OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG
FACULTY OF COMPUTER SCIENCE
SwarmLab

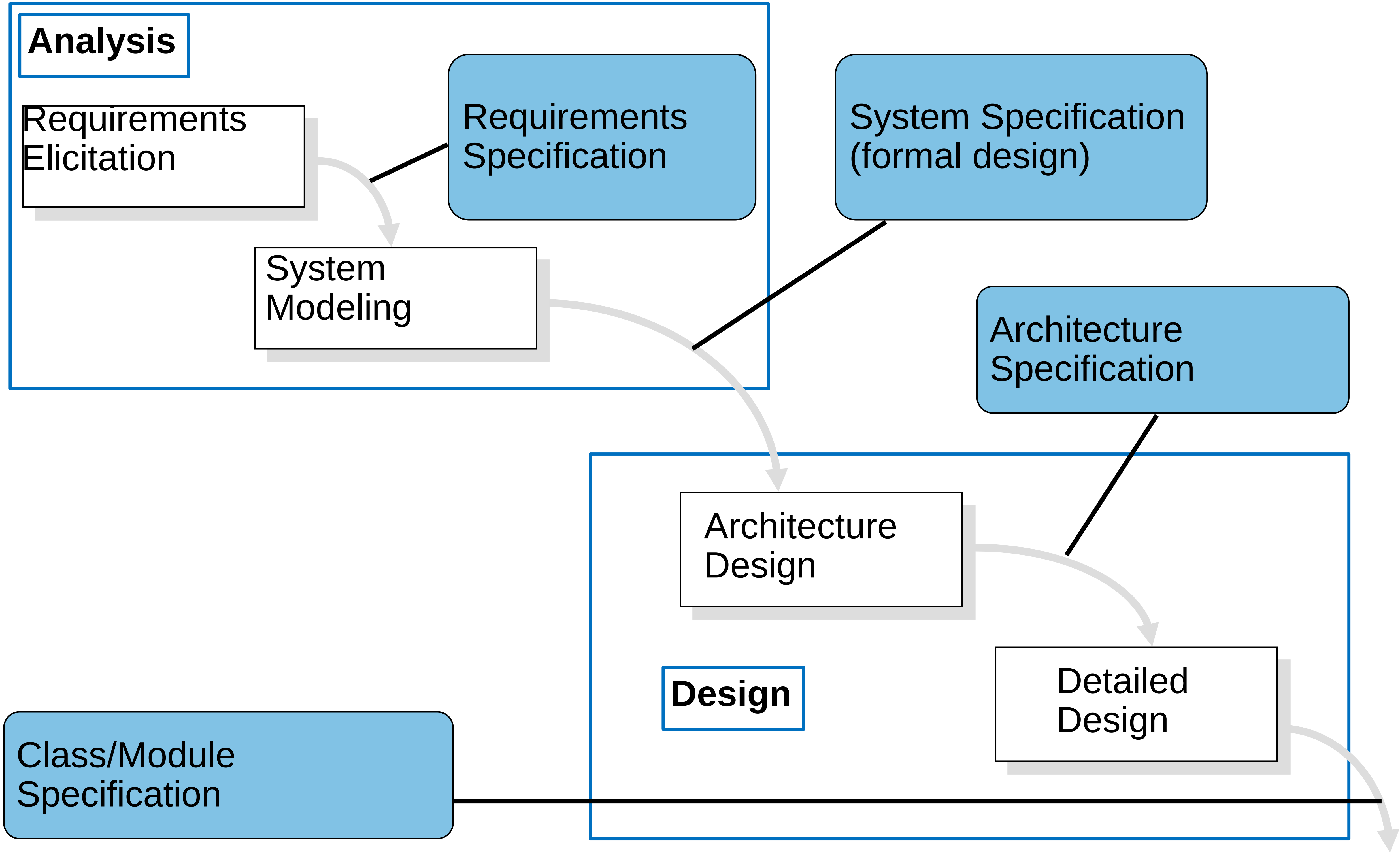
Introduction to Software Engineering for Engineers L-07: Detailed Design & Design Patterns
Part 1: UML Diagrams Revisited

Dr.-Ing. Christoph Steup

Content

- Repetition & Introduction
- UML class diagrams for analysis and design
- Design Pattern

Design Phases

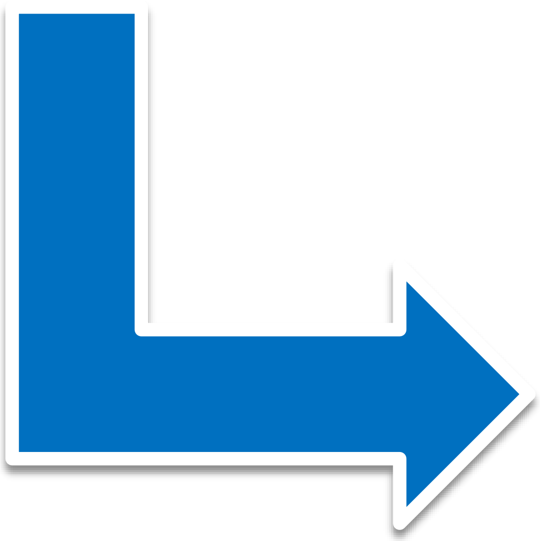

Analysis
Requirements Elicitation
Requirements Specification
System Specification (formal design)
System Modeling
Architecture Specification
Architecture Design
Design
Detailed Design
Class/Module Specification

Object-Oriented (Detailed) Design

Starting Point:

Architecture Design:

- ➢ Decomposition in subsystems (possibly using reference architectures)
- ➢ distribution concept
- ➢ workflow model



Result:

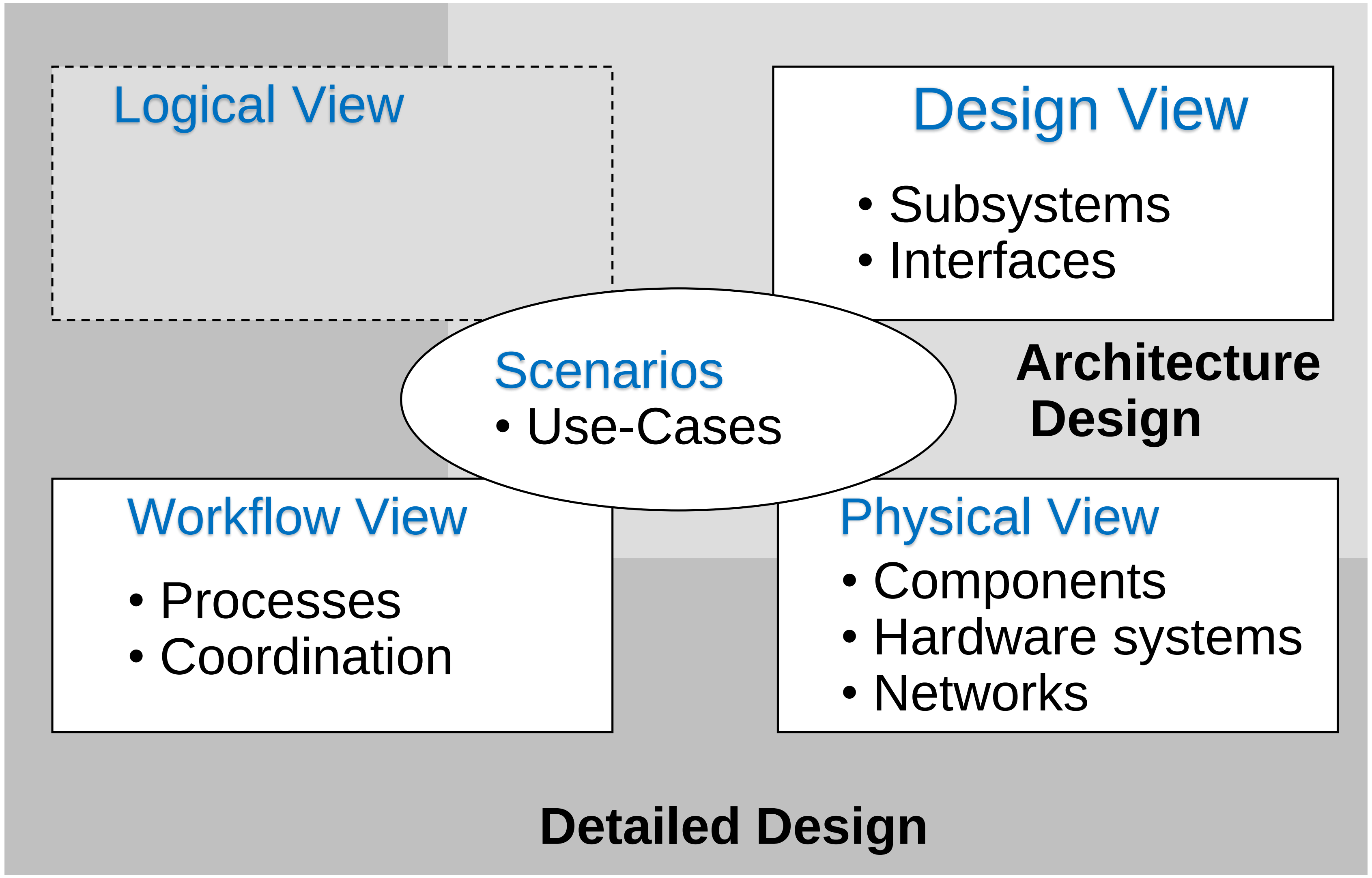
OO model for each subsystem of the architecture

OO model for supporting subsystems taking selected technologies into account

specification of classes

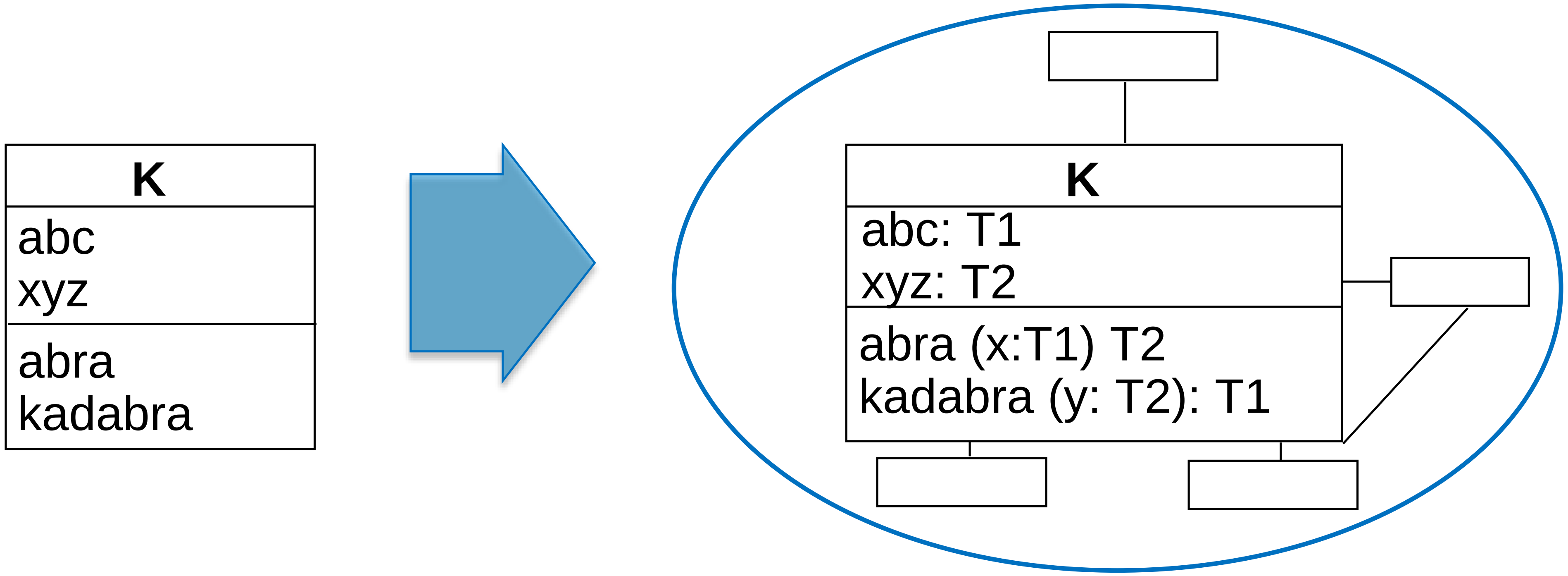
specification of interfaces

Object-Oriented (Detailed) Design



Refinement of Analysis Model

General view:



Refinement of Analysis Model

What is refined?	Additional details (compared to analysis model):
Functional core	➢ Lists of attributes/operations: complete ➢ Attributes and operations: data types, accessibility ➢ Operations: specification (e.g., pre-/post-conditions) ➢ Associations/aggregations: direction, order, qualification
Additional classes/packages:	➢ Integration into infrastructure, connection to legacy systems etc. ➢ adaptation and decoupling layers for selected technologies (e.g., data access layer, CORBA interfaces, XML connection...)

UML in the Design Phase

General:

Analysis models are rebuilt/modified in the Design Phase

Detailing

Specification

Classes
→real-world objects

Classes
→Part of software system

UML in the Design Phase

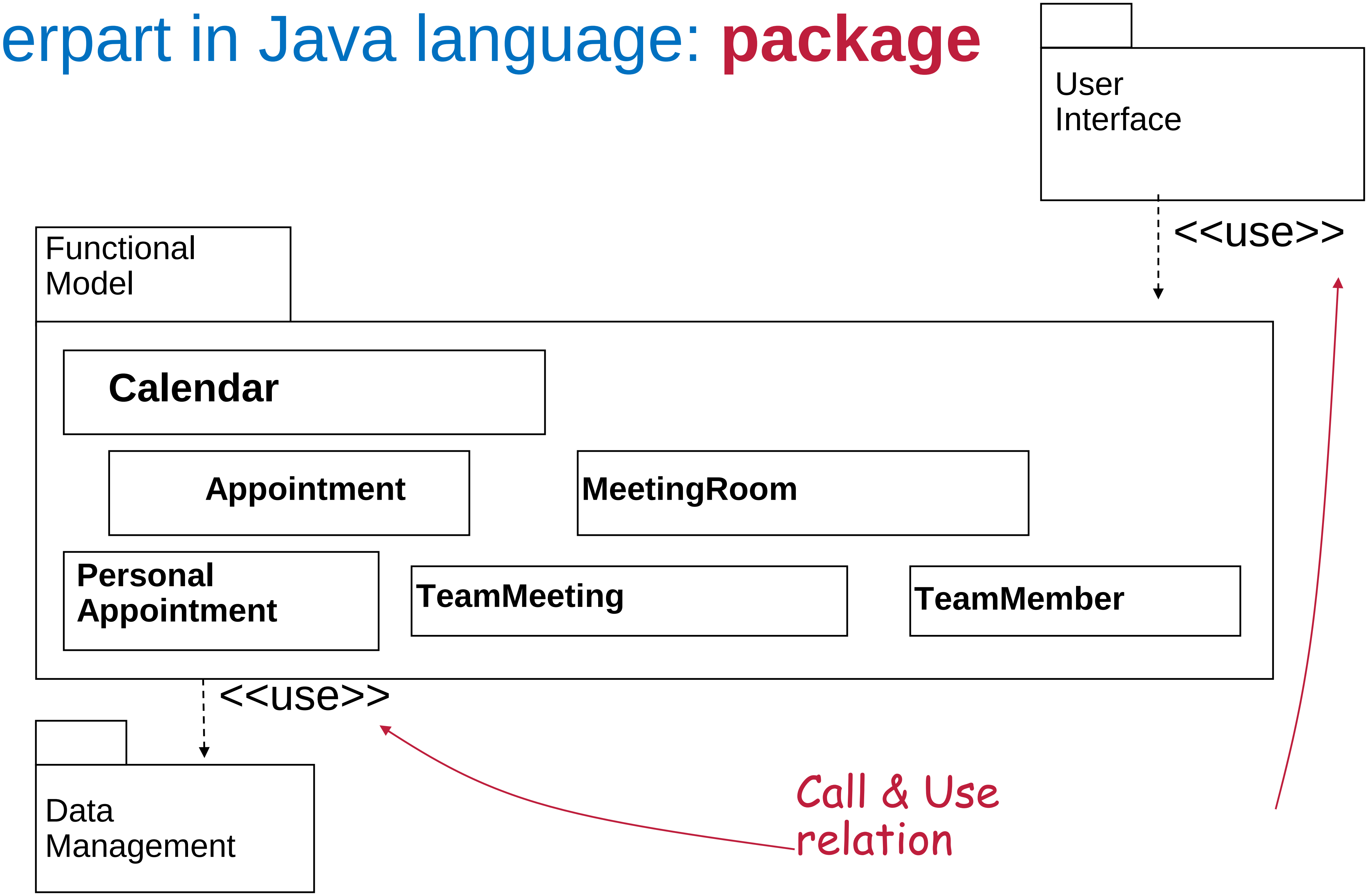
General	Analysis models are rebuilt in Design Phase
Class diagrams	Analysis: Classes represent real-world objects. Design: Classe are parts of the software system. This is achieved by means of detailing and specification.
State charts	If not already decomposed into single specifications of methods, state charts are detailed as well.
Templates	Activity, sequence, and use-case diagrams

UML for Logical (Detailed) Design

	Analysis Model	Design Model
Objects	Technical artefacts	Objects: software units
Classes	Technical terms	Classes: Schemata
Inheritance	Term structure	Program derivation
General	▪ Assumption: perfect technology ▪ Functional essence ▪ Entirely project-specific	▪ fulfils concrete conditions ▪ entire structure of the system ▪ Similarities between related projects ▪ Precise definition of the structure
Notation	UML	UML
	Abstract outline of structure	***More structure & more details***

Packages and Subsystems

- UML: Packages for structuring of models
- Component: realization of an architectural unit
- Counterpart in Java language: **package**

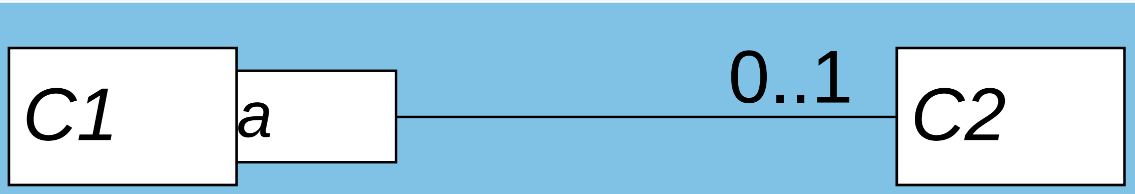
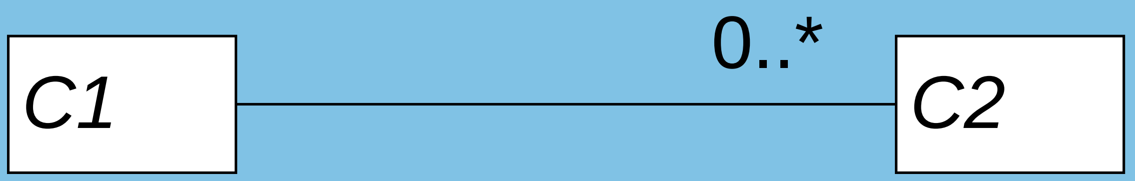


Visibility (Access Modifiers)

	Visibility			
UML	+	#	-	
Java	public	protected	private	(default)
Visible for:				
Same class	Yes	Yes	No	Yes
Other class, same package	Yes	Yes/No*	No	Yes
Other class, other package	Yes	No	No	No
Sub class, same package	Yes	Yes	Yes	Yes
Sub class, other package	Yes	Yes	No	No

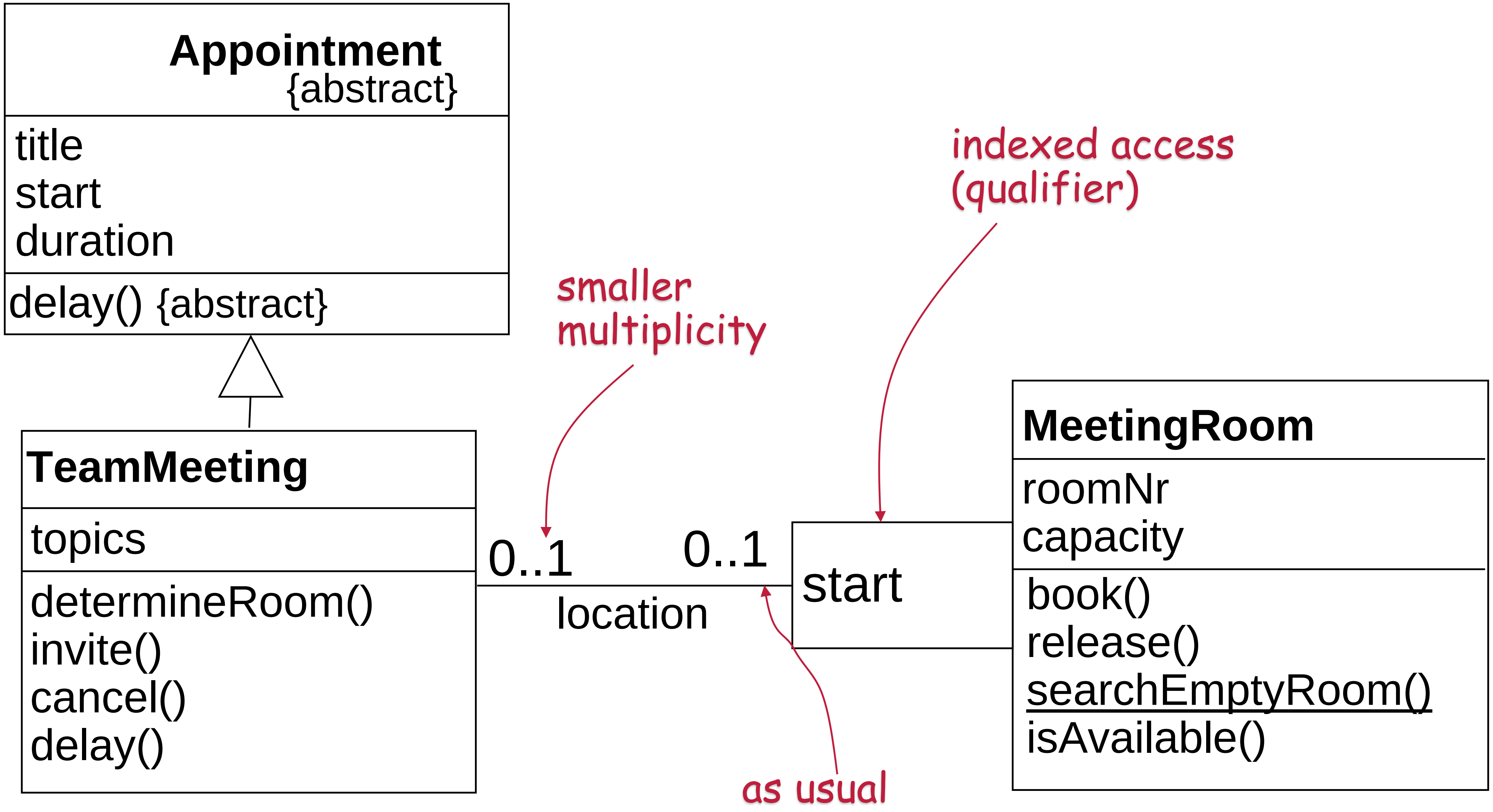
* In UML and C++ “No”, in Java “Yes”.

Qualified Associations

	Qualified Association
Definition (informal)	A *Qualifier* is an attribute for an association between classes C1 and C2 so that the set of C2 objects, associated with C1 objects, are *partitioned*. Purpose: direct access (by avoiding a search).
Notation	 as detailed specification of: 
Note	Importance especially in relation with databases (indices), but can also be represented with Java (with appropriate data structures).

Qualified Associations

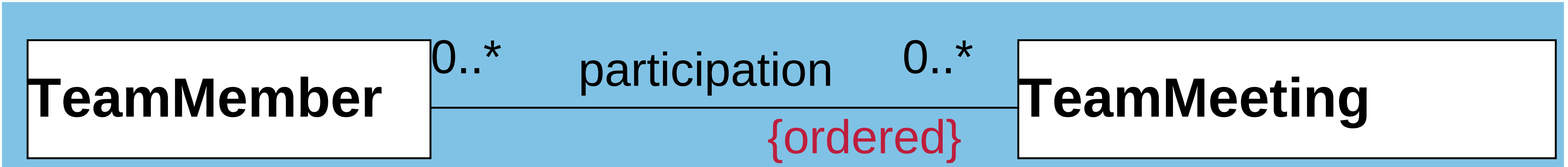
Example

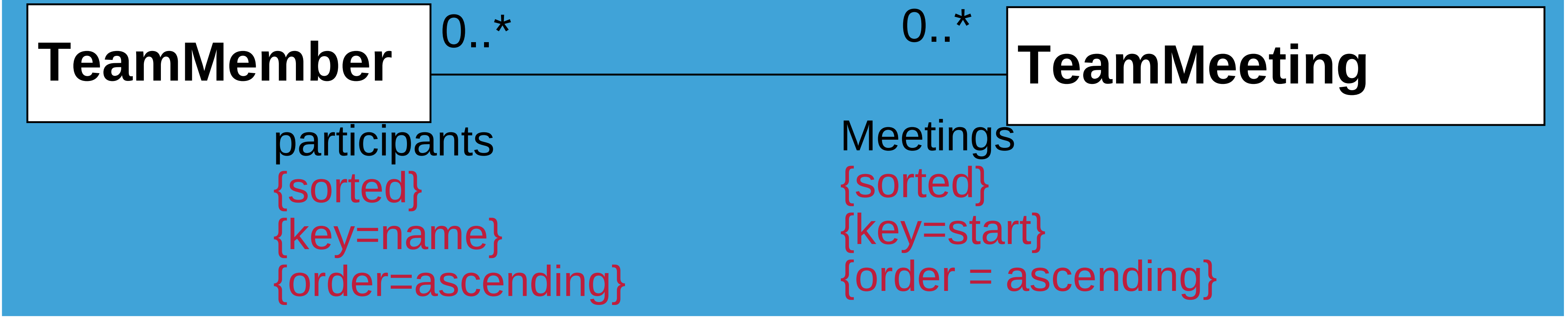


Room12.isAvailable (start=04.05.02 10:00, duration=60min);

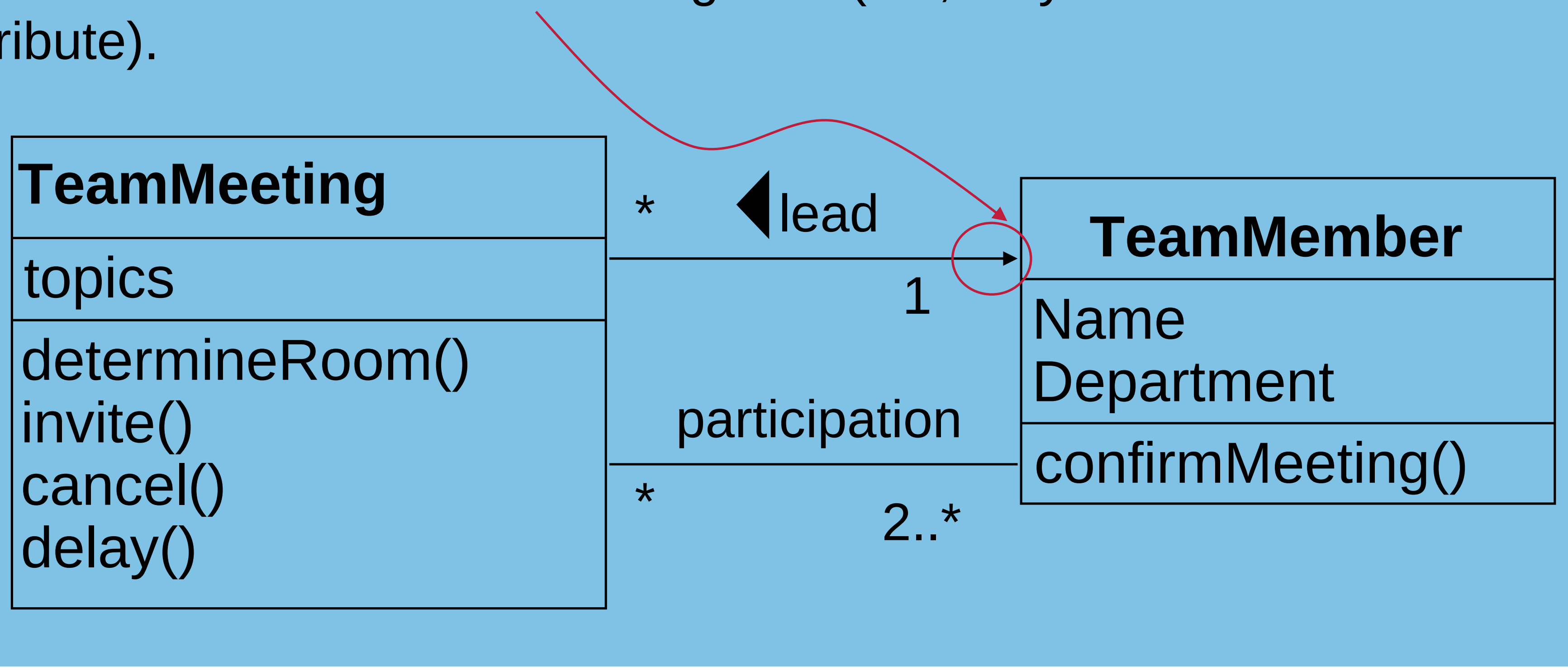
can be queried directly by date, whether an association exists

Ordered and Sorted Association

Definition (informal)	{ordered} at one association end: ▪ fixed order for traversal of associated objects (e.g., by access via iterators). ▪ no duplicates of an object
Example	 TeamMember 0..* participation 0..* TeamMeeting {ordered}

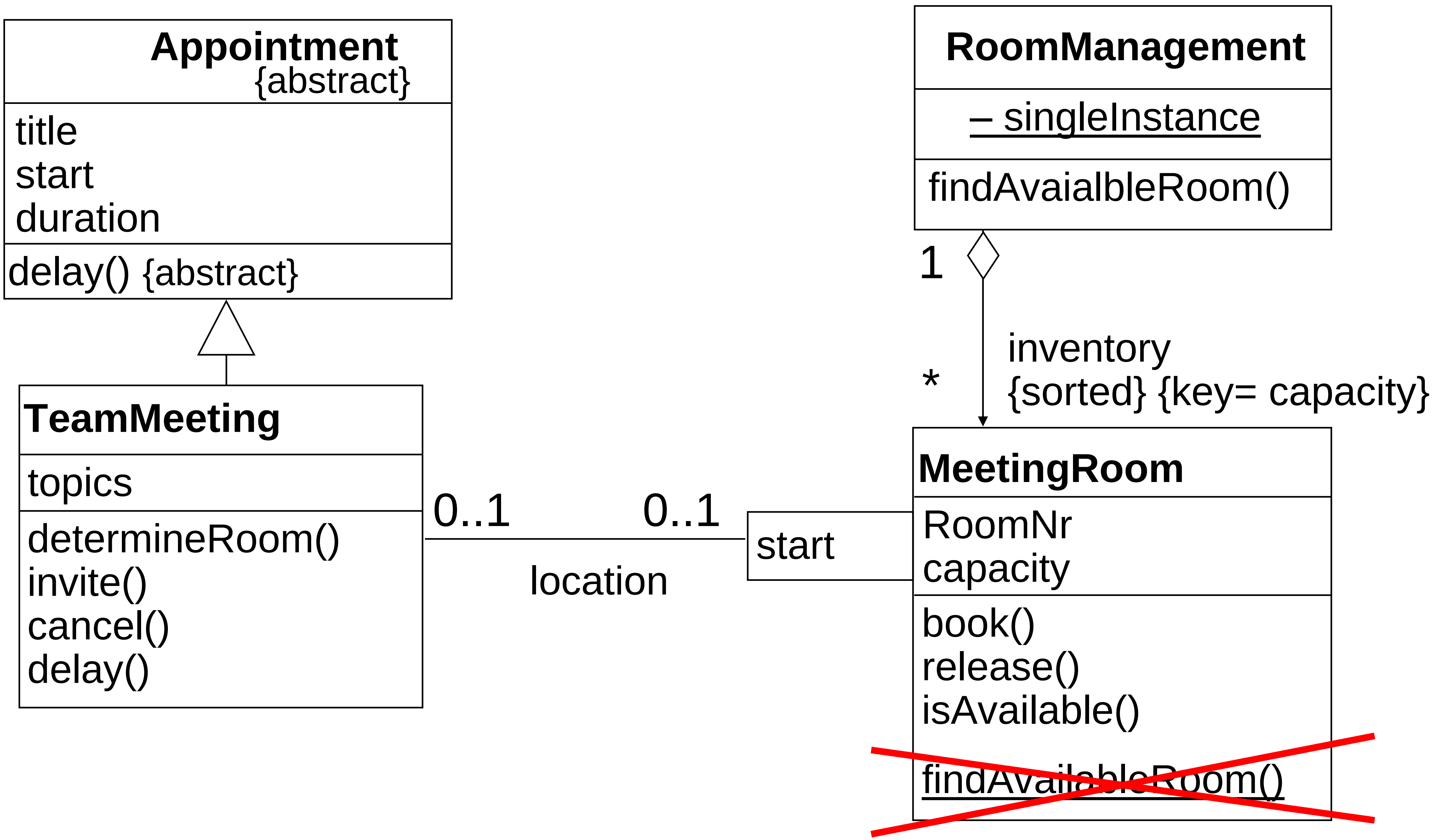
Extension	Further restriction possible, e.g., the requirement for sorting ({sorted}) with respect to particular attributes
Example	 TeamMember 0..* 0..* TeamMeeting participants {sorted} {key=name} {order=ascending} Meetings {sorted} {key=start} {order = ascending}

Direction of Associations

Definition	Associations are used to *navigate* in the network of objects. Usually, associations can be navigated in both directions (i.e., are handled on both sides like an attribute).
Example	Special case: unidirectional navigation (i.e., only on one side handled like an attribute).  TeamMeeting topics determineRoom() invite() cancel() delay() * ◀ lead 1 participation * 2..* TeamMember Name Department confirmMeeting()

Management Classes

Example



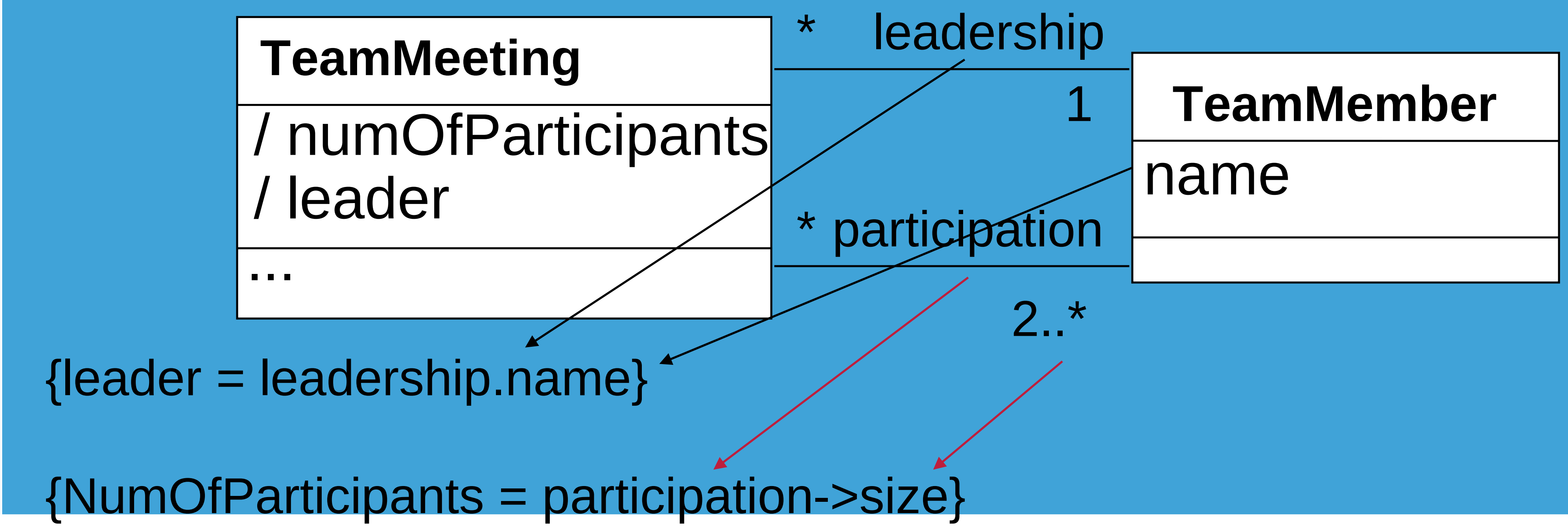
Management Classes: Textual Notes for the Example

- Class *RoomManagement* is only used to realize the class method *findAvailableRoom*() more efficient.

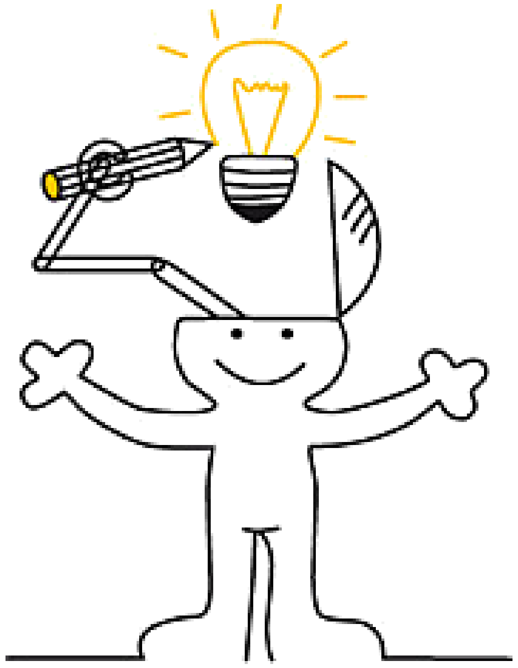
- Only one instance of this class should exist, which is indicated by the class attribute (so called *Singleton*).

Note: Management classes are recommended, if access on all instances of a particular class is required, e.g., on all existing rooms. For instance, in Java there is no other (explicit) possibility for having access to all instances of a particular class.

Derived (Redundant) Elements

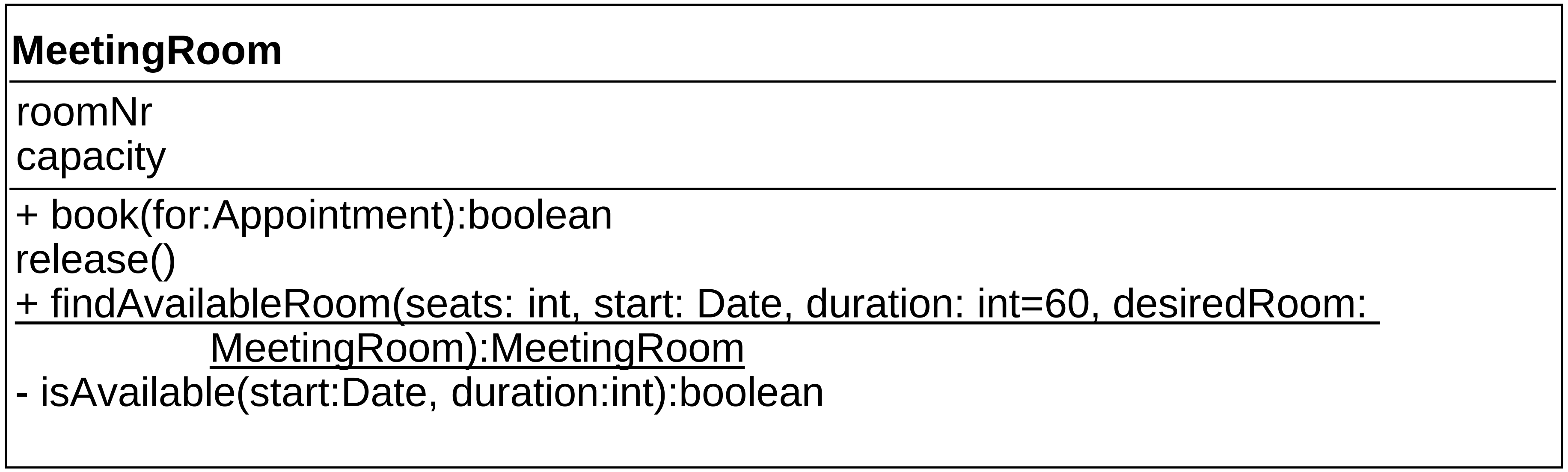
Definition	A ***derived*** model element (e.g., attribute, association) is a model element that can be reconstructed at any time from other (non-derived) elements.
Notation	*/model element or model element* {derived}
Example	 **TeamMeeting** / numOfParticipants / leader ... * leadership 1 * participation 2..* **TeamMember** name {leader = leadership.name} {NumOfParticipants = participation->size}

Parameter and Data Types for Operations



Precisely determine parameter and data types for operations!

Examples (Class *MeetingRoom*):

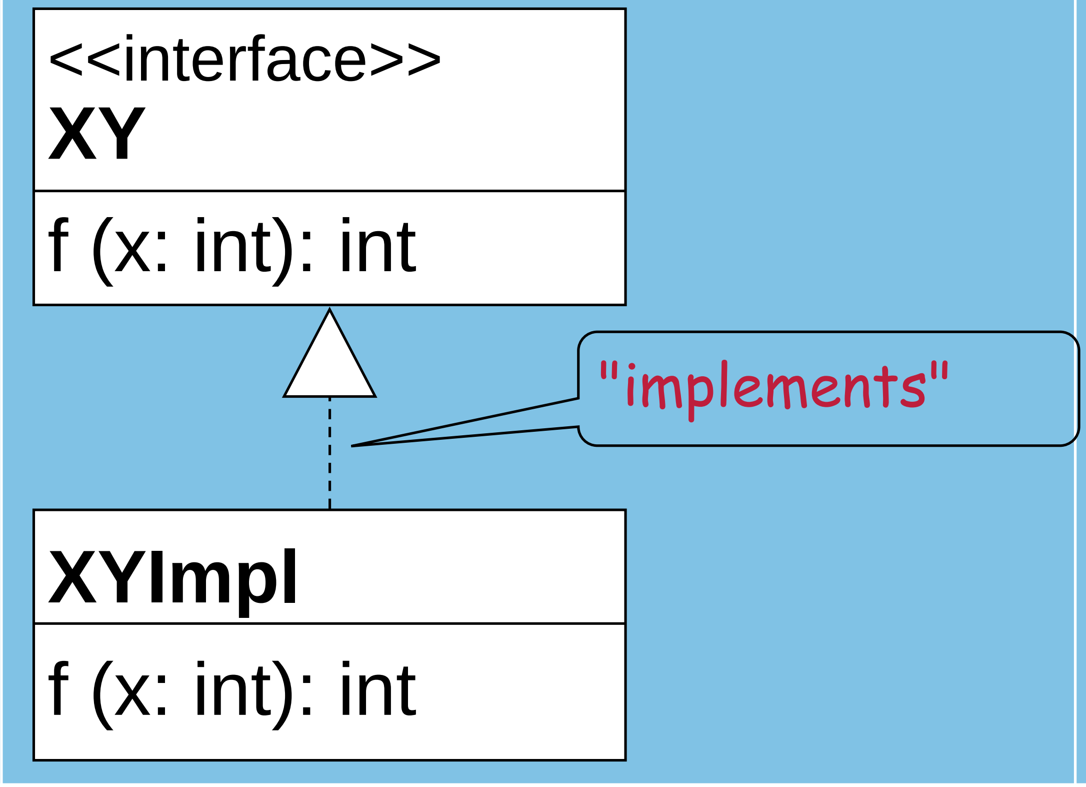


```
MeetingRoom
roomNr
capacity
+ book(for:Appointment):boolean
release()
+ findAvailableRoom(seats: int, start: Date, duration: int=60, desiredRoom:
                MeetingRoom):MeetingRoom
- isAvailable(start:Date, duration:int):boolean
```

Specification of Operations

Definition	**Definition:** The *Specification* of an operation determines its behaviour without defining a particular algorithm.
Basic principle	The *"What"* is described but not the*"How"*.
Forms of specification	➢ Text in natural language (usually with specific conventions) ▪ often embedded in source code (comments) ▪ tool support for generating a documentation, e.g., "javadoc" ➢ Pre- and post-conditions (e.g., in JML) ➢ Tables, specific notations ➢ "Pseudocode" (text similar to a programming languages) ▪ use with care - usually this results in too many details!

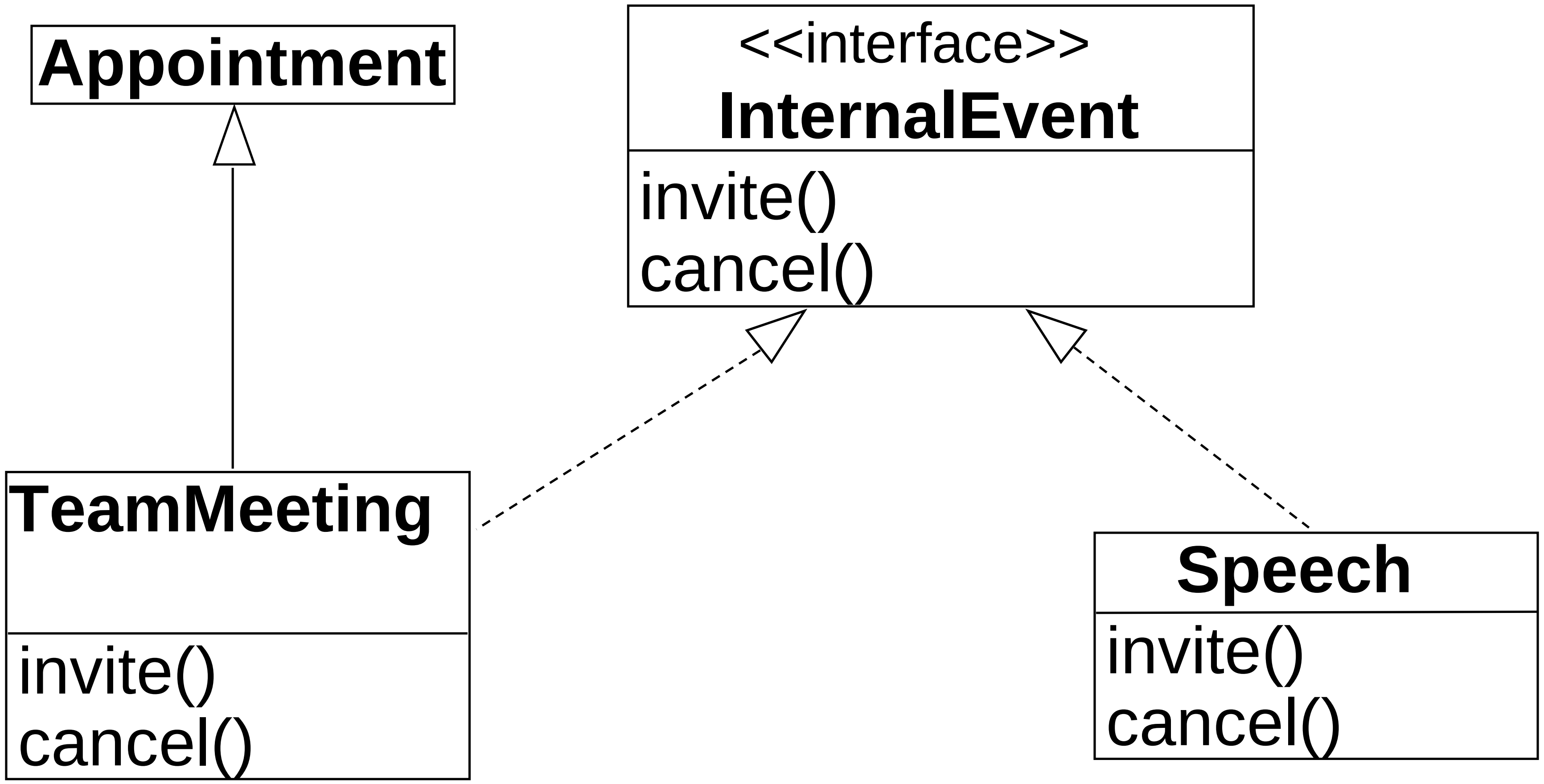
Interfaces

Why	Good software design ensures *homogeneity* and *ergonomics*. Similar functionality should be addressable in the same way.	
Definition	An interface is an abstract class without attributes and method bodies (i.e., implementation).	
Example	UML	Java
	 <<interface>> XY f (x: int): int "implements" XYImpl f (x: int): int	

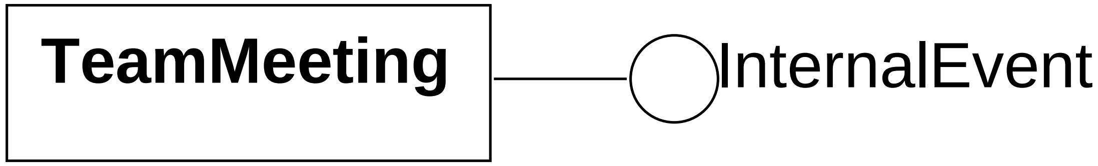
```
interface XY {
  int f (int x);
}

class XYImpl implements XY {
  public int f (int x) {
    ... method body of f ...
  }
...
}
```

Simple Inheritance Using Interfaces

Example



Note: “*lollipop*” notation for interfaces is commonly used and even allowed in UML (and equal):

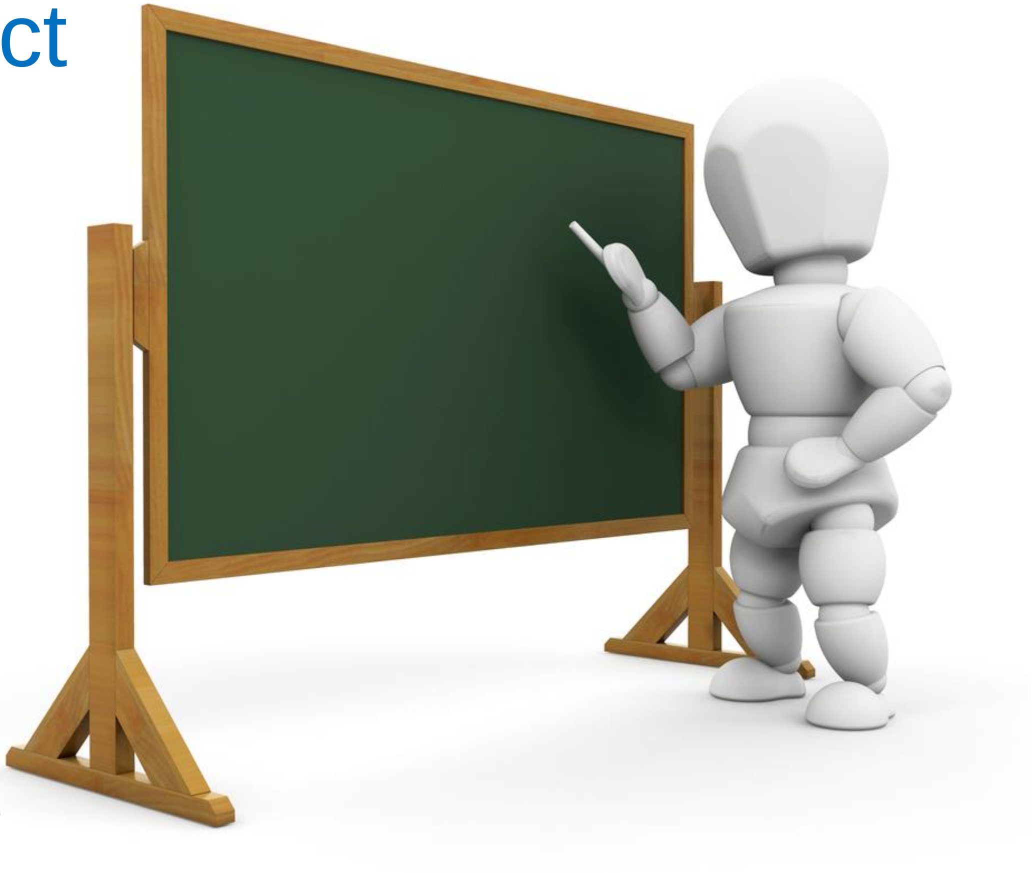


Detour: Interfaces and Abstract Classes

	Abstract Class	**Interface**
General	Contains attributes and operations	▪ Contains operations only (and, if so, constants) ▪ interface provides specific view on a class
Default Behavior	➢ can define default behavior ➢ default behaviour can be redefined in subclasses	➢ can NOT define default behaviour ➢ subclasses must define behavior
Java	Subclass can only inherit behavior from one class	A class can implement multiple interfaces

This was relaxed in newer versions of Java

Summary:
UML Class diagrams in Analysis Models



- Draft: partially incomplete with respect to attributes and operations
- data types and parameters can be omitted
- less established relation to language used for realization
- No consideration how to realize associations

Summary:
UML Class diagrams in Design Models

- complete information for all attributes and operations
- complete specification of data types and parameters
- in relation to the programming language used for realization
- Navigation information, qualifier, order, Management classes
- decision about data structures
- Preparation for connect the functional core (back-end) to user interface and data management

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

FACULTY OF COMPUTER SCIENCE

SwarmLab

Introduction to Software Engineering for Engineers L-07: Detailed Design & Design Patterns
Part 2: Design Patterns

Dr.-Ing. Christoph Steup

Pattern Specification

A pattern consists of

- a context or situation in which a recurring design problem occurs,
- which can be solved by a generic, established solution.

Patterns are described with:

- pattern templates provide a uniform structure, divided into context, Problem, and solution (and respective subitems).
- Each aspect of this pattern template is filled with plain text and diagrams.

Application of Patterns

No automated "Pattern Matching"!	
Design pattern are abstract solutions for „recurring problems“!!	
Rather propagation of idea of the respective pattern	
basic structure of pattern should be observable, if needed, adapt/modify the existing design	
Also, the behavioral scheme must exist in the source code, similar to the pattern idea.	

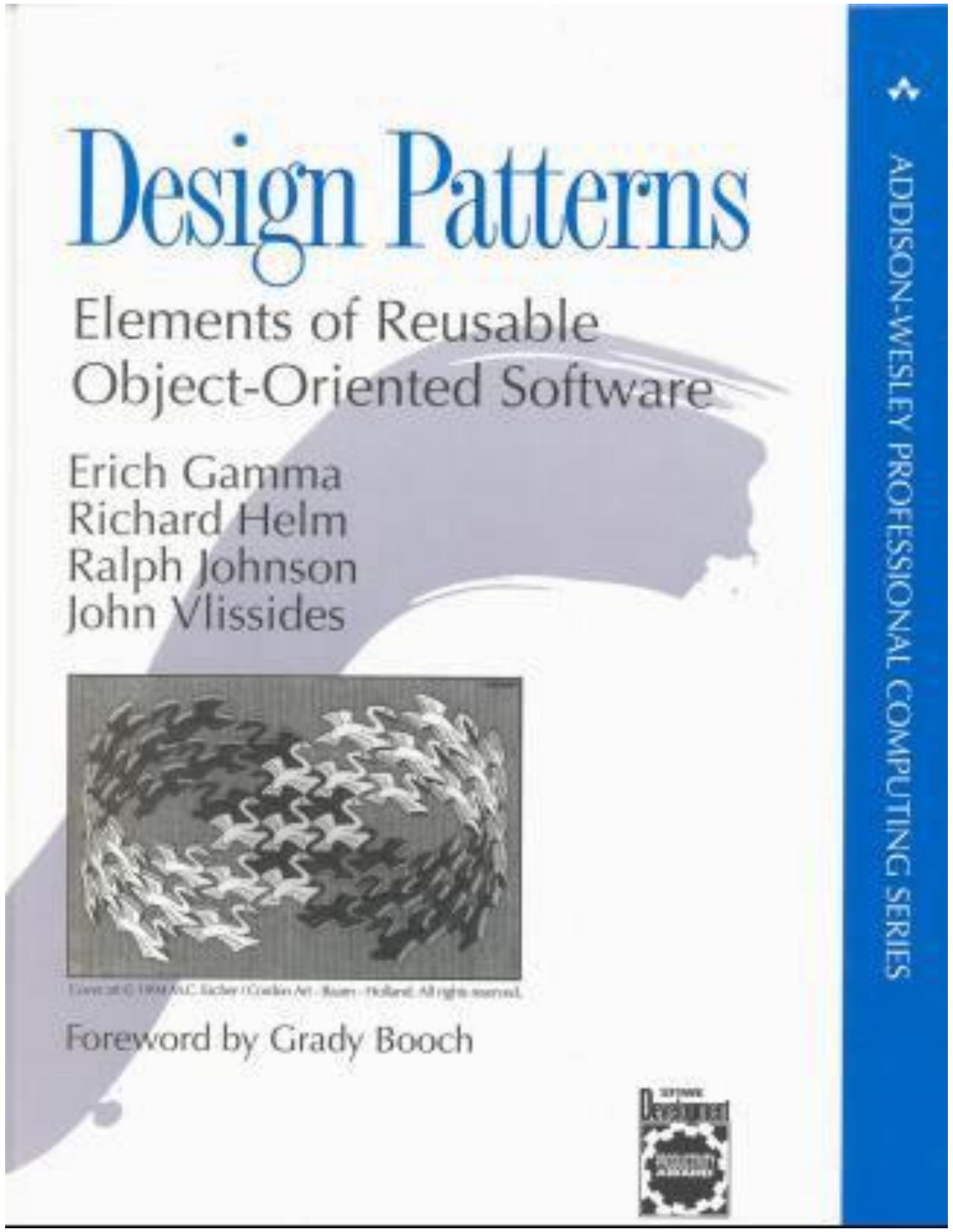
Users of Patterns

Developer	▪ support for design decisions ▪ reuse of approved knowledge of experienced programmers ▪ code examples can ease the start and serve as starting point for own solutions ▪ help for conveying and persisting own knowledge
Team	▪ creating an own language/terminology ▪ standard for documentation ▪ knowledge transfer for new employees
Company	▪ standardized knowledge of company’s knowledge ▪ reuse of approved solutions ▪ consistent architecture of company’s systems

Problems of Patterns

Problem	Reason
Effort	can be high for familiarisation and selection of pattern
Organization	categorization and classification of pattern ➢ currently no uniform schema for describing pattern
Usage	requires experience in design of software systems ➢ balancing of strengths of a particular pattern ➢ selection and combination of pattern
Detection	hard to detect in source code without documentation ➢ almost no tool support
Development	difficult and tedious for pattern to be useful ➢ „over use of specific patterns“ (What is actually no pattern?) ➢ „singular patterns“ (pattern that occur only once)

Object-Oriented Design Patterns


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns of GoF

Introduction	Definition, description, and usage of design pattern
Content	➢ catalogue of 23 design pattern, each with at least two application examples
Goals	supporting means for object-oriented design “in the small” ✓ How to develop efficient and flexible mechanisms? ✓ How to find classes and operations for technical problems? ✓ How can objects interact in a meaningful way? ✓ How to implement proposed mechanisms? ✓ How to keep code readable and maintainable?

Classification of the Design Pattern

Category	Task
creational patterns	Object creation (e.g., Factory, Singleton, Prototype)
structural patterns	Composition of classes and objects (e.g., Composite, Proxy)
behavioral patterns	Managing of control flow and interactions between classes and objects, respectively

Describing a Design Pattern (proposed by GoF)

Characteristics:	
Name	
Problem	Motivation
	Application domain
Solution	Structure (class diagram)
	Elements (usually names of classes, associations, and operations): ➢ “role names", i.e., placeholder for elements of the application ➢ fixed elements of the implementation
	object interaction (workflows, maybe sequence diagrams)
Discussion	Advantages & disadvantages
	known application
	special cases
	dependencies, limitations

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

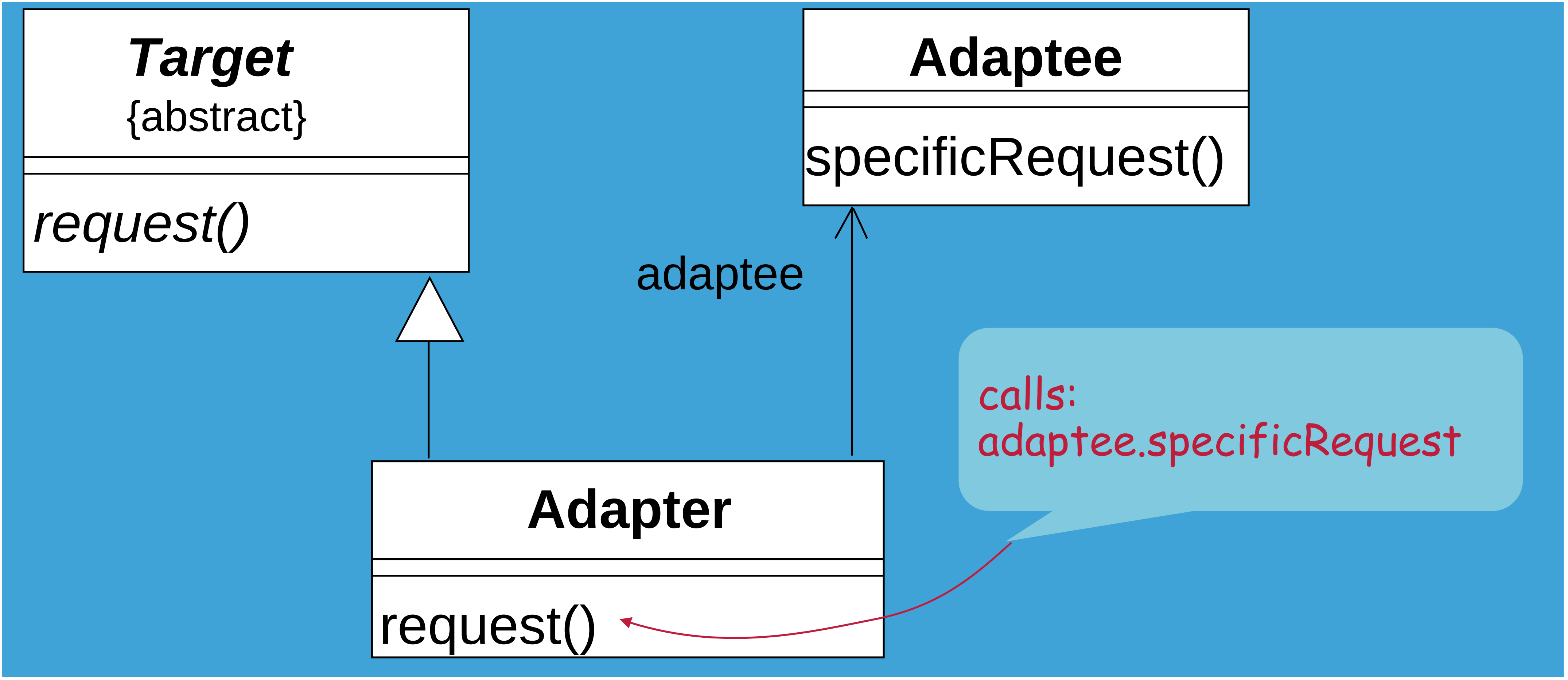
FACULTY OF COMPUTER SCIENCE

SwarmLab

Introduction to Software Engineering for Engineers L-07: Detailed Design & Design Patterns
Part 3: Exemplary Design Patterns

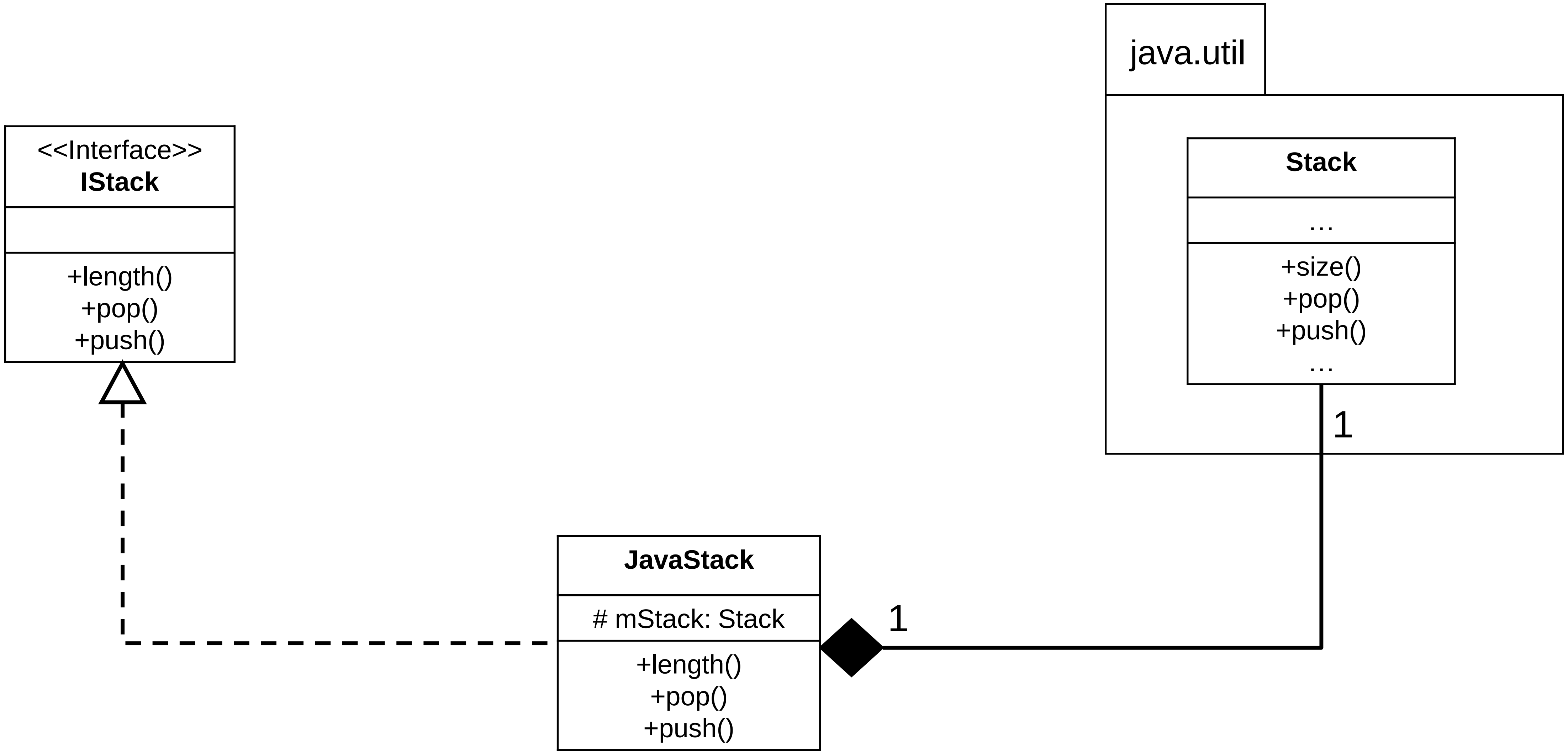
Dr.-Ing. Christoph Steup

Structural Pattern:
Adapter — Variant 1: Object Adapter

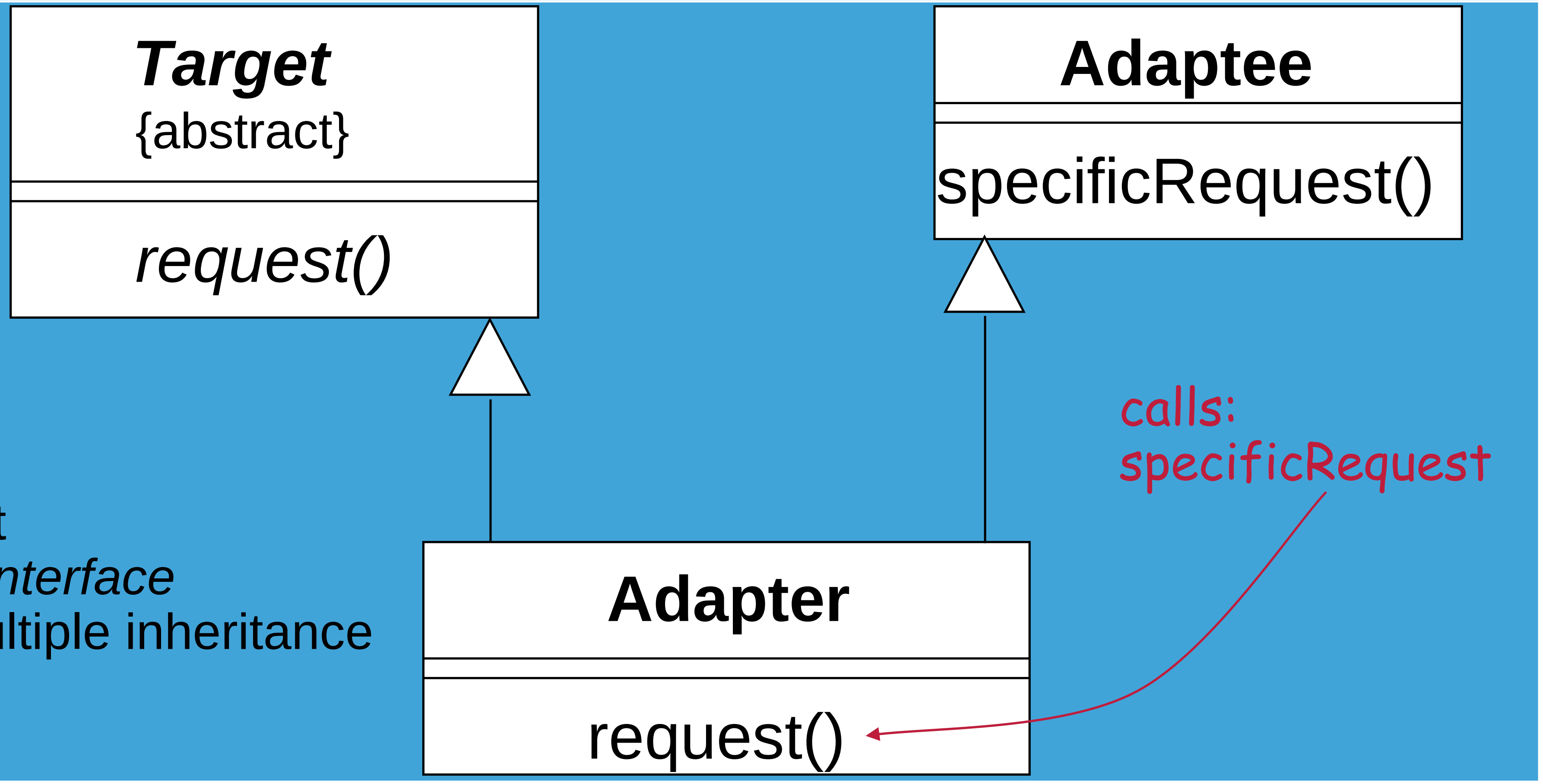
Characteristics: Object Adapter	
Name	**Adapter**
Problem	Adjusting the interface of a given object (*adaptee*) to a desired interface (*target*)
Solution	

Object Adapter — Example in UML

Example

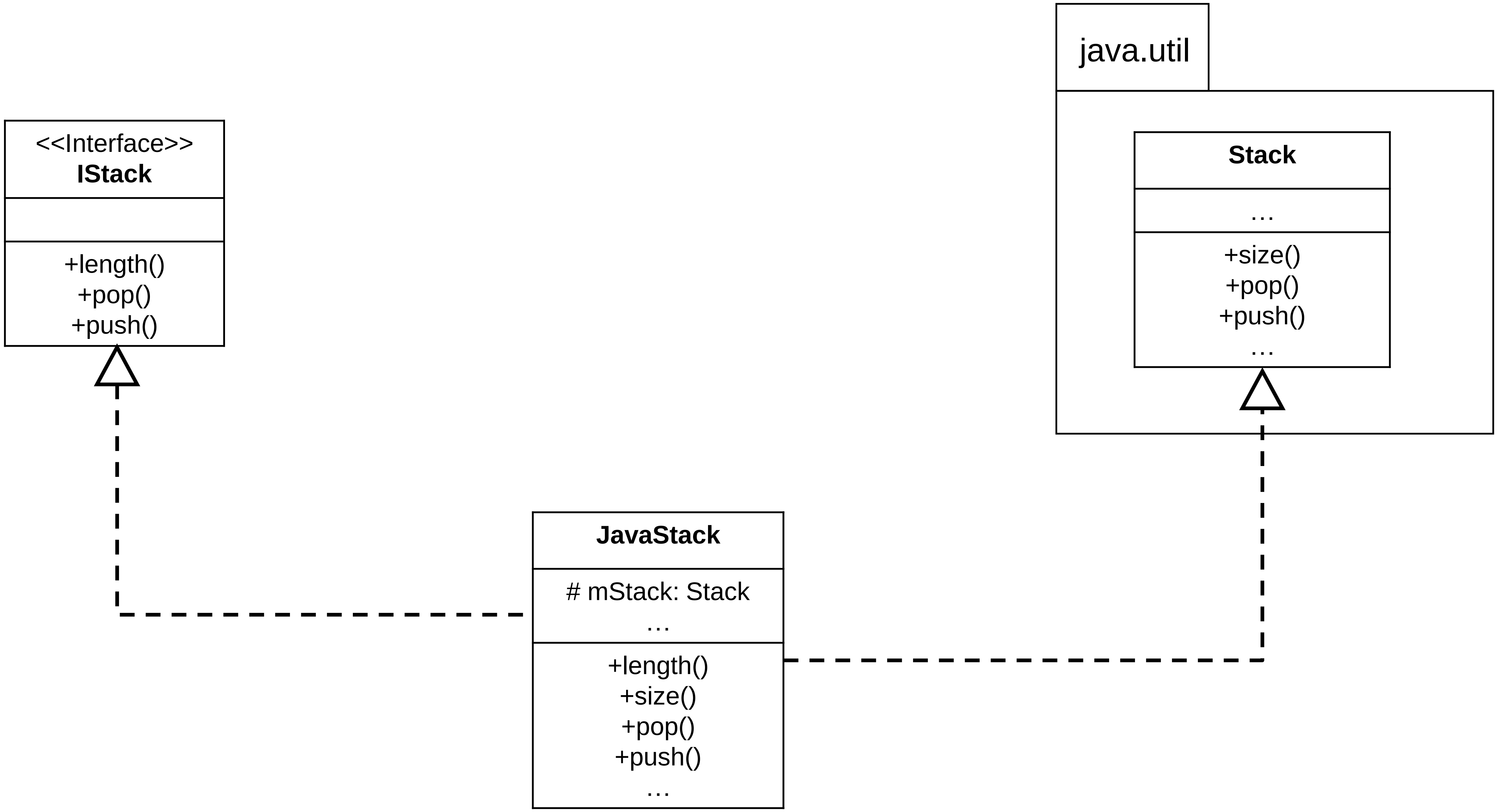


Adapter — Variant 2: Class Adapter

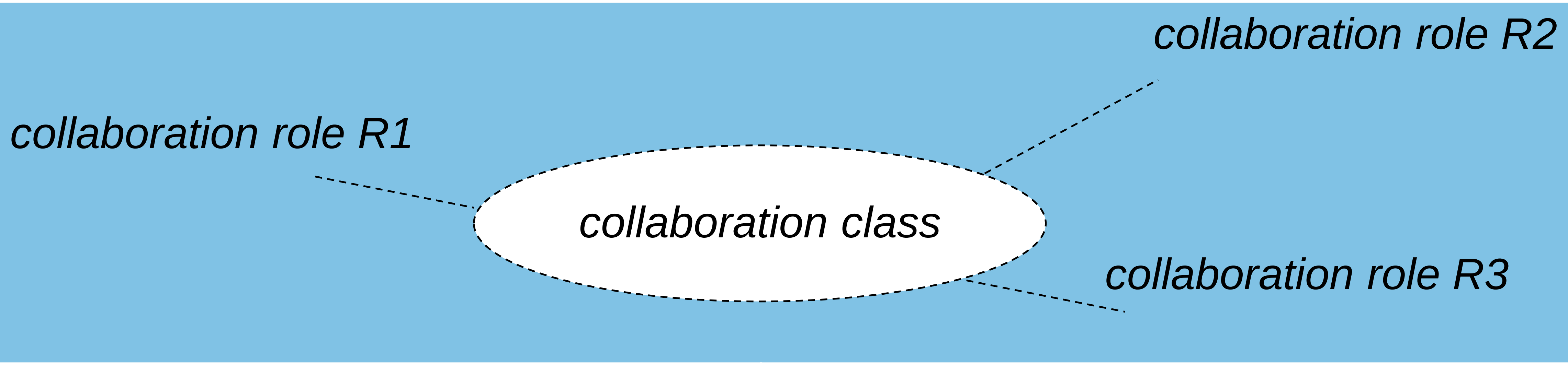
Characteristics: Class Adapter	
Name	**Adapter** (aka: Wrapper)
Problem	➢ Adjusting the interface of a given object (*adaptee*) to a desired interface (*target*) ➢ Many operations are identical in *Adaptee* and *Target*, but have different names or different order of parameters
Solution	 ***Java:*** Target must be an *interface* (since no multiple inheritance is possible)!

Object Adapter — Example in UML

Example



UML — Notation for Design Pattern



collaboration role R2

collaboration role R1

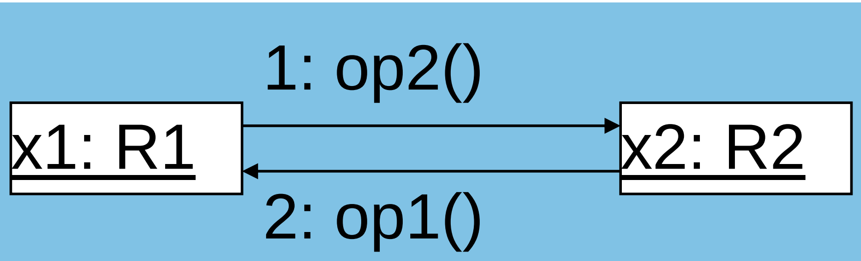
collaboration class

collaboration role R3

oval as pattern description

collaboration class = pattern name,
roles = names of placeholders

process of collaboration described by collaboration diagram (equivalent to sequence diagram):

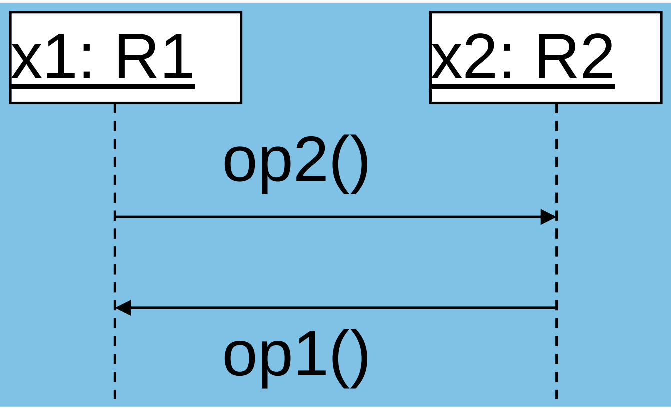


1: op2()

x1: R1

x2: R2

2: op1()



x1: R1

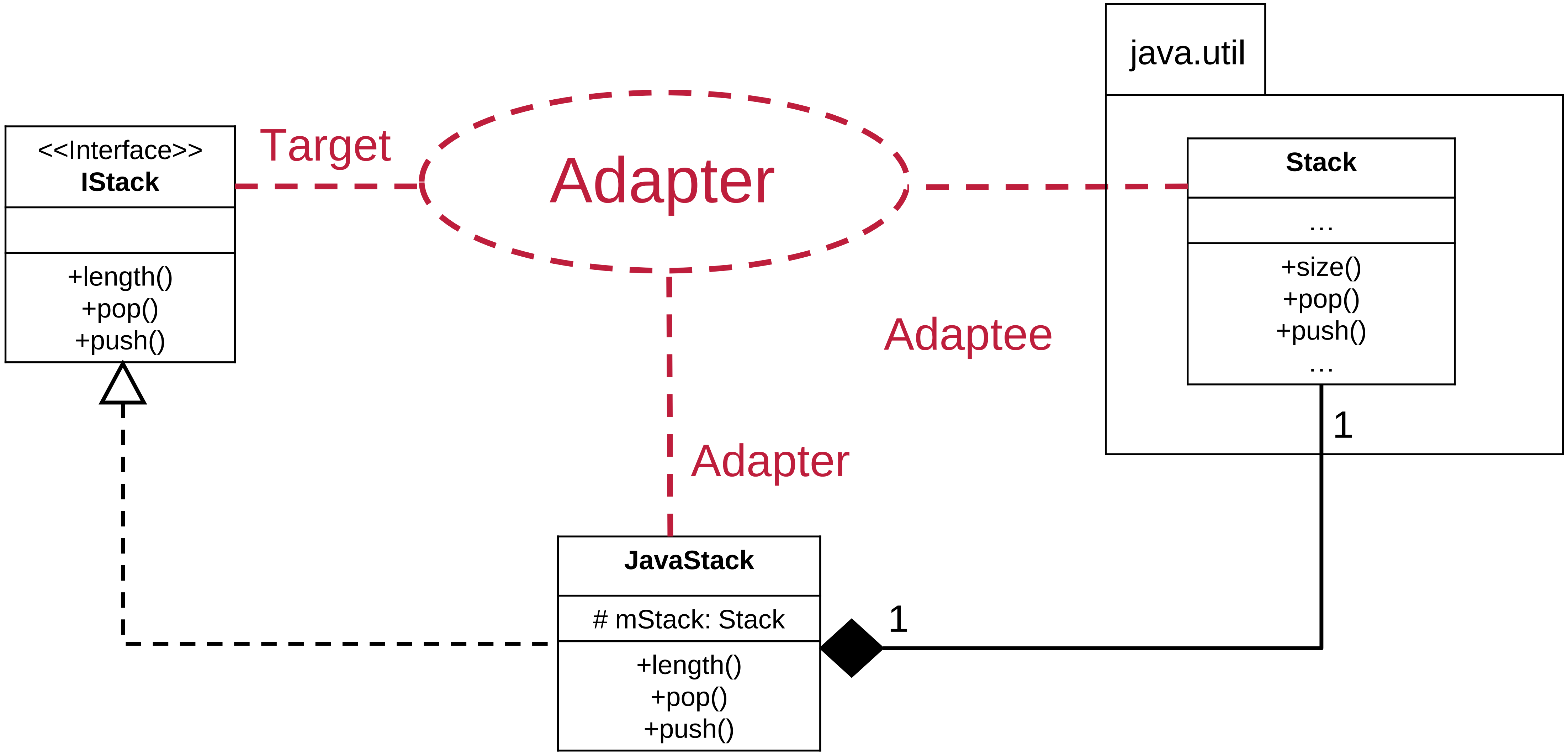
x2: R2

op2()

op1()

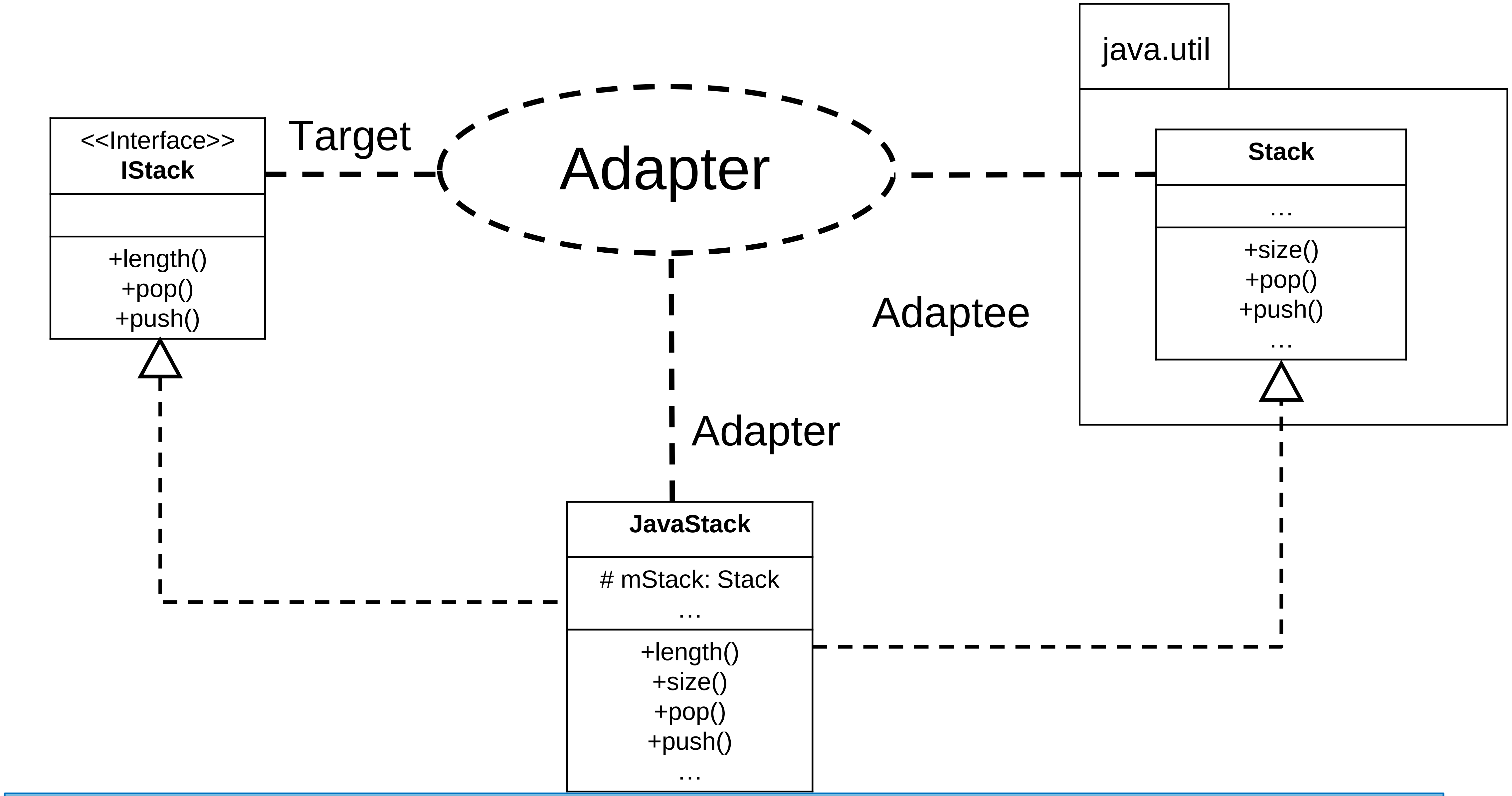
Object Adapter — Example in UML

Example



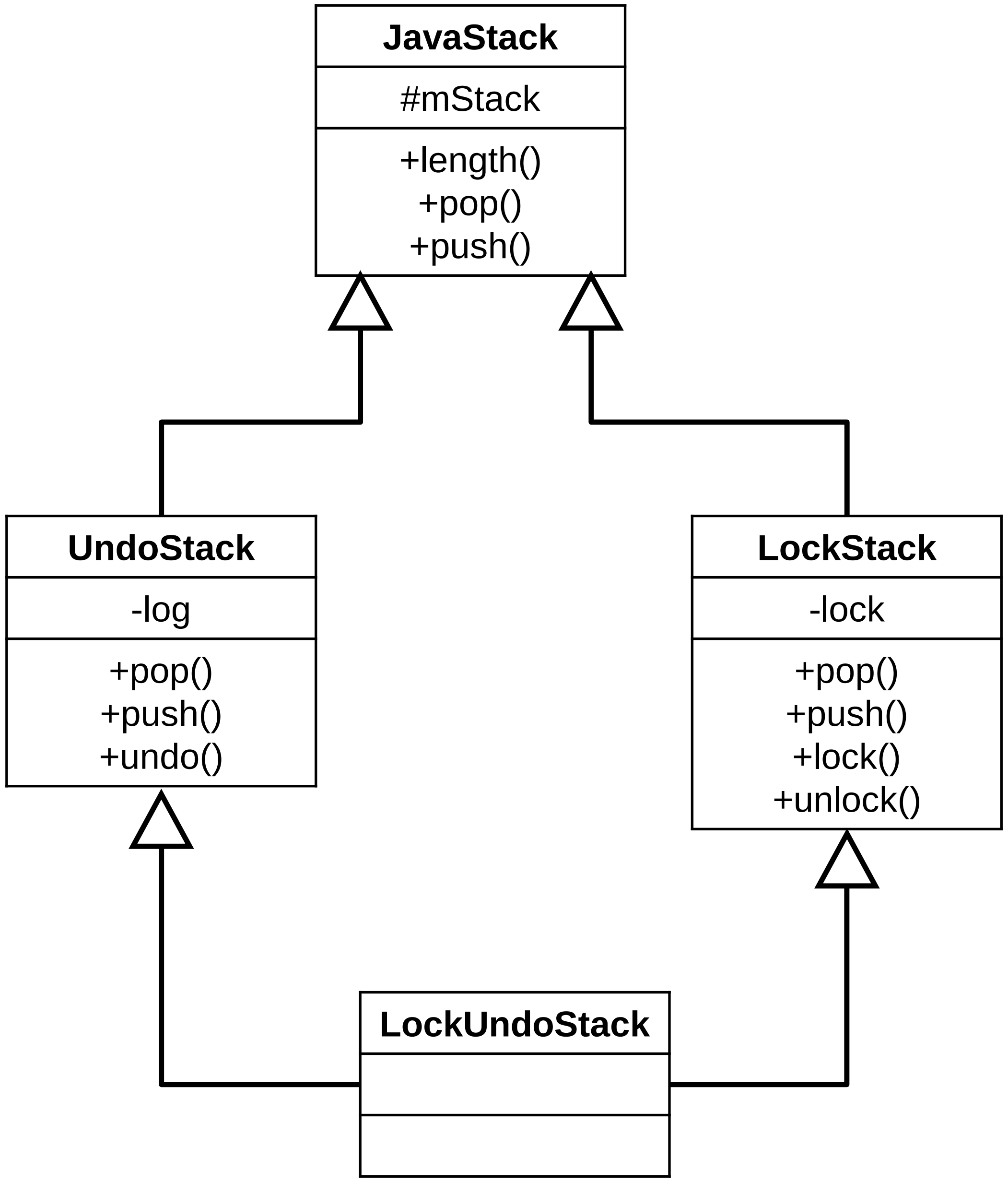
Object Adapter — Example in UML

Example



Caution:
All operations of the Adaptees are inherited by the Adapter, including those that are possibly undesired!

Decorator Pattern — Why is it needed?



Diamond Problem !

What happens?

new LockedUndoStack().length()

Answer:

- You don‘t know
- Multiple length*()* operations exist
- One in **LockedStack**
- One in **UndoStack**

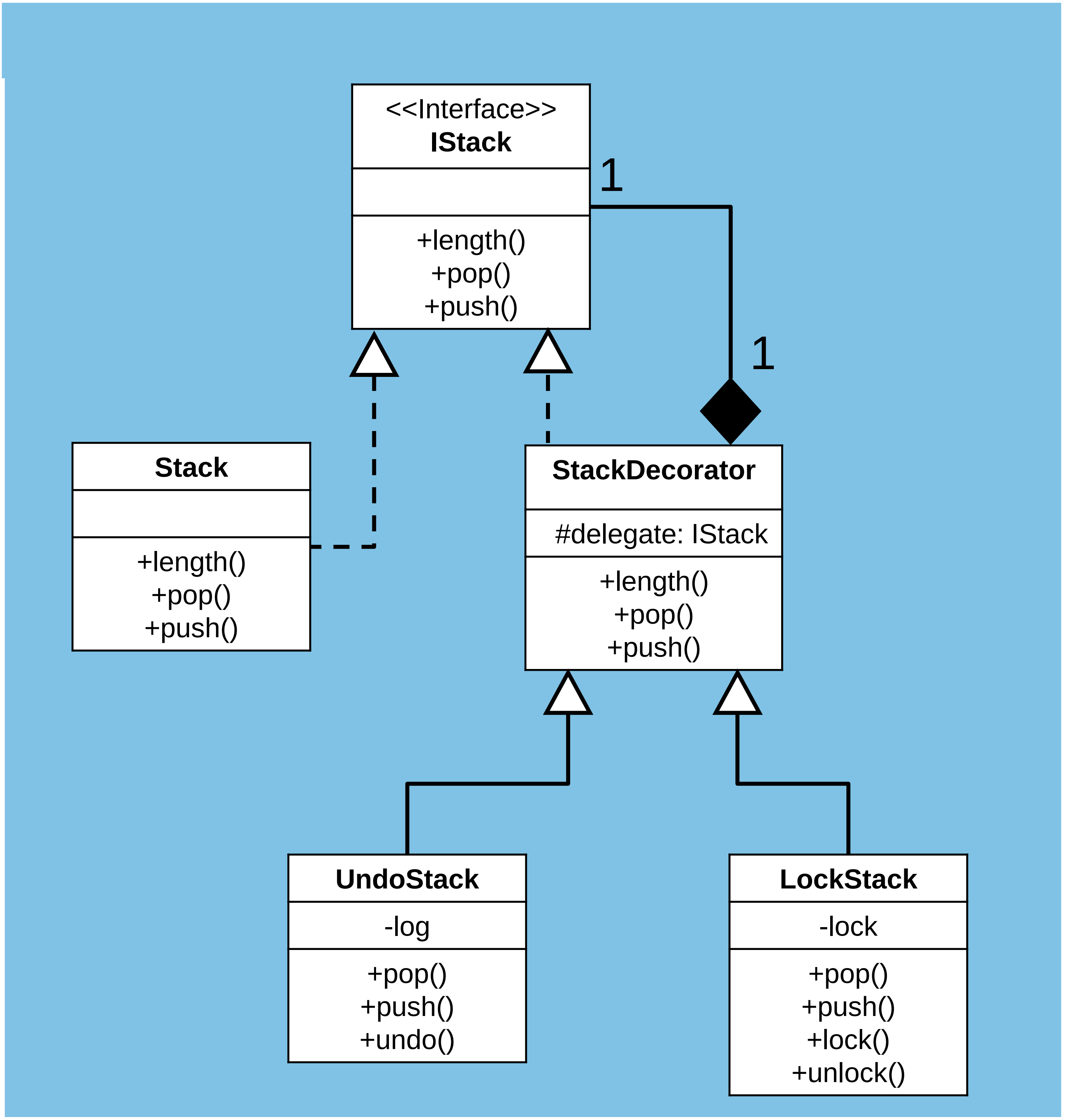
Solutions:

- **C++:** Virtual Inheritance
- **Java, C++: Compile Error**
- **Python3:** Default Dispatch Order

This doesn‘t work in Java

Structural Pattern: Decorator

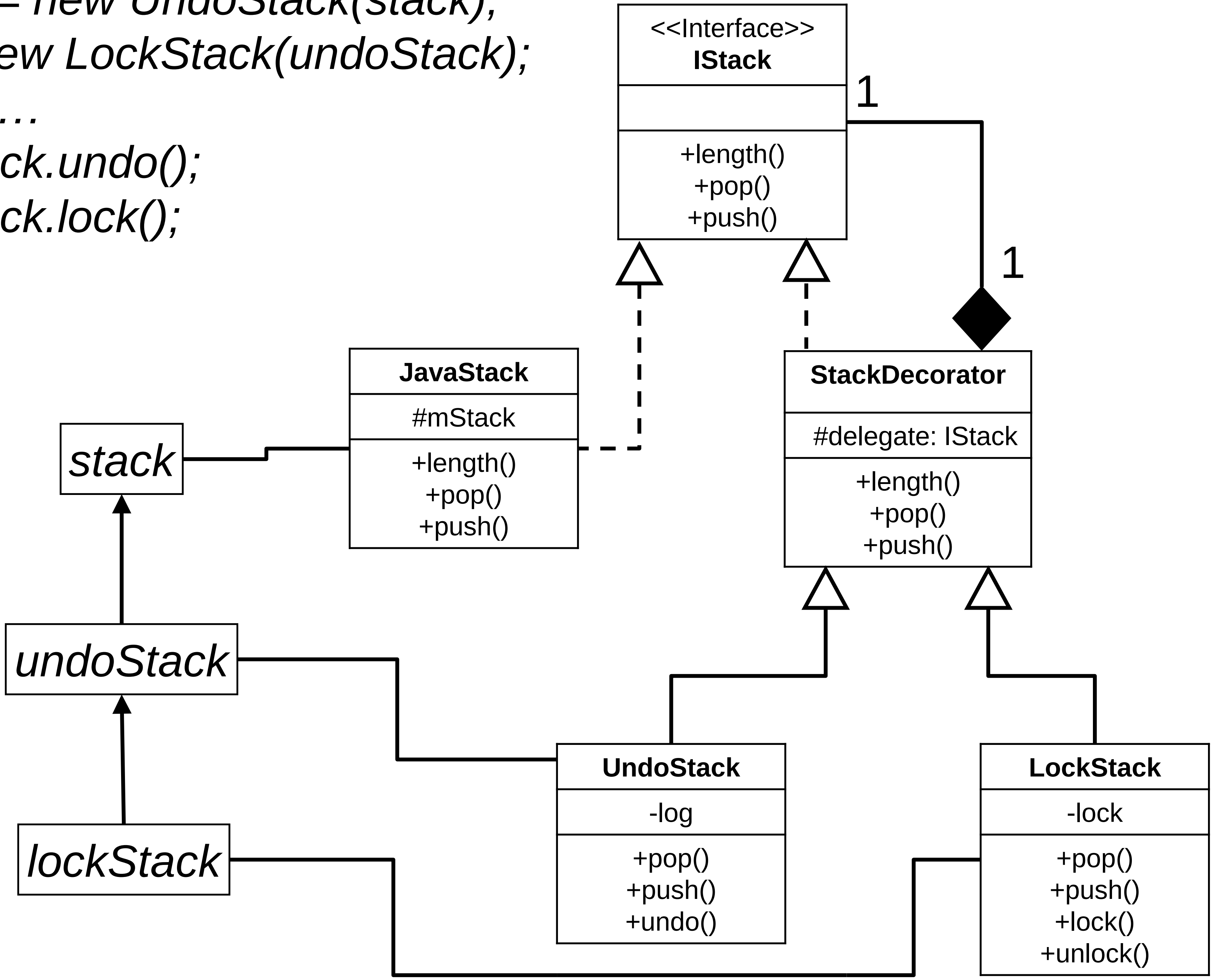
Characteristics	
Name	**Decorator**
Problem	Add additional properties and/or operations to existing class. Goal: flexibility Constraint: Class interface shall stay the same
Solution	Definition of a helper class for intermediate objects. Delegation of untouched operations to original class. Encapsulation of new functionalty in Subclasses of abstract class Decoration is done on Object creation



Structural Pattern: Decorator

Example

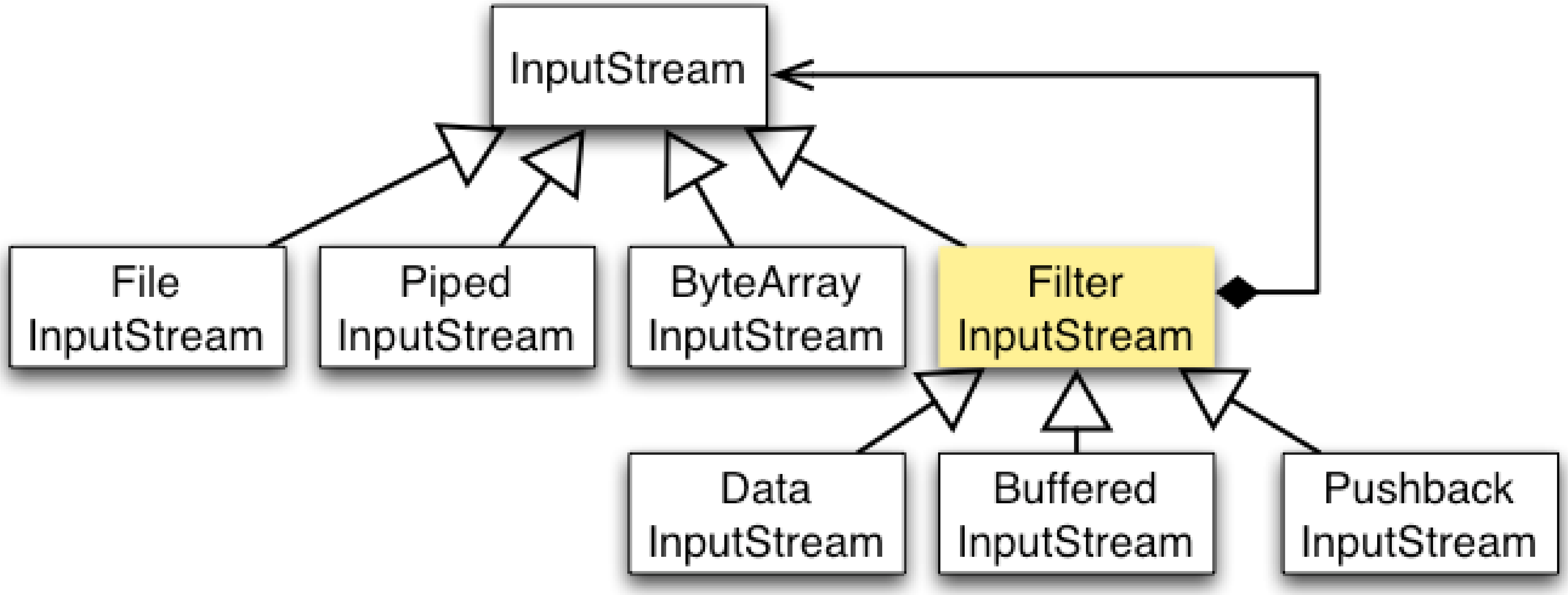
JavaStack stack = new JavaStack();
UndoStack undoStack = new UndoStack(stack);
LockStack lockStack = new LockStack(undoStack);
…
undoStack.undo();
lockStack.lock();



Caution:
In Contrast to inheritance the various decorators need to be kept to use the additional operations

Decorator: Decorator in java.io

Example

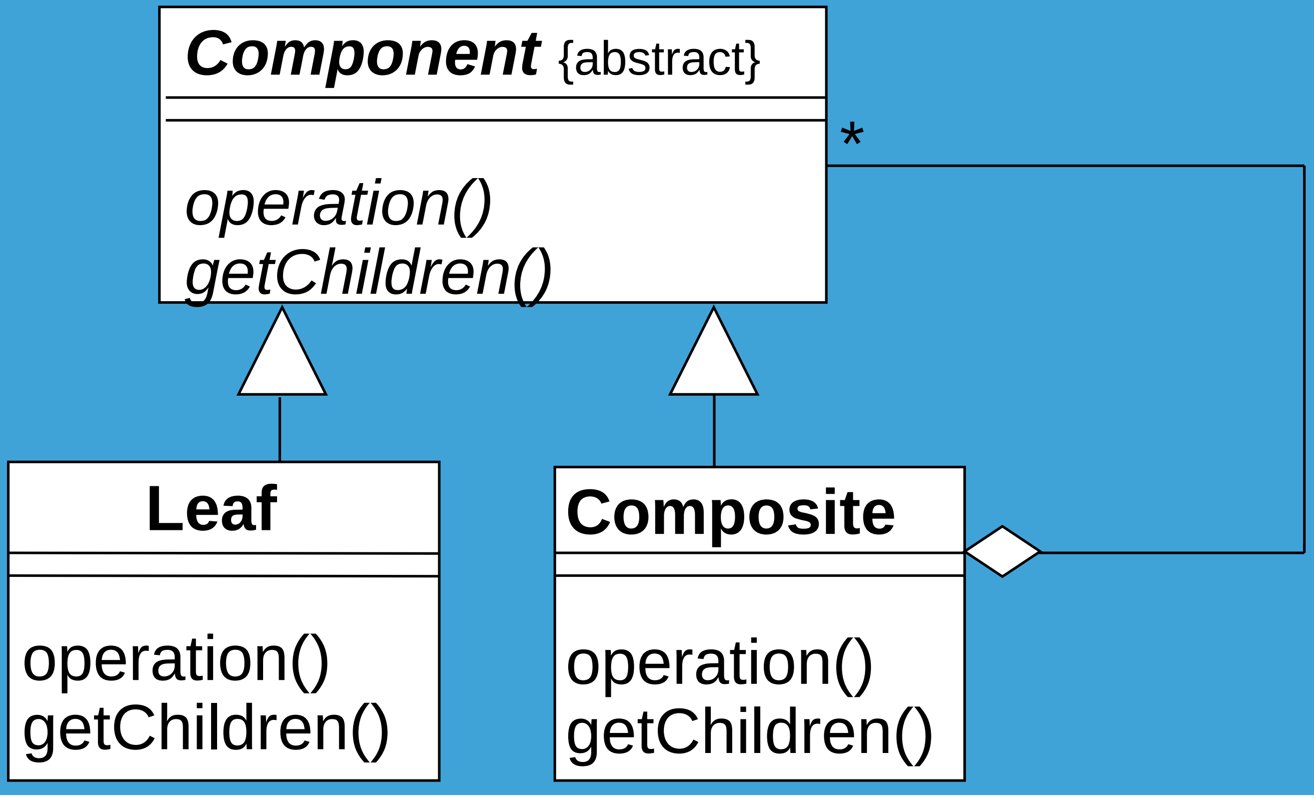


java.io contains different functions for input/output:

- Programs **operate on stream objects** ...
- **Independent** data source/target and kind of data

Structural Pattern: Composite

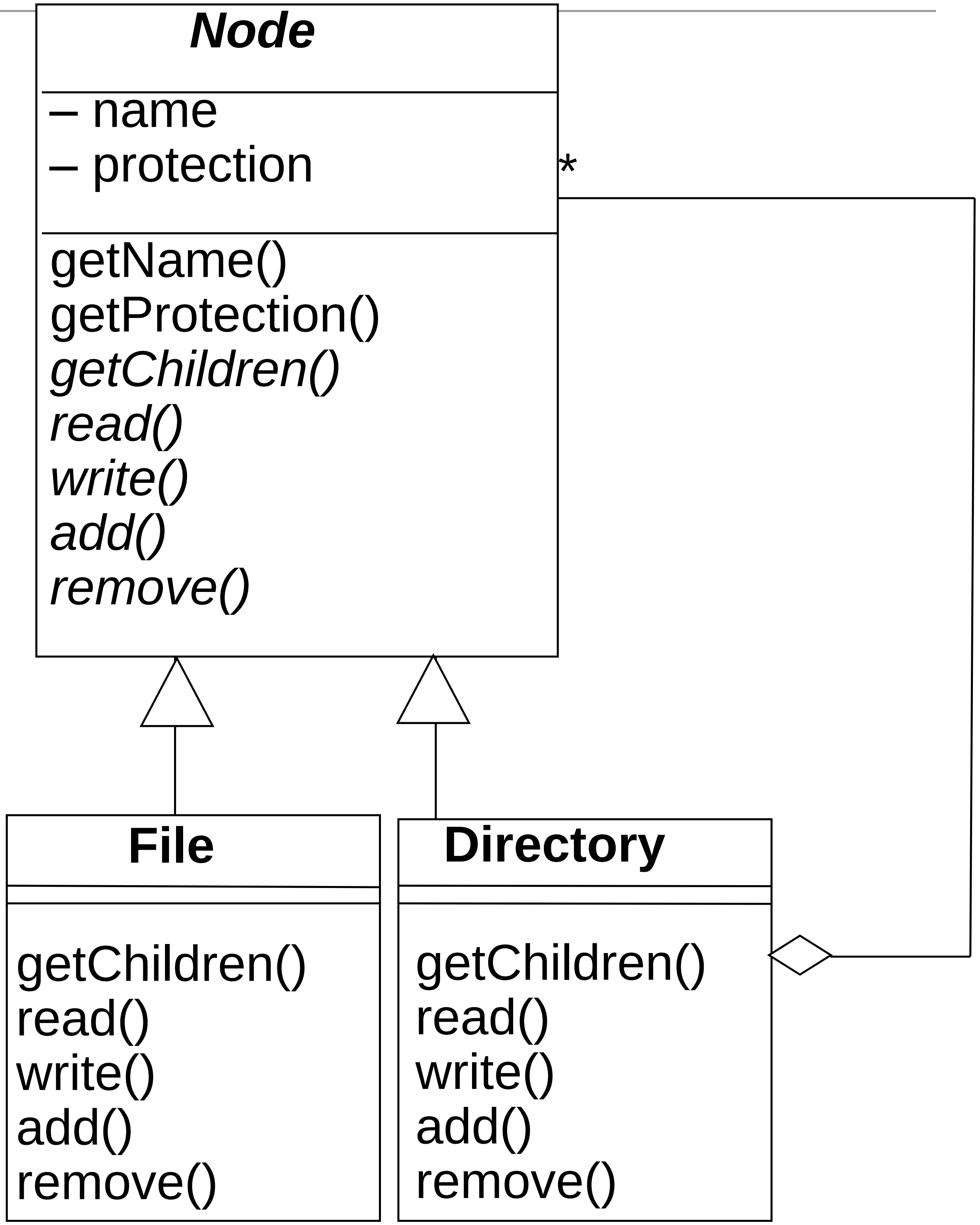
Characteristics	
Name	**Composite**
Problem	Hierarchical structure of objects

Solution	Consistent, abstract interface for „leaves“ branching and nodes in a tree
	
	 Component {abstract}
	operation()
	getChildren()
	*
	Leaf
	operation()
	getChildren()
	Composite
	operation()
	getChildren()

Application of Composite Pattern

Example

- ➢ flexible access layer for file systems
  - ▪ Common operations on files and directories:
  - ▪ name, size , access rights, ...
- ➢ part structure for devices
- ➢ genealogical tables (trees...)

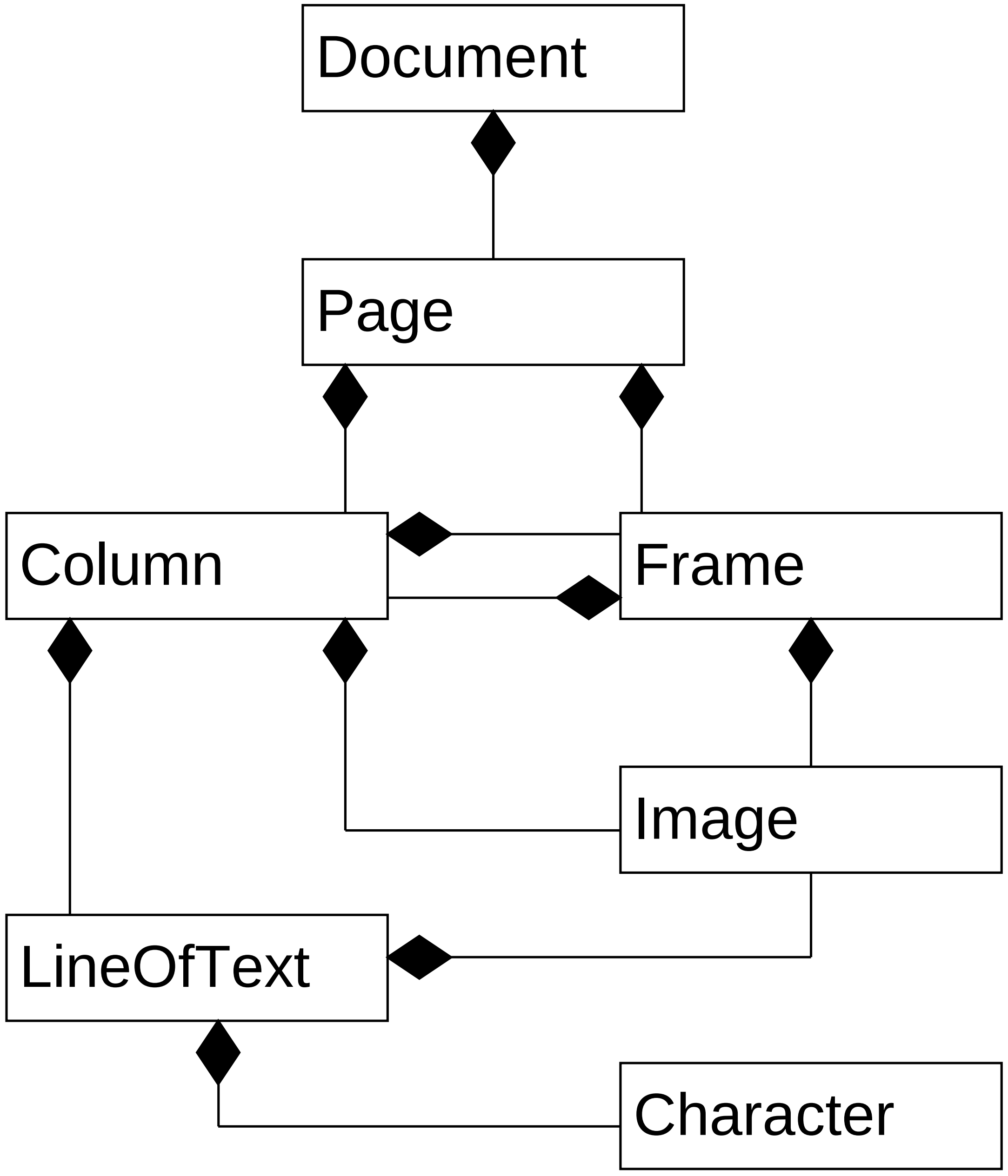


Composite - Detailed Example

Example

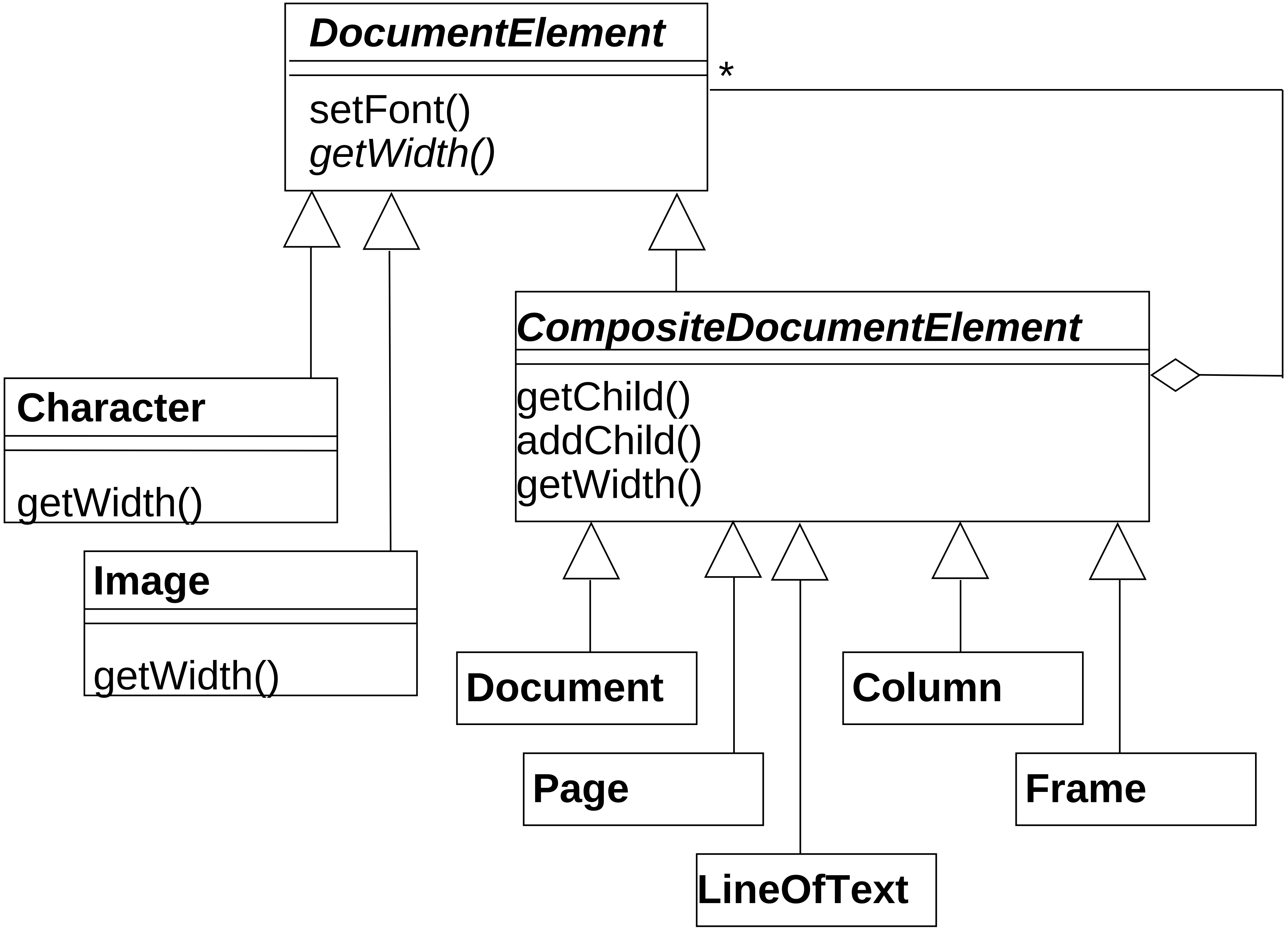
- Task: document structure and formatting

- Initial dass diagram (from analysis)



Application of Composite Pattern

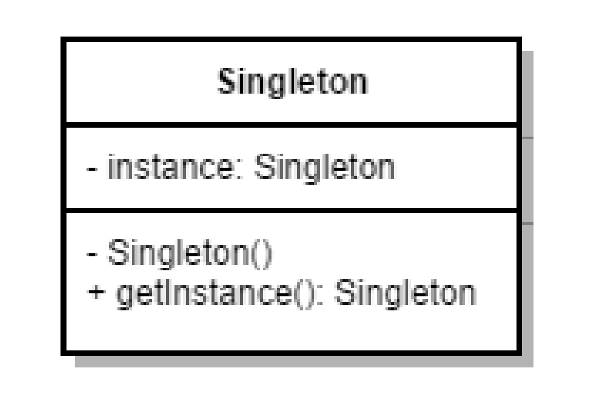
Example



Creational Pattern: Singleton

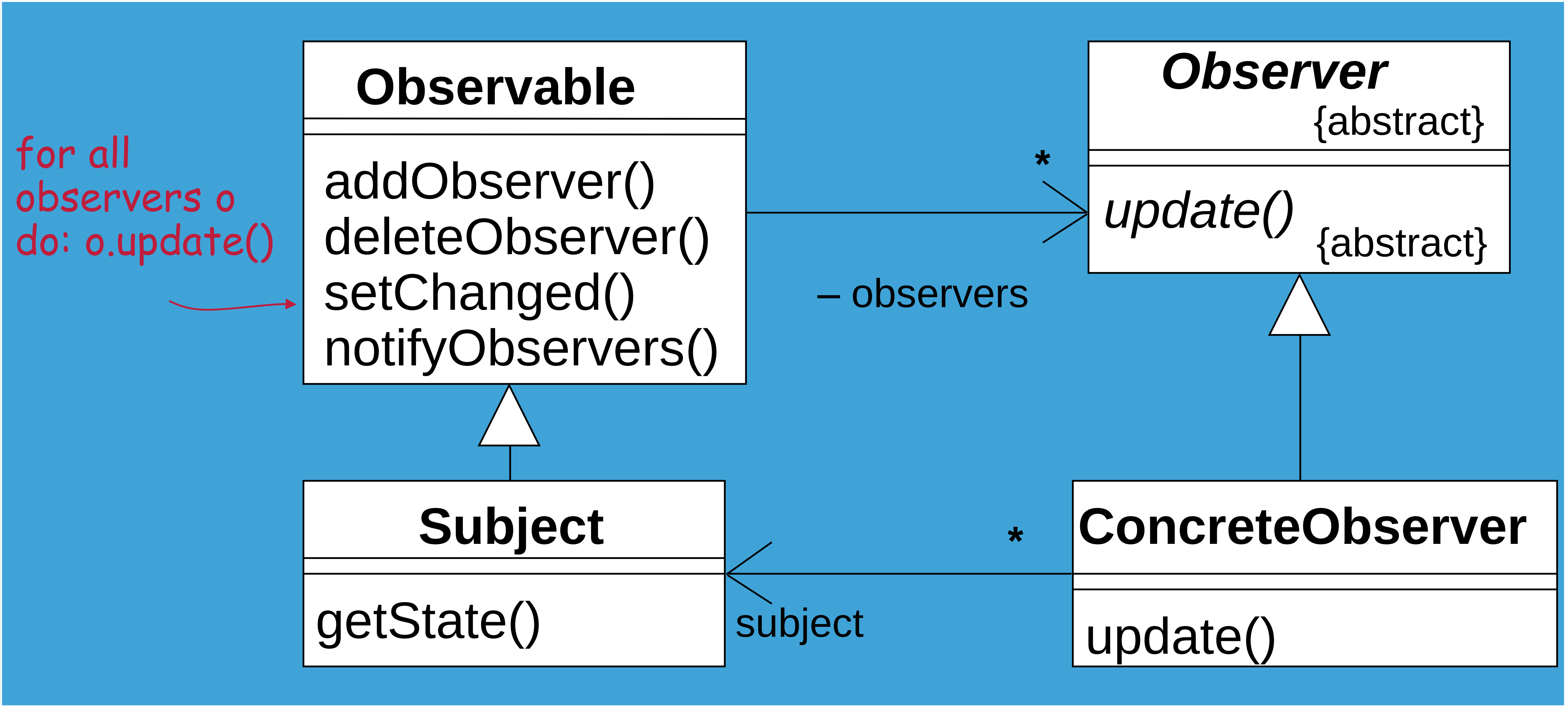
Characteristics	
Name	**Singleton**
Problem	Some classes only meaningful when it is guaranteed that at most one instance of this class exists (which can be created on demand).
Solution	model level: declare classes as Singleton program level: language-dependent

```
class Singleton {
  private static Singleton theInstance;
  private Singleton () {
  }
 public static Singleton getInstance() {
    if (theInstance==null)
      theInstance = new Singleton();
    return theInstance;
  }
}
```



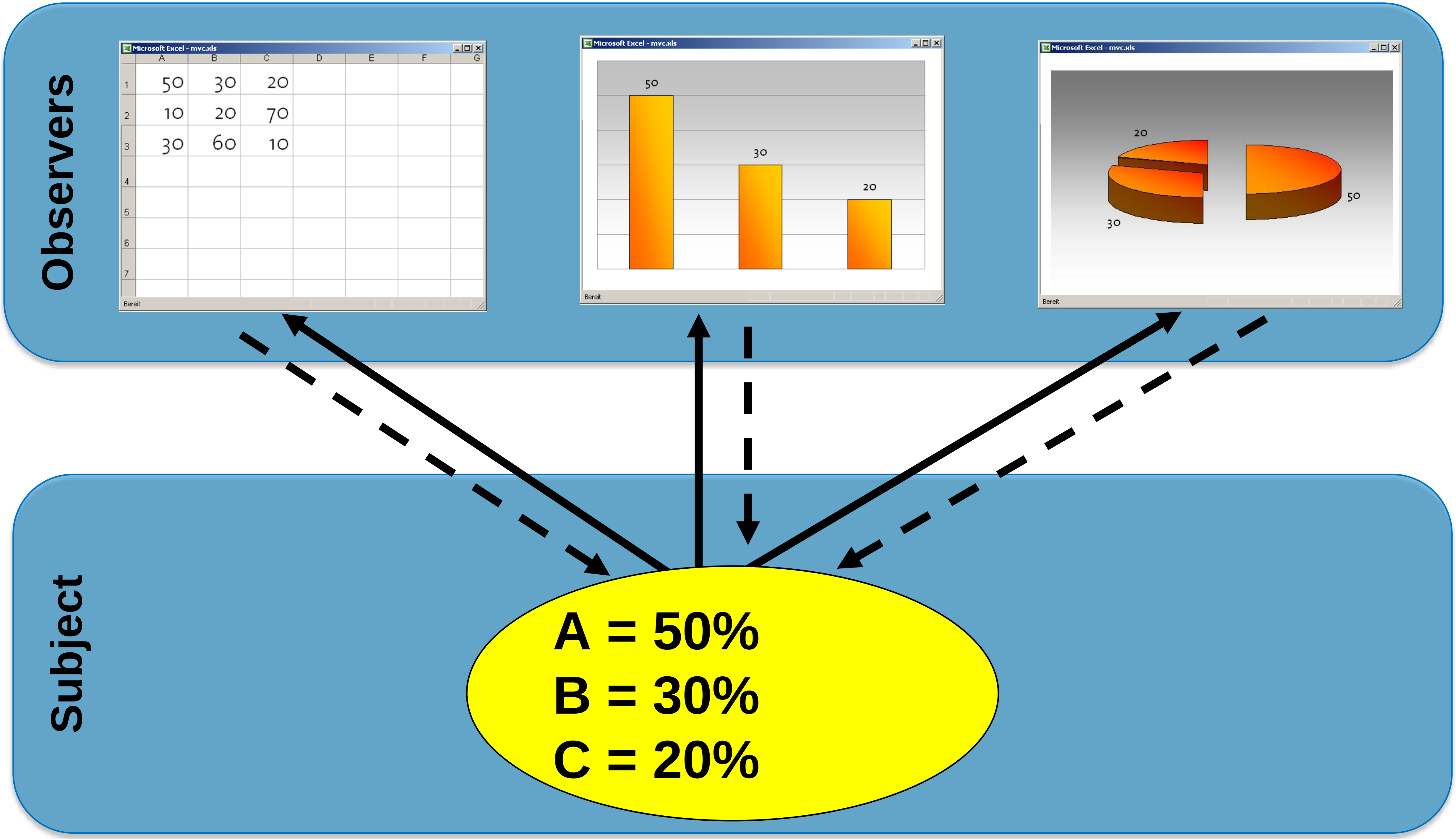
This implementation is not thread-safe

Behavioral Pattern: Observer

Characteristics	
Name	**Observer**
Problem	Multiple objects are interested in particular behavioral changes of another object
Solution	

Application of Observer Pattern

Example



Observers

	A	B	C	D	E	F	G
1	50	30	20				
2	10	20	70				
3	30	60	10				
4							
5							
6							
7							

Microsoft Excel - mvc.xls
Bereit
50
30
20
Microsoft Excel - mvc.xls
Bereit
20
50
30
Microsoft Excel - mvc.xls
Bereit

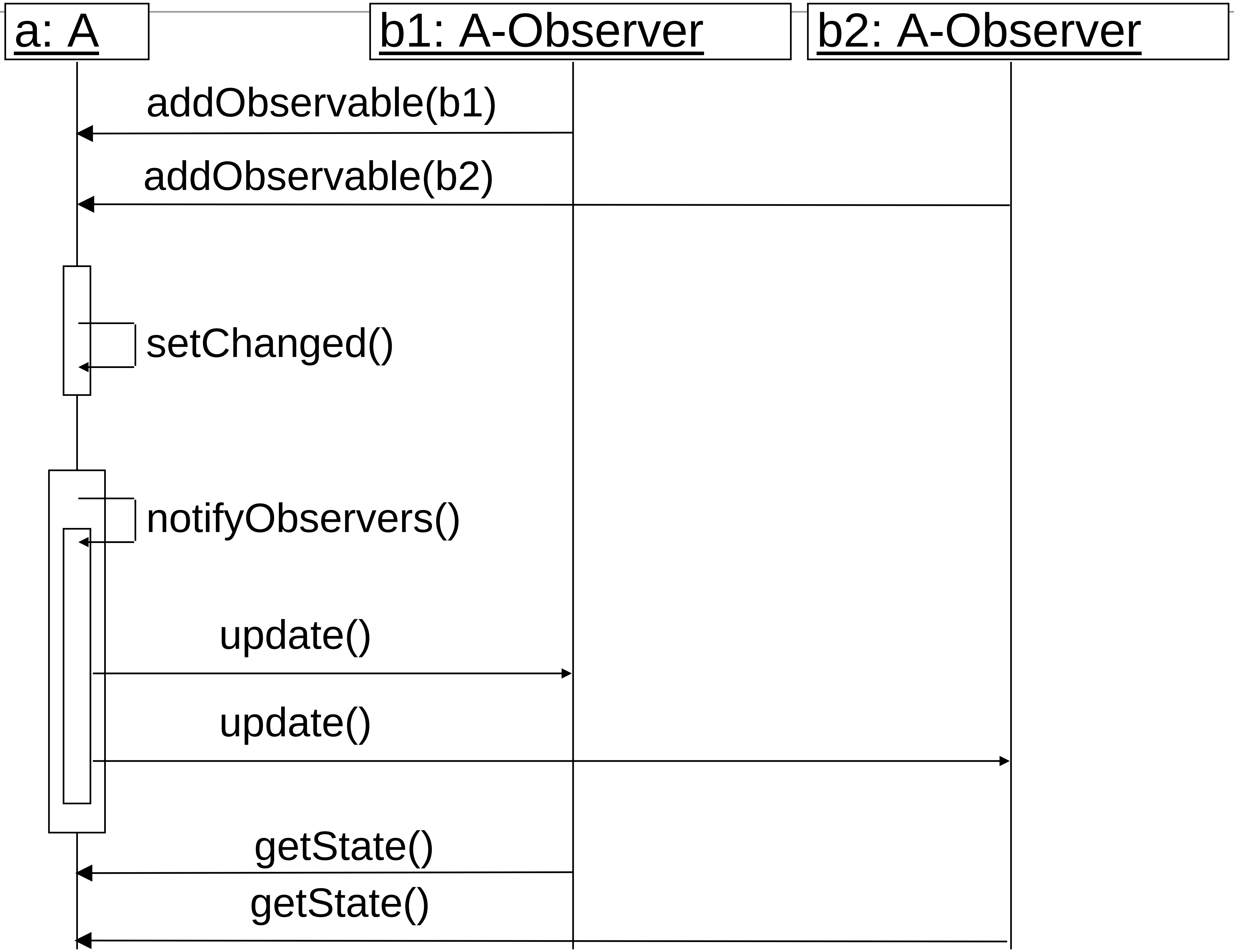
Subject

A = 50%
B = 30%
C = 20%

[Quelle: Meyer/Bay]

Exemplary Workflow of Observer

Example

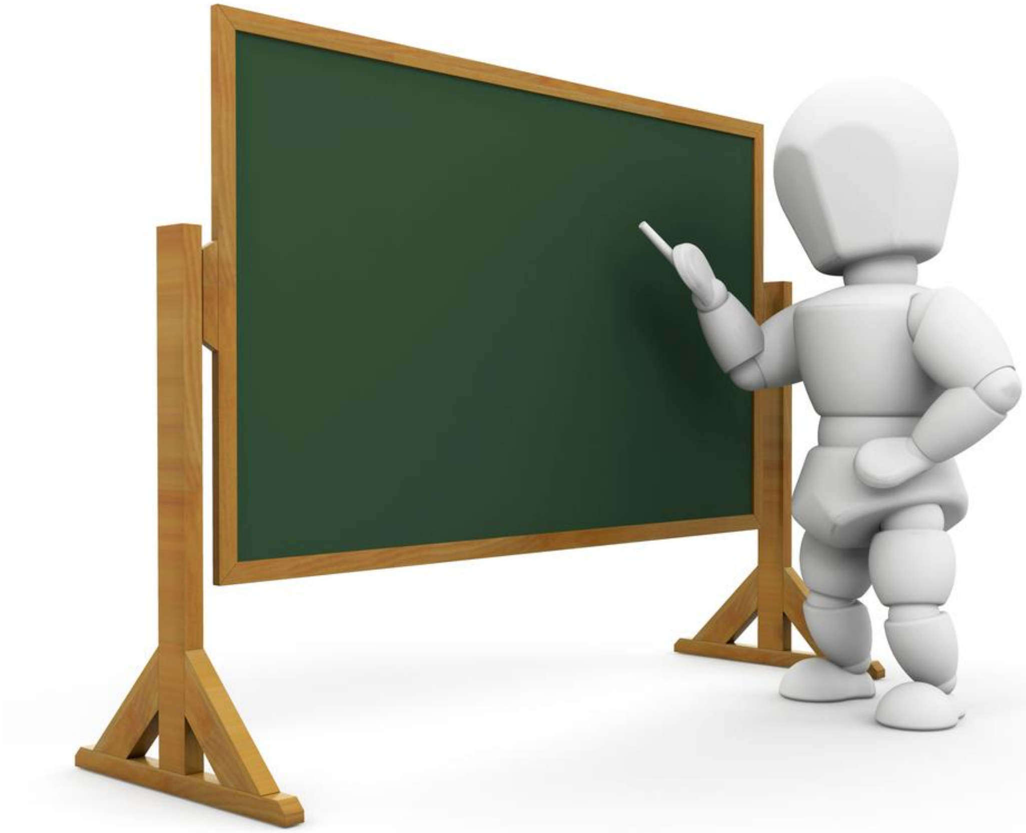


Summary: Design Patterns

- A couple of further design pattern has been developed in recent years, partially to manage (domain)-specific problems

- Design pattern are a useful mean for:
  - Improving the structure of the source code
  - communicating about design decisions

Note: „pattern“ are not realized directly,
but adapted to the requirements:
pattern are rather templates for design ideas.



Excurse: Anti-Patterns

- Patterns describing generic misbehavior
- In software design
- In programming
- In project management
- Often more important to prevent anti-patterns then to follow design patterns

https://en.wikipedia.org/wiki/Anti-pattern

Summary

➢ Refinement of analysis models in the design phase

- UML for detailed design
- Packages and Visibility
- Refinement of associations
- Refinement of classes and methods
- interface specification

➢ Designpatterns by example

- Structural patterns: Adapter, Decorator, Composite
- Creational patterns: Factory, Singleton
- Behavioral patterns: Observer