledge.

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

Application v. Solution Domain Objects

- Application domain objects represent concepts of the problem domain that are relevant to the system.
  - They are identified by the application domain specialists and by the end users.
- Solution domain objects represent concepts that do not have a counterpart in the application domain,
  - They are identified by the developers

Reuse Concepts

- Main goal:
  - Reuse knowledge from previous experience
  - Reuse of already available functionality
- Customization
- Application objects versus solution objects
- Specification inheritance and implementation inheritance
- Delegation
- The Liskov substitution principle
- Delegation and inheritance in design patterns
- Selecting design patterns and components

A Little Bit of Terminology: Activities

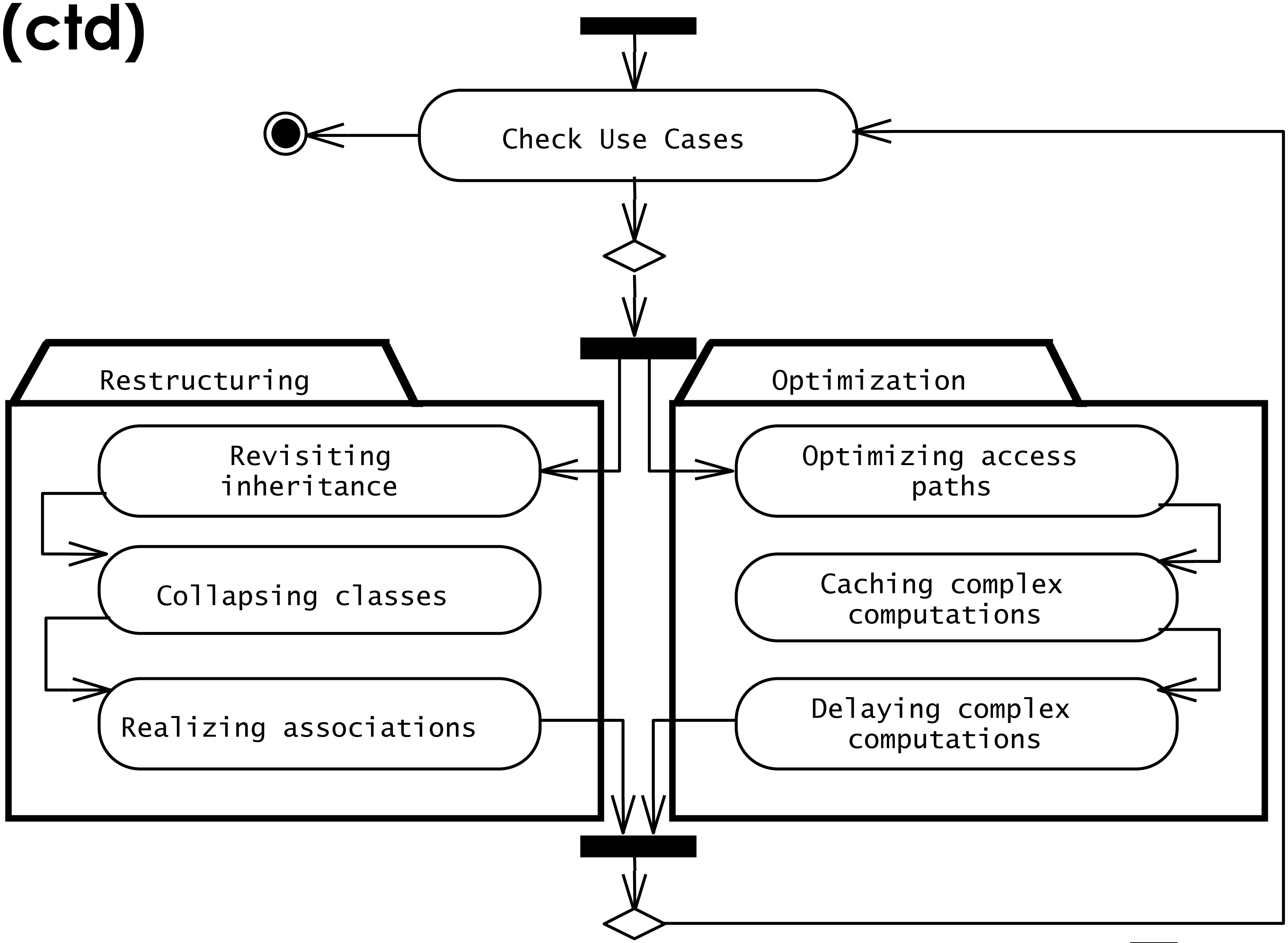
Object-oriented software engineering (OOSE):

- System Design
  - Decomposition into subsystems
- Object Design
  - Implementation language chosen
  - Data structures and algorithms chosen

Structured analysis/structured design (SA/SD):

- Preliminary Design
  - Decomposition into subsystems
  - Data structures are chosen
- Detailed Design
  - Algorithms are chosen
  - Data structures are refined
  - Implementation language is chosen
  - Typically in parallel with preliminary design, not a separate activity

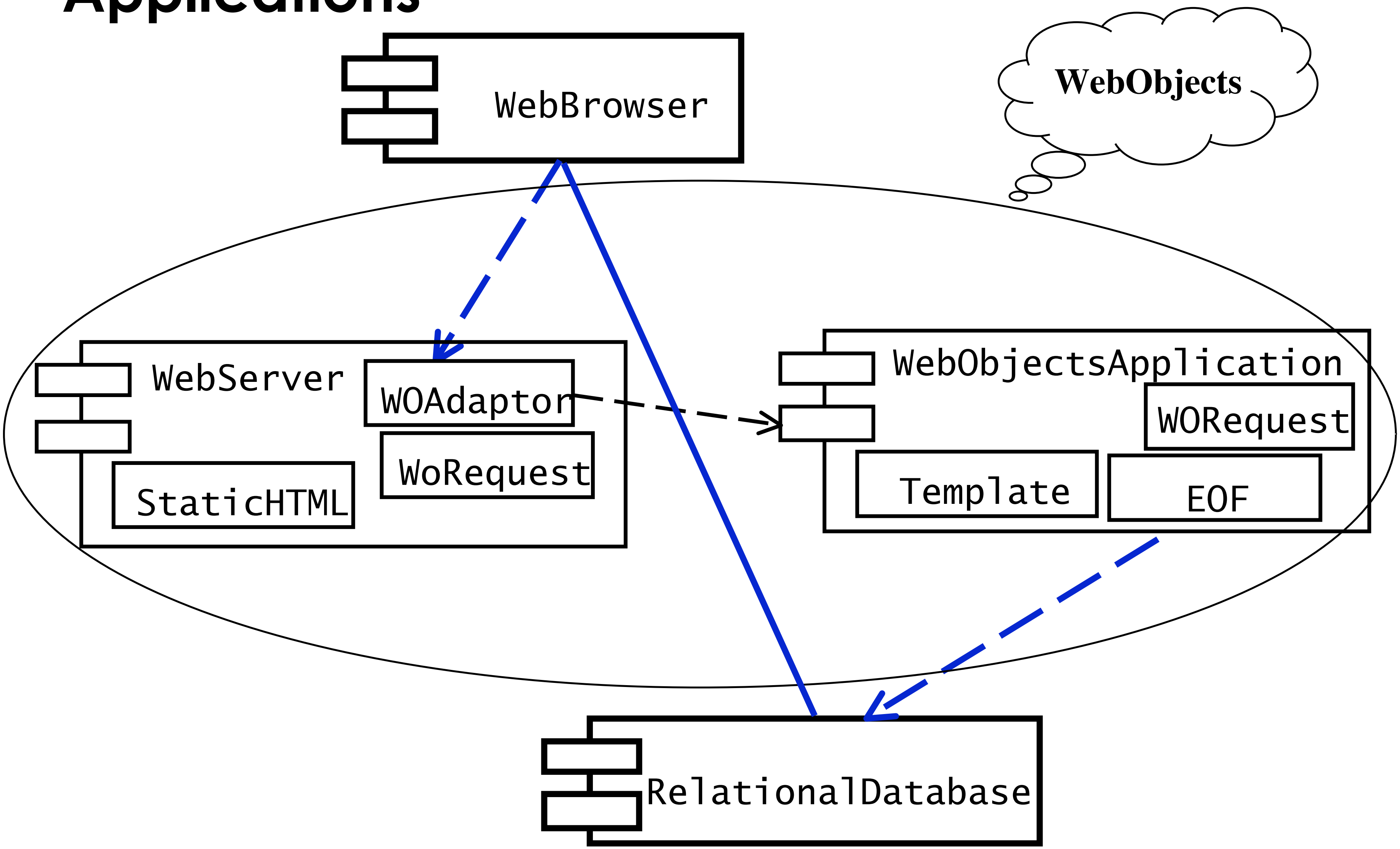
Detailed View of Object Design Activities (ctd)



Typical of Object Design Activities

- Identification of existing components
- Full definition of associations
- Full definition of classes
  - System Design => Service, Object Design => API
- Specifying contracts for each component
- Choosing algorithms and data structures
- Identifying possibilities of reuse
- Detection of solution-domain classes
- Optimization
- Increase of inheritance
- Decision on control
- Packaging

Example: Framework for Building Web Applications



JavaDoc

- Add documentation comments to the source code.
- A doc comment consists of characters between /** and */
- Doc comments may include HTML tags
- Example of a doc comment:

```
/**
* This is a <b> doc </b> comment
*/
```

More on JavaDoc

- Doc comments are only recognized when placed immediately before class, interface, constructor, method or field declarations.

- Class and Interface Doc Tags
- Constructor and Method Doc Tags

Class and Interface Doc Tags

@author name-text

- Creates an “Author” entry.

@version version-text

- Creates a “Version” entry.

@see classname

- Creates a hyperlink “See Also classname”

@since since-text

- Adds a “Since” entry. Usually used to specify that a feature or change since a certain release number

- @deprecated deprecated-text
  - Adds a comment that this method can no longer be used. Convention is to describe the replacing method
    - Example: @deprecated Replaced by setBounds(int, int, int, int).

Constructor and Method Doc Tags

Can contain @see tag, @since tag, @deprecated as well as:

@param parameter-name description

Adds a parameter to the "Parameters" section.

@return description

A description of the return value.

@exception fully-qualified-class-name description

Name of the exception that may be thrown by the method.

@see classname

Adds a hyperlink "See Also" entry to the method.

Example of a Class Doc Comment

```
/**
        * A class representing a window on the screen.
        * For example:
        * <pre>
        *    Window win = new Window(parent);
        *    win.show();
        * </pre>
        *
        * @author  Sami Shaio
        * @version %I%, %G%
        * @see     java.awt.BaseWindow
        * @see     java.awt.Button
 */
class Window extends BaseWindow {
        ...
}
```

Example of a Method Doc Comment

```
/**
 * Returns the character at the specified index. Index ranges
 * from <code>0</code> to <code>length() - 1</code>.
 *
 * @param     index  the index of the desired character.
 * @return    the desired character.
 * @exception StringIndexOutOfRangeException
 *            if the index is not in the range <code>0</code>
 *            to <code>length()-1</code>.
 * @see       java.lang.Character#charValue()
 */
public char charAt(int index) {
        ...
}
```

Example of a Field Doc Comment

A field comment can contain only the @see, @since and @deprecated tags

```
/**
 * The X-coordinate of the window.
 *
 * @see window#1
 */
int x = 1263732;
```

Example: Specifying a Service in Java

```
/** Office is a physical structure in a building. It is
  possible to create an instance of an office; add
  an occupant; get the name of occupants */
public class Office {
        /** Adds an occupant to the office
        @param  NAME  name is a nonempty string
        */
        public void AddOccupant(string name);
        /** @Return Returns the name of the office.
          Requires, that Office has been initialized with a
          name
        */
        public string GetName();
    ....
}
```

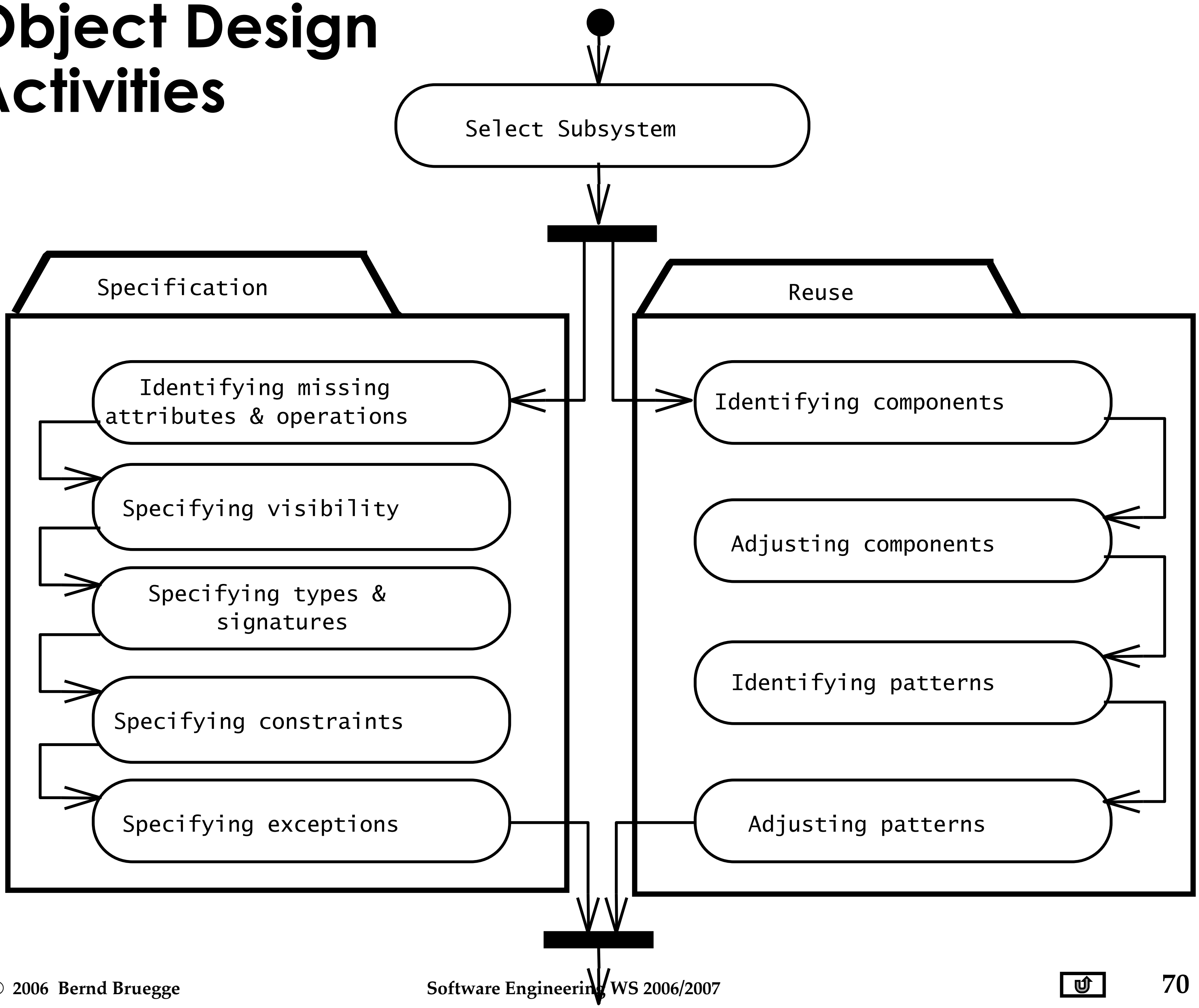
Package it all up

- Construct physical modules
  - Ideally use one package for each subsystem
- Two design principles for packaging
  - Minimize coupling:
    - Classes in client-supplier relationships are usually loosely coupled
    - Large number of parameters in some methods mean strong coupling (> 4-5)
  - Maximize cohesion:
    - Classes closely connected by associations => same package

Packaging Heuristics

- Each subsystem service is made available by one or more interface objects within the package
- Start with one interface object for each subsystem service
  - Try to limit the number of interface operations (7+-2)
- If the service has too many operations, reconsider the number of interface objects
- If you have too many interface objects, reconsider the number of subsystems

Object Design Activities



Customization Projects are like Advanced Jigsaw Puzzles

http://www.puzzlehouse.com/_