Software Engineering

Lesson 06
Object Oriented Design
v1.0

Uwe Gühl

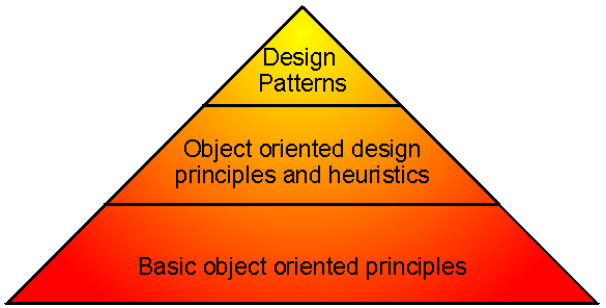
Fall 2007/ 2008

Contents

- Introduction
- Architecture
- Components
  - Introduction
  - Definitions
  - Characteristics
  - UML Diagram
  - Interfaces
  - Example
  - Proceeding
- Interfaces

Contents

- Good Object Oriented Design
  - Basic OO principles to modeling
  - OO Design Principles and Heuristics
  - Design Pattern
- Transition from OOA to OOD
- Good Object Oriented Code
- Sources



OOD
Introduction

- Design means
  to develop a solution
  for a given problem
  in consideration of given surrounding conditions

OOD Introduction

- Goal of Object Oriented Design
  - "The goal of [object-oriented design] is to manage dependencies within a program. It achieves this goal by dividing the program into chunks of manageable size, and the hiding those chunks behind interfaces..." (Robert C. Martin).
  - A major goal of object-oriented design is maximizing reusability of classes and methods [AR00]
  - The main goal of Object Oriented Design is to decompose the system into modules, that is identifying the software architecture so that it should maximize the cohesion and minimize the coupling [She05]
  - The goal of object-oriented design is to develop an object model of a system to implement the identified requirements [RV04], [Mol05]

OOD Introduction

- Questions
  - Now I know about Object Oriented Design – so maybe the OOA Model is not sufficient
  - Why do we need all these models in Software Development?
  - What's the difference between an OOA and OOD model? Must they be separated?

OOD Introduction

- Discussion
  - A model is always wrong, some are helpful
  - A model is not identical with the subject
  - A model is something like a statement about its subject, focusing on a specific aspect disregarding other aspects
  - Multiple statements about a subject can be combined in one model

OOD Introduction

- Discussion
  - Sometimes one model is not sufficient to integrate all necessary statements about a subject – more models are necessary
  - Example out of physics: Wave-particle dualism
    - 1803 Thomas Young showed in double-slit experiments that light behaves as waves
    - The photoelectric effect proves that light exists of particles

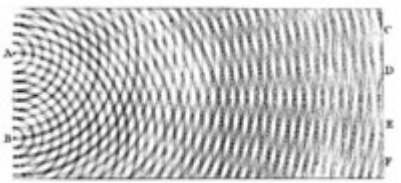
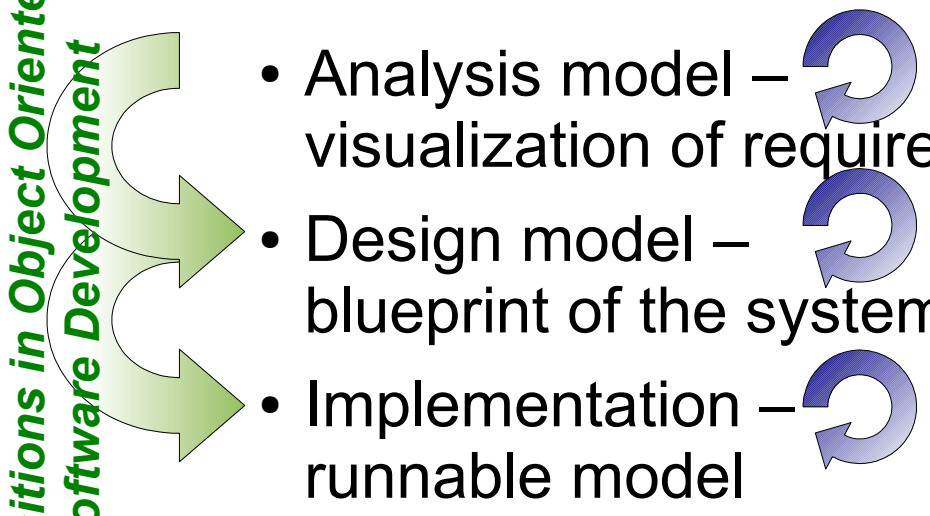


Image source: http://en.wikipedia.org/wiki/Wave-particle_duality

OOD Introduction

- Discussion concerning OOA, OOD, and OOP
  - Even if the notation is integrated, different models have different intentions



  - All models should be iterated, there is a dependency as well, but they should exist parallel

OOD
Introduction

- Discussion concerning OOA, OOD, and OOP
  - The transition from one model to another means creative mental effort
  - This creative effort is part of the development process and – typically – can not be automated
  - That's why treat “Roundtrip Engineering” carefully – attention with tools arguing code visualization means object oriented modelling

OOD
Introduction

- Process oriented aspects
  - Not everything is an object: There is behaviour, that can not be assigned to “real world entities”
  - Example: Usually a sort algorithm needs a behaviour located outside of the objects to be sorted
  - Customers or contracts are entities of the real world and could be modeled as objects with states and behaviour
  - So, a complete design must cover functional objects and their relationships and process oriented aspects

OOD
Introduction

Proceeding – Proposal

- Typical steps in a software design
  - Define the application architecture
  - Structure contents to components
  - Develop components
  - Develop the collaboration of the components
  - Define the interfaces

OOD Architecture

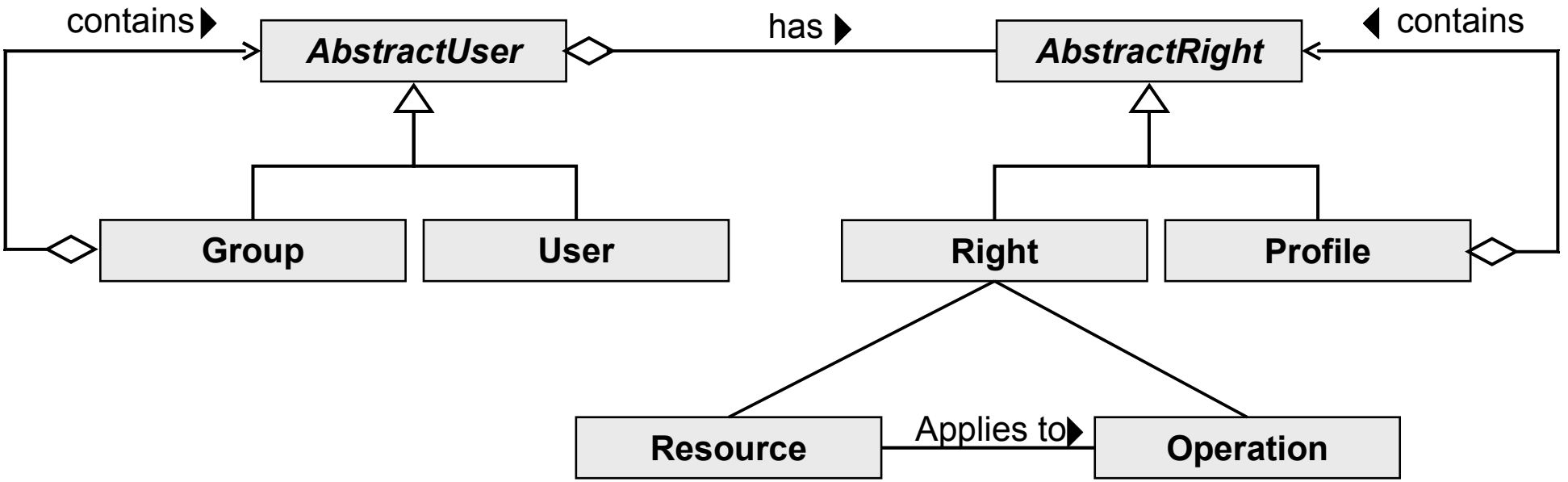
- Definition of the principle structure of a program
- Identification of layers, typical layers are
  - Communication or presentation layer (e. g. Java applets, HTML interface, IBM 3270 display terminal, Java GUI, and so on)
  - Application logic (e. g. business logic in Java applications, applets, or on a CORBA or J2EE application server, Web-Services, etc.)
  - Data management (e. g. relational database with object relational mapping, access on a database via JDBC, file system, etc.)

OOD Architecture

- Definition of interfaces and communication protocols
- An application architecture is often already part of the general constraints of a project
- Additionally to basic decisions – like application logic into the application server or into the client – frameworks often influence further details of the architecture

OOD
Components – Introduction

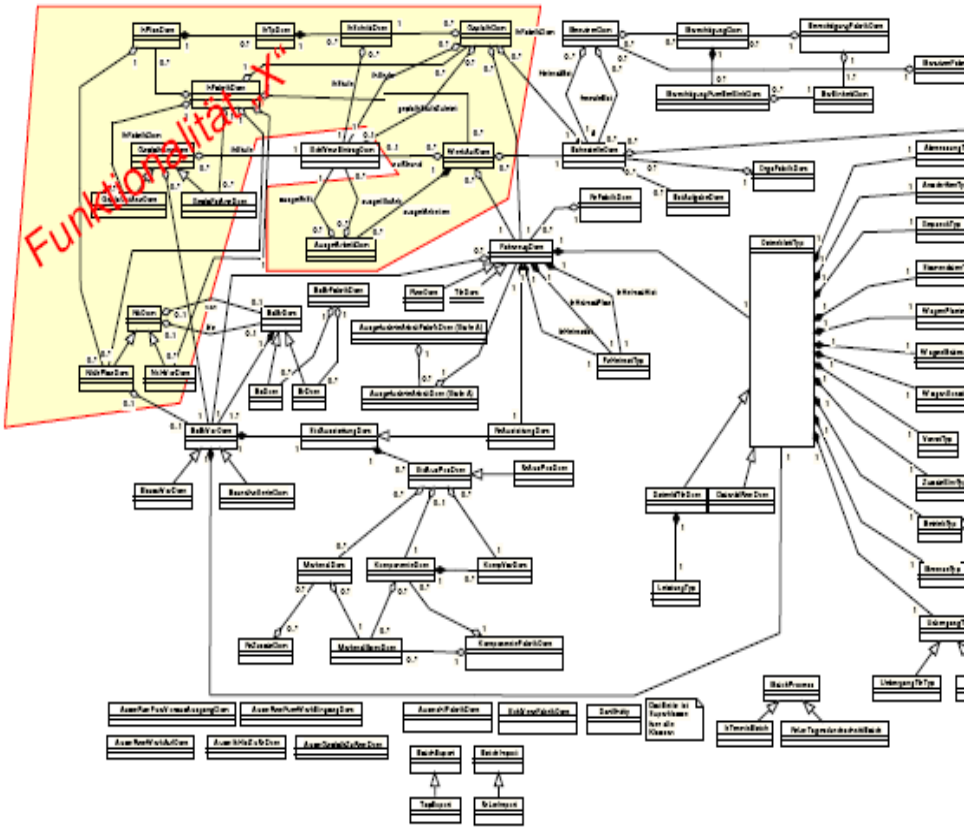
Working example: Authorization



- Class model is alright for specified problem
- Model works fine in small applications, but scaling problems expected for bigger ones

OOD
Components – Introduction

Limits of
object orientation



- Class as structuring element is too small
- Application concepts and function groups are difficult to find

Image source: Volker Wurst, www.ba-stuttgart.de/~vwurst, 6_komponenten.pdf

OOD
Components – Introduction

- The Taligent Project was one of the biggest C++ project in the 1990's
- Dependency graph when the project was stopped

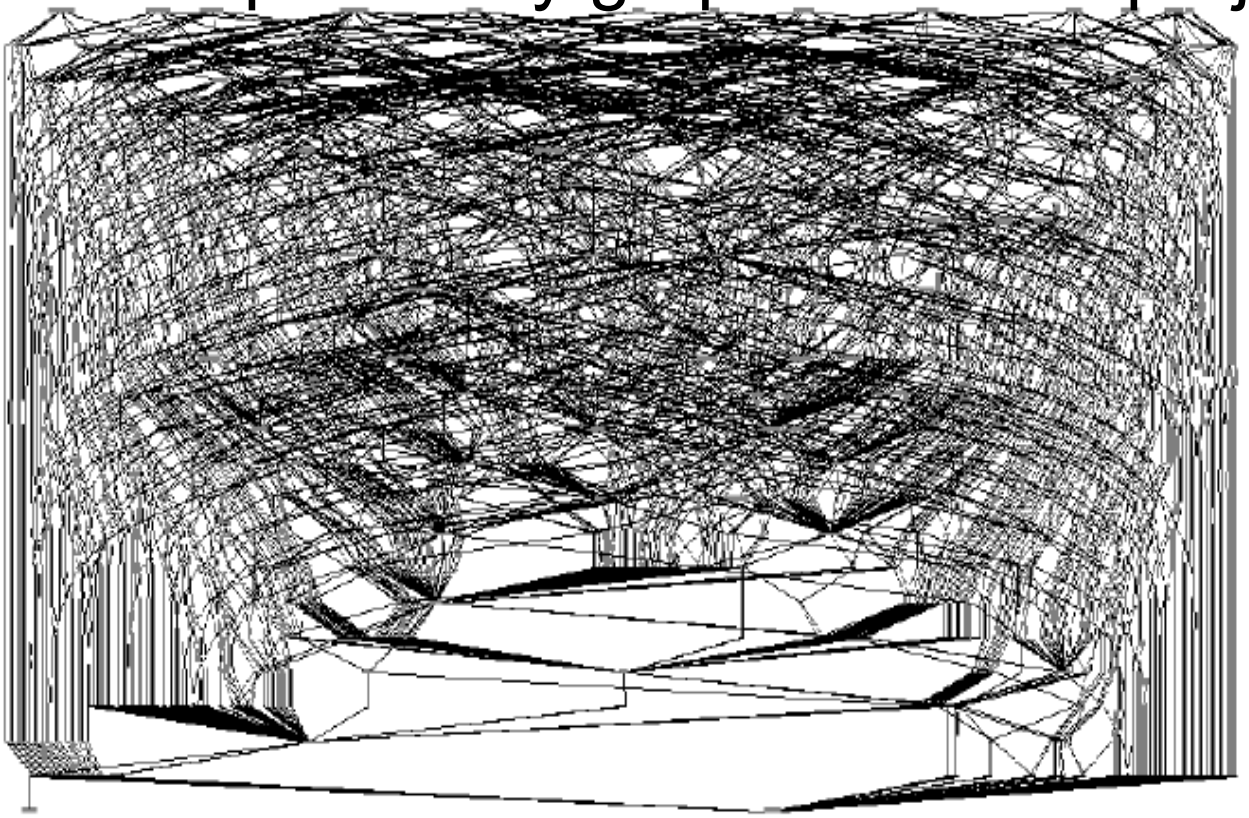


Image source: Volker Wurst, www.ba-stuttgart.de/~vwurst, 6_komponenten.pdf

OOD
Components – Introduction

- Meanwhile exists an own approach Component-based software engineering (CBSE) that focus on software reuse
- Summarized components are more abstract than object classes and can be understood as independent service providers

OOD
Components – Definitions

- A component (latin componere = to put something together) is part of a system or may serve as a part of a system
- "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." (Szyperski, ECOOP Workshop WCOP 1997)

OOD
Components – Definitions

- “A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.”
  (William T. Councill, George T. Heineman:
  Component-Based Software Engineering. Addison-Wesley, 2001)
- A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction. (Grady Booch)
- A software component is a static abstraction with plugs. (Nierstrasz/Dami)

OOD
Components – Definitions

- As Components deal with ports and interfaces it's important to know the differences ...
- Interfaces
  - An Interface is collection of operations provided anywhere. It gives a name to such a collection.
  - Interfaces don't provide behaviour
  - An interface is something like a contract between a service provider and a service user
  - An interface is like a phone book, naming a service

OOD
Components – Definitions

- Interfaces
  - An interface is a specification of required behavior (but not the implementation) - The benefit of an interface is that it lets you separate the specification of behavior from its implementation (James Brucker)
- Ports
  - Ports are instantiable (in contrast to interfaces)
  - A port is a connection to an instance of a class
  - Ports have an identity
  - A port is like a phone distribution box, accepting incoming call and connecting to the serving location

OOD
Components – Characteristics

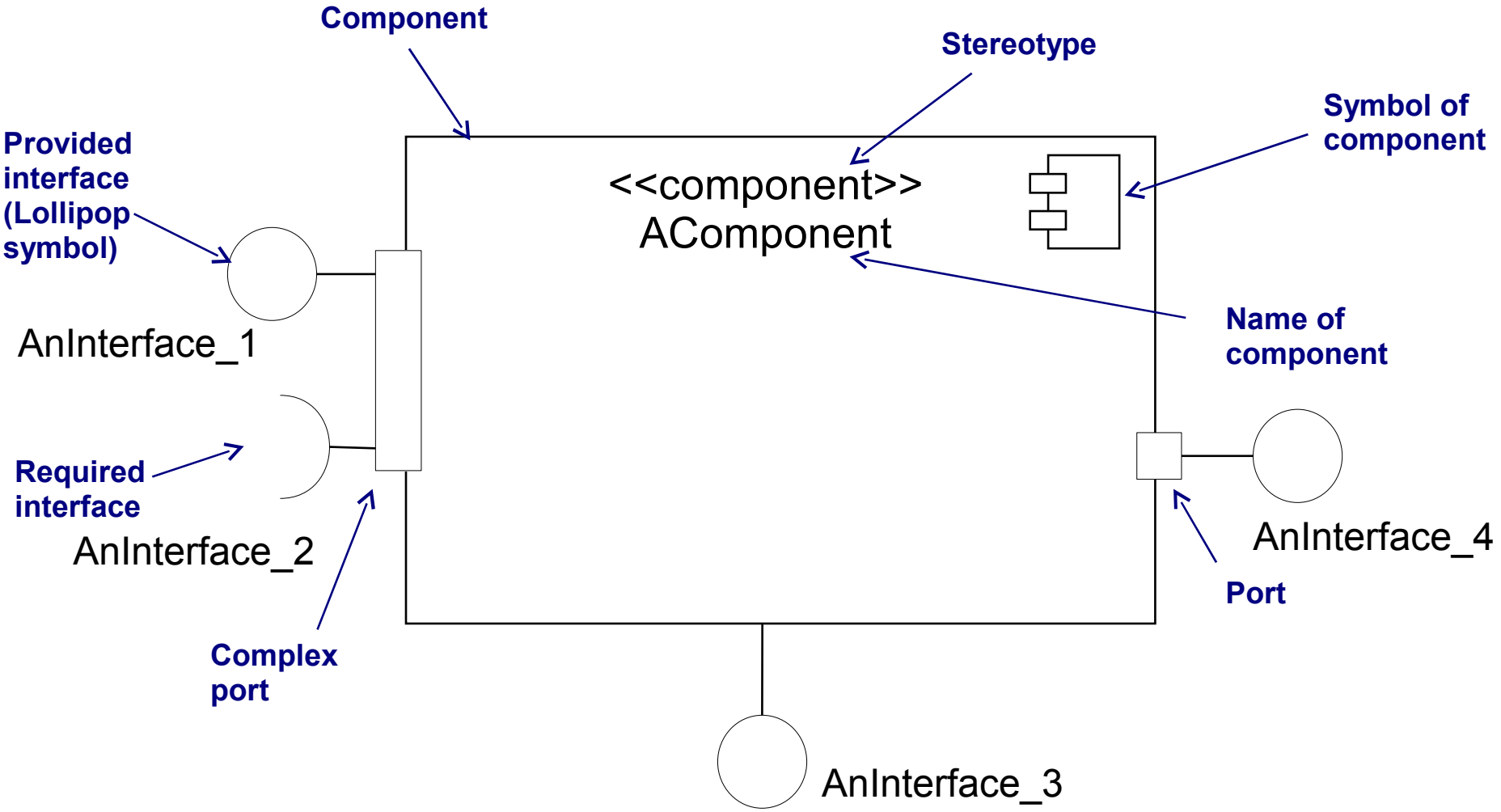
1. A component exports one or several interfaces, that are guaranteed like a contract, especially the exact semantics of the interfaces.
   Every Component C exporting the Interface I is an Implementation of I.
2. A component imports other interfaces meaning that the component is using the methods of this imported interfaces.
   The component is only executable, if all interfaces are available.
   This is the task of the configuration.

OOD
Components – Characteristics

3. A component hides the implementation and is so interchangeable with another component using the same interface
4. A component can easy be reused as it does know anything about the environment where it is running. It makes only minimal assumptions
5. A component could content other components, a component hierarchy is possible
6. Beside interfaces components are the most important utilities in design and implementation

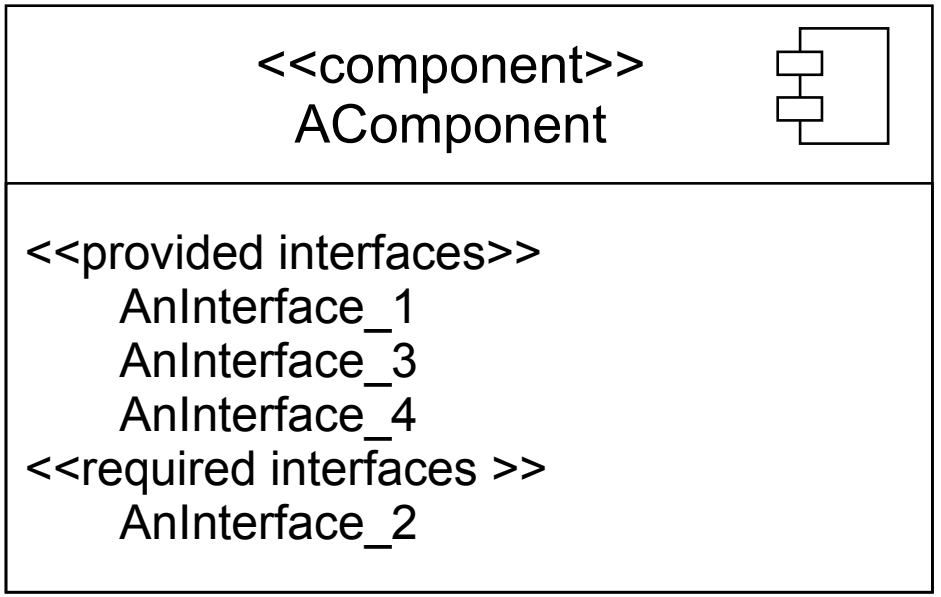
OOD
Components – UML Diagram

- Black Box representation



OOD
Components – UML Diagram

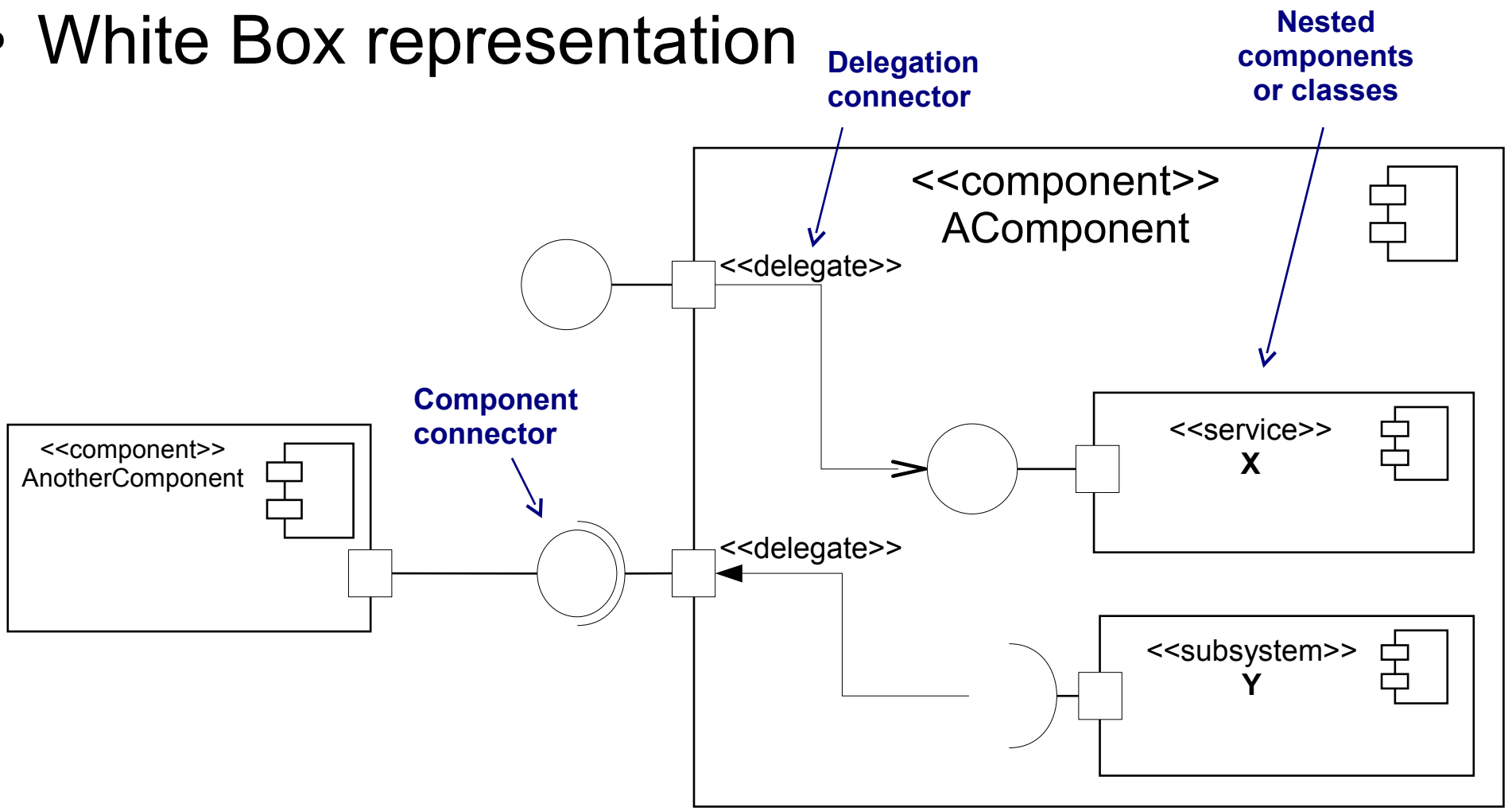
- Black Box representation - alternative



Stereotypes for components could be for example
<<specification>>
<<implement>>
<<entity>>
<<service>>
<<subsystem>>.

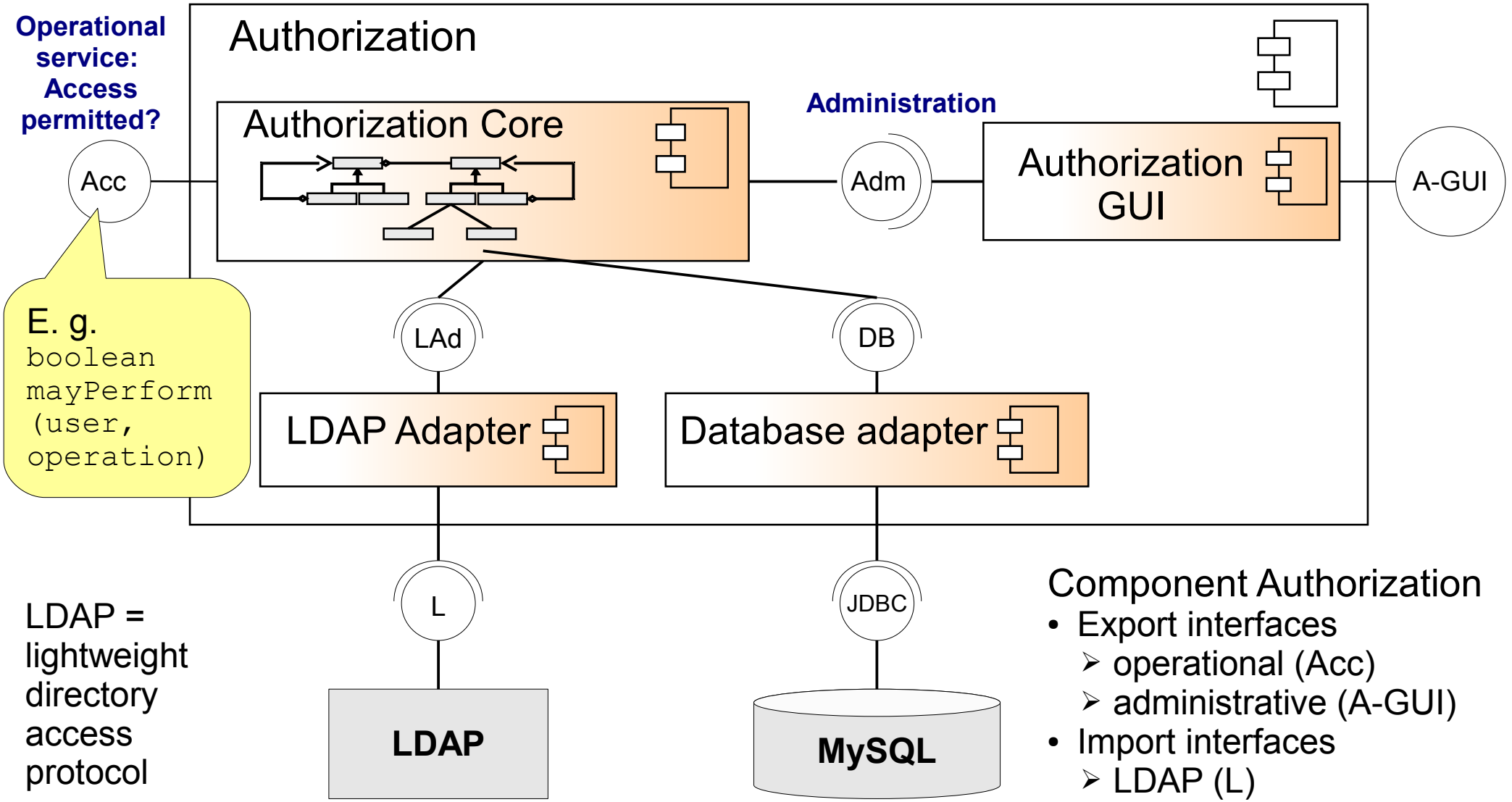
OOD
Components – UML Diagram

- White Box representation



OOD – Components

Working example: Authorization (cont'd)



LDAP = lightweight directory access protocol

Component Authorization
- Export interfaces
  - operational (Acc)
  - administrative (A-GUI)
- Import interfaces
  - LDAP (L)
  - Database (JDBC)

OOD
Components – Interfaces

- The interfaces of a component are typically for different kind of users
- Rule of thumb: The more user an interface has the easier it should be usable
- Interfaces between components could help to structure a project
  - For example: Different components could be developed in subprojects

OOD
Components – Interfaces

- Working example: Authorization (cont'd)

Interface	will be used by ...
– Acc	Application programmer *... to get out, if a user may access or not*
– A-GUI	Administration
– Adm, Acc, LAd, DB	Authorization expert
– LAd, L	LDAP expert
– DB, JDBC	Database expert
– Adm, A-GUI	GUI programmer

OOD
Components – Example

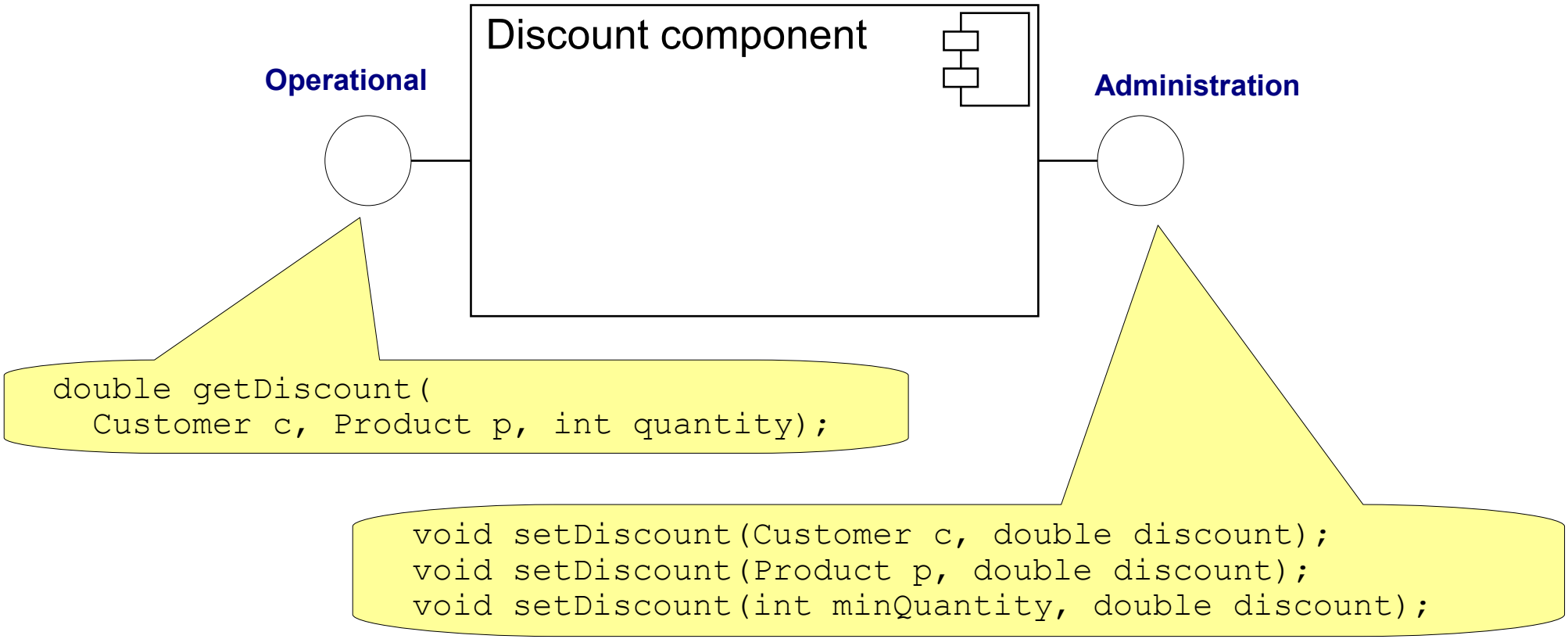
- Discount calculation
  - Description:
    Discounts are given dependent on customer, products and quantity
  - Problem:
    Which discount gets a customer if he orders a specified product in a specified quantity?

OOD
Components – Example

- Discount calculation
  - Proposed solution:
    - Algorithm is in a black box
    - Simple interfaces
    - Data model for the discount problem is not necessary

OOD
Components – Example

- Discount calculation
  - Possible implementation



OOD
Components – Proceeding

Formation of components – Considerations

- Difficult design decisions should be encapsulated in separated components (problem hiding)
- The logical dependencies of a component should be clear
- Consideration concerning decoupling:
  Could it be possible to use the component in a completely different context?

OOD
Components – Proceeding

Formation of components – Considerations

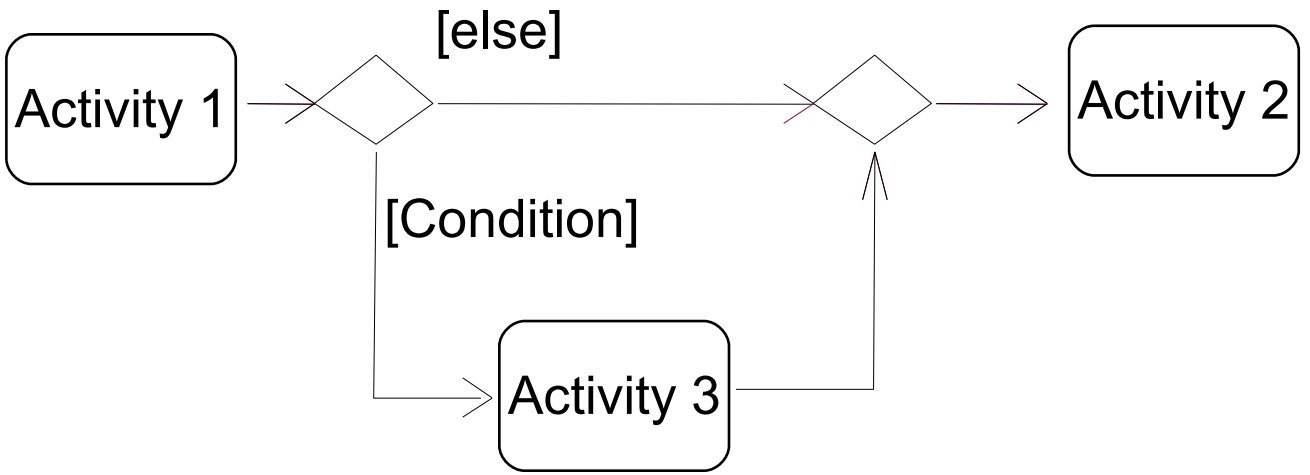
- The data in a component should have the same life cycle
  - Example: Master data concerning variable data only necessary for a specific transaction

OOD
Components – Proceeding

- Formation of components based on Functional Requirements / OOA Model
  - General recommendation out of [Oes06]:
    - Per business process a workflow component
    - Per Use Case a Use Case control component
    - Per external system a component
    - Functional components
  - Advantage of this breakdown:
    - The process oriented aspects of a system get explicit and are separated from the modelling of the real world entities

OOD
Components – Proceeding

- Recommendation for modelling of process oriented components:
  - Use of UML Activity Diagrams
  - Activities in Use Cases could be mapped directly into Activity Diagrams



OOD
Components – Proceeding

- Recommendation for modelling of functional components
  - In a functional component principally the class model could be established out of the OOA class model
  - Functional components get distinguished in the way that loose coupling could be achieved
  - Criteria for functional components
    - Following functionality: close functionality like for example composition should not be separated
    - Classes in an inheritance hierarchy should be in one component (Exception if a framework is used)
    - Tightly coupled classes should be in one component

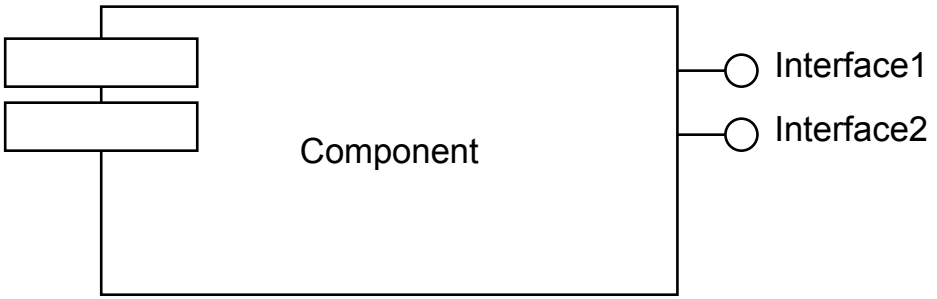
OOD
Components – Proceeding

- Criteria for functional components
  - Technical classes (for example for security, persistence, middleware) and classes with application logic (typically business objects or process objects) should be in separated components
  - Small number of relationships between classes of different components
  - Small volume of message exchange between classes of different components
  - Requirements out of the architecture of distributed systems
  - There should be no cyclical dependencies
  - For expected changes as few components as possible should be adapted
  - Components used by other components should have similar stability

OOD
Components – Proceeding

Collaboration of components

- Develop external interface of components
  - Which classes and methods are accessible from outside?
- Definition of collaboration of the components, especially of the process oriented components with functional components



OOD
Component – Proceeding

- Step by step
  - The formation of components should be done iteratively. Special criteria – like the volume of message exchange – could be measured not until the first “trial and error” .
  - Good formation of components could be critical, especially in distributed systems, if components are located on different systems (message exchange)
  - In big projects it's not a bad idea to use organizational boundaries, like for example team boundaries, to define external interfaces of components

OOD Interfaces

- Goal should be to develop smart interfaces – general principle
  - The `public` interface of a class should be preferably small
  - The implementation of a functionality should be `private`
  - The Test-First-Approach supports this idea

OOD
Interfaces

- Interfaces – Considerations
  - Adequacy
    Who should use the interface?
    - Application programmer versus technology specialist
    - Operative versus administrative access
    - Many versus less user

OOD
Interfaces

- Interfaces – Considerations
  - Coupling / Complexity
    - In general:
      The user of an interface should only see what he needs – not more not less

OOD Interfaces

- Interfaces – Considerations
  - Coupling / Complexity
    - Possible designs concerning the parameter of an interface
      - Flat interfaces
        - using only basic data types like `String`, `int`, ...
        - Loose Coupling
      - Deep interfaces
        - using complex objects as parameter
        - Close coupling
    - Proposal: Instead of using internal objects of a component one should better use “transport objects” (value objects) for the interface to decouple the interface from the internal implementation

OOD Interfaces

- Value Objects / Data types
  - “Value object” and “transport object” are synonyms
  - Value objects
    - hold only references to basic objects like `String` or `Container` (with `Array`, `Hashtable`, ...)
      Attention concerning container: Which objects are referenced? Structures should be flat and basic
    - have no functionality – typically only accessor methods
    - are created typically only once, and then used only for reading – change operations should be offered as different services

OOD Interfaces

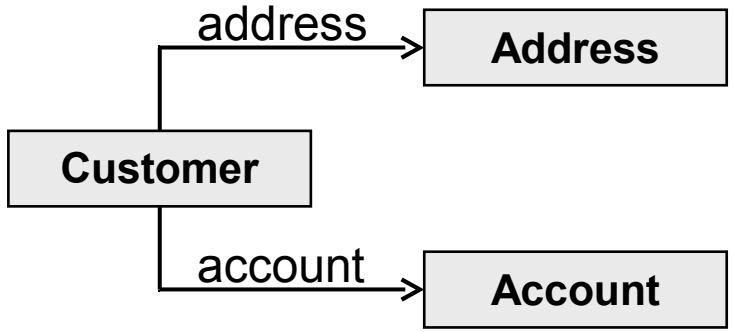
- Value Objects / Data types
  - Data types are intelligent Value Objects
    - They contain own check logic
    - Example: An ISBN object could content a check, if the format is valid

OOD Interfaces

- Comparison Domain Object / Value Object
  - Example

References to other domain objects

```
public class Customer {
   int id;
   String firstName;
   String lastName;
   Address address;
   Account account;
}
```




```
public class CustomerTO {
   int id;
   String firstName;
   String lastName;
   String cityName;
   String zipcode;
   String streetName;
}
```

flat structure, no references, only data needed in the context

OOD
Interfaces

- Interfaces – Proceeding
  - Completeness concerning data types of the interface
    - It must be clear where all the used data types are defined
    - Basic data types like `String` or `Container` are unproblematic
    - Complex data types; handling possibilities:
      - The interface could offer a query returning the corresponding data type – with a default initialization if required
      - The data type is defined together with the interface but has to be instantiated by client
      - Another interface offers a query returning the parameter
        But this results in a dependency to this other interface which should be avoided

OOD
Interfaces

- Interfaces – Proceeding
  - Independence from techniques
    - First an interface should be defined constraint on functionality and not on techniques
    - Concerning the decision which concrete technique to use maybe adaption are necessary because of constraints
    - Reaction on programming errors or technical exceptions like network problems are not part of the interface, but functional exceptions could be part of it

OOD
Interfaces

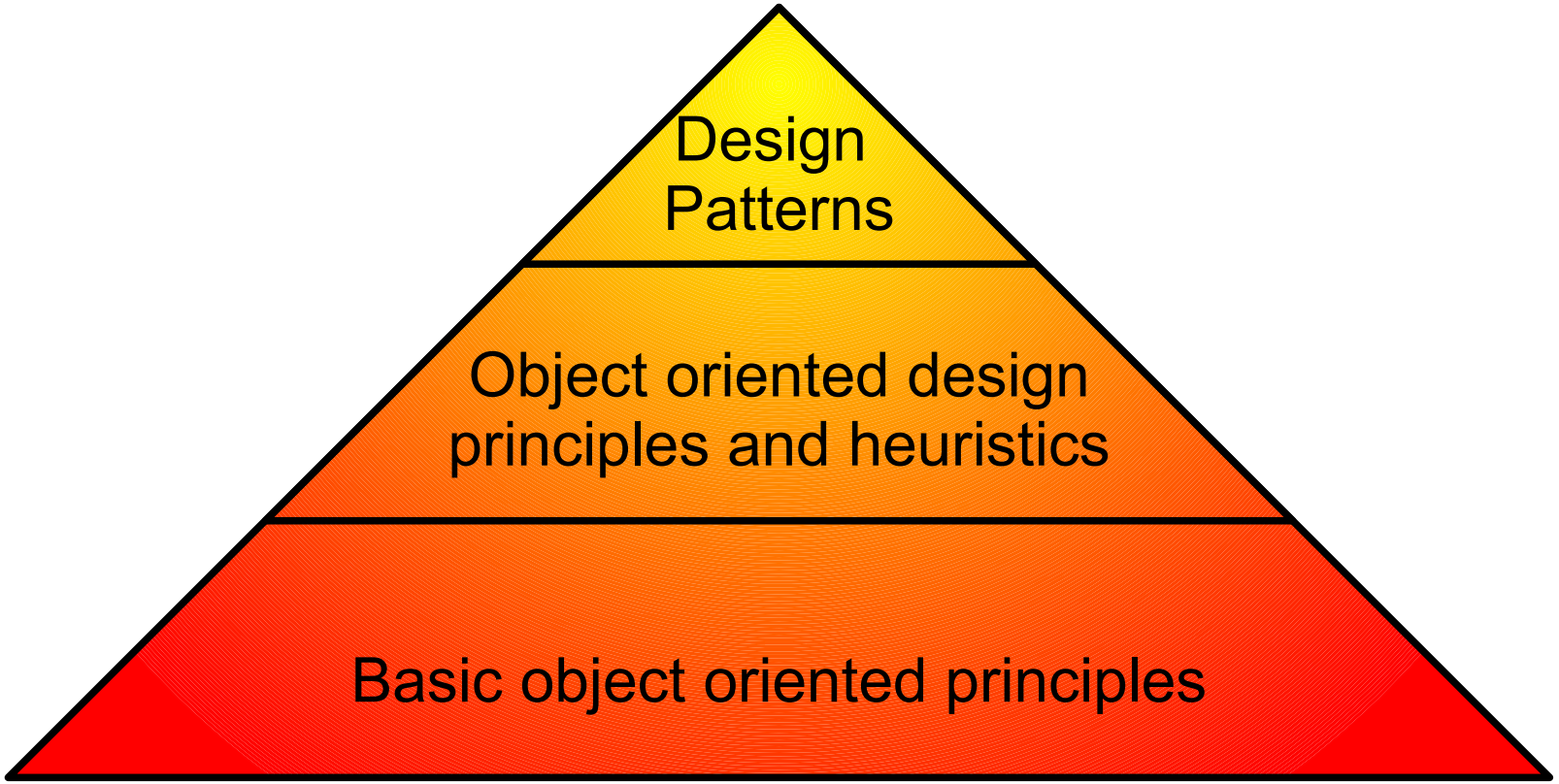
- Interfaces – Proceeding
  - Complete scope
    - Exists for every operation a request to check the results of it?
    - Is it possible to test the preconditions of an operation?
    - Is it possible to cancel a specific operation?
      It is a sensible design decision if this is necessary or should be possible

OOD Interfaces

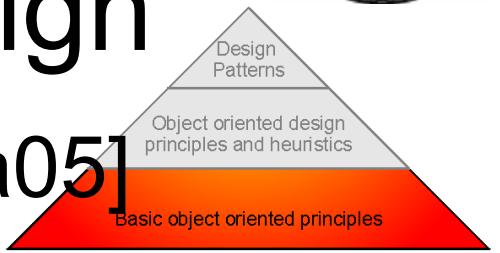
- What is specified in an interface?
  - Syntax of the interface — *Always*
    - Methods, parameter, return values
  - Possible errors and exceptions — *Mostly*
  - Semantic of the interface — *Vague*
    - Side effects (if so)
    - Preconditions and postconditions
    - Description of the functionality or result
  - Non functional requirements — *Virtually never*
    - Performance
    - Robustness

OOD
Good Object Oriented Design

- The OOD pyramid [Sha05]

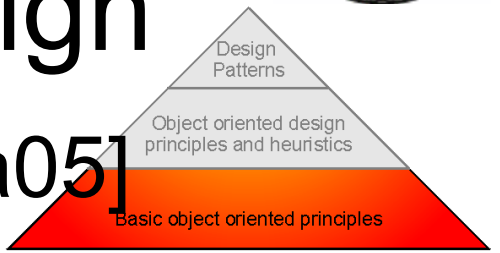


OOD
Good Object Oriented Design



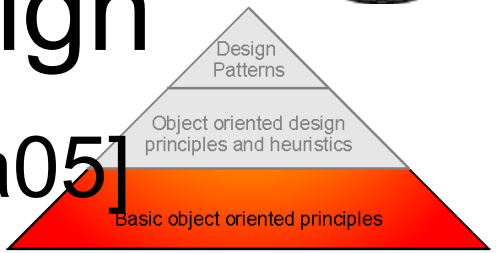
- Basic OO principles to modeling [Sha05]
  - Encapsulation
    - Data and behavior are integrated and encapsulated in a programming unit
    - Goal: Assuring the highest level of decoupling between classes
    - Information hiding
      Accessing of data only with methods
    - Implementation hiding
      Clearly defined interfaces hide internal implementation details

OOD
Good Object Oriented Design



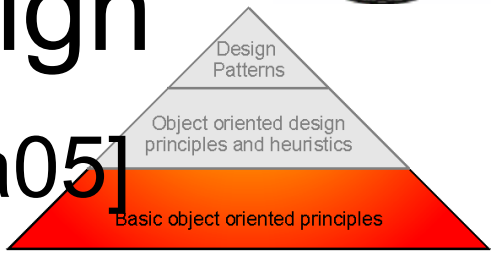
- Basic OO principles to modeling [Sha05]
  - Inheritance
    - Goal: Extend the behavior of a base class
    - Interface inheritance describes a new interface in terms of one or more existing interfaces
    - Implementation inheritance defines a new implementation in terms of one or more existing implementations

OOD
Good Object Oriented Design



- Basic OO principles to modeling [Sha05]
  - Polymorphism
    - Ability of different objects to respond differently to the same message
    - Goal: Clients can easier interact with similar objects using the same operations
    - Polymorphism is closely related to inheritance as well as to encapsulation
    - Inheritance polymorphism works on an inheritance chain
    - Operational polymorphism specifies similar operations for non-related out-of-inheritance classes or interfaces.

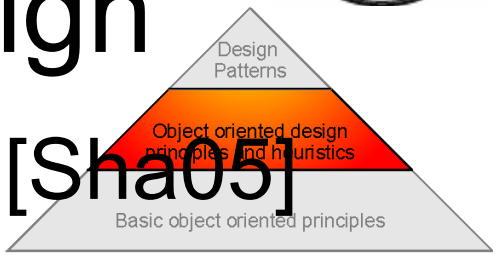
OOD
Good Object Oriented Design



- Basic OO principles to modeling [Sha05]
  - Discussion
    - Some OO principles are controversial in the sense that they are inconsistent with one another.
    - For example, to be able to inherit from a class, one should know the internal structure of that class, while encapsulation's goal is exactly the opposite – it tries to hide as much of the class structure as possible
    - Tradeoff between these two principles necessary

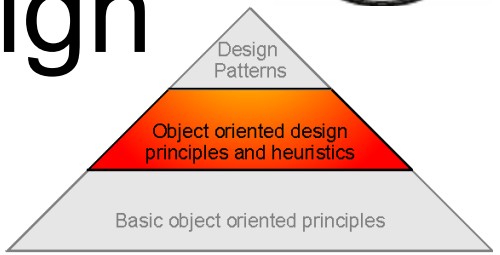
      The art of OOD

OOD
Good Object Oriented Design



- OO Design Principles and Heuristics [Sha05]
  - Introduction:
    - Collected
      - About a dozen OO design principles
      - Four dozens OO design heuristics
    - OO evangelists: Grady Booch, Bertrand Meyer, Robert C. Martin, Barbara Liskov, and others
    - OO design principles define the most common scientifically derived approaches for building robust and flexible systems
    - These approaches proved to be the best tools in solving numerous OO design issues that can't be captured by fundamental OO principles

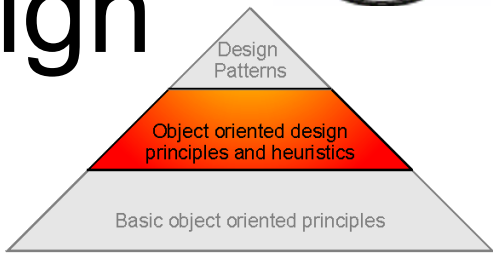
OOD
Good Object Oriented Design



- OO Design Principles [Sha05]

Class structure and relationships group	Package cohesion group	Package coupling group
• Single Responsibility Principle (SRP) • Open/Closed Principle (OCP) • Liskov Substitution Principle (LSP) • Dependency Inversion Principle (DIP) • Interface Segregation Principle (ISP) • Don't Repeat Yourself (DRY) • Keep it simple, stupid (KISS)	• Reuse/Release Equivalency Principle (REP) • Common Closure Principle (CCP) • Common Reuse Principle (CRP)	• Acyclic Dependency Principle (ADP) • Stable Dependency Principle (SDP) • Stable Abstractions Principle (SAP)

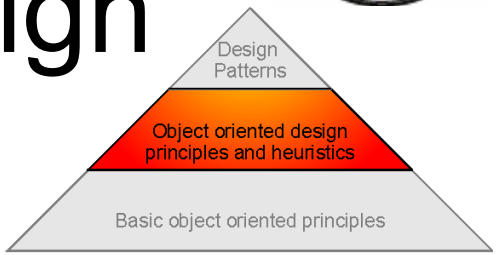
OOD
Good Object Oriented Design



- OO Design Principles [Sha05]
  - Class structure and relationships group

    Design principles:
    - Single Responsibility Principle (SRP)
      - Also known as the cohesion principle
      - One class should have only one responsibility or cover only one functional unit
      - A class should have only one reason to change
      - No big “Swiss army knife®” classes
      - Rather many small classes with high locality
      - Advantages:
        - Well arranged code
        - Reusability easier

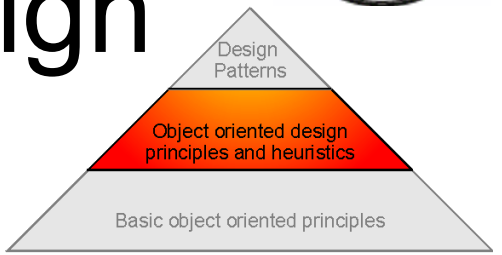
OOD
Good Object Oriented Design



- OO Design Principles [Sha05]
  - Class structure and relationships group

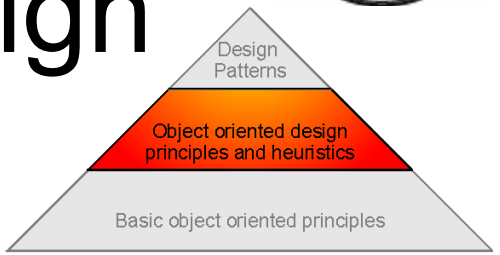
    Design principles:
    - Open/Closed Principle (OCP)
      - Classes should be open to extension but closed to modification
      - Modules should be written so that they can be extended without being modified
      - Developers should be able to change what the modules do without changing the modules' source code

OOD
Good Object Oriented Design



- OO Design Principles [Sha05]
  - Class structure and relationships group Design principles:
    - Liskov Substitution Principle (LSP)
      - also known as “Design by Contract”
      - Subclasses should be able to substitute for their base classes
      - Clients that use references to base classes must be able to use the objects of derived classes without knowing them
      - This principle is a generalization of a "design by contract" approach that specifies that a polymorphic method of a subclass can only replace
        - its pre-condition by a weaker one
        - its post-condition by a stronger one

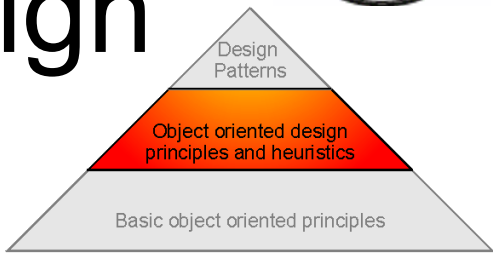
OOD
Good Object Oriented Design



- OO Design Principles [Sha05]
  - Class structure and relationships group

    Design principles:
    - Dependency Inversion Principle (DIP)
      - High-level modules shouldn't depend on low-level modules.
      - Abstractions shouldn't depend on details.
      - Details should depend on abstractions.
    - Interface Segregation Principle (ISP)
      - clients shouldn't depend on the methods they don't use
      - Multiple client-specific interfaces are better than one general-purpose interface

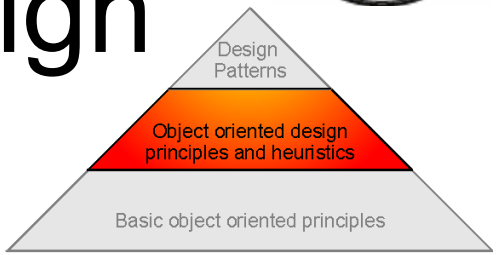
OOD
Good Object Oriented Design



- OO Design Principles [MPW06]
  - Class structure and relationships group

    Design principles:
    - Don't Repeat Yourself (DRY)
      - also known as “Once and Only Once” or “Single Point of Truth (SPOT)”
      - Code should be written only once, duplication should be avoided
      - If similar code is used more often, it should be concentrated e. g. in an abstract parent class.
      - Advantage: Easier to maintain, as common code has to be changed at only one place

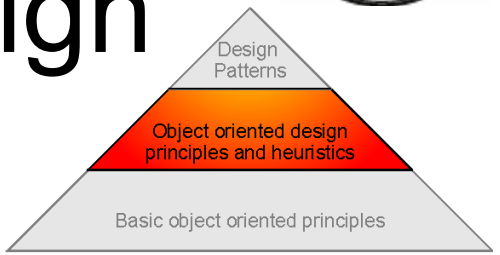
OOD
Good Object Oriented Design



- OO Design Principles
  - Class structure and relationships group

    Design principles:
    - Keep it simple, stupid (KISS)
      - No including of needless abstraction levels / generalizations etc.
      - Advantages:
        - The less code exists, the less effort has a maintenance programmer later to get orientation in the system
        - No „dead code“ in the system

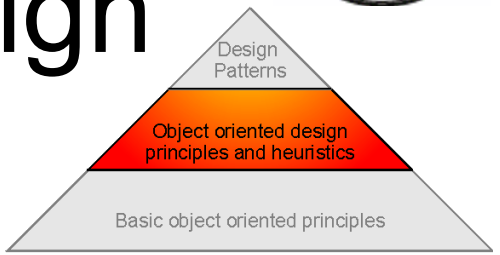
OOD
Good Object Oriented Design



- OO Design Principles [Sha05]
  - Package cohesion group

    This group deals with the principles that define packaging approaches based on class responsibilities (i. e., how strongly related the responsibilities of classes are)

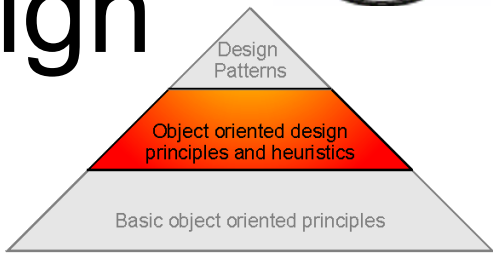
OOD
Good Object Oriented Design



- OO Design Principles [Sha05]
  - Package cohesion group

    Design principles:
    - Reuse/Release Equivalency Principle (REP) [Mar96]
      - makes release granularity equal to reuse granularity
      - only components that are released through a tracking system can be effectively reused. This granule is the package.
      - Example:
        With this principle code could be reused without need to look at the source code (other than the public portions of header files). Whenever these libraries are fixed or enhanced, a new version gets released which can then be integrated into a system when opportunity allows.
        That is the reused code is to be treated like a product.

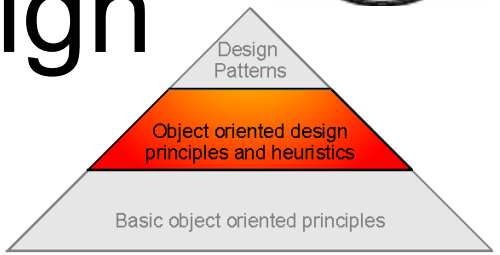
OOD
Good Object Oriented Design



- OO Design Principles [Sha05]
  - Package cohesion group

    Design principles:
    - Common Closure Principle (CCP)
      - Classes that change together belong together
      - Classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in the package [Mar96]
      - That means: More important than reusability, is maintainability

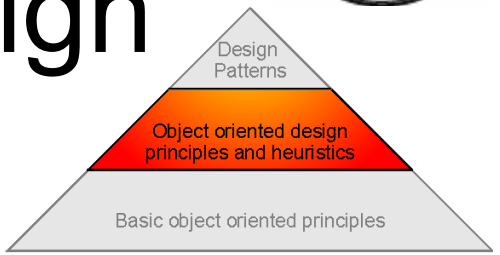
OOD
Good Object Oriented Design



- OO Design Principles [Sha05]
  - Package cohesion group

    Design principles:
    - Common Reuse Principle (CRP)
      - Classes that aren't reused jointly shouldn't be grouped together
      - Classes in a package are reused together. If you reuse one class in a package, you reuse them all [Mar96]

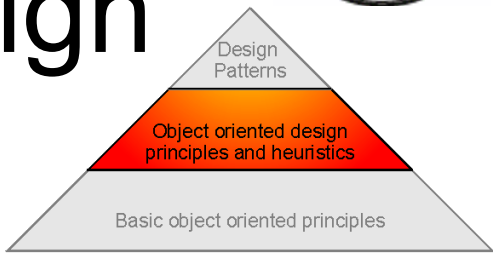
OOD
Good Object Oriented Design



- OO Design Principles [Sha05]
  - Package coupling group

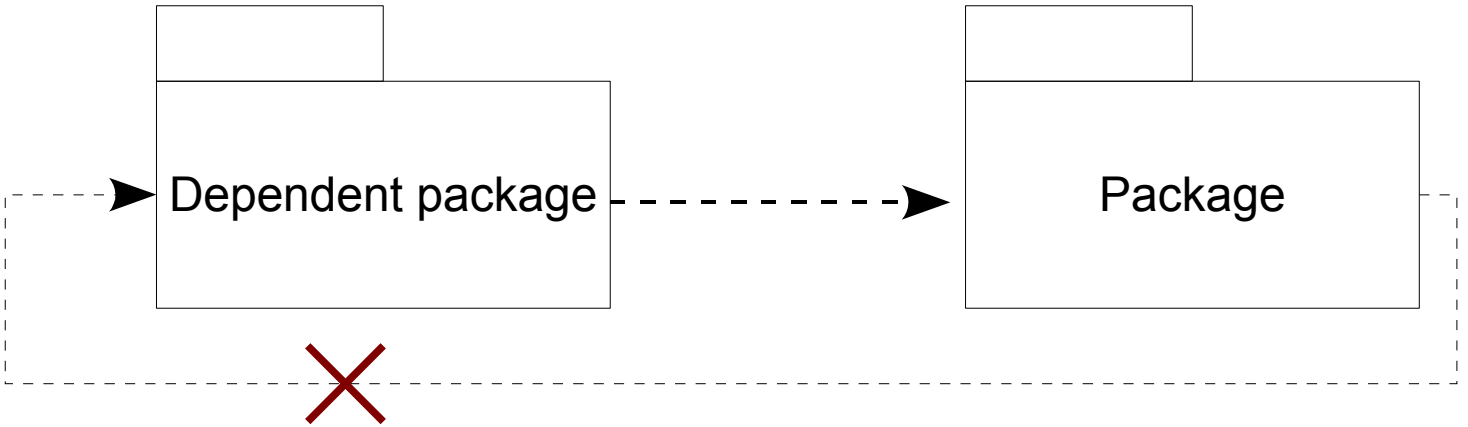
    This group deals with principles that define packaging approaches based on the packages' collaboration (i. e. how much one package relies on or is connected to another)

OOD
Good Object Oriented Design

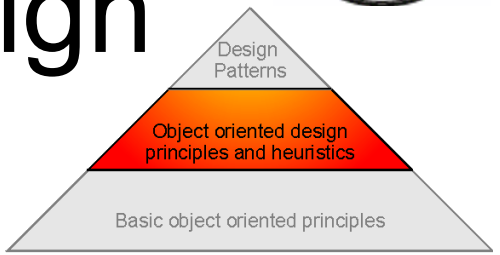


- OO Design Principles [Sha05]
  - Package coupling group

    Design principles:
    - Acyclic Dependency Principle (ADP)
      - prohibits forming cyclic dependencies among packages
      - The dependency structure between packages must be a directed acyclic graph. That is, there must be not cycles in the dependency structure [Mar96]

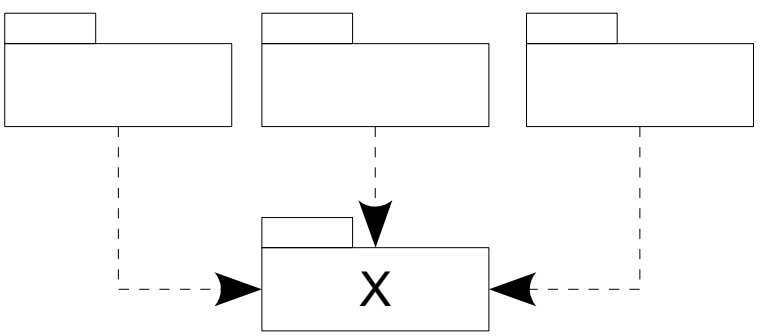


OOD
Good Object Oriented Design

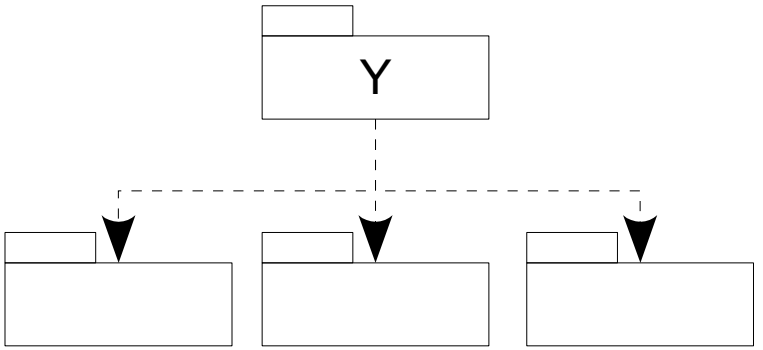


- OO Design Principles [Sha05]
  - Package coupling group

    Design principles:
    - Stable Dependency Principle (SDP) [Mar00]
      - package dependency should be allowed to reinforce package stability
      - Stability is related to the amount of work required to make a change.

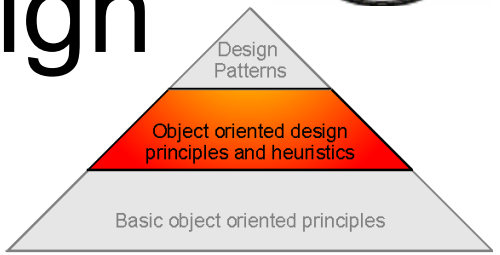


X is stable - independent



Y is unstable – depending on 3 packages

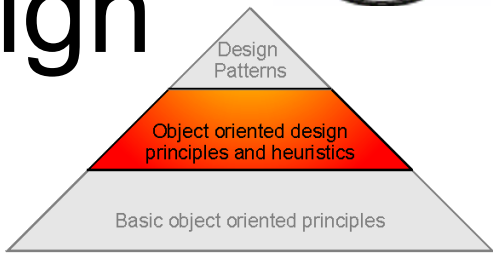
OOD
Good Object Oriented Design



- OO Design Principles [Sha05]
  - Package coupling group

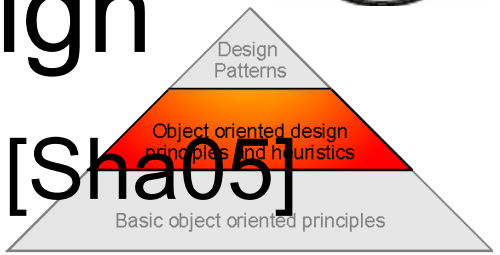
    Design principles:
    - Stable Abstractions Principle (SAP) [Mar00]
      - stable packages should be abstract packages
      - Idea is to create a packages structure of an application as a set of interconnected packages with instable packages at the top, and stable packages on the bottom.
        In this view, all dependencies point downwards.
        Hence, those packages at the top are instable and flexible.
        But those at the bottom are very stable and should be difficult to change.
        These packages should be highly abstract, so they could easily be extended

OOD
Good Object Oriented Design



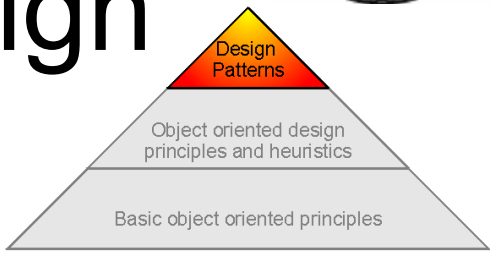
- OO Design Heuristics [Sha05]
  - Design heuristics derive from the practical experience of OO developers
  - Heuristics can extend design principles to several specific implementations
  - Design heuristics are grouped by their application: class structure, object-oriented applications, relationships between classes and objects, inheritance and association relationships, etc.
  - Heuristics are less fundamental than design principles, but they clarify, explain, and expand design principles

OOD
Good Object Oriented Design



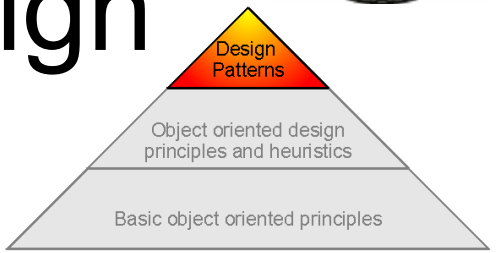
- OO Design Principles and Heuristics [Sha05]
  - Both design principles and heuristics can be controversial - some design principles and heuristics have internal dissension, while others contradict each other.
  - Examples
    - Conforming to the Open/Closed Principle can be expensive and lead to unnecessary complexity - the class model should be pertinent to a specific context
    - Liskov Substitution Principle restricts the use of inheritance while the Open/Closed Principle embraces it

OOD
Good Object Oriented Design



- Design Pattern [Sha05]
  - Design patterns represent common solutions to design problems solved in a particular context
  - So far collected
    - 23 basic design patterns [GHJV95]
    - 21 core J2EE patterns by the Sun Java Center
    - 51 patterns of enterprise application architecture identified by Martin Fowler et al.
    - 65 enterprise integration patterns
    - ... lot of patterns specific to particular problem domains

OOD
Good Object Oriented Design



- Design Pattern [Sha05]
  - Design Pattern represent good design practices and span a wide range of solutions from general topics like object lifecycle and structure to more specific themes such as integration tiers, data transfer, and transformation.
  - Rule of thumb:
    - Try to apply patterns where application design would benefit from performance and flexibility
    - However, sometimes you have to choose patterns based on just one "benefit"

OOD
Transition from OOA to OOD

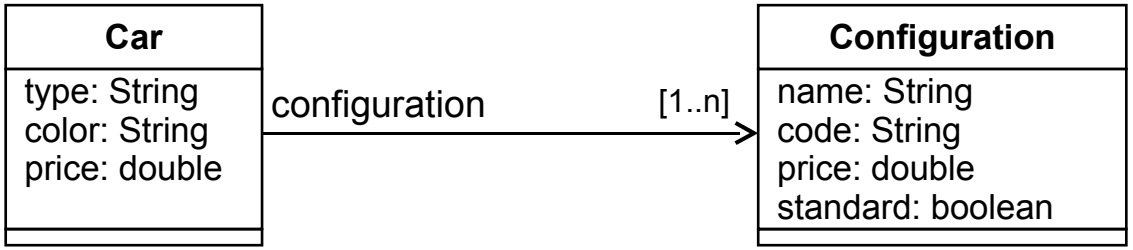
- Inheritance
  - Classifications could be modeled with inheritance but only applicable classifications should be chosen
  - In general: Use inheritance economically
  - In inheritance hierarchies additional classes could be necessary
  - If changes are expected find and use appropriate Design Pattern

OOD
Transition from OOA to OOD

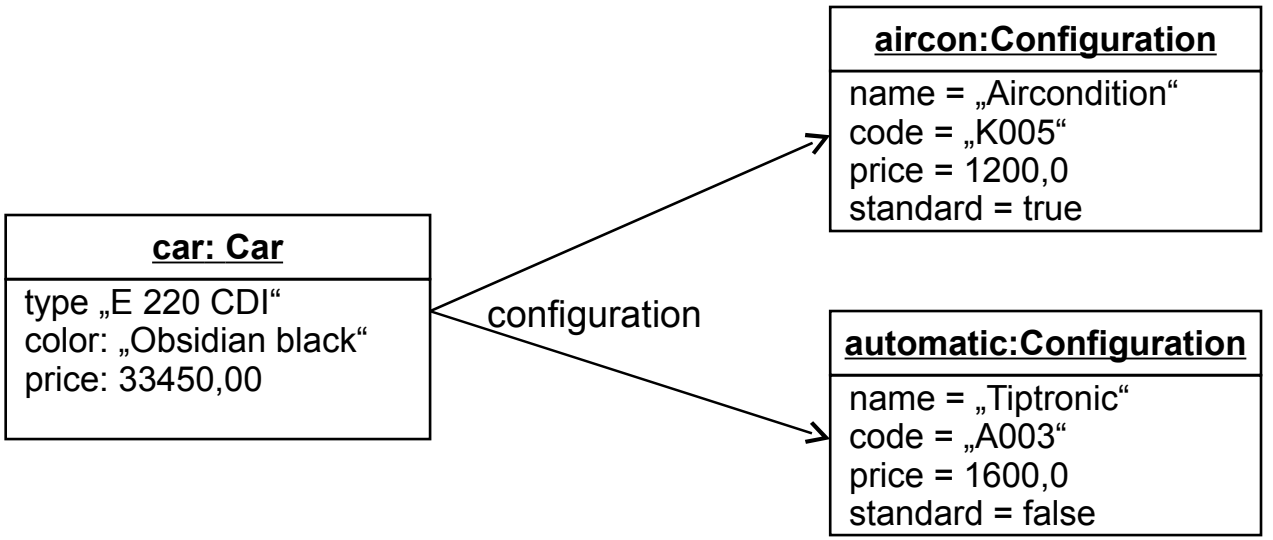
- Attributes of entities – consider
  - Quantity structure
  - Data types and data structures to existing interfaces
  - Data types and data structures in databases
- Definition of technical classes, for example
  - Collection classes, iterators, utility classes, data storage classes, classes for process oriented aspects
- Definition of persistent objects – defining the mapping into the database

OOD
Transition from OOA to OOD

- Example
  OOA Model



  - Example for instances

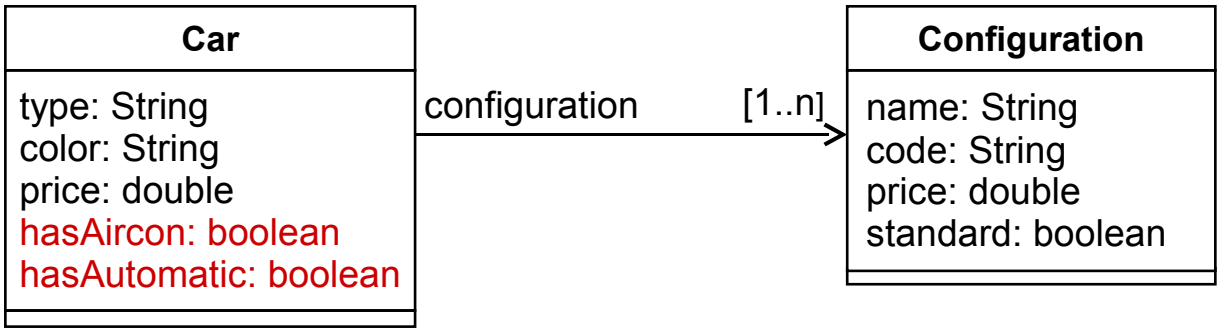


OOD
Transition from OOA to OOD

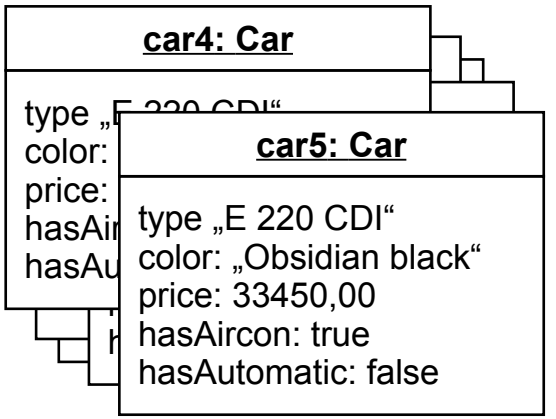
- Example OOD Model (1)
  - Requirement:
    - It should be possible to search quickly e. g. for cars which have automatic gear and air condition
  - Design considerations
    - A 1:1 realization of the analysis model would be too slow, because all the cars have to be initialized with all configurations
  - Design decision
    - Redundant keeping of data in class Car

OOD
Transition from OOA to OOD

- Example OOD Model (1)



  - Precondition is the possibility to use partly initialized object structures

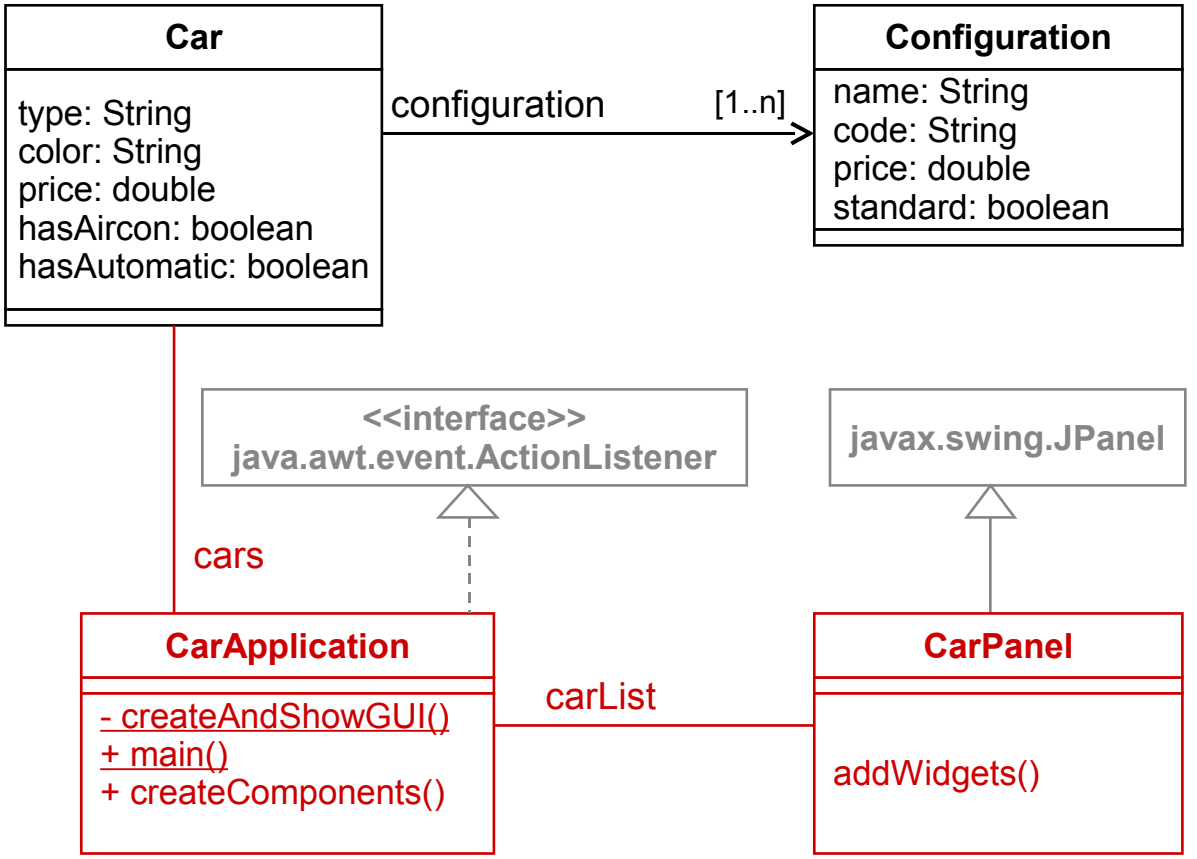


OOD
Transition from OOA to OOD

- Example OOD Model (2)
  - Requirement:
    - The cars should be presented in a GUI
  - Design decision
    - Use of Java Swing classes

OOD
Transition from OOA to OOD

- Example OOD Model (2)

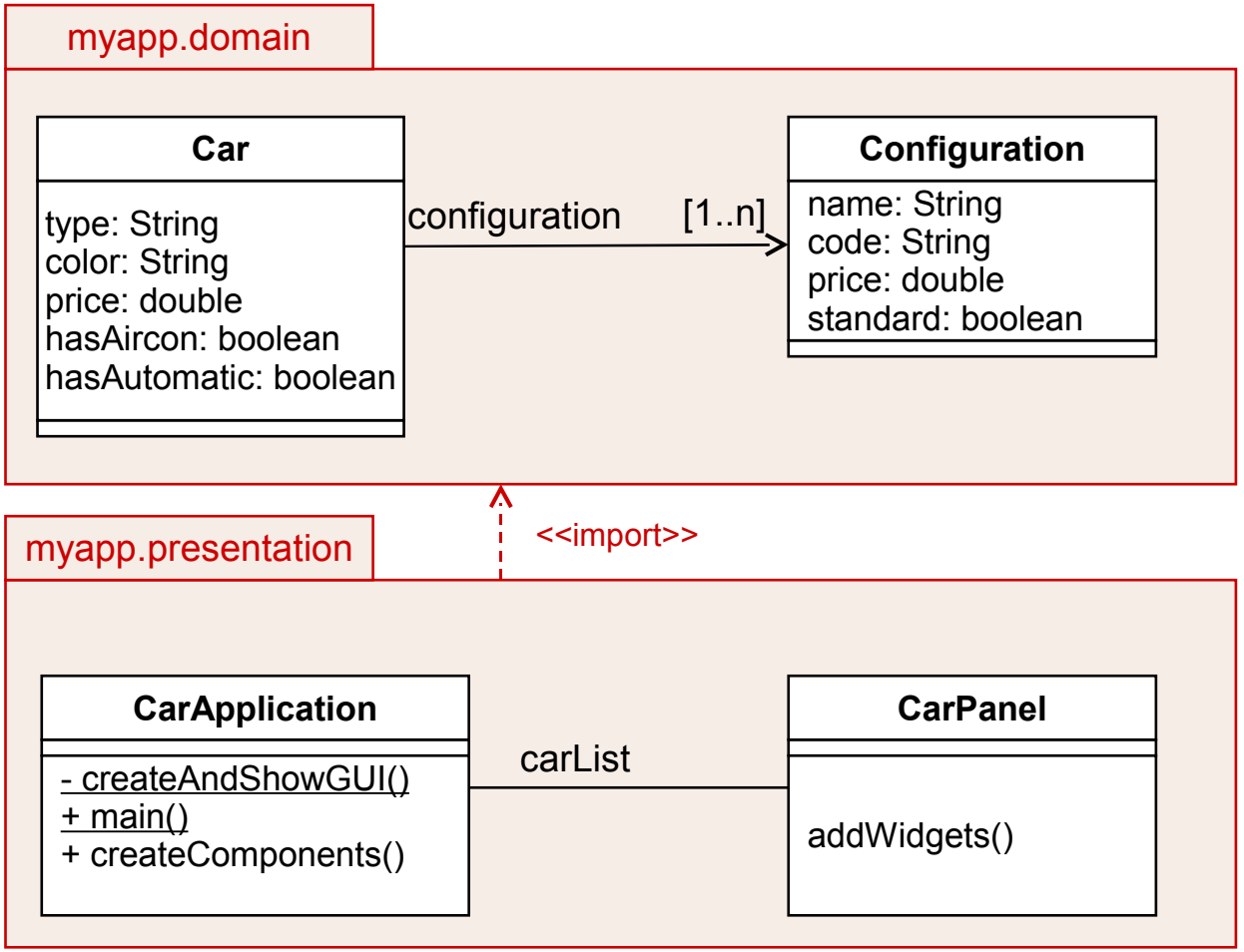


OOD
Transition from OOA to OOD

- Example OOD Model (3)
  - Requirement
    - Arrangement of the application in layers
  - Design decision
    - Grouping of classes belonging together in packages

OOD
Transition from OOA to OOD

- Example OOD Model (3)

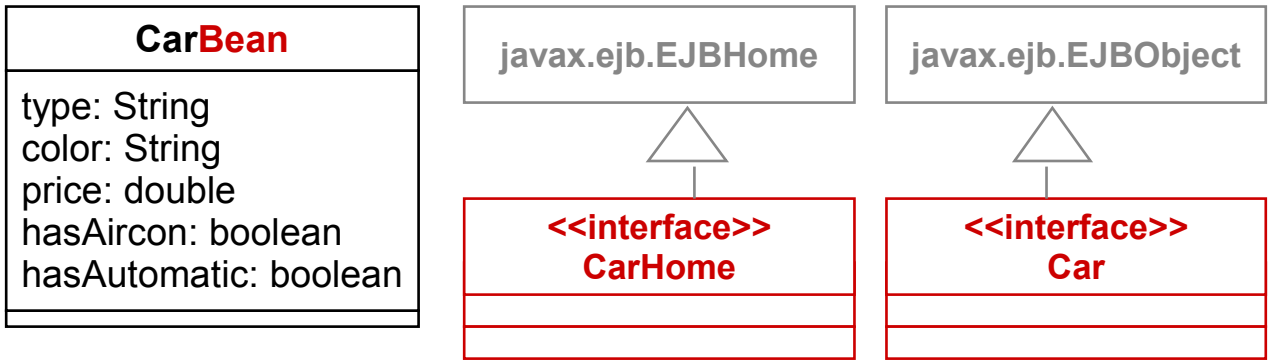


OOD
Transition from OOA to OOD

- Example OOD Model (4)
  - Requirement
    - Data should be stored in a database
  - Design decision
    - Use of Enterprise Java Beans (EJB)
    - Implementation of needed EJB classes
    - More classes get generated typically automated

OOD
Transition from OOA to OOD

- Example OOD Model (4)



OOD
Transition from OOA to OOD

- Summary
  - OOA focus on the functional class model, OOD considers possible reuse, modification issues, maintainability, and implementation aspects

  ⇨ That's why the models are different, typically the OOD model is changed, and / or extended

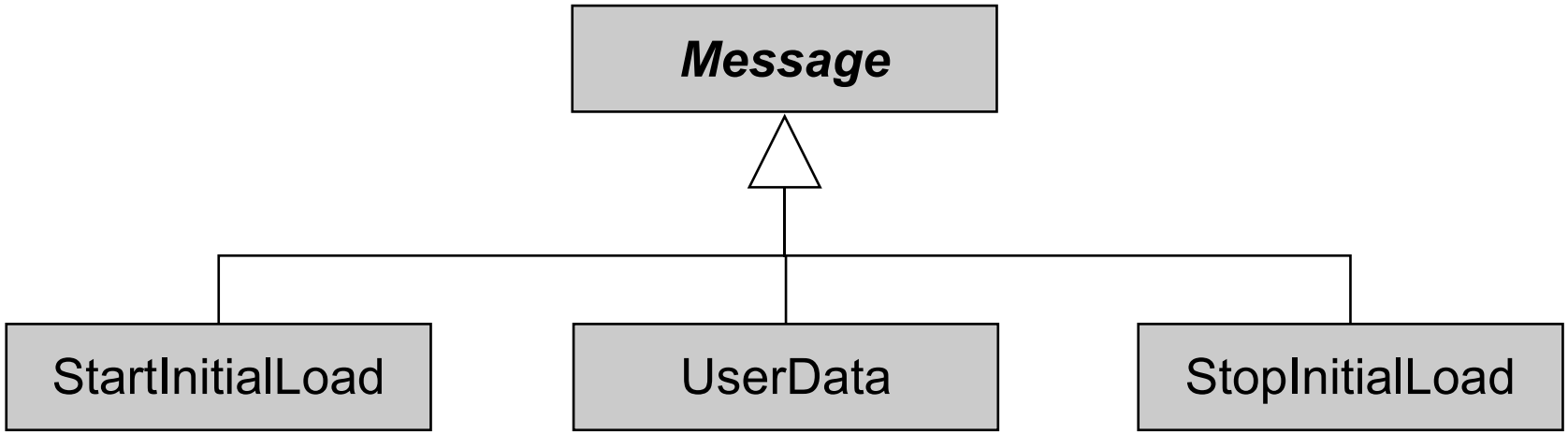
  - More reasons for a different OOD model
    - Resolution of multiple inheritance
    - Memory restriction (every class has some memory overhead - many objects means extended memory demand)
    - Performance (network traffic, restricted database access)

OOD
Good Object Oriented Code

Task:

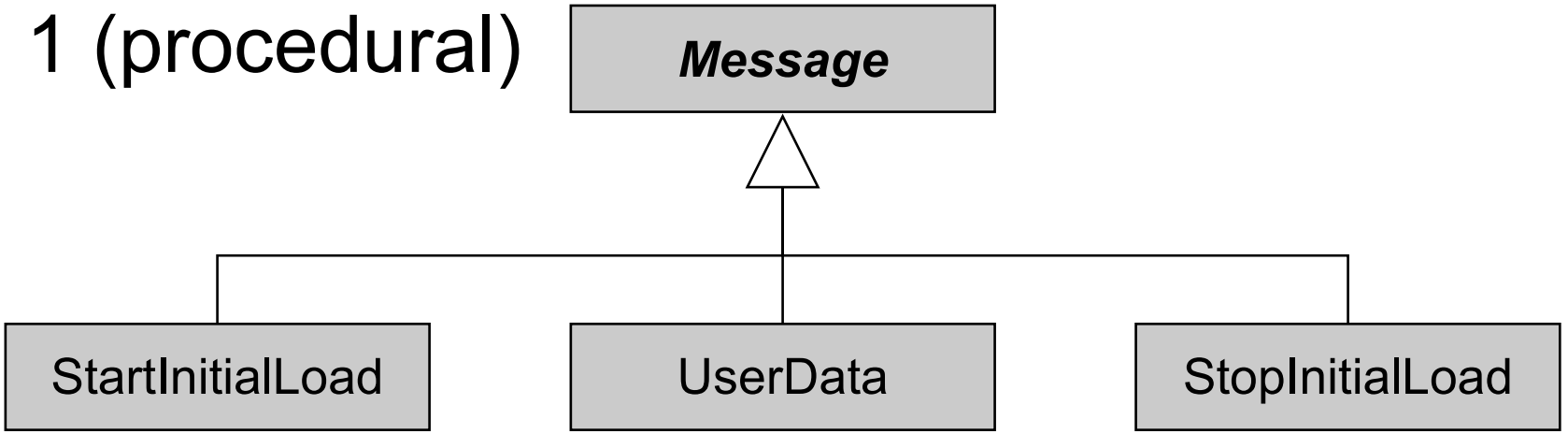
- A message system delivers different messages
  - User data
  - Start of initial load
  - End of initial load
- Depending on the kind of message follows a different processing

OOD
Good Object Oriented Code



OOD
Good Object Oriented Code

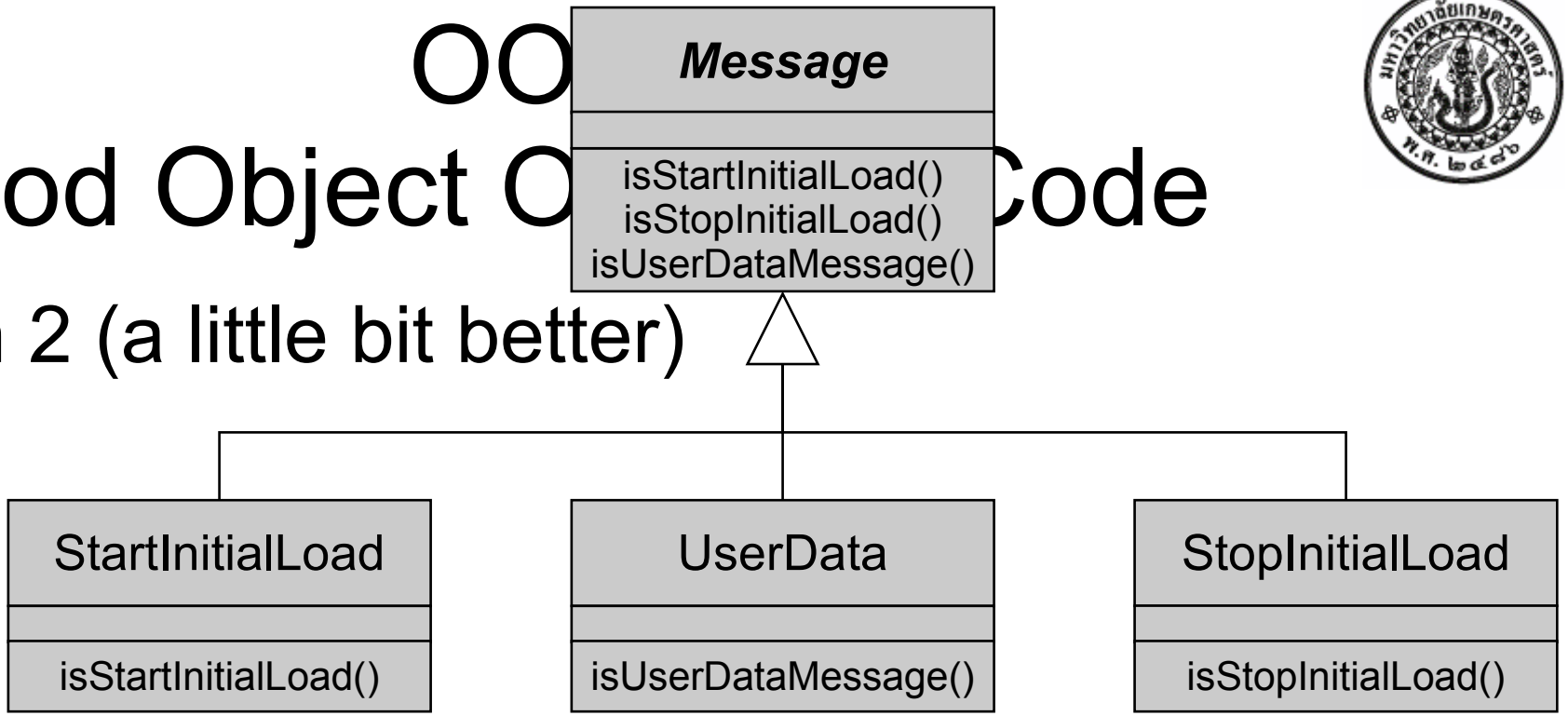
Solution 1 (procedural)



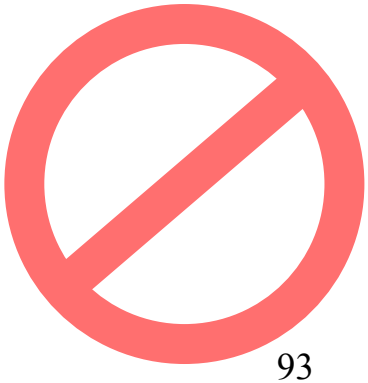

```
public class Importer {
....
  public void main(...) {
    Message message = getNextMessage();
    if       (message instanceof StartInitialLoad)
        processStartInitialLoad();
    else if (message instanceof StopInitialLoad)
        processStopInitialLoad();
    else
        processUserDataMessage(message);
  }
} 
```

OO Good Object O Code

Solution 2 (a little bit better)

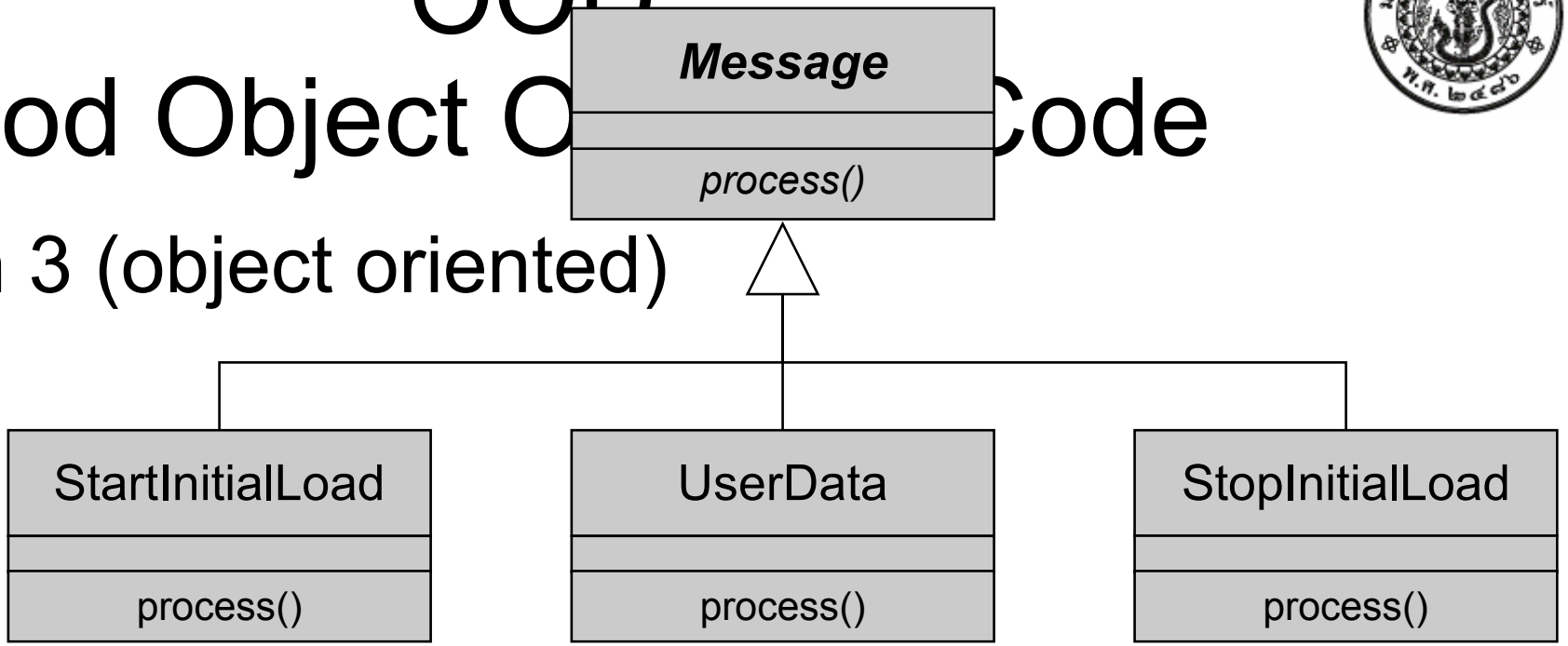



```
public class Importer {
....
  public void main(...) {
    Message message = getNextMessage();
    if      (message.isStartInitialLoad())
        processStartInitialLoad();
    else if (message.isStopInitialLoad())
        processStopInitialLoad();
    else
        processUserDataMessage(message);
  }
}
```



OOD
Good Object O Code

Solution 3 (object oriented)




```
public class Importer {
....
public void main(...) {
   Message message = getNextMessage();
   message.process();
}
```

Every subclass overwrites `process()` to handle the different message types

Substituting the `if else` statements with the polymorph method `process()`

Sources

[AR00] Mohamed Abdelrahman, Abdul Rasheed, "A Methodology for Development of Configurable Remote Access Measurement System", Transactions of Instrumentation Society of America Transactions, 2000.

[Mar96] Robert C. Martin, Granularity, http://www.objectmentor.com/resources/articles/granularity.pdf, 1996

[Mar00] Robert C. Martin, Principles and Patterns, http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf, 2000

[Mol05] Muhammad K. Bashar Molla: An Overview Of Object Oriented Design Heuristics, Master Thesis, Department of Computer Science, Umeå University, Sweden, http://www.cs.umu.se/~ens03mbr/thesis/finalreport.pdf, January 27, 2005

[MPW06] Brett D. McLaughlin, Gary Pollice, David West: Head First Object Oriented Analysis and Design, O'Reilly, 2006

[RV04] Steve Roach, Javier C Vásquez: A Tool to Support the CRC Design Method, http://succeednow.org/icee/Papers%5C339_Roach-Vasquez_(1).pdf, 2004

[Sha05] Gene Shadrin: Three Sources of a Solid Object-Oriented Design; Design heuristics, scientifically proven OO design guidelines, and the world beyond the beginning, http://java.sys-con.com/read/84633.htm, May. 11, 2005

[She05] Girish Shetty: C++ Design and Coding Tips, http://newlc.com/C-Design-and-Coding-Tips.html, 2005