Object-Oriented Software Engineering

Using UML, Patterns, and Java

Chapter 8, Object Design: Reuse and Patterns

Object Design

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

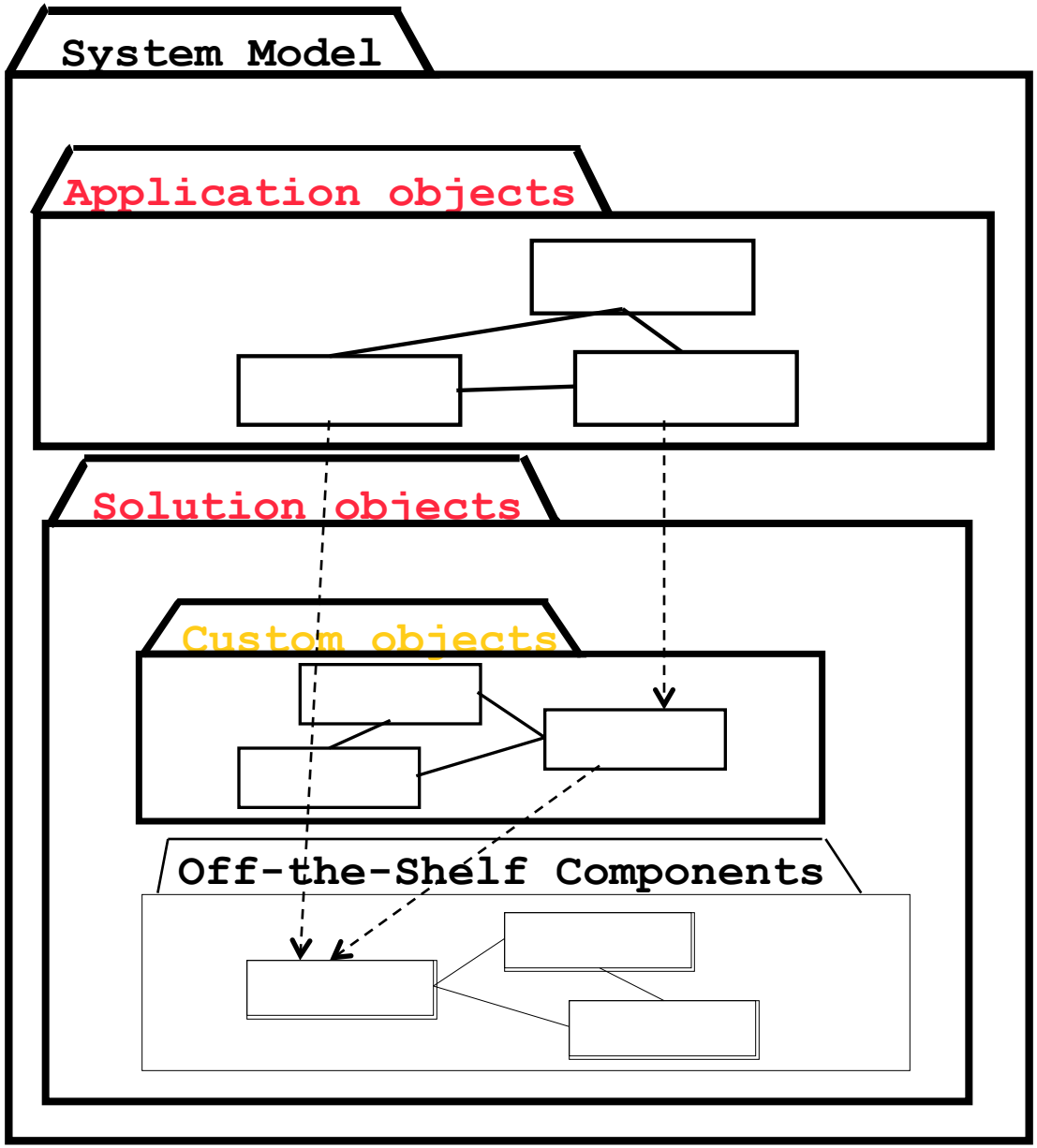
Terminology: Naming of Design Activities

Methodology: Object-oriented software engineering (OOSE)

- *System Design*
  - Decomposition into subsystems, etc

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

System Development as a Set of Activities



Problem

Analysis

Design

- Object Design

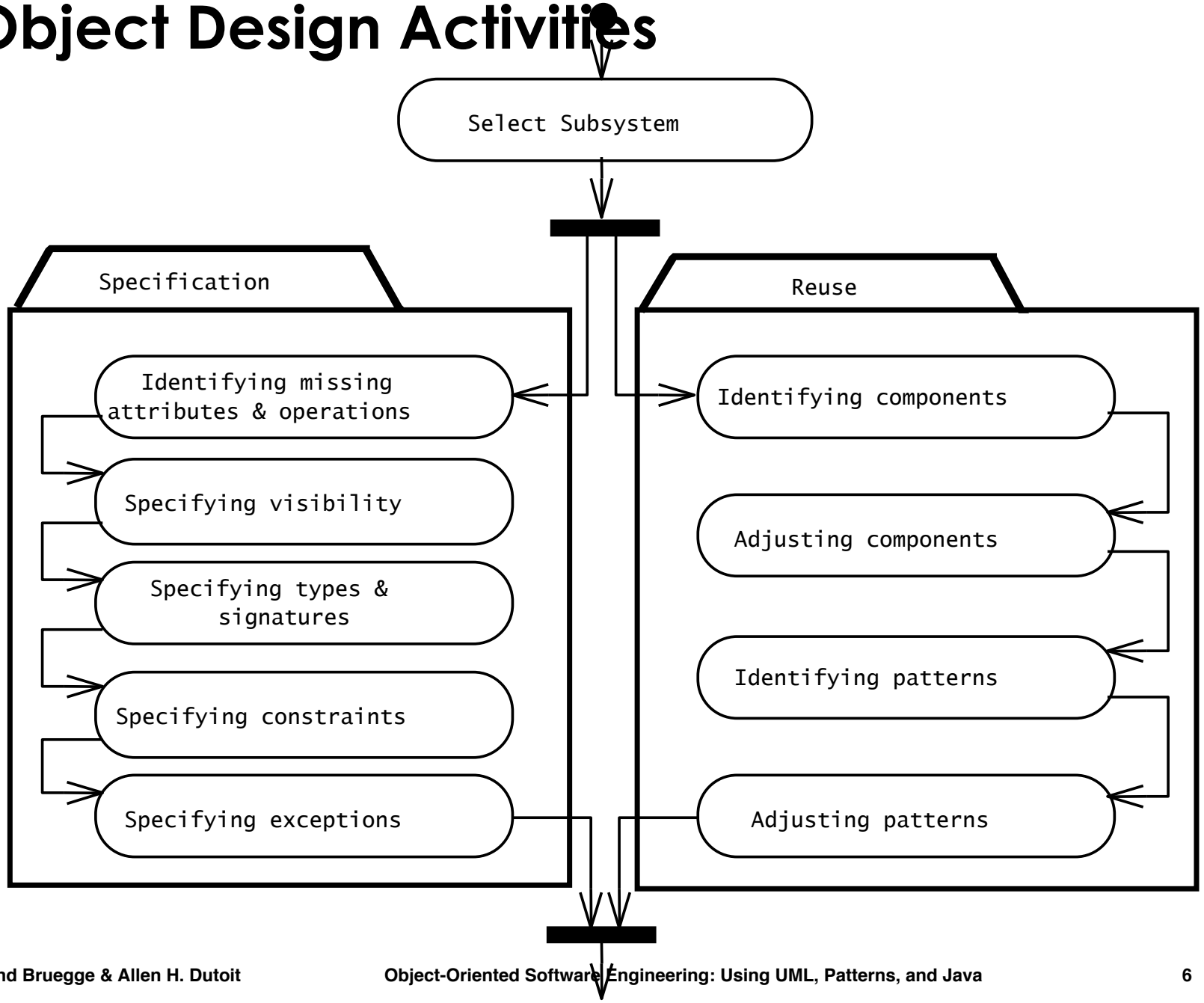
- System Design

Existing Machine

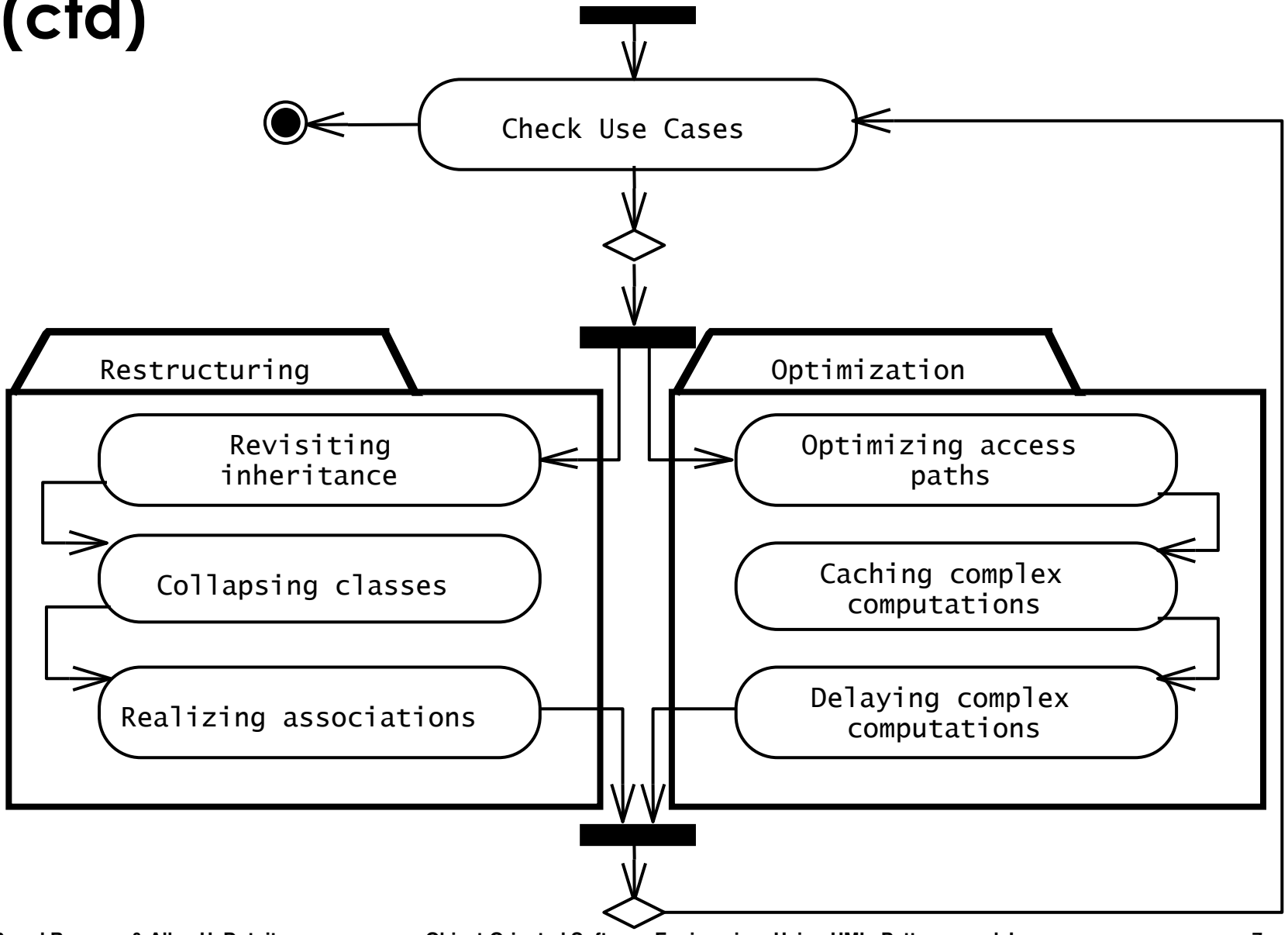
Object Design consists of 4 Activities

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.

Object Design Activities



Detailed View of Object Design Activities (ctd)



One Way to do Object Design

1. Identify the missing components in the design gap
2. Make a build or buy decision to obtain the missing component

=> Component-Based Software Engineering:

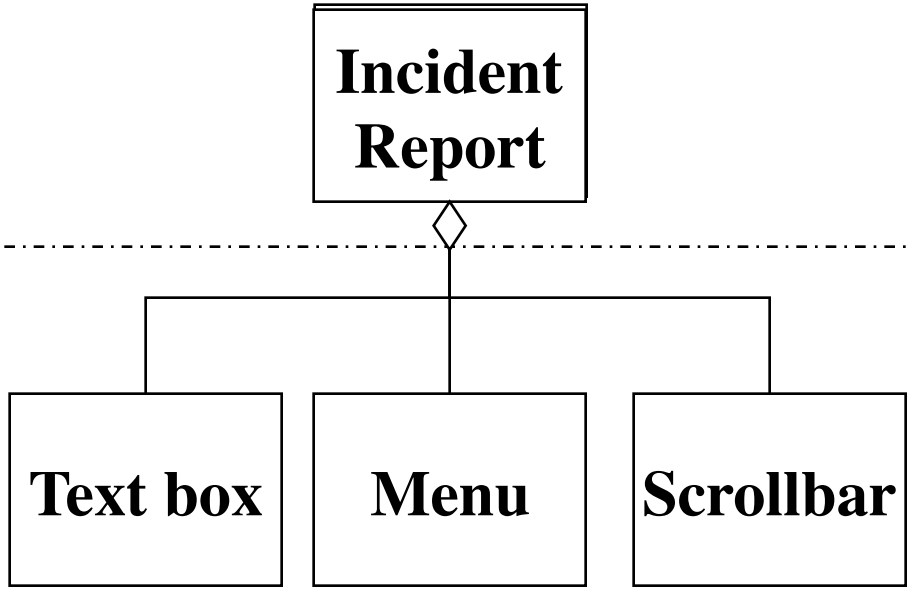
The design gap is filled with available components ("0 % coding").

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

=> Design with standard components.

Identification of new Objects during Object Design

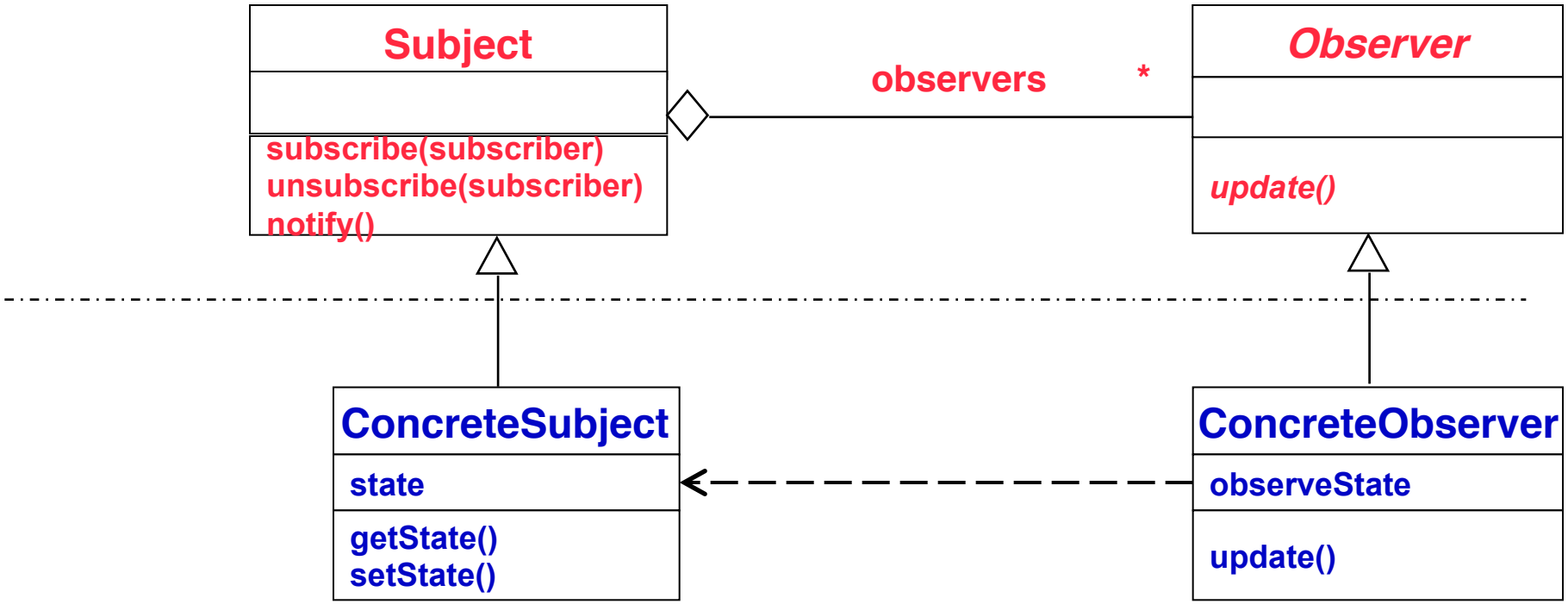
Requirements Analysis (Language of Application Domain)



Object Design (Language of Solution Domain)

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.

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 existing classes

- I have an implementation for a list of elements of type 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

Inheritance comes in many flavors

Inheritance is used in four ways:

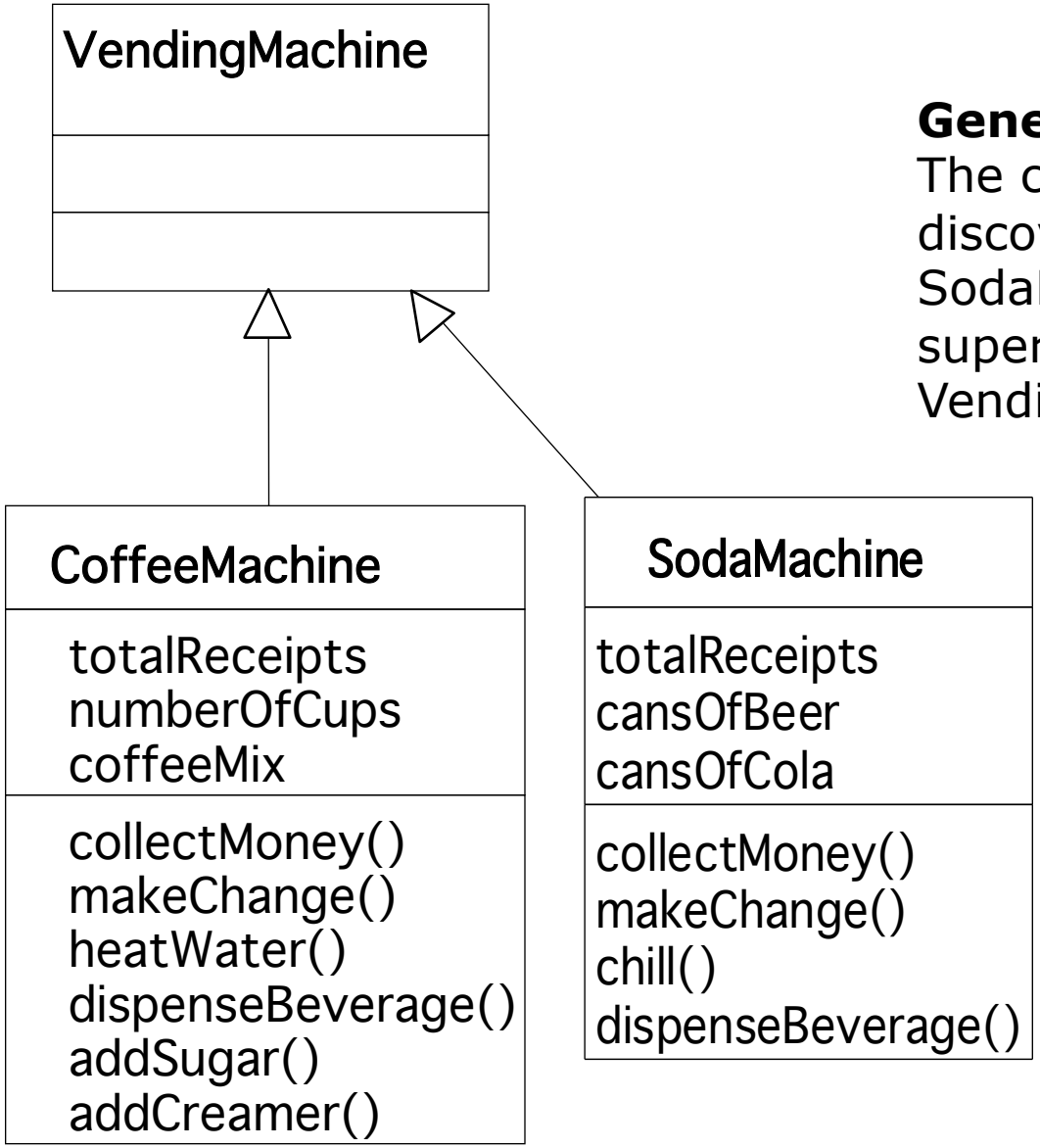
- Specialization
- Generalization
- Specification Inheritance
- Implementation Inheritance

Discovering Inheritance

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

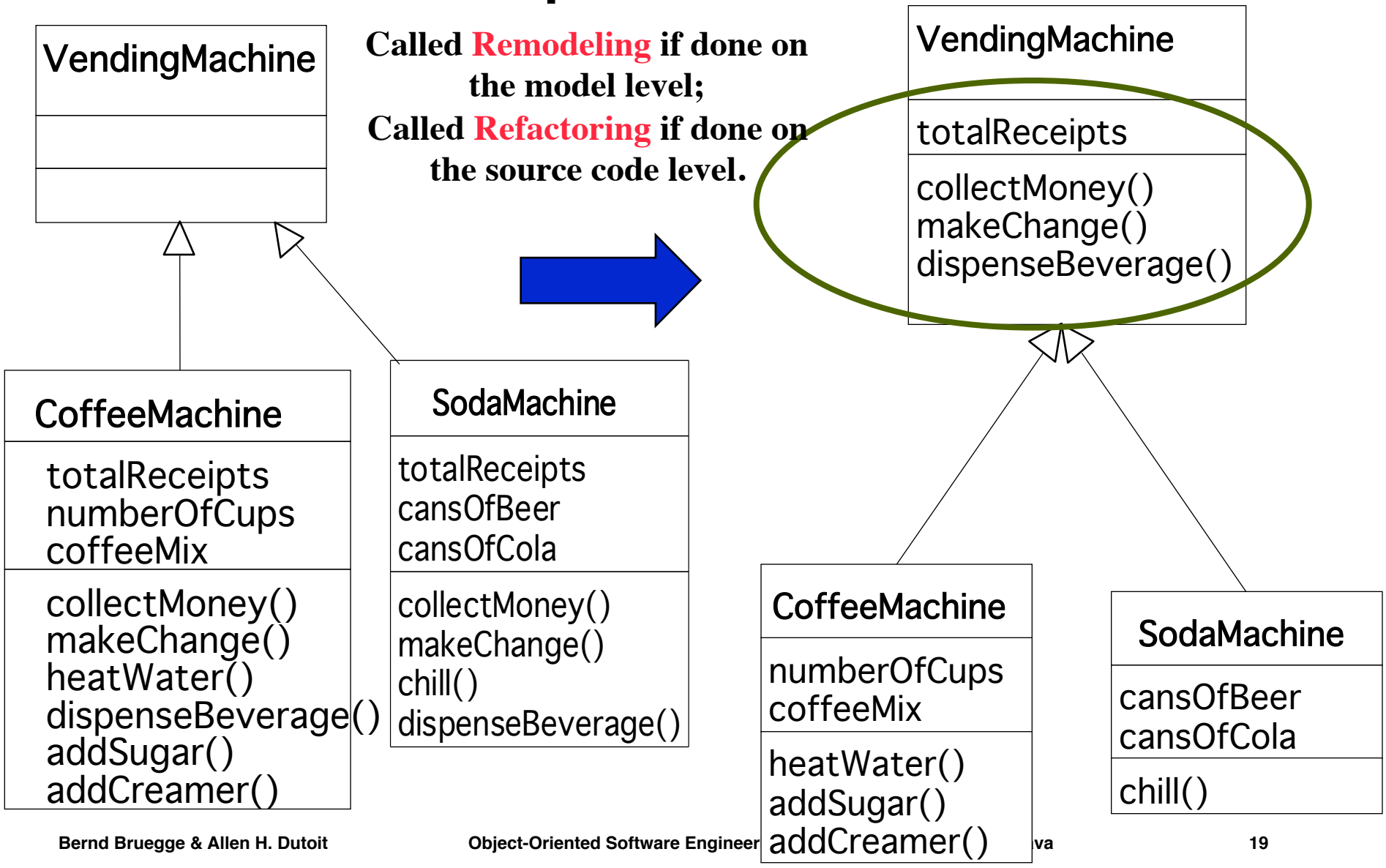
- Generalization: the discovery of an inheritance relationship between two classes, where the sub class is discovered first.
- Specialization: the discovery of an inheritance relationship between two classes, where the super class is discovered first.

Generalization Example: Modeling a Coffee Machine

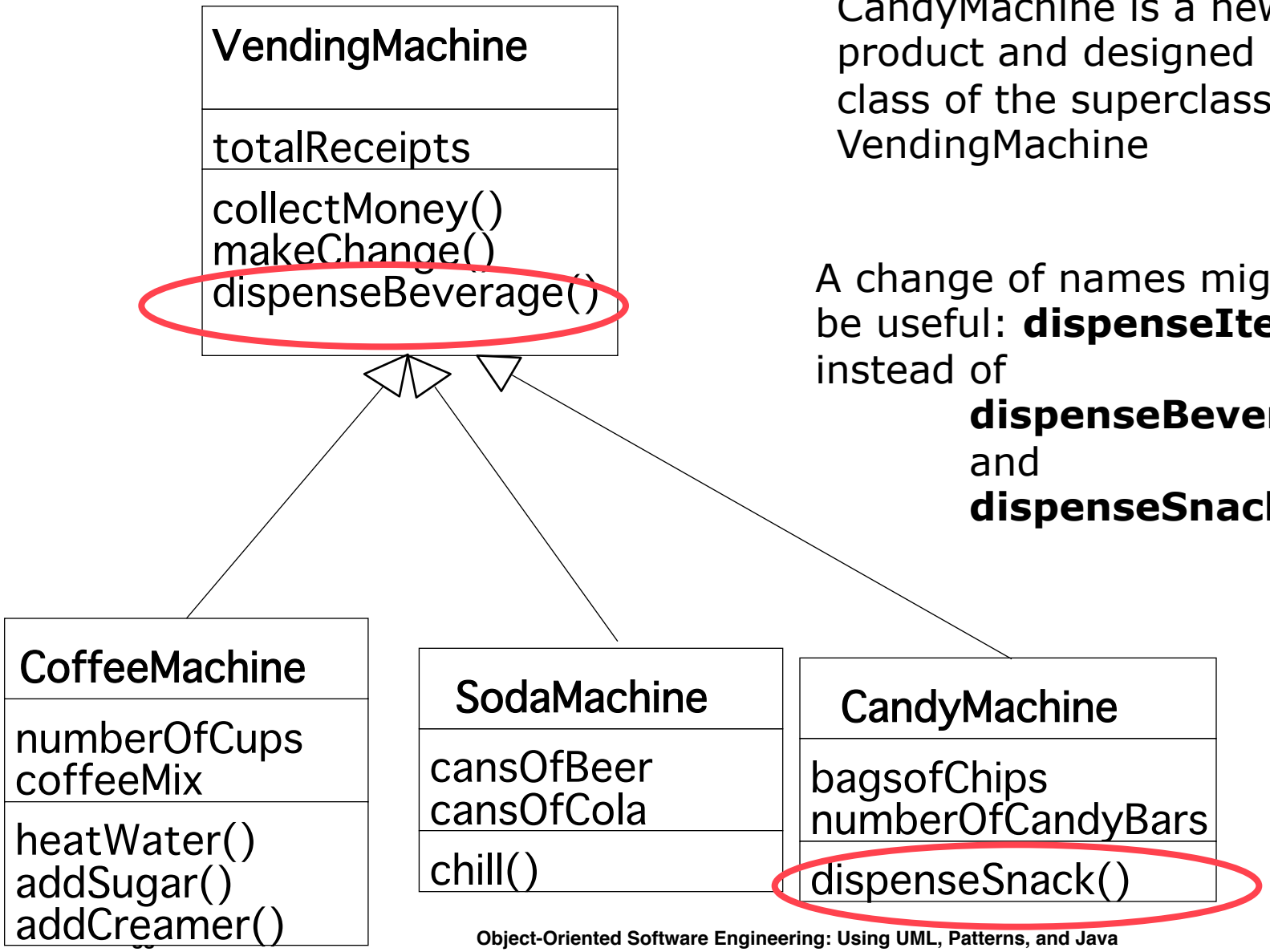


Generalization:
The class CoffeeMachine is discovered first, then the class SodaMachine, then the superclass VendingMachine

Restructuring of Attributes and Operations is often a Consequence of Generalization



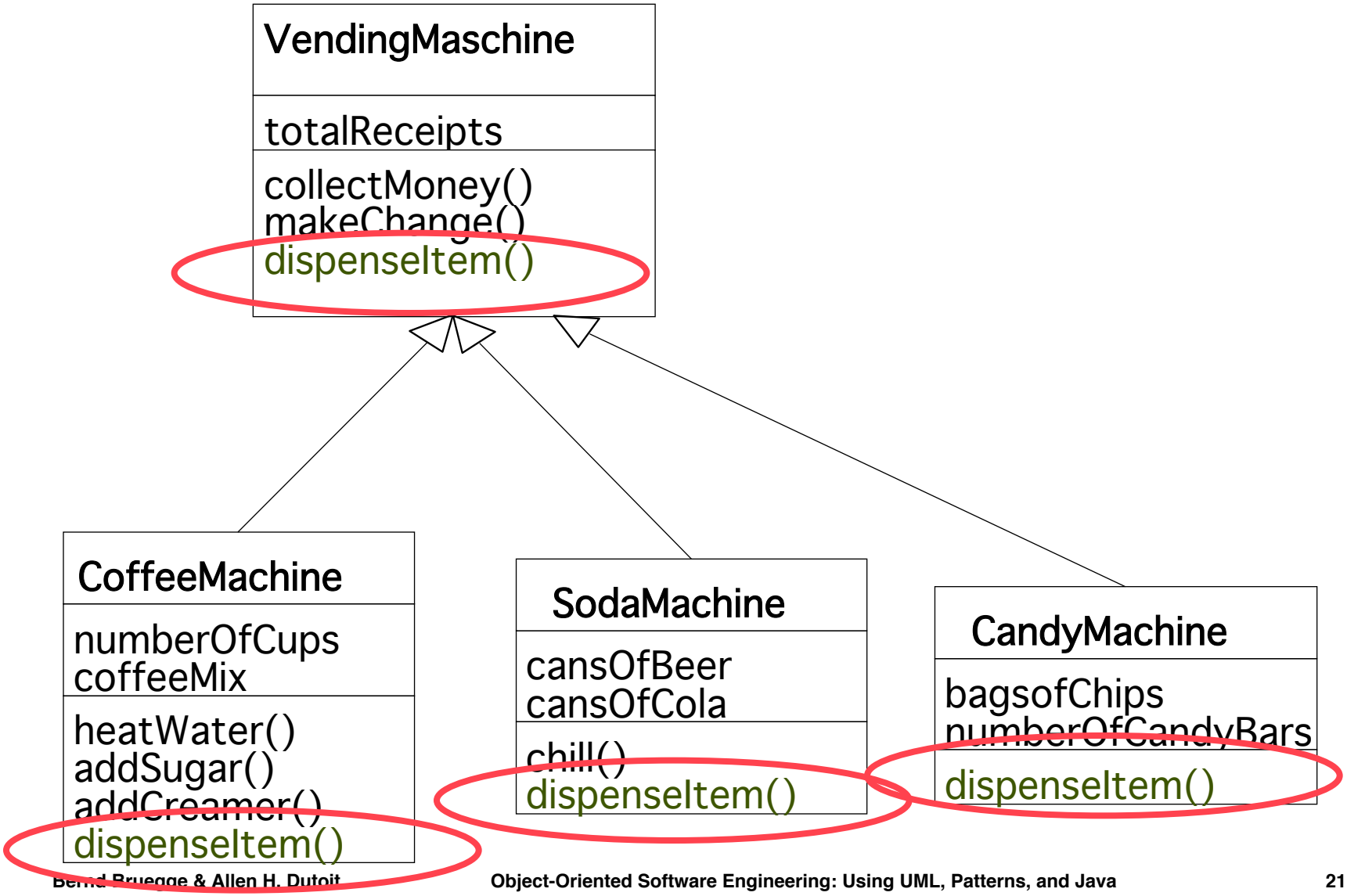
An Example of a Specialization



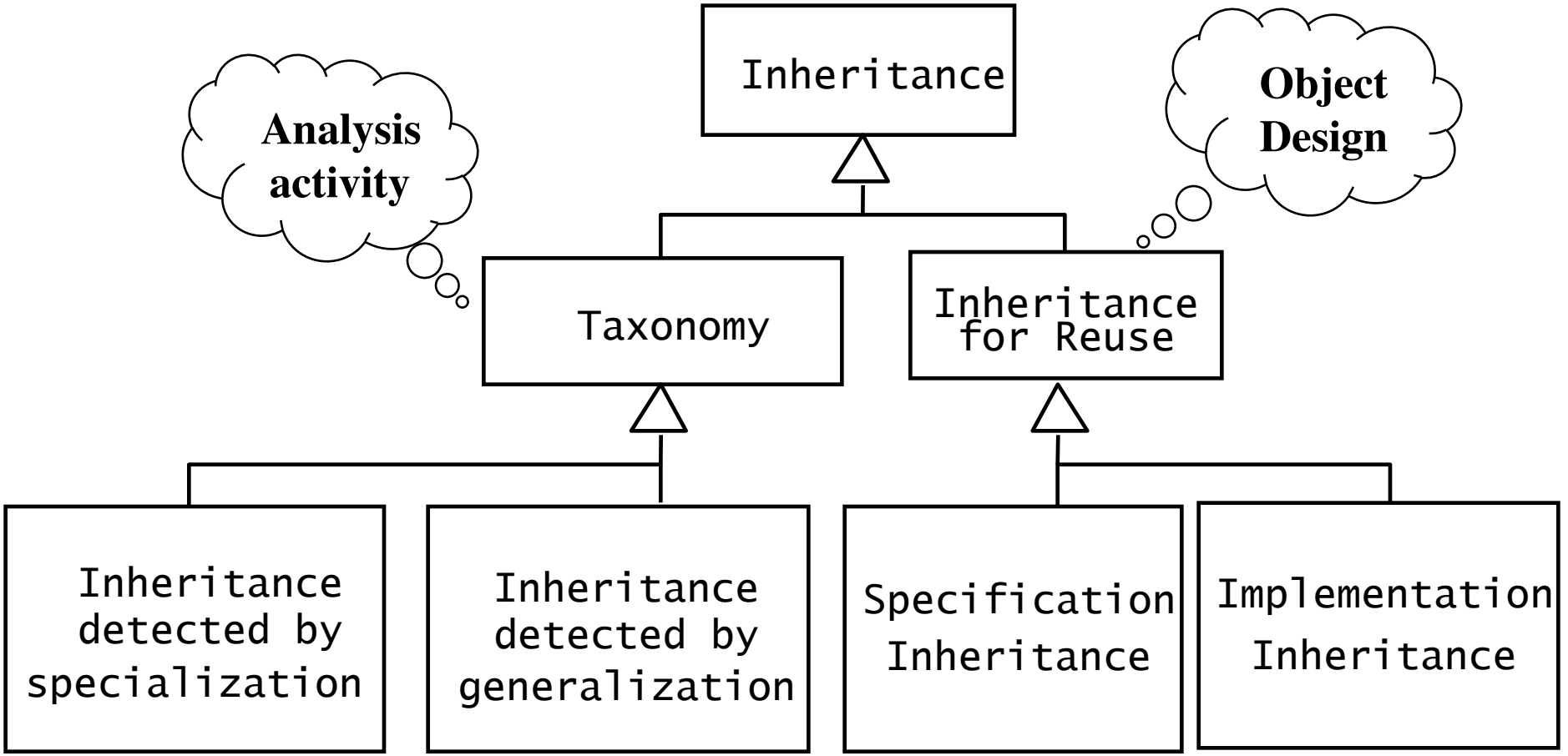
CandyMachine is a new product and designed as a sub class of the superclass VendingMachine

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

Example of a Specialization (2)



Meta-Model for Inheritance



For Reuse: Implementation Inheritance and Specification Inheritance

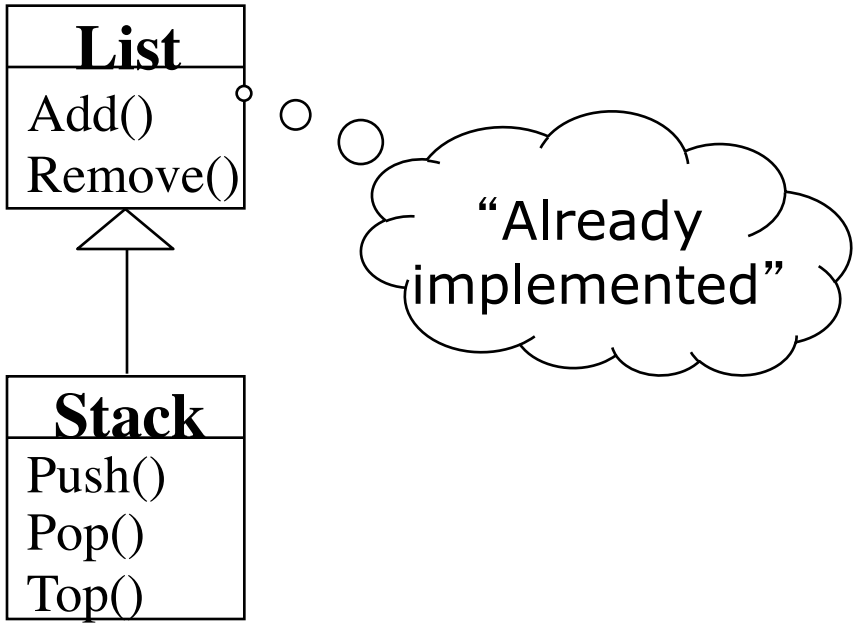
- Implementation inheritance
  - Also called class inheritance
  - Goal:
    - Extend an applications' functionality by reusing functionality from the super class
    - Inherit from an existing class with some or all operations already implemented
- Specification Inheritance
  - Also called subtyping
  - 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:

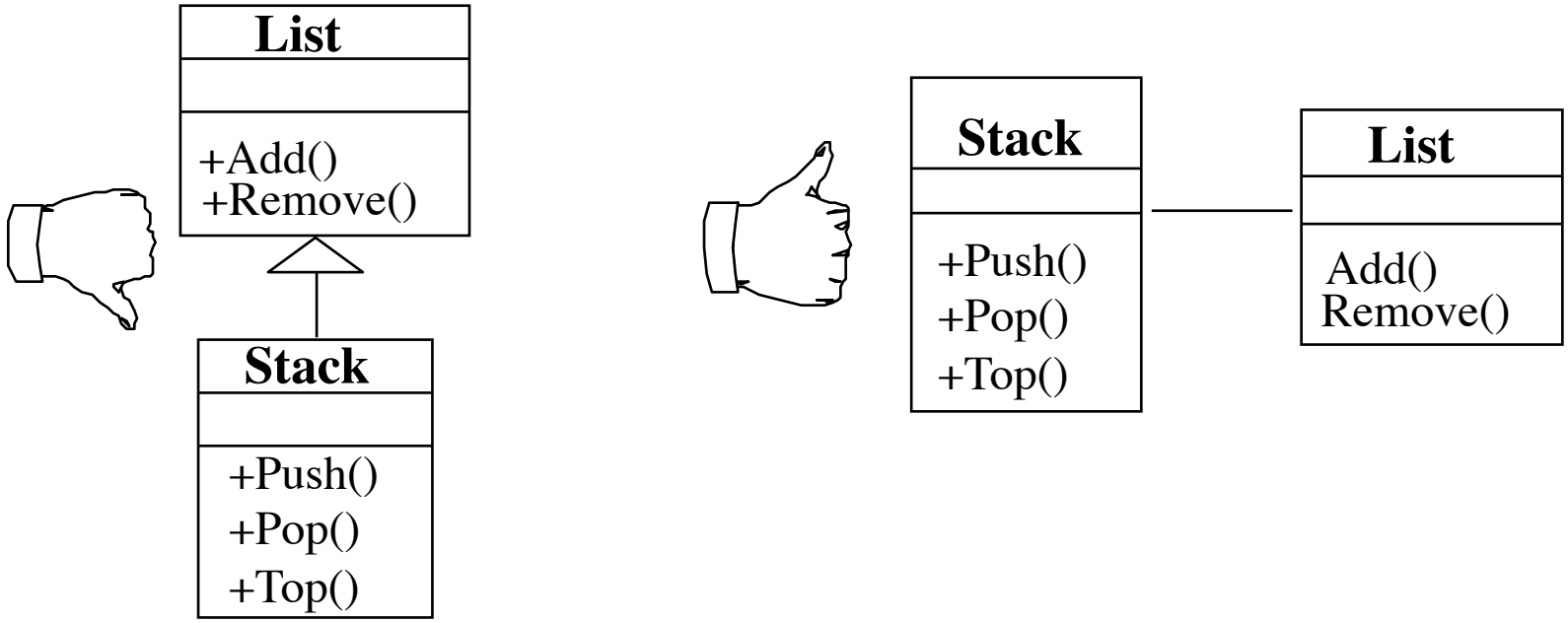
- I have a **List** class, I need a **Stack** class
- How about subclassing the **Stack** class from the **List** class and implementing **Push()**, **Pop(), Top()** with **Add()** and **Remove()**?



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

Delegation instead of Implementation Inheritance

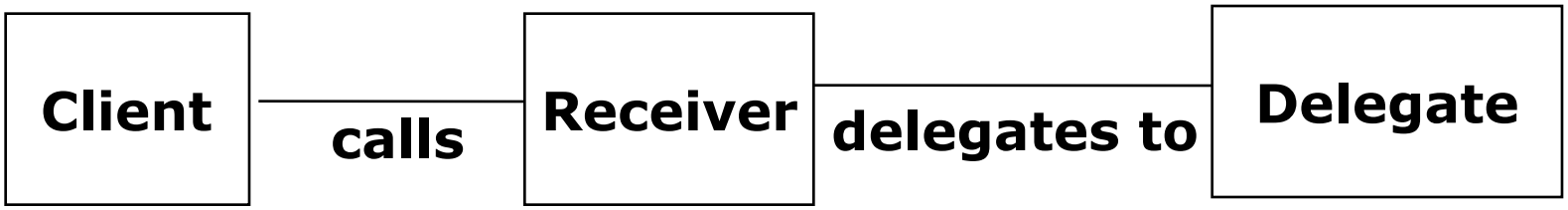
- Inheritance: Extending a Base class by a new operation or overriding an operation.
- Delegation: Catching an operation and sending it to another object.
- Which of the following models is better?



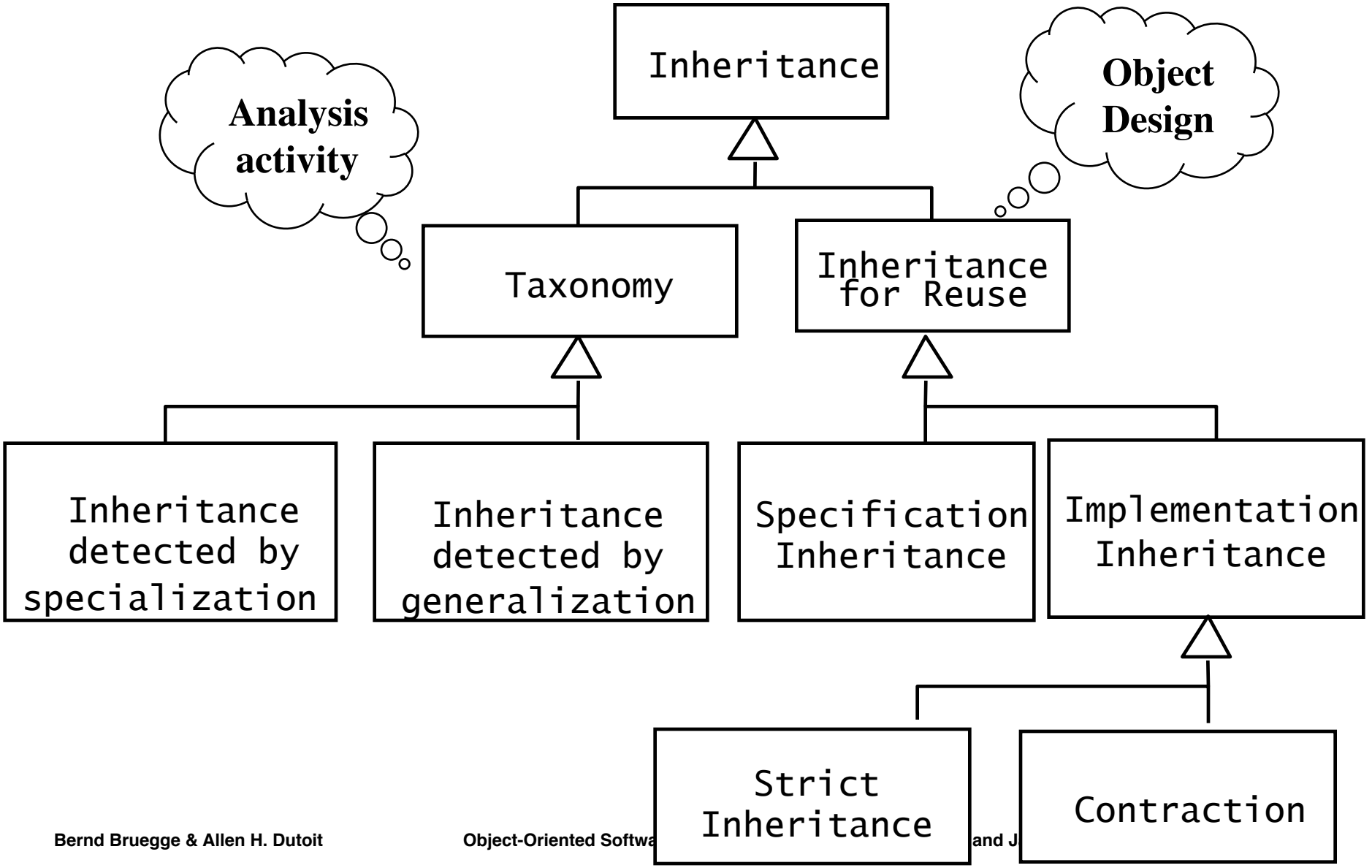
Delegation

- 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.



Revised Metamodel for Inheritance



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

Package it all up

- Pack up design into discrete units that can be edited, compiled, linked, reused
- Construct physical modules
  - Ideally use one package for each subsystem
  - System decomposition might not be good for implementation.
- Two design principles for packaging
  - Minimize coupling:
    - Classes in client-supplier relationships are usually loosely coupled
    - Avoid large number of parameters in methods to avoid strong coupling (should be less than 4-5)
    - Avoid global data
  - Maximize cohesion: Put classes connected by associations into one 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 an interface object has too many operations, reconsider the number of interface objects
- If you have too many interface objects, reconsider the number of subsystems
- Interface objects vs Java interface:
  - Interface object: Used during requirements analysis, system design, object design. Denotes a service or API
  - Java interface: Used during implementation in Java (May or may not implement an interface object).