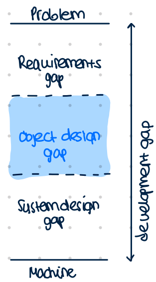
LECTURE 6

27.05.21

OUTLINE
1) Object design
2) Reuse
3) Generalization vs. specialization
4) Design Patterns
   4.1) Composite pattern
   4.2) Bridge pattern
   4.3) Proxy pattern

1) OBJECT DESIGN

PURPOSE:
- design decisions to prepare the system implementation
  → object design = basis for implementation
- Transform system model (→ optimize it)
- Investigate alternative ways to implement the system model
  → based on design goals: min execution time, memory, ...



4 activities of object design

1. REUSE: Identification of existing solutions
- use inheritance
- off-the-shelf components and additional solution objects
- use of design pattern

2. Interface specification
- Describes each class interface precisely

Focus on reuse and specification

3. Object model restructuring
- Transforms the object design model to improve its understandably and extensibility

4. Object model optimization
- Transforms the object design model to address performance criteria

Towards mapping models to code

IDENTIFY COMPONENTS
1. Identify the missing components to the object design gap
2. Make a build or buy decision to obtain the missing components

→ Component based software engineering: the design gap is filled with available components (0% coding)

Modeling the real world

=> Leads to a system that reflects today's realities but not necessary tomorrow's
- There is a need for reusable and extendable designs

2) REUSE

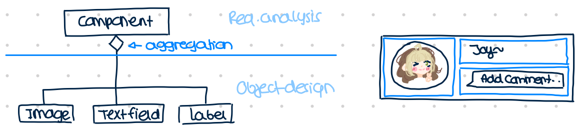
Types of reuse:
1) Reuse of design knowledge
2) Reuse of existing classes
3) Reuse of existing interfaces

Techniques to close the object design gap with reuse

1) BLACK-BOX-REUSE (aggregation and decomposition) — Modeled with ◇ ◆
   → creation of new class through aggregation of existing class
   → new class offers aggregated functionality of existing class

2) WHITE-BOX-REUSE: (inheritance) — Modeled with △
   → new class is created by subclassing
   → new class reuses functionality of superclass and may offer new functionality

Example: black box reuse



Inheritance in software engineering

Two different goals

1) Description of taxonomies
→ used during req. elicitation and analysis

ACTIVITY: Identify application domain objects that are hierarchically related

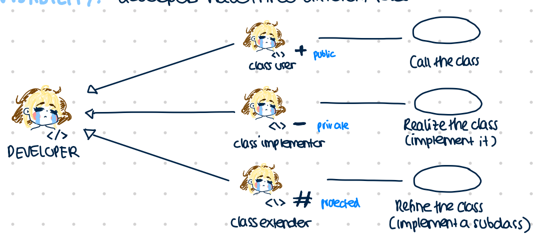
GOAL: Make analysis model more understandable

2) Interface specification
→ used during object design

ACTIVITY: Identify the signatures, visibility and return type of all identifiable objects

GOAL: Increase the reusability, enhance the modificability and the extensibility

VISABILITY: developers have three different roles



DISTINCTION

During analysis:

Course
name req. Semester times
enroll()

During: object design

Course
- courseID : int + name: String + req. Semester: int + times: Date
+ enroll(s: Student): void

Interface specification

1) Implementation inheritance
- Subclassing from an implementation
- REUSE: Implemented functionality in the superclass

2) Delegation
- Catching an operation and sending it to another object where it's already implemented
- REUSE: Implemented functionality = existing object

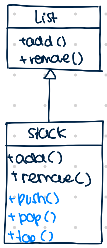
3) Specification inheritance
- Subclassing from a specification
- Specification = abstract class where all operation are specified but not implemented
- REUSE: Specified functionality in superclass

1) Implementation inheritance:

A class is already implemented that does almost the same as the desired class

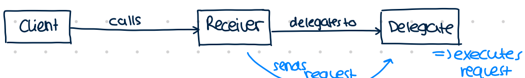
PROBLEM:

The inherited operations might exhibit unwanted behavior

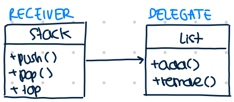


2) Delegation

=> REUSE functionality of existing object using object instantiation and method calls
- 3 objects involved



=> Existence of reciever makes sure that client cannot „misuse" the delegate



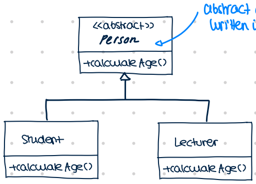
This one is better than the one with inheritance!

Implementation inheritance vs delegation

Implementation inheritance	Delegation
↳ Extending a base by a new operation or overriding an existing operation	↳ catching an operation and sending it to another object
+ Straight forward to use + Supported by many coding languages + Easy to implement new functionalities in the subclass	+ Flexible, because any object can be replaced at runtime by another one
- Inheritance exposes some methods of the parent class - Changes in parent class → forces changes in subclass	- Inefficient, because objects are encapsulated

3) Specification inheritance:

EXAMPLE



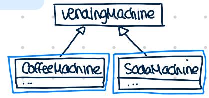
abstract operations written in italics

→ Abstract operation must be implemented by subclass

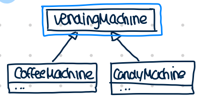
3) Generalization vs. Specialization

Discovering inheritance

1. GENERALIZATION: SUB class → SUPERclass
   => finds subclass first, then superclass

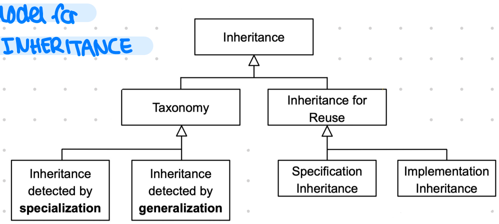


2. SPECIALIZATION: SUPER class → SUBclass



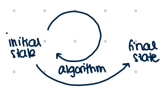
=> finds superclass first, and then subclasses

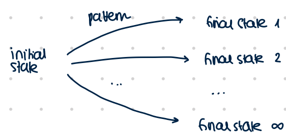
Model for INHERITANCE



4) DESIGN PATTERNS

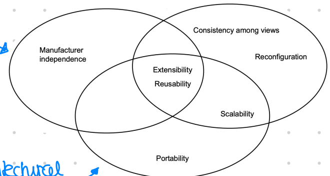
Algorithm:	vs	Pattern:
- solving a problem with a - finite sequence of - well-defined instruction		- describes a problem which - occurs over and over again - describes core of the solution, so you can - use the solution multiple times with different outcomes





Patterns address nonfunctional requirements

Analysis pattern



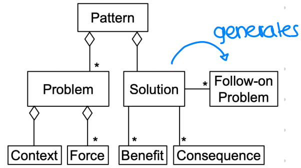
Design patterns

Architectural pattern

PATTERN: ORIGINAL DEFINITION:

"A pattern is a three part rule, which expresses a relation between a certain context, a problem and a solution"

Modeling a PATTERN in UML



CATEGORIZATION OF PATTERNS

Patterns for development activities
- Analysis
- Design
- Architecture
- Testing

Patterns for cross-functional activities
- Process
- Agile
- Build and release management

Antipatterns
- Smells and Refactoring

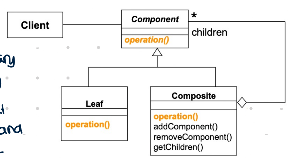
TYPES OF DESIGN PATTERNS

- Structural patterns
  - reduces coupling between classes
  - abstract classes to enable future extensions
  - Encapsulates complex structures
- Behavioral patterns:
  - allow choice between algorithms
  - allows assignments of responsibility to objects
  - Model complex control flows that are difficult to follow at runtime
- Creational patterns
  - Allow to abstract
  - Make system independent from the way its objects are created, composed & represented

4.1) Composite Pattern

PROBLEM: There are hierarchies with arbitrary depth or width (folders, files,..)

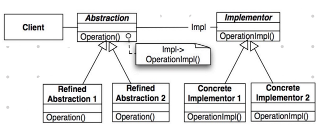
SOLUTION: Composition pattern lets a Client treat an individual class called Leaf and Composition Leaf classes uniformly



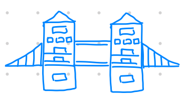
4.2) Bridge Pattern

PROBLEM: Many design decisions are made final at design time or at compile time
=> Sometimes it is desirable to delay design decisions until runtime

SOLUTION: Bridge pattern allows to delay a decision until the startup time

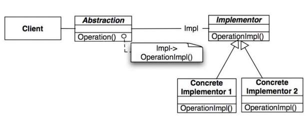


=>



looks like a bridge :3
(bridge between application & solution domain)

„Degenerated" bridge pattern:



4.3) PROXY PATTERN

PROBLEM #1:
=> Object = complex, instantiation is expensive

SOLUTION #1:
- delay instantiation until object is actually used
- if object is never used → no instantiation cost

PROBLEM #2:
=> Object is located on another node
accessing the object is expensive

SOLUTION #2:
- Instantiate & initialize a "smaller" local object → representative (: proxy) for remote object
- Try to access mostly the local object
- Remote object only if necessary

=> Reduces cost of accessing objects
Proxy as stand in for remote object
Location transparency

Use cases of the proxy pattern

- Caching (remote proxy)
  - The proxy object is a local representative for an object in a different address space
    - Caching is good if information does not change too often
    - If information changes, the cache needs to be flushed
- Substitute (virtual proxy)
  - The proxy object acts as a stand-in for an object which is expensive to create or download
    - Good for information that is not immediately accessed
    - Good for objects that are not visible (not in line of sight, far away)
- Access control (protection Proxy)
  - The proxy object provides access control to the real object
    - Good when different objects should have different access and viewing rights

Summary:

- Inheritance can be used in analysis as well as object design
  - => during analysis: describes taxonomies
  - => during object design: used for interface specification & reuse
- Blackbox vs. Whitebox reuse
- Interface specification: Implementation & specification inheritance delegation
- Discovering inheritance: generalization / specialization
- Design patterns: composite, bridge, proxy

ty for studying with me ♡
good luck~