# Lecture 5
# Classes, Components, and Nodes

Rogardt Heldal

# Architecture: three facts

- Every application has an architecture
  - The architecture of a system can be characterized by the principal design decisions made during its development
- Every application has at least one architect
  - Perhaps not known or recognized by that title
- Architecture is not a phase of development
  - Where did the software architecture come from?
  - How does it change over time?

**Taylor, R. N., Medvidovic, N., and Dashofy, E. M. 2009 Software Architecture: Foundations, Theory, and Practice. Wiley Publishing.**

# Challenge

- The **actual** architecture of a system is not always exactly the one conceived by the architects
  - The architecture is also emerging during development (bottom-up)
  - Some architectural decisions are made unconsciously
    - Which decisions have an impact on the architecture? –not easy
  - Some “actual” architects do not have the title of architect

**High-level architecture**

Ideas/vision of the system to be realized

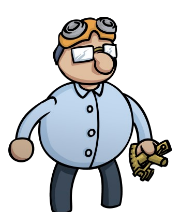
**Working architecture**

Actual blueprint for the implementation teams, used in their daily work

# Problem to be solved

Descriptive models of the Ideas/vision of the system to be realized

GAP



Prescriptive models to develop the system even through automated code transformations

# Analysis

- So far we have done:
  - Requirements
  - Domain Models
  - Use Cases
- These models are based on:
  - Interviews
  - Observations
  - Workshops
  - Looking at similar systems

Generate domain knowledge

Descriptive models of the Ideas/vision of the system to be realized

Comments: without a good analysis, one cannot obtain a good system.

# Problem to be solved

Descriptive models of the Ideas/vision of the system to be realized

GAP

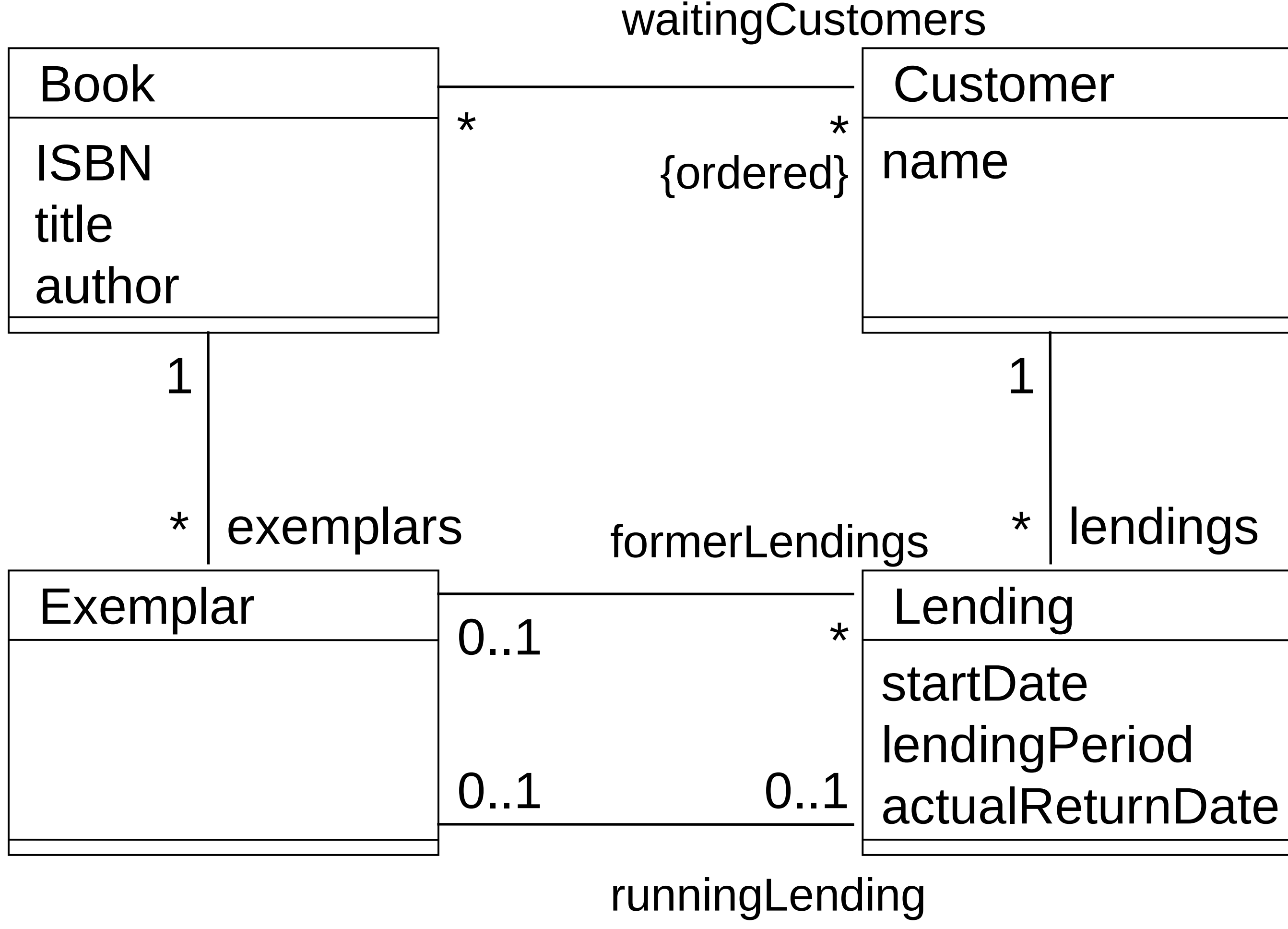
Need creativity

Prescriptive models to develop the system even through automated code transformations

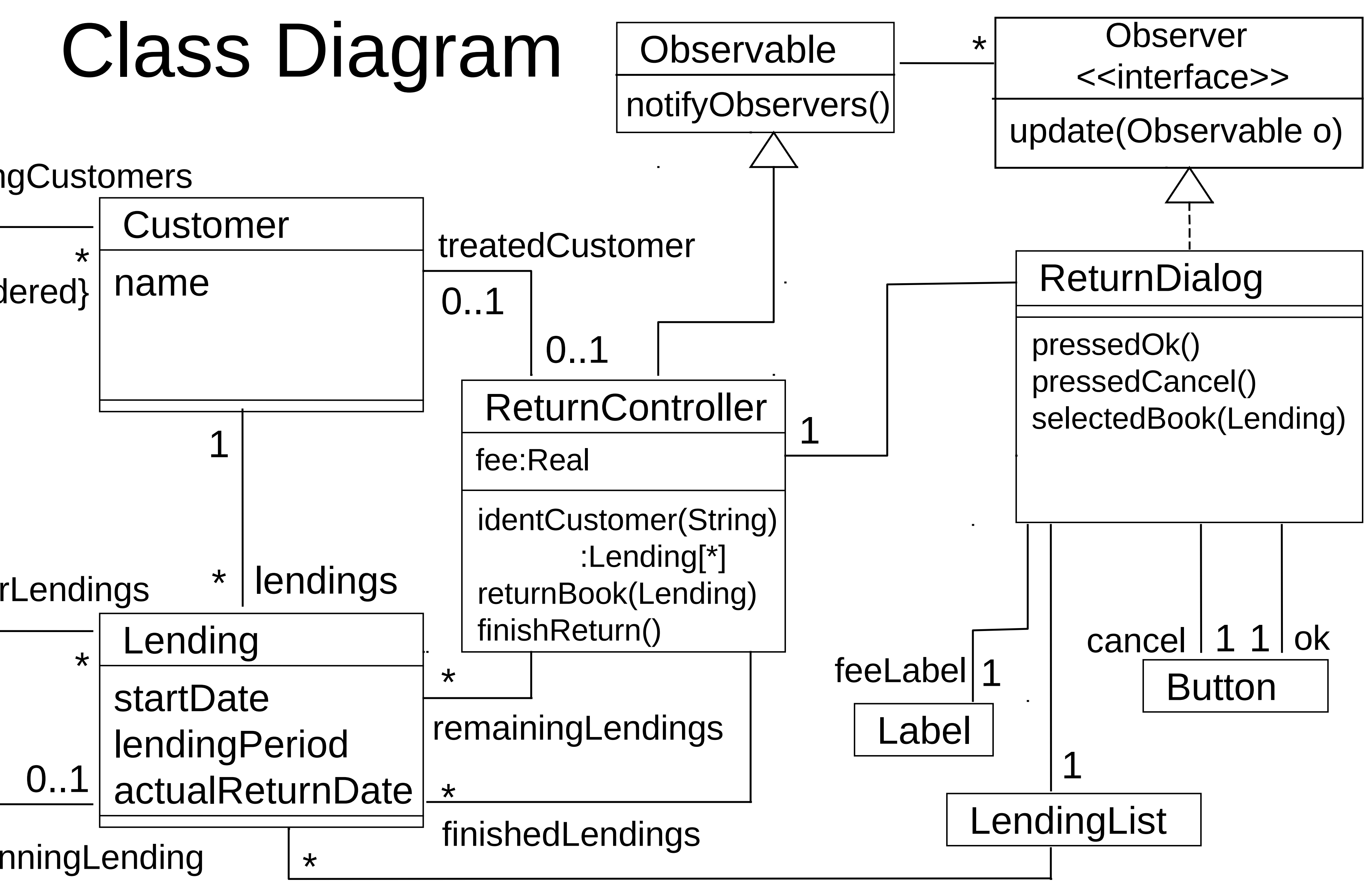
# Example

## Domain model to class diagram

# Problem Domain: A Library



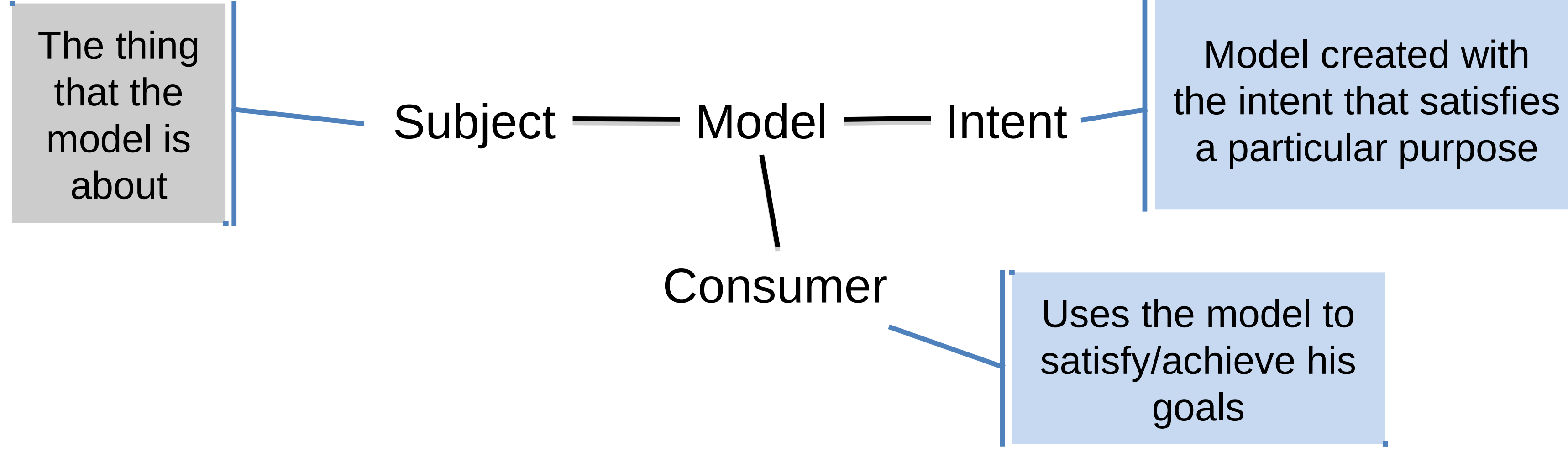
# Class Diagram



# Reflection

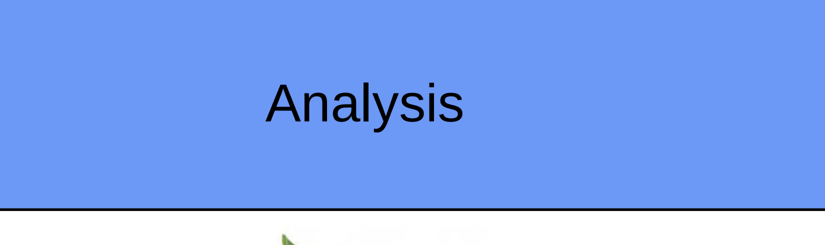
- Both use the syntax of class diagram
- But, different:
  - Subject
  - Consumer
  - Intent

# Models



# Gap between analysis and design

A lot of creativity to produce the analysis



Design

But also a lot of creativity to produce design from the analysis

# Analysis

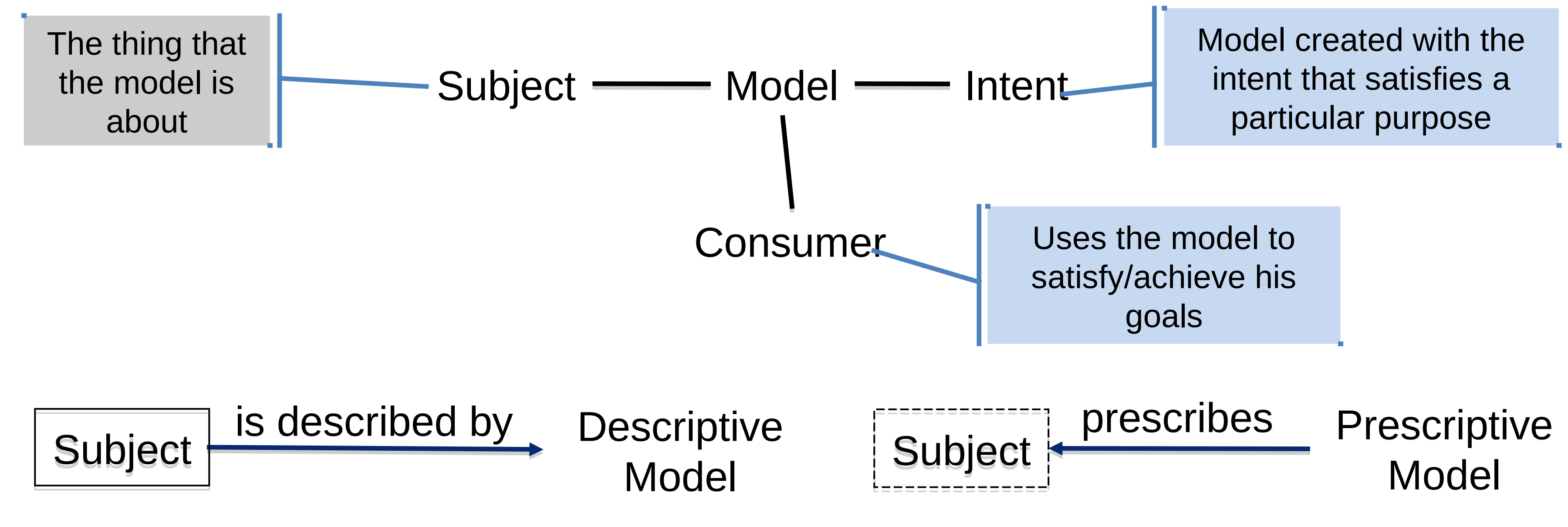
- So far we have done:
  - Requirements
  - Domain Models
  - Use Cases
- These models are based on:
  - Interviews
  - Observations
  - Workshops
  - Looking at similar systems

Descriptive models of the Ideas/vision of the system to be realized

Generate domain knowledge

Can other types of models be part of the analysis phase?

# Descriptive vs Prescriptive



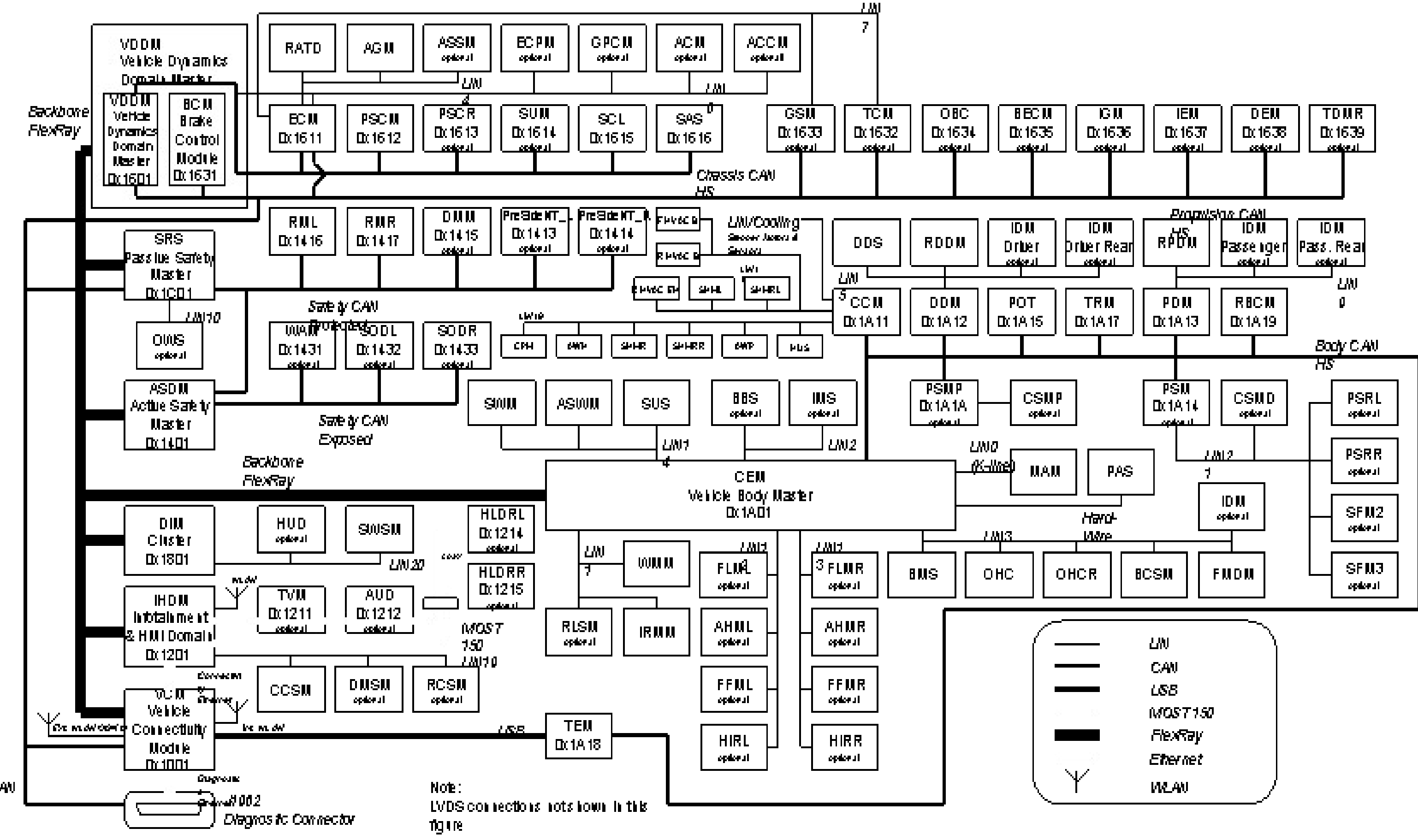

Ideas/vision of the system to be realized — is described by → High-level Architecture

System to be realized ← prescribes — Working Architecture

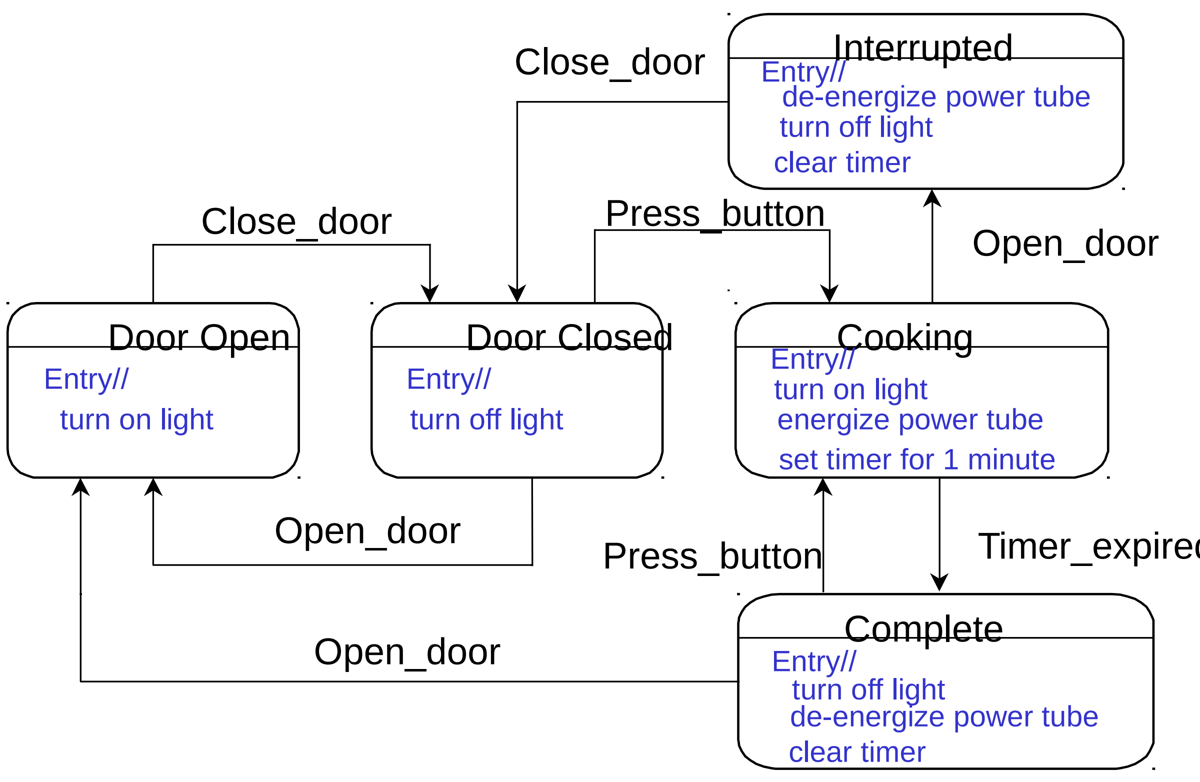
# Complex system

- Might need other types of diagram in the analysis phase:
  - Components
  - Nodes
  - State machines
  - …

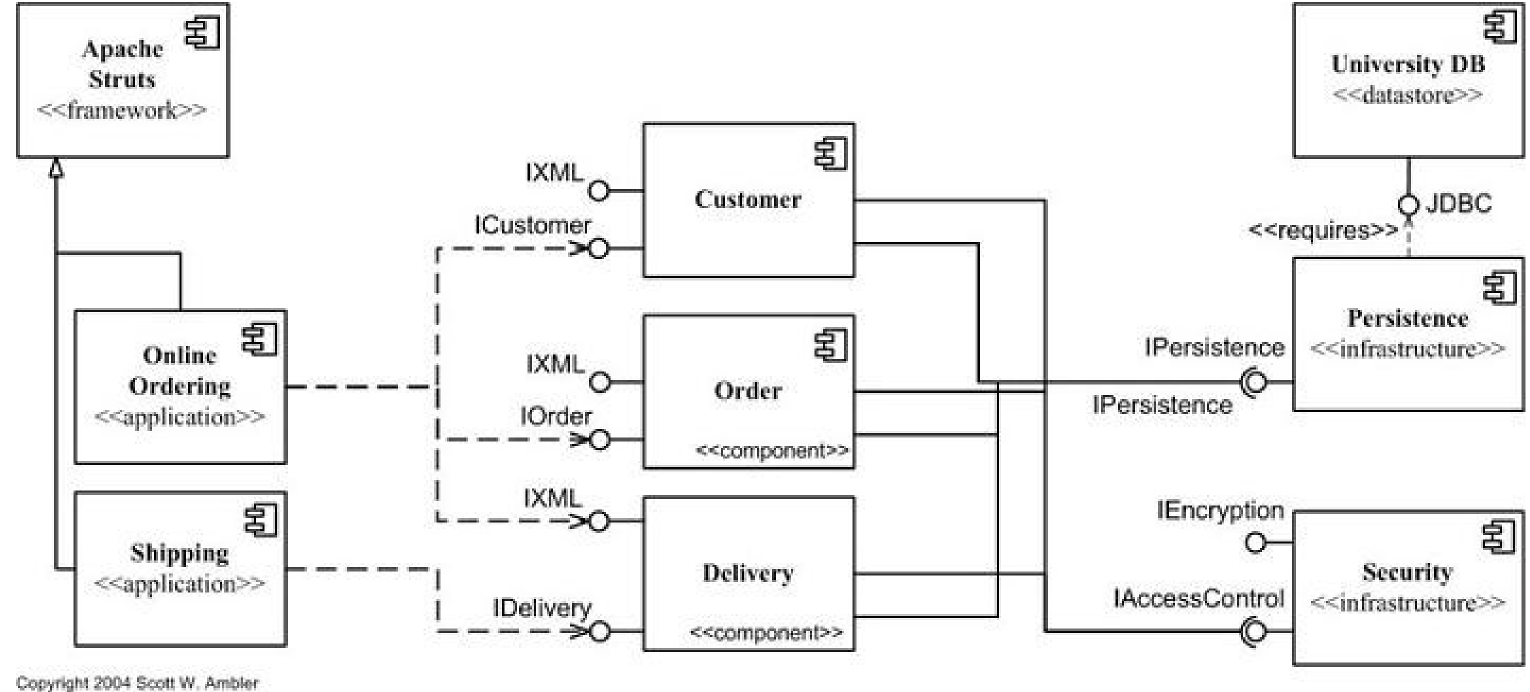
# Example: show the ECUs in a car



# Example: State machine to explain the behavior of an oven



# Example: component diagram to split up a large system

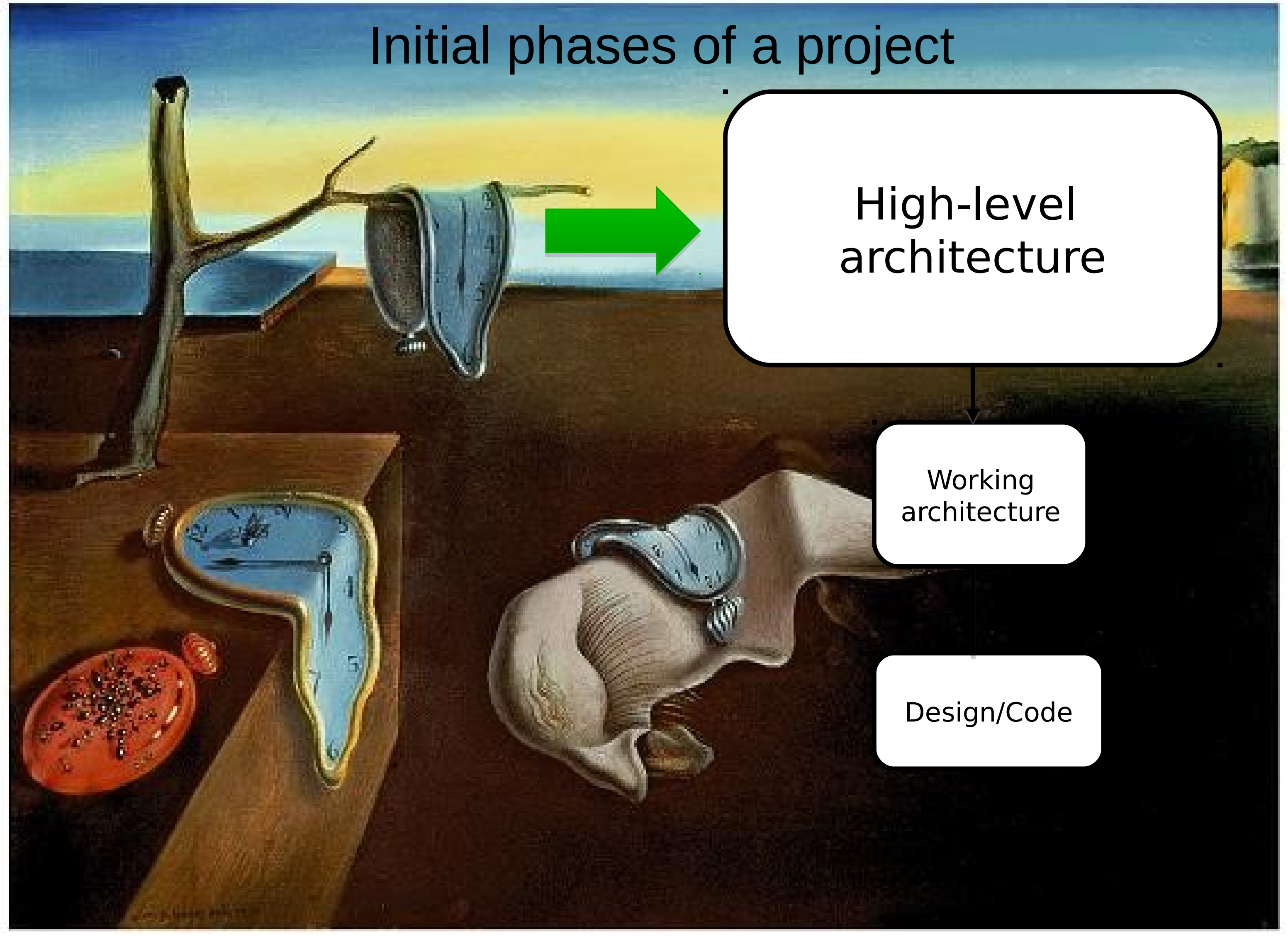


# GAP

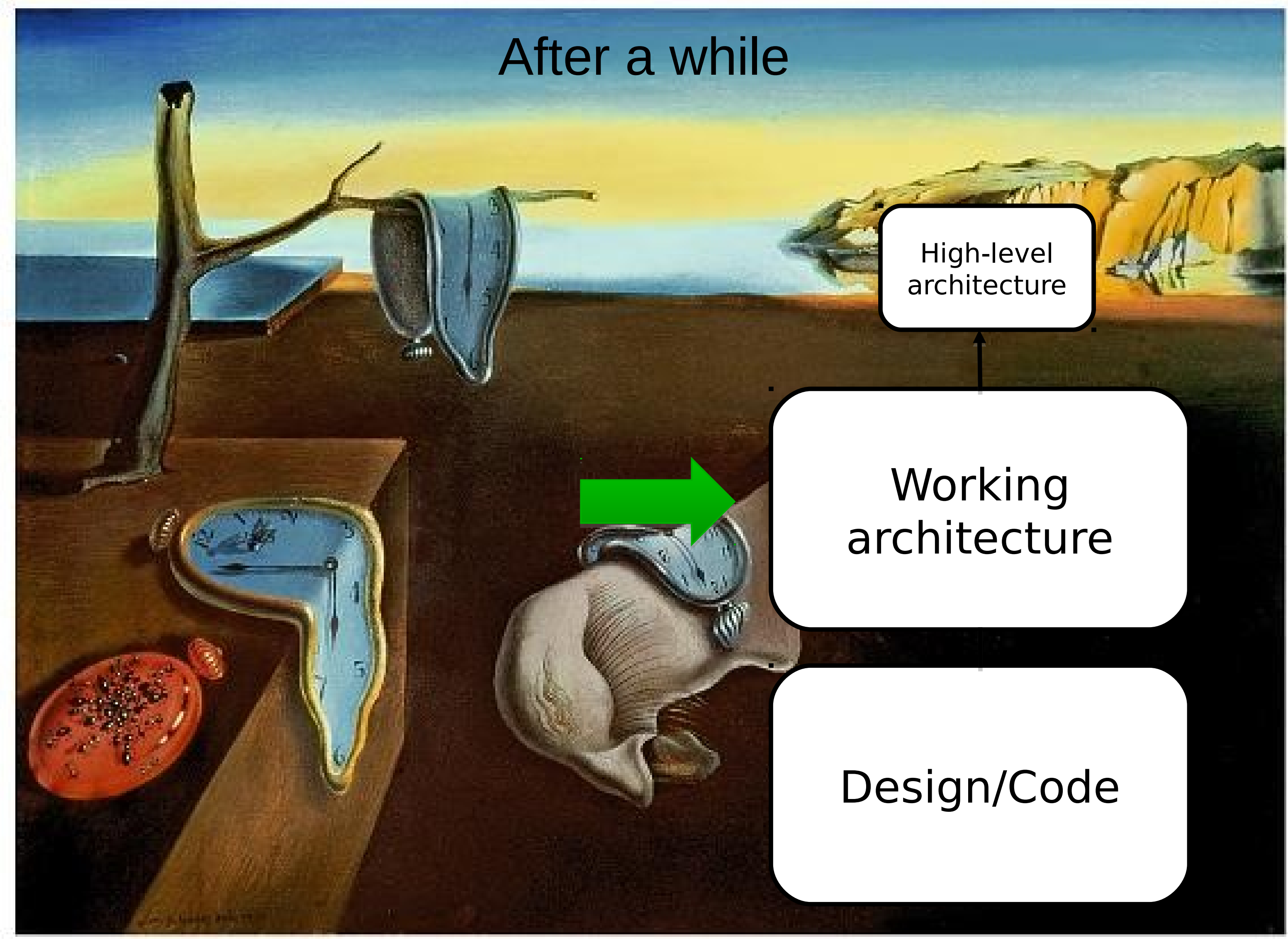
Descriptive
models of the
Ideas/vision
of the system to
be realized

Prescriptive
models to
develop the
system even
through
automated code
transformations

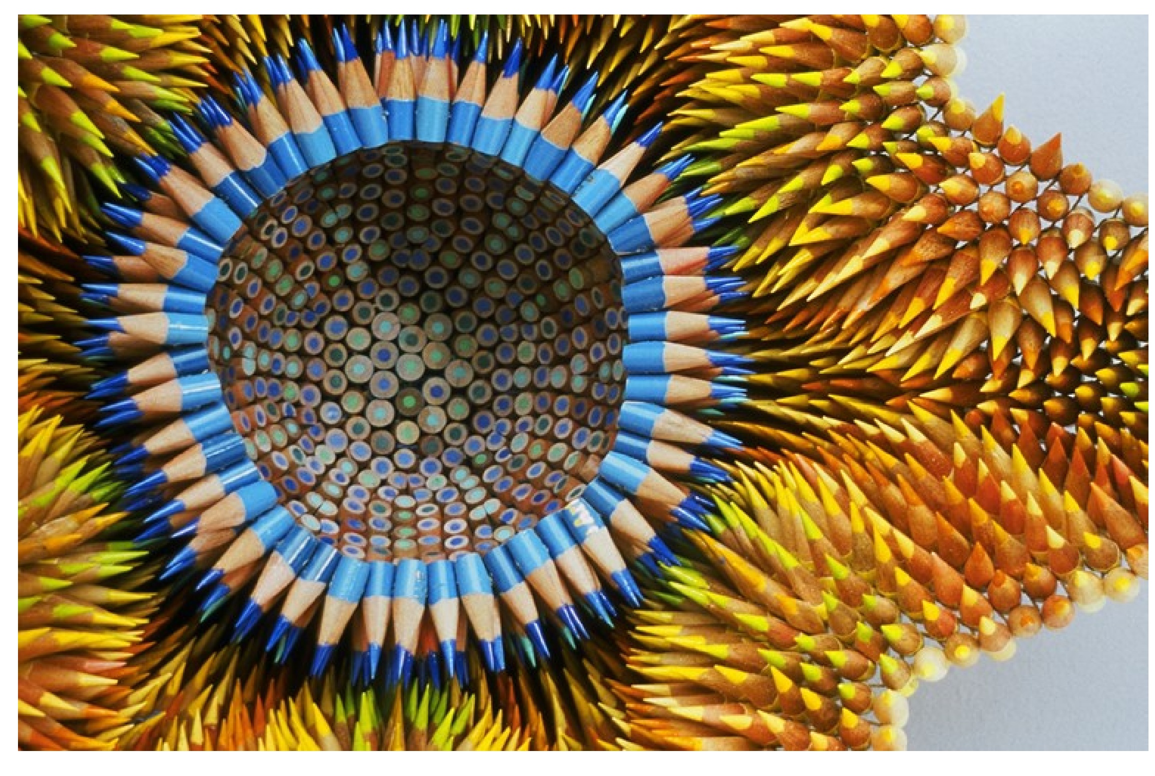
# A question of time



# A question of time



# Problems of the high-level architecture



Too many details

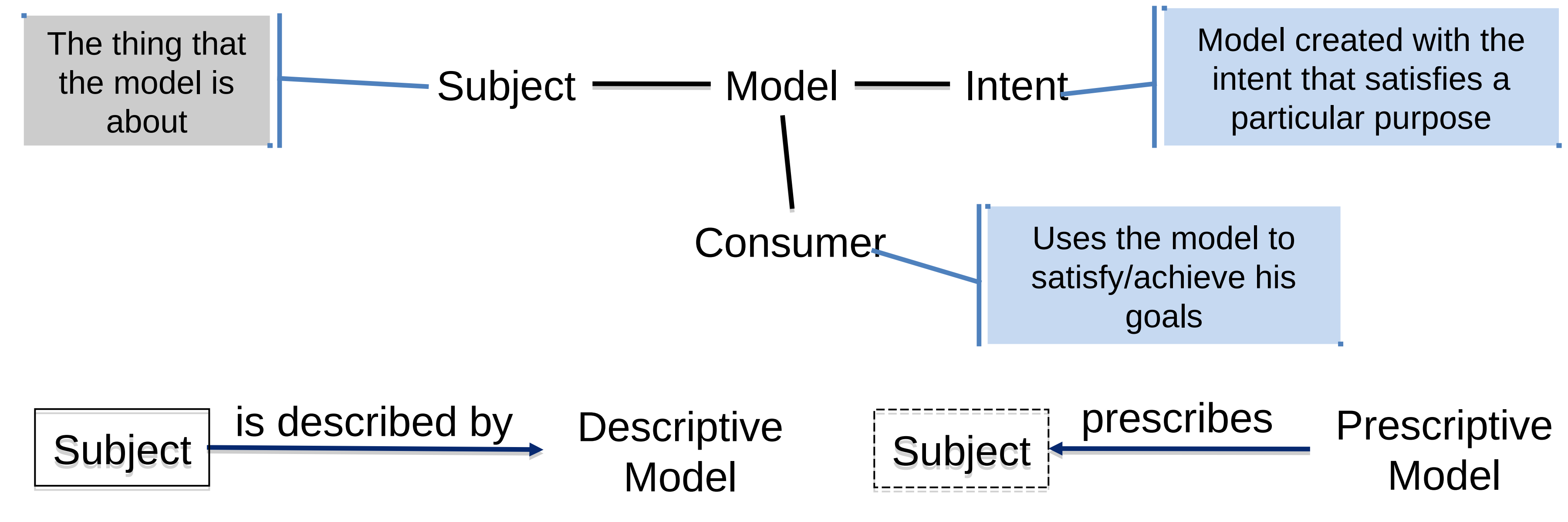
# Problems of the high-level architecture

Easily becomes
out of date

# Problems of the high-level architecture

Present and Future mixed in the same document

# Descriptive vs Prescriptive




Ideas/vision of the system to be realized

is described by

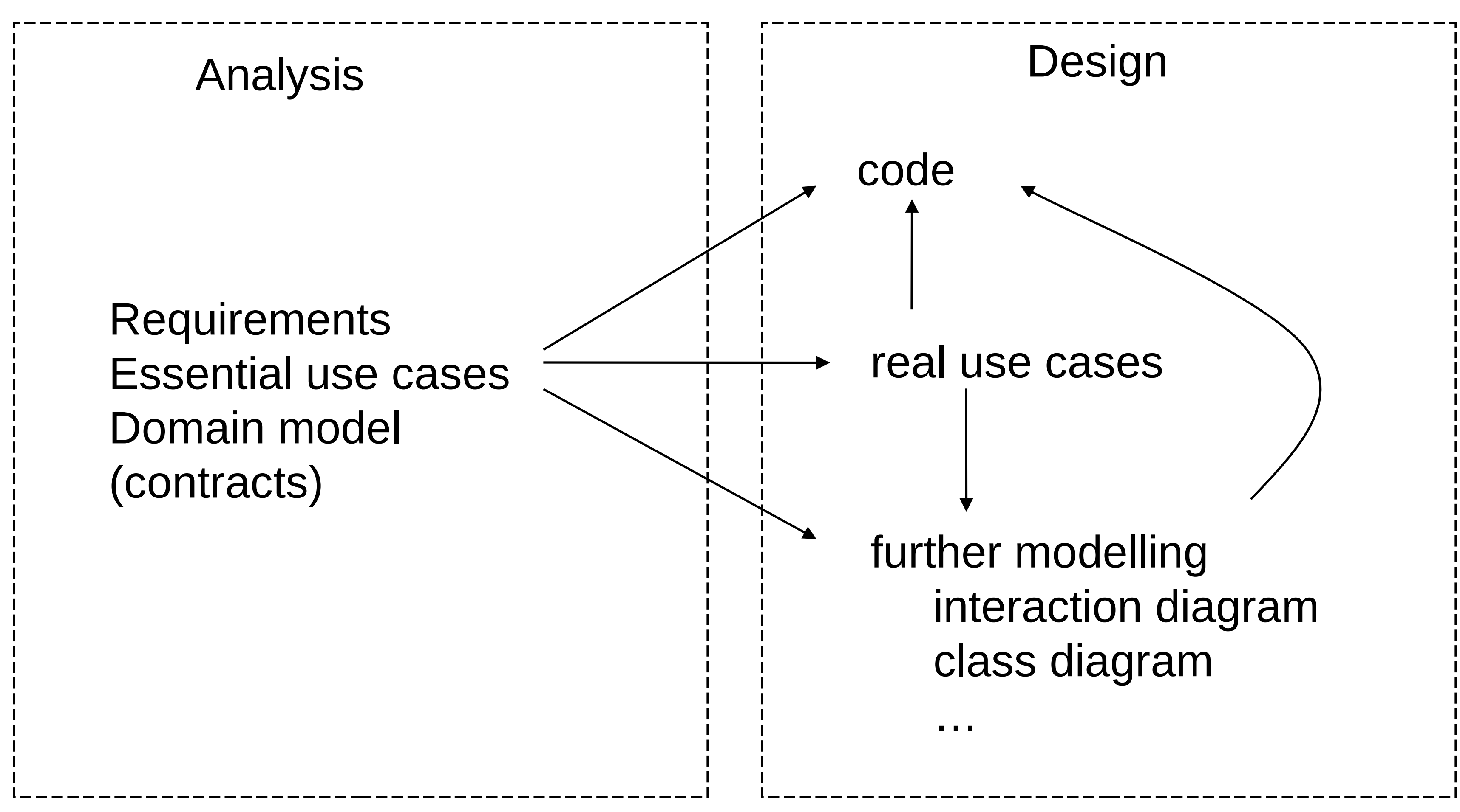
High-level Architecture

System to be realized

prescribes

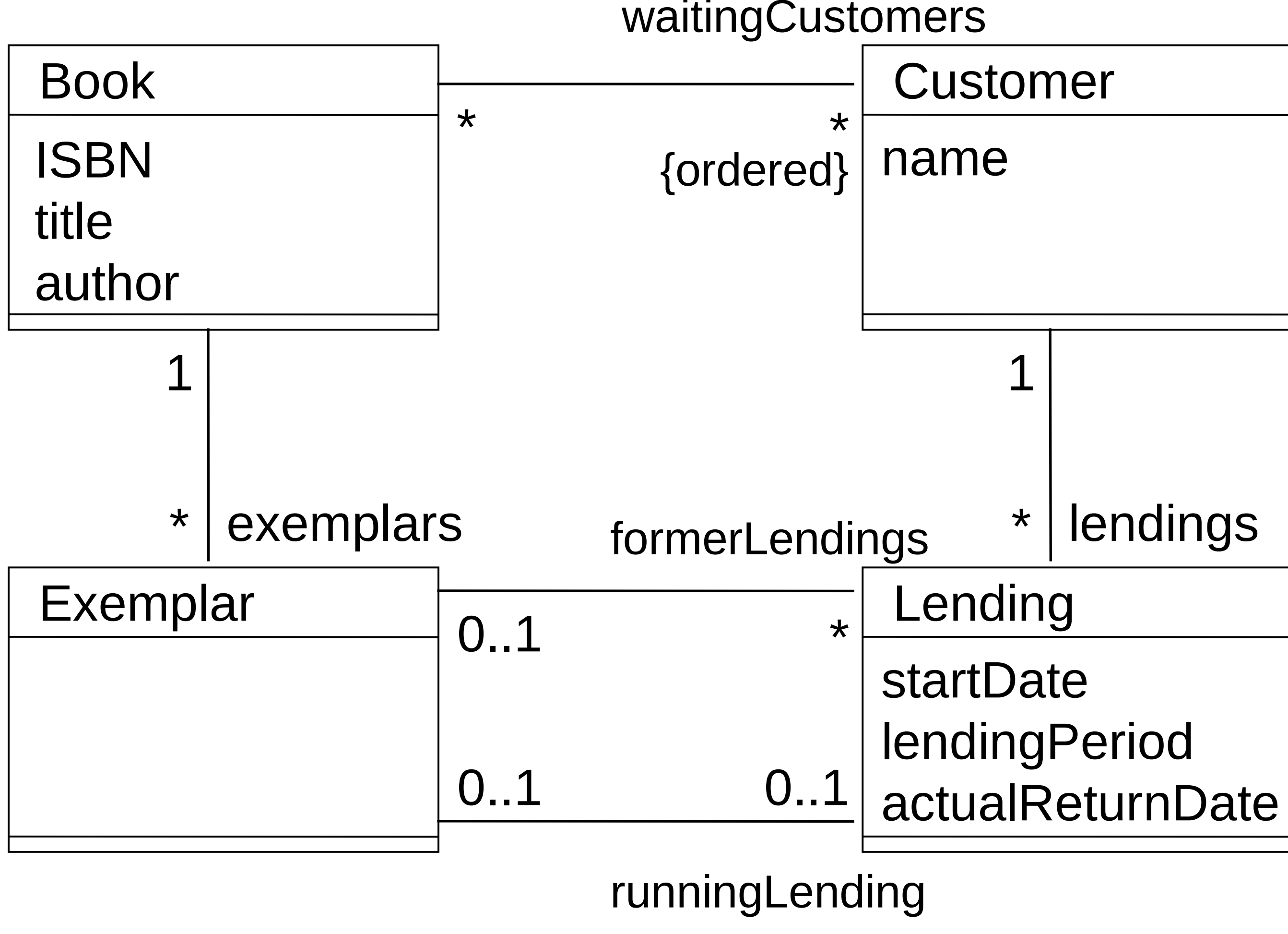
Working Architecture

# What next?

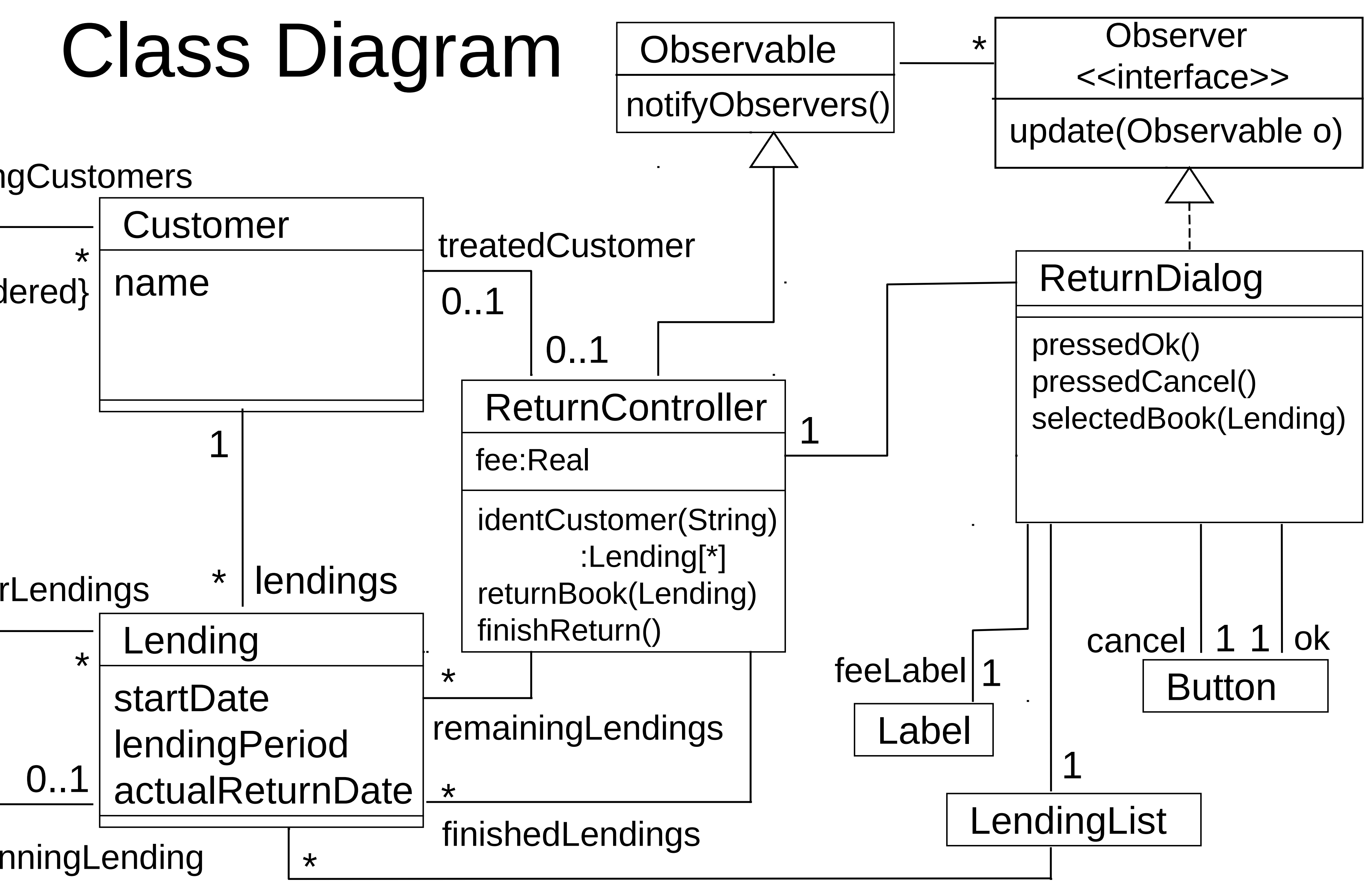


# Classes

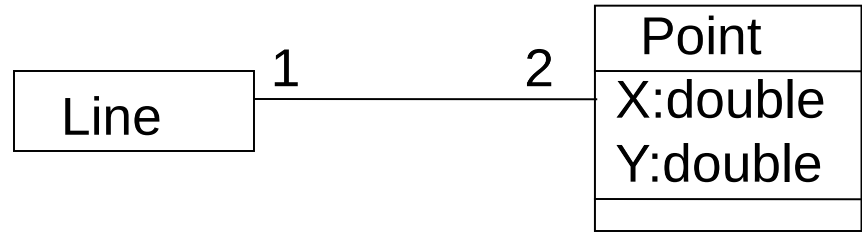
# Problem Domain: A Library

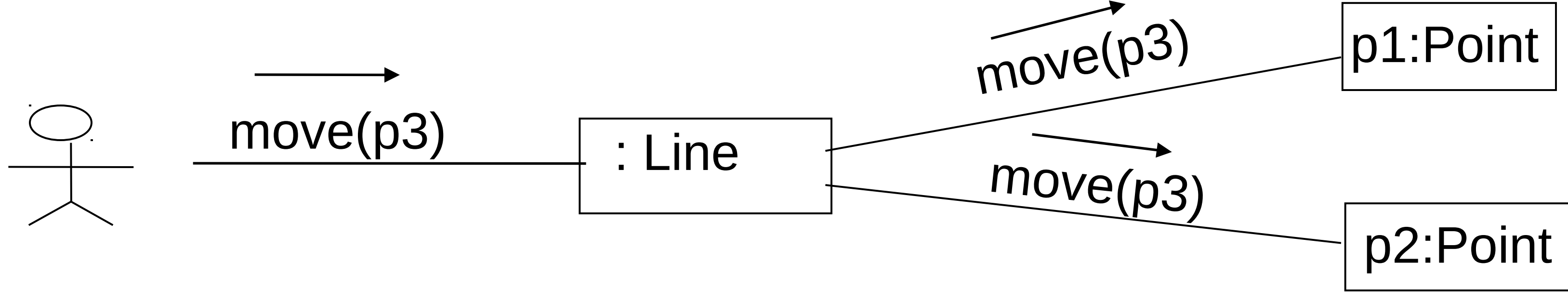


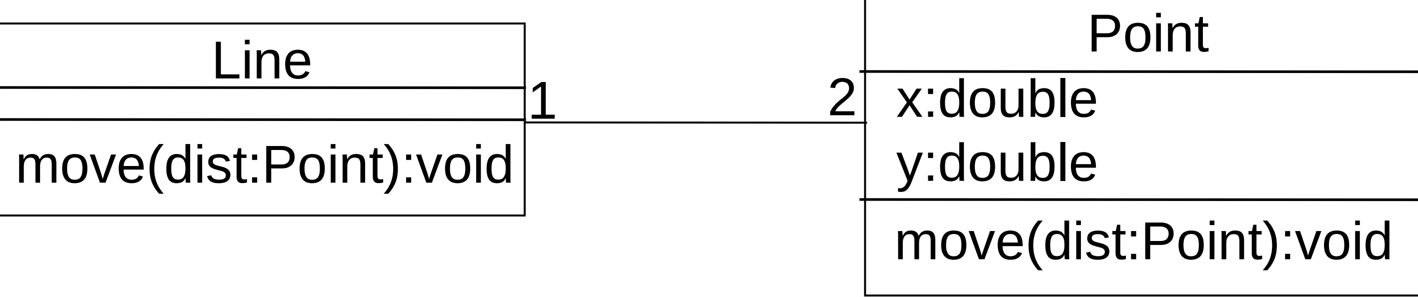
# Class Diagram



# Obtaining operations

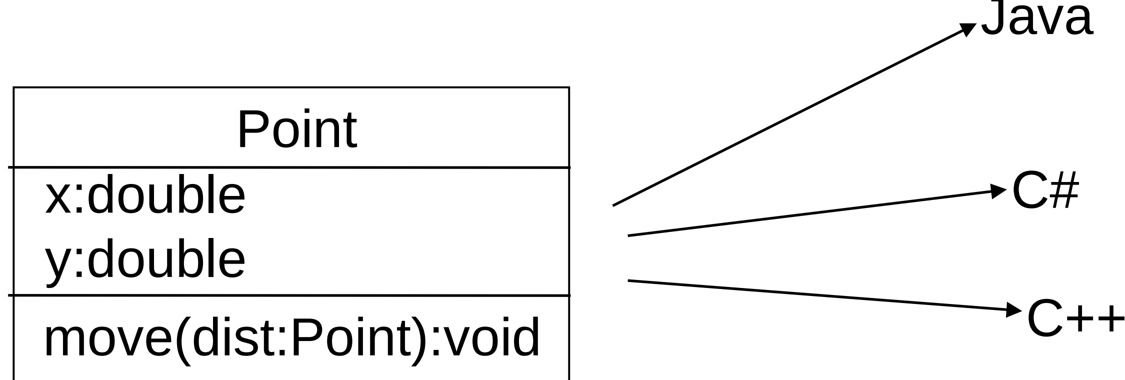


# Mapping to code

- One can map a UML class to many different code skeletons in different programming languages such as:



# UML Classes: Visibility

| Point |
| --- |
| - x:double<br>- y:double |
| + move(dist:Point):void |

Mapping visibility to java:

- **-** **->** **private**
- **#** **->** **protected**
- **+** **->** **public**
- **~** **->** **package**

(In this case the semantics of -,#,+,~ will be the ones of Java.)

# UML attribute

**UML:**

[visibility] name [multiplicity] [:type] [= initial value] [{properties}]

Properties could be:

- **changeable** (Variable may be changed.)
- **addOnly** (When **multiplicity** is bigger than one you can add more values, but not change or remove values.)
- **frozen** (Cannot be changed after it has been initialized.)

- **Example:**
  - **x : int {frozen}**

# Operations/methods

**UML:**

[visibility] name [(parameter list)] [: return type] [{properties}]

You can have zero or more parameters. Syntax for parameters:

[direction] name : type [= default value]

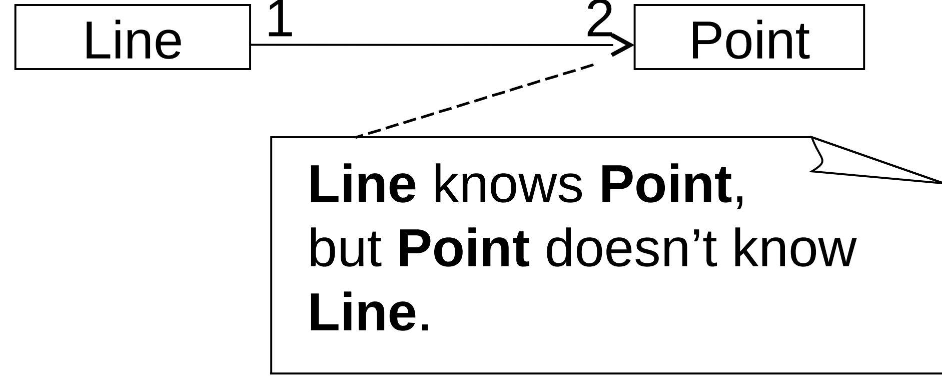
- direction: in, out, inout

- Example of a property
  - **isQuery** (no ”side effects”)

# Relations

- All the associations we consider when drawing domain models can also be used in class diagrams.
- But there are some interesting issues to consider …

# Navigability



# Association constraint

**Constraint:**

- **changeable** (Links may be changed.)
- **addOnly** (New links can be added by an object on the opposite side of the association.)
- **frozen** (When new links have been added from an object on the opposite side of the association, they cannot be changed.)
- **ordered** (Has a certain order)
- **bag** (multisets instead of sets)
- **…**

Person {ordered} Company

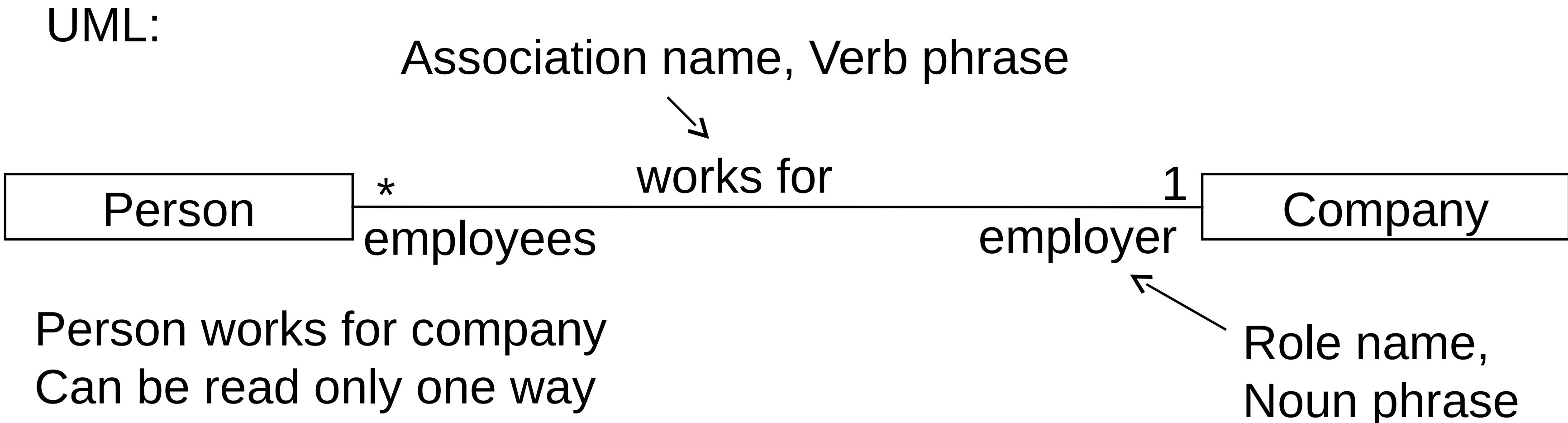
# Class methods and class variables

Account

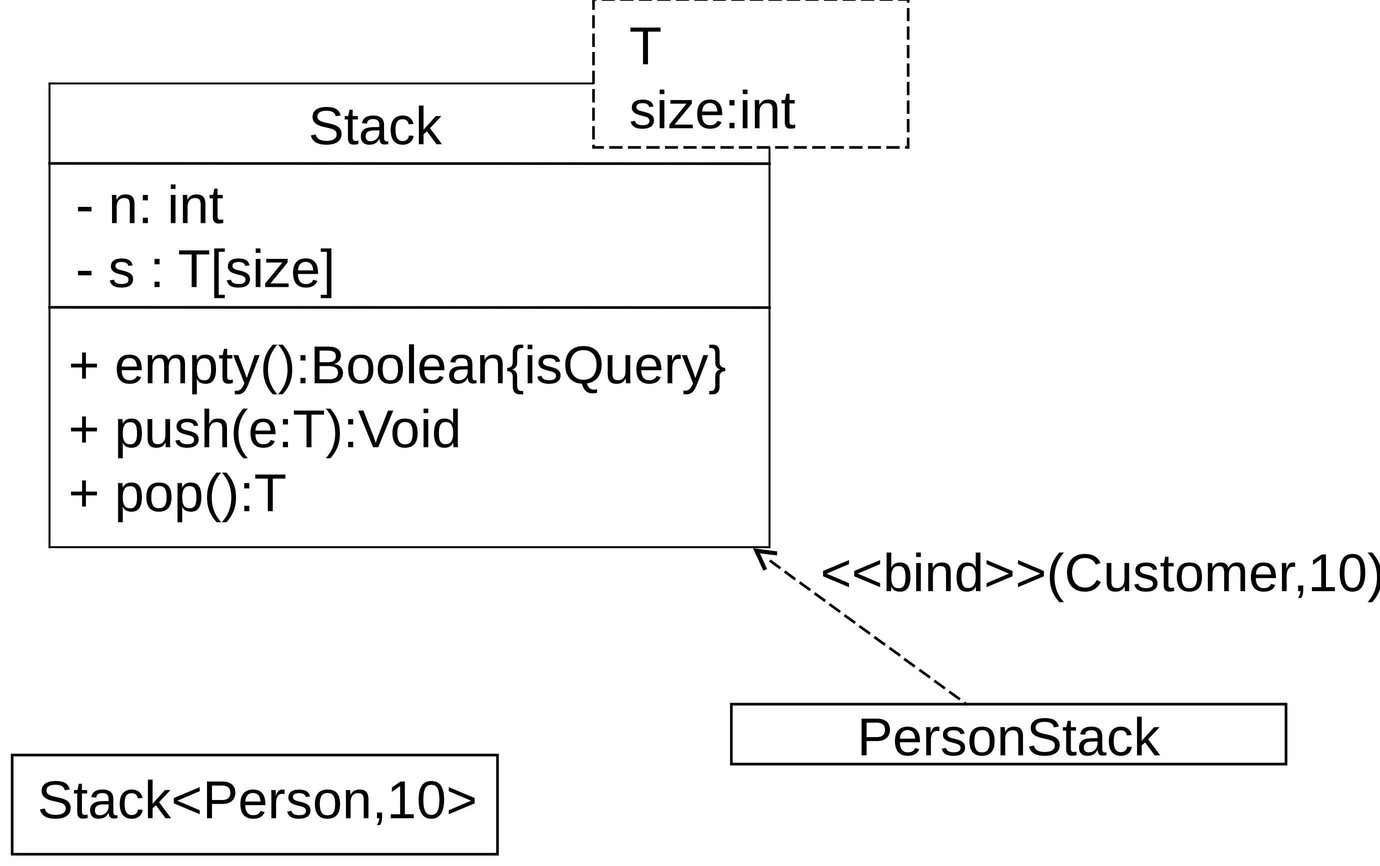
-interestRate:double
-balance:double

+changeInterestRate(newinterestrate:double)

# Association names UML



# Class templates

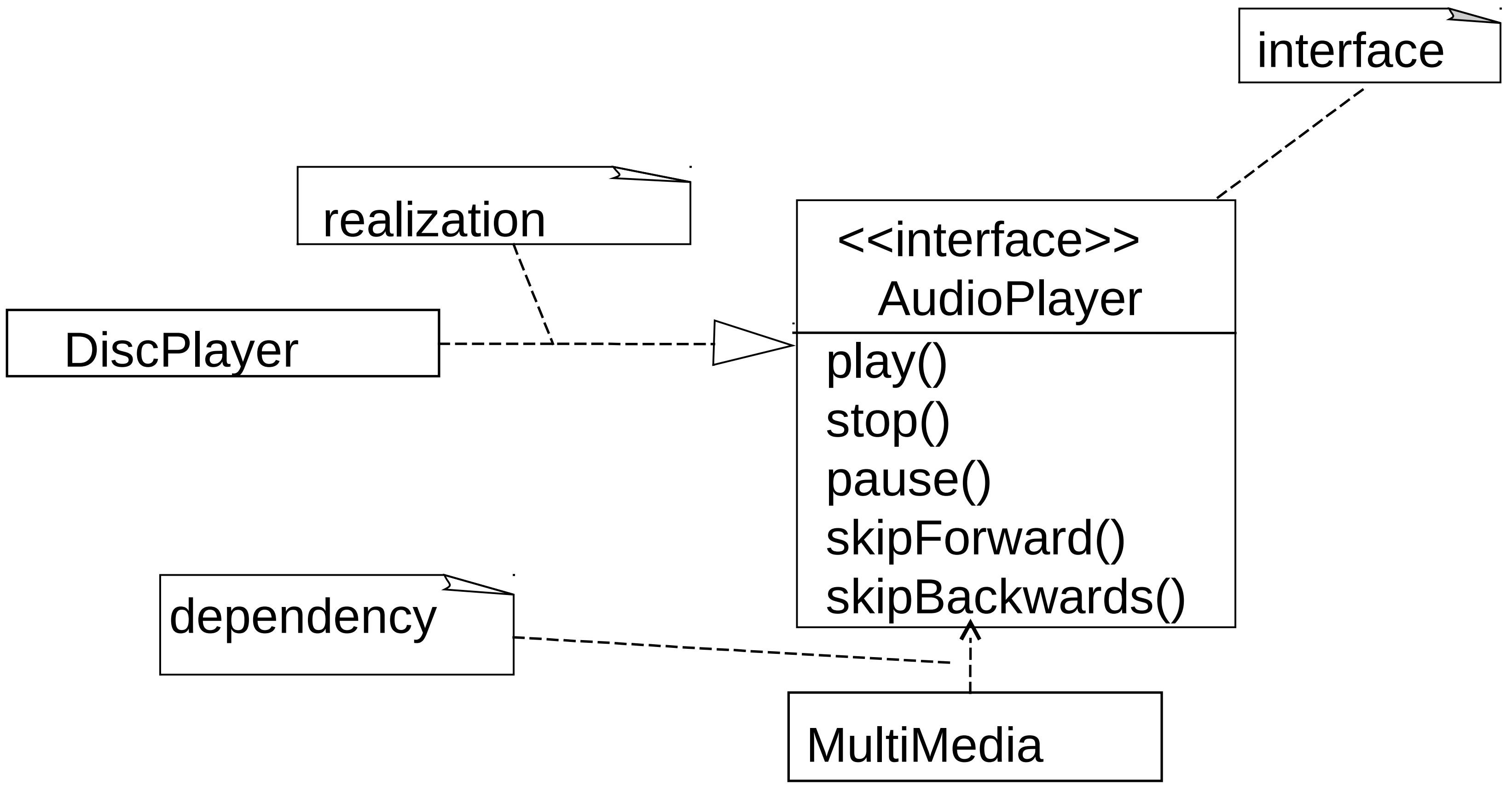


# Interfaces

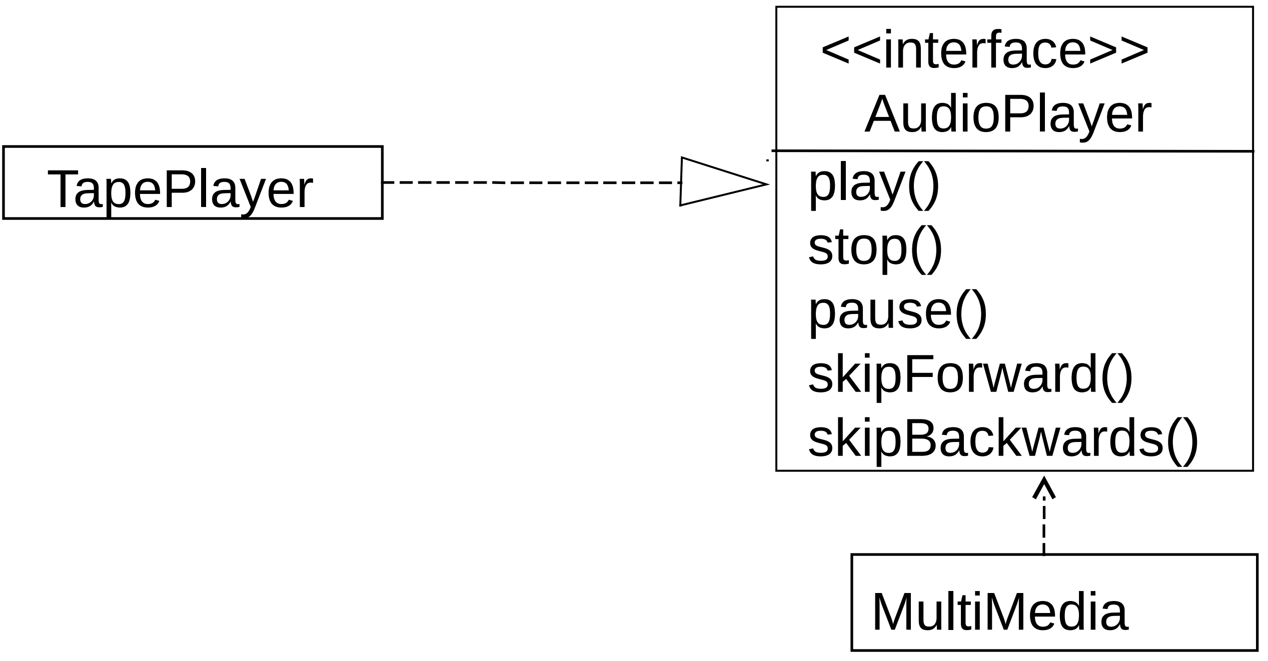
# Interfaces

- Interfaces are very important. By using an interface you can separate implementation from specification.
- An interface specifies a service of a class or component.

# Interfaces in UML

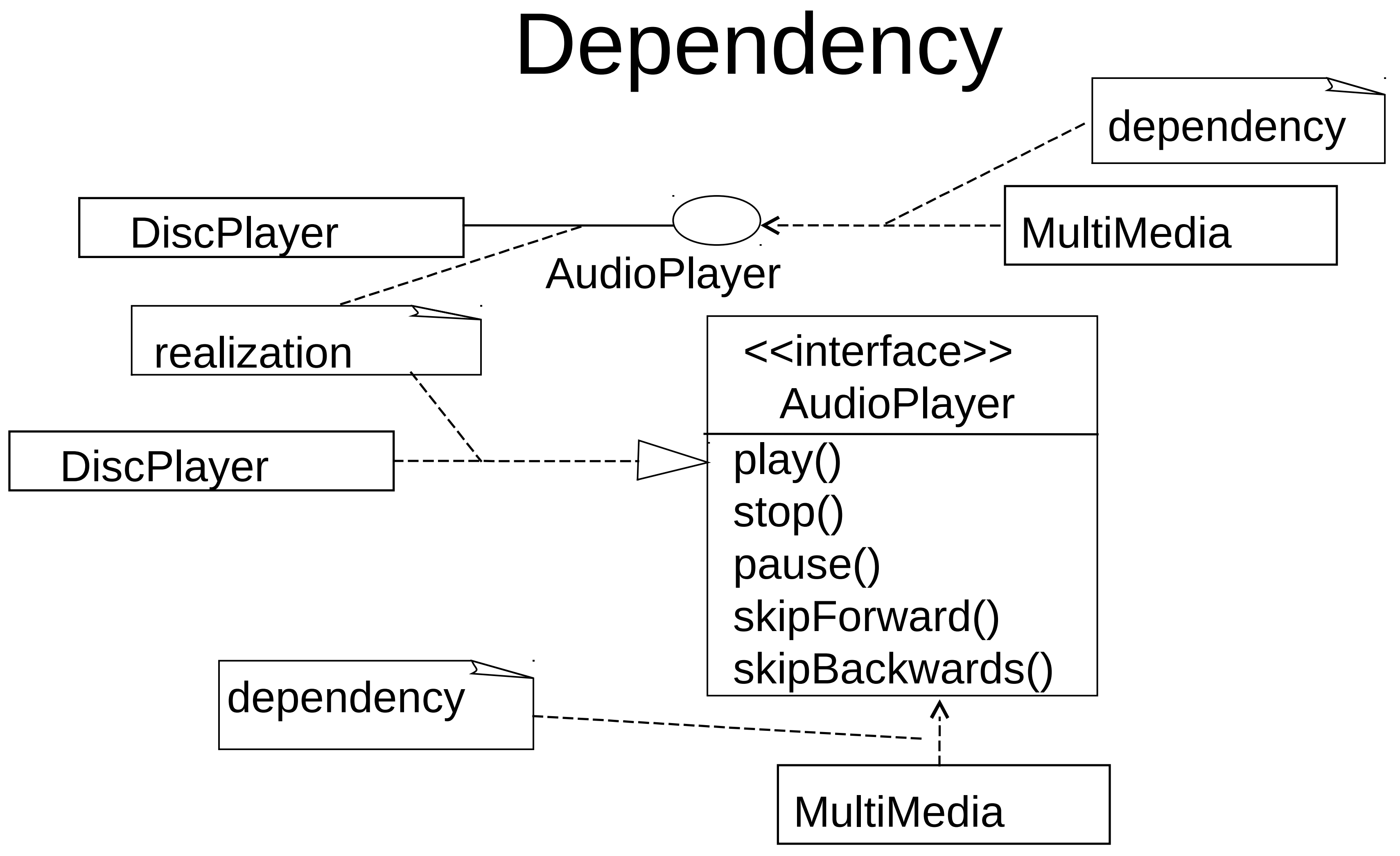


# The same interface




Here **TapePlayer** is a new implementation of **AudioPlayer**. If you have done everything correctly you only have to change the implementation of the methods in the interface, the rest of the program remains the same.
The **MultiMedia** doesn’t need to be changed!

# Dependency



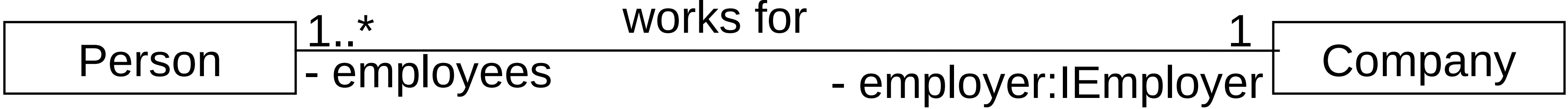

The class **MultiMedia** uses the methods in the interface, which is implemented by **DiscPlayer**.

# Interface Specifiers

<<interface>>
IEmployer

getCompensation()
getBenefits()

Roles can be shown using interfaces.

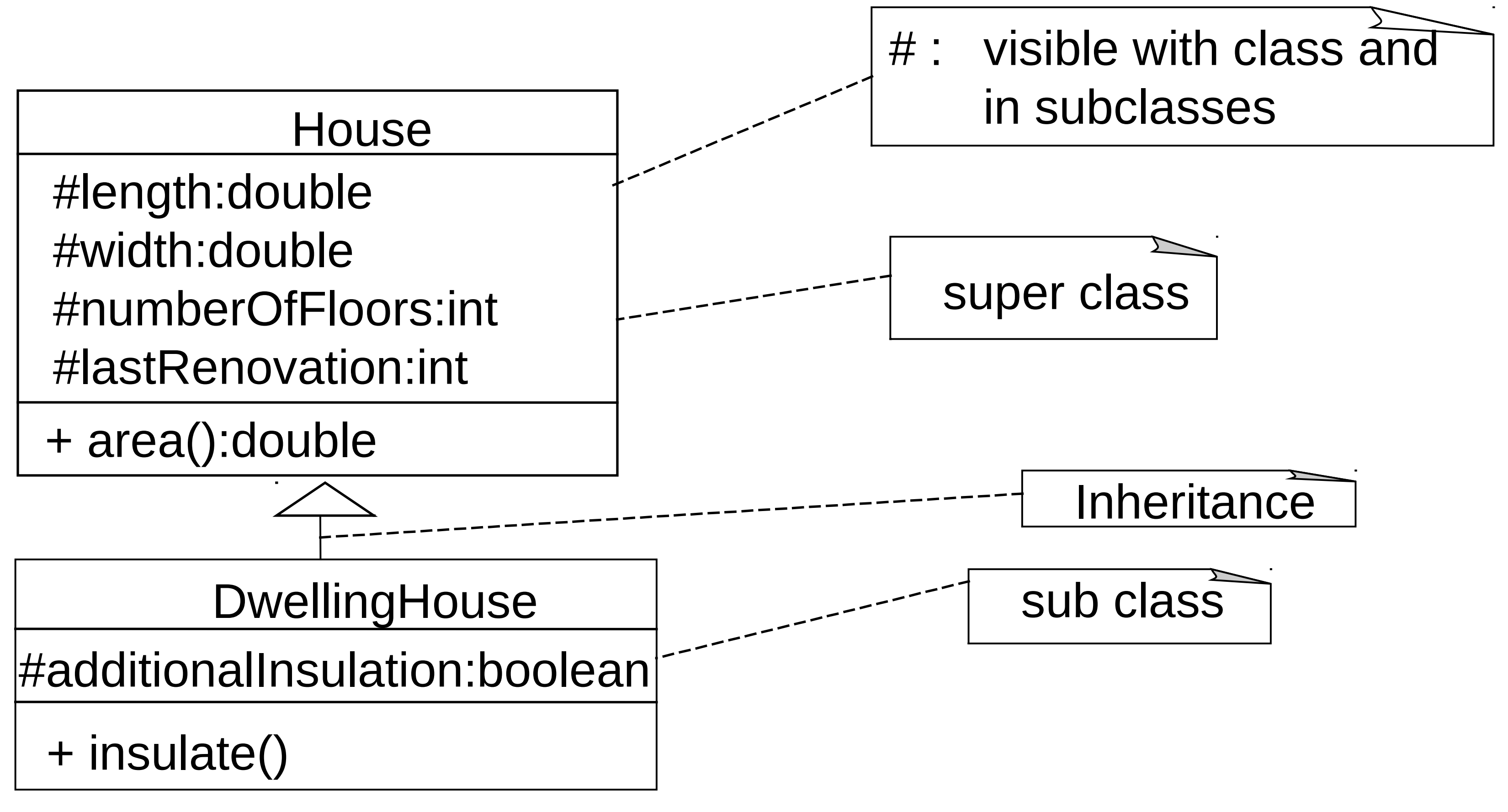



A person can have many other roles, such as customer, boss, father, pilot etc.

- worker:IEmployee
*
1
Person
- supervisor:IManager

# Inheritance

# Example: Dwelling-house



# Instances

Sometimes you want to work with instances of House and sometimes with instances of DwellingHouse etc.

| :House |
| --- |
| length = 20<br>width = 15<br>numberOfFloors = 2 |

| :DwellingHouse |
| --- |
| length = 30<br>width = 20<br>numberOfFloors = 3<br>additionalInsulation = true |

# leaf: stops inheritance

```
public final class A {
      ...
}
```

A
{leaf}

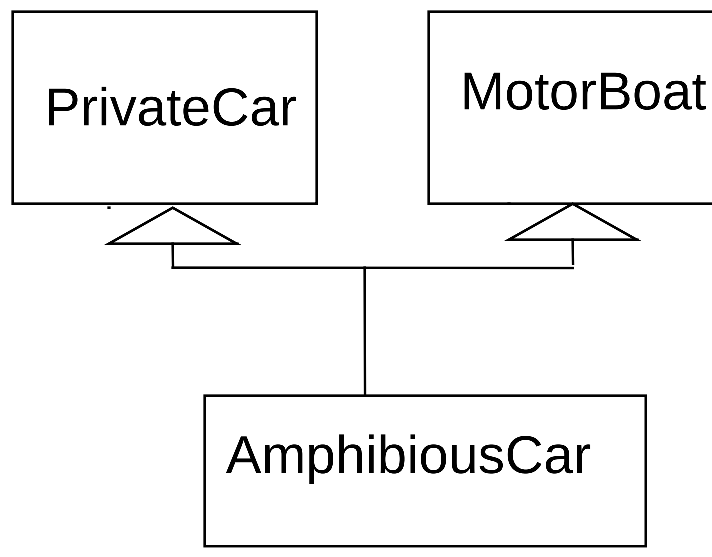
B

Not allowed!

Note that also a method can be final. Then the method must not be changed in the sub classes, e.g.

```
public final int test (int x) {
      ...
}
```

# Multiple inheritance

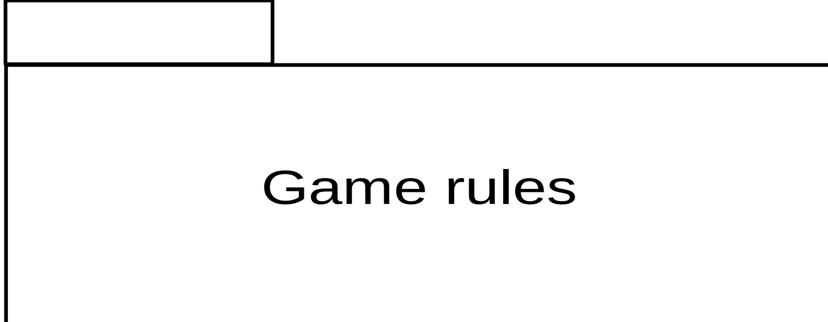



- This is allowed in C++, but not in Java.

  (But: For interfaces in Java multiple inheritance is allowed)

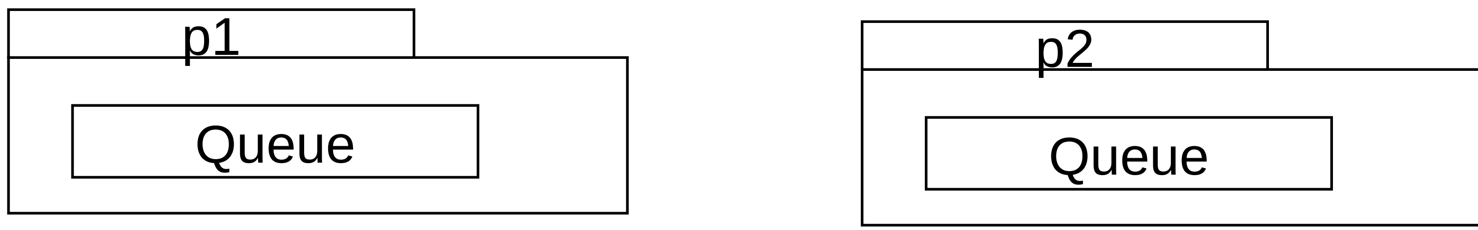
# UML: Package

- In UML, one can use packages to group elements, for example group use cases, classes, components etc.



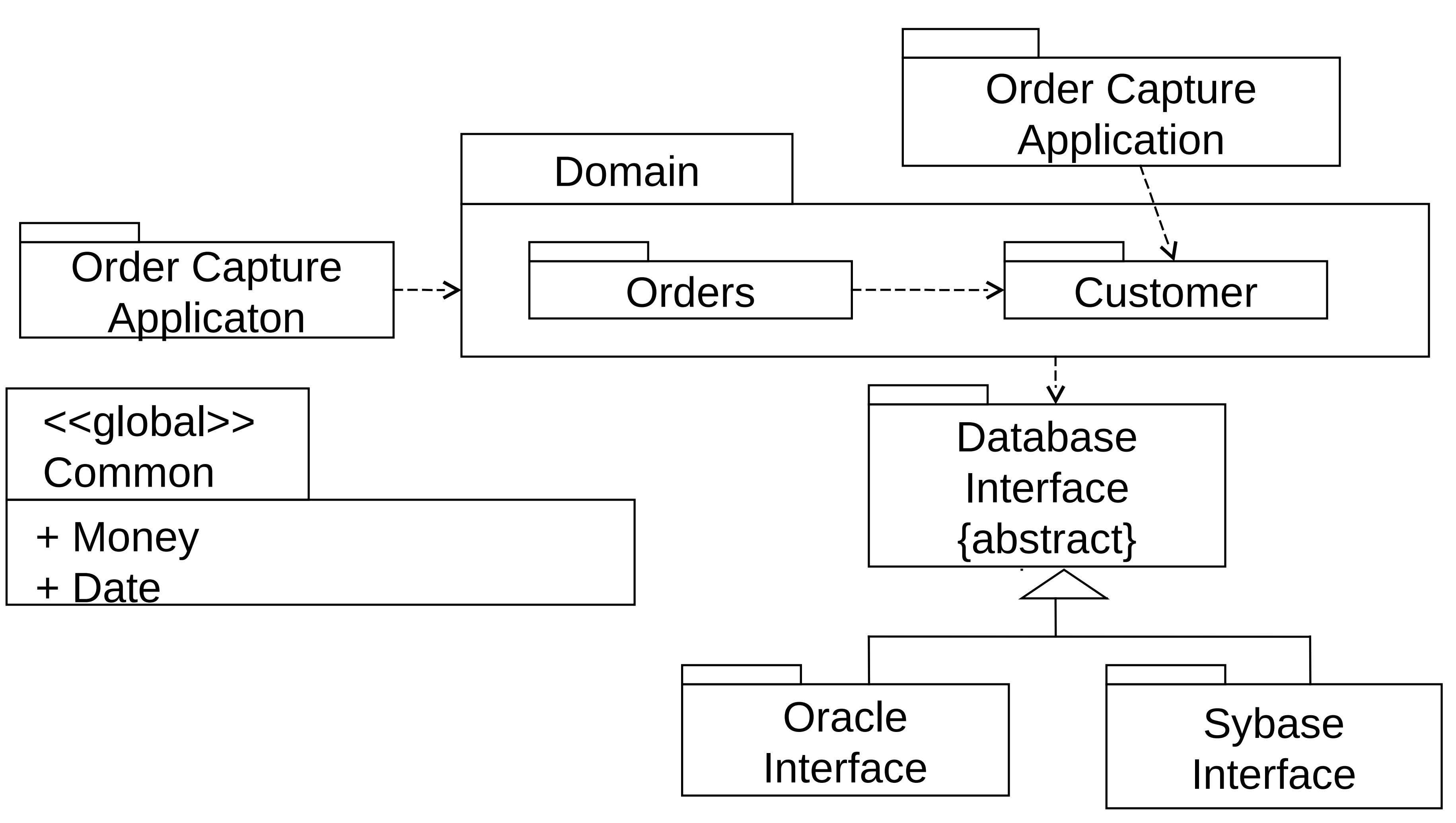
# Name conflicts

- One can resolve name conflicts with packages, for example:

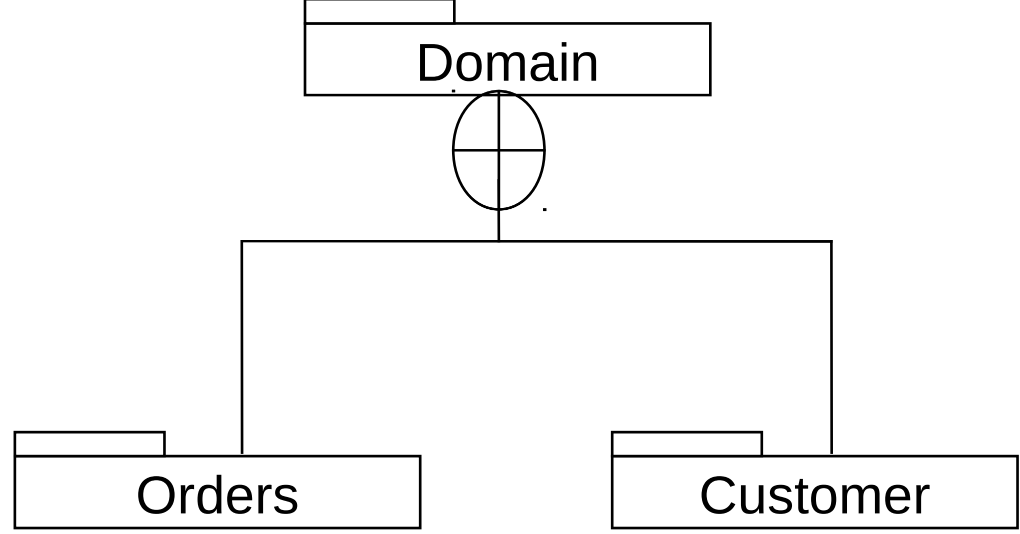



- **p1::Queue** and **p2::Queue** are two different classes with the same name.

# Package Diagram

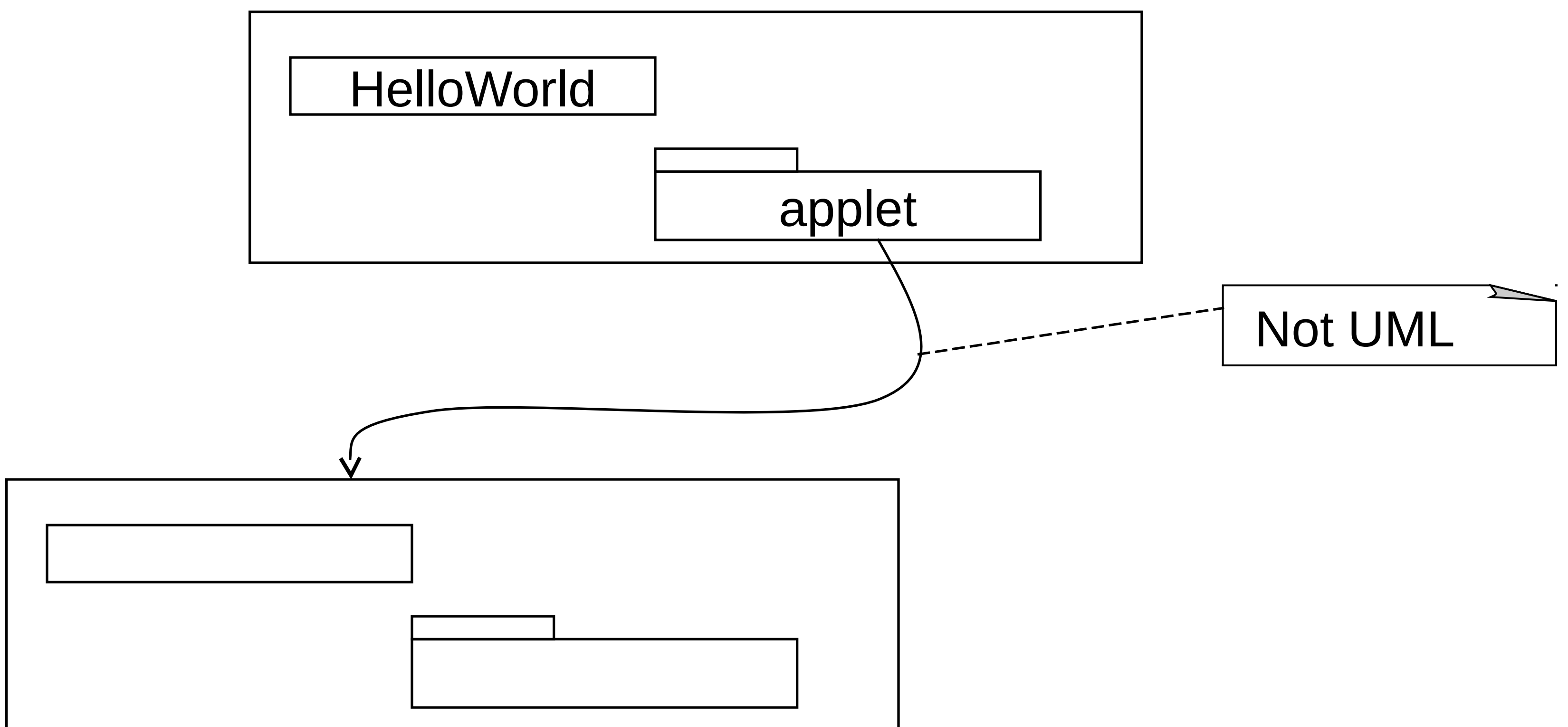


# A hierarchy of packages

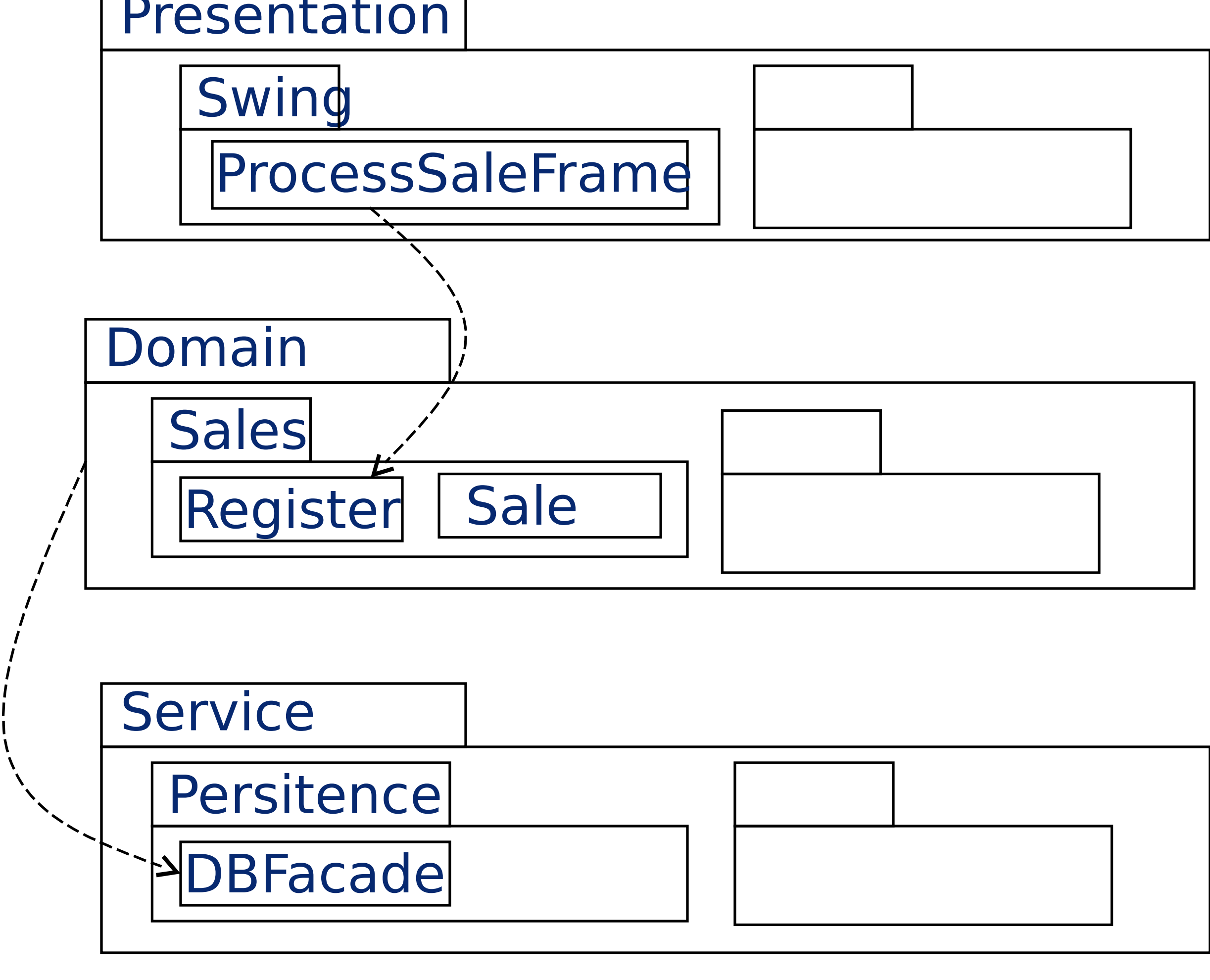


# Tools

In tools one usually does not visualise the contents of a package, rather one has a link to a file showing the contents.

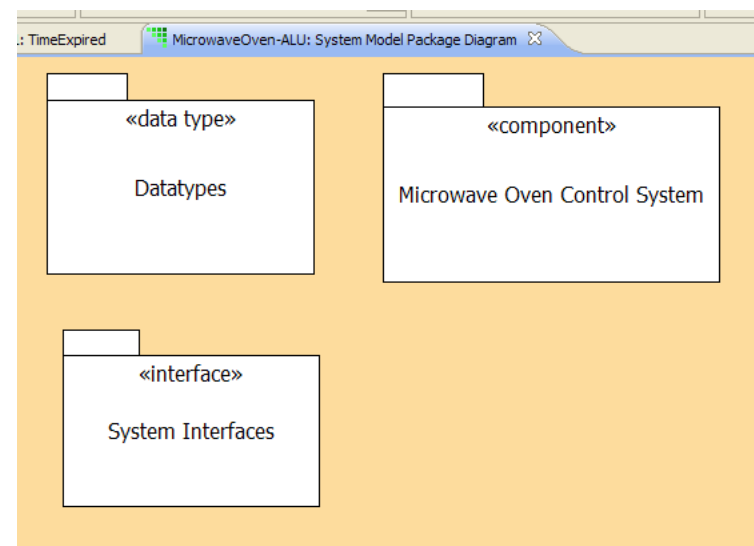


# Combining architectural patterns: Layers and Call-Return Systems



# Packages

- Currently, can contain
  - Package
  - Activity, Communication, Sequence, Use Case
  - Component, Interface, Data Type
- Currently, transparent
  - No namespace
  - No limitation on visibility
- Future, per UML
  - Namespace
  - Visibility controls
  - Separate diagram and package concepts



# Components

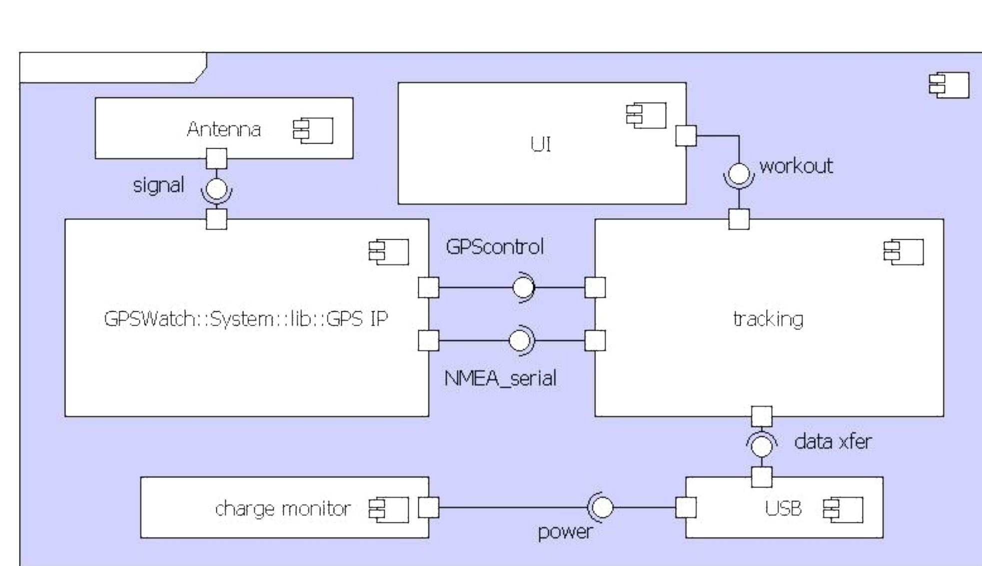
- There are many definitions for “component”, but Clements Szyperski probably gives the most well-known:
  - A software component is a unit of composition with contractually specified and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

# UML 2.0 Component Definition

- A modular part of a system design that hides its implementation behind a set of external interfaces
- Within a system, components satisfying the same set of interfaces may be substituted freely

# Components and substitution

- A component may be:
  - Behavioral system level component
  - Implementation component
  - Test stub
  - External code
  - Others…



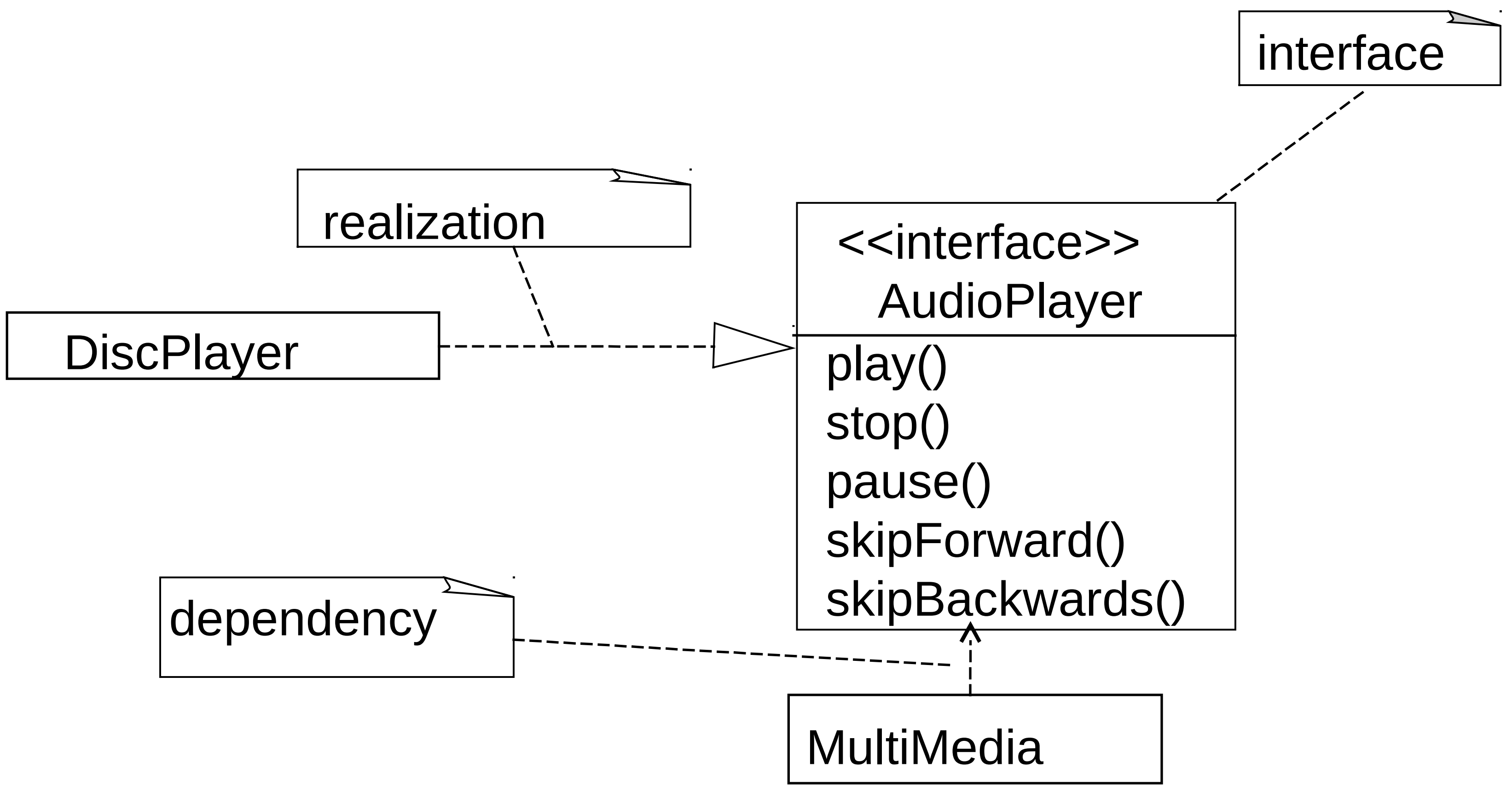
# Interfaces

- Separate implementation from specification.
- An interface specifies a service of a classifier such as a class, component or subsystem.

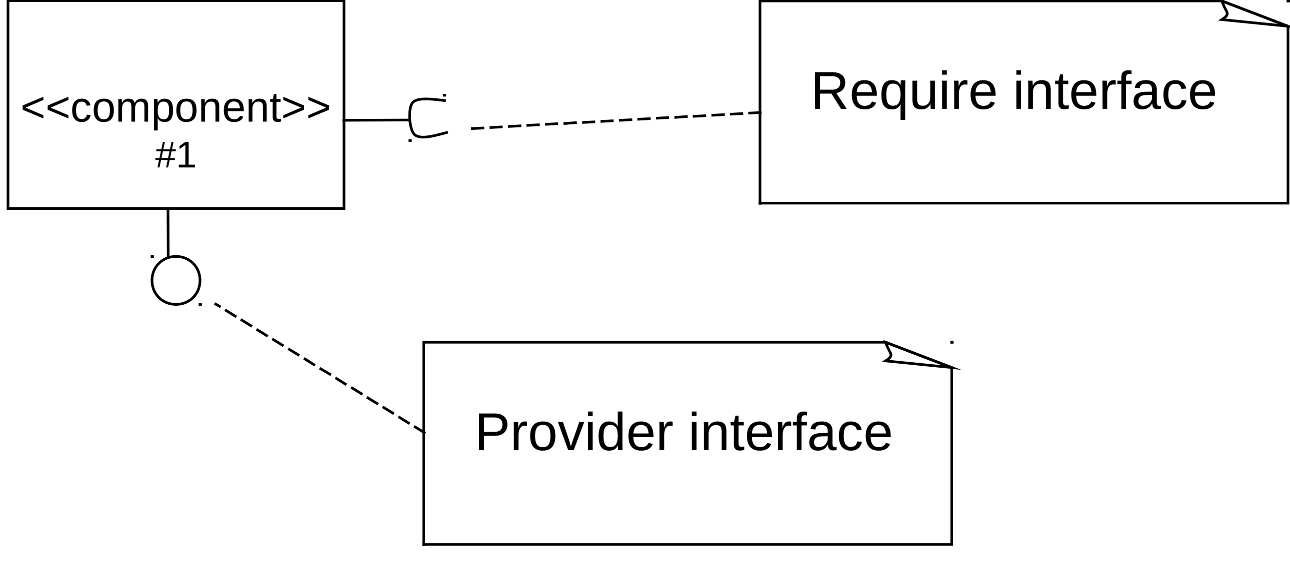
# Interface Specifies

- Operation
  - Realizing classifier must have an operation with the same signature and semantics.
- Attribute
  - Realizing classifier must have public operations to set and get the values of the attribute
- Association
  - Realizing classifier must have an association to the target classifier.
- …

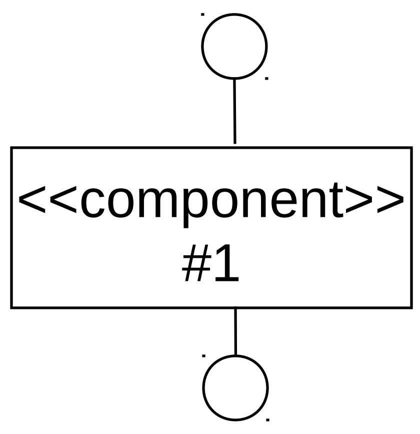
# Interfaces in UML

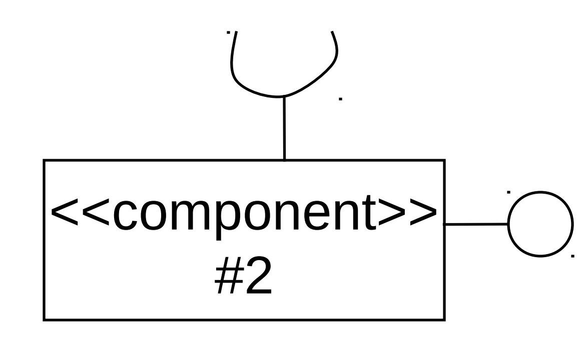


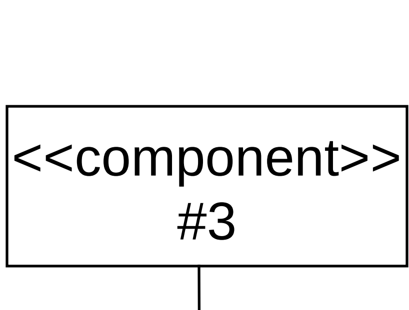
# UML Components

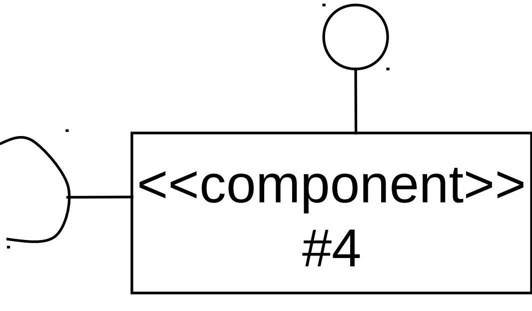


# Components

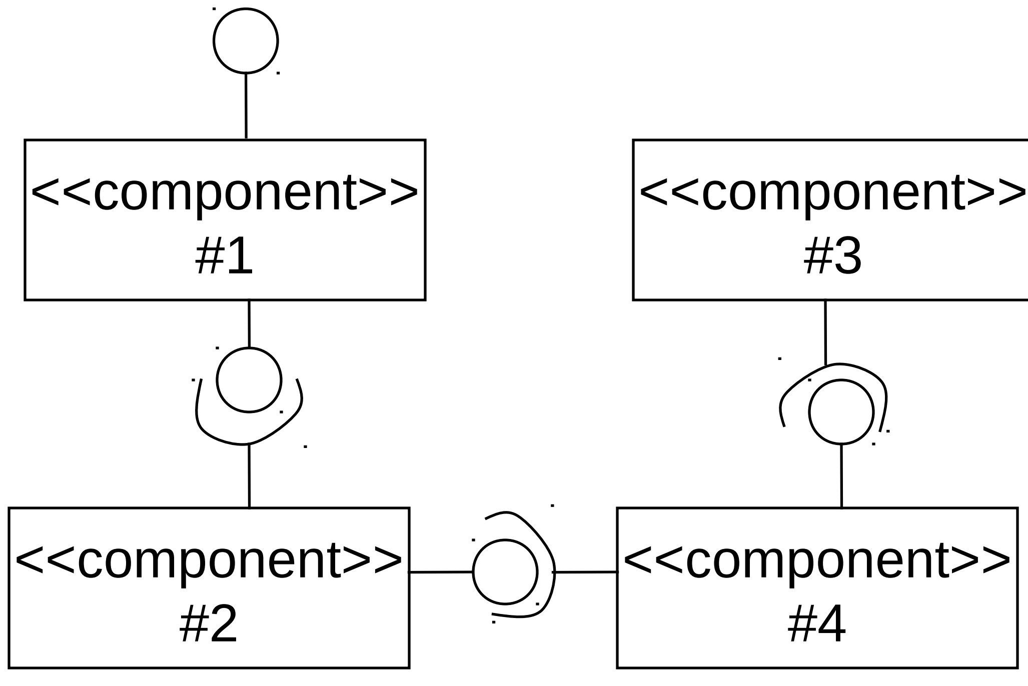


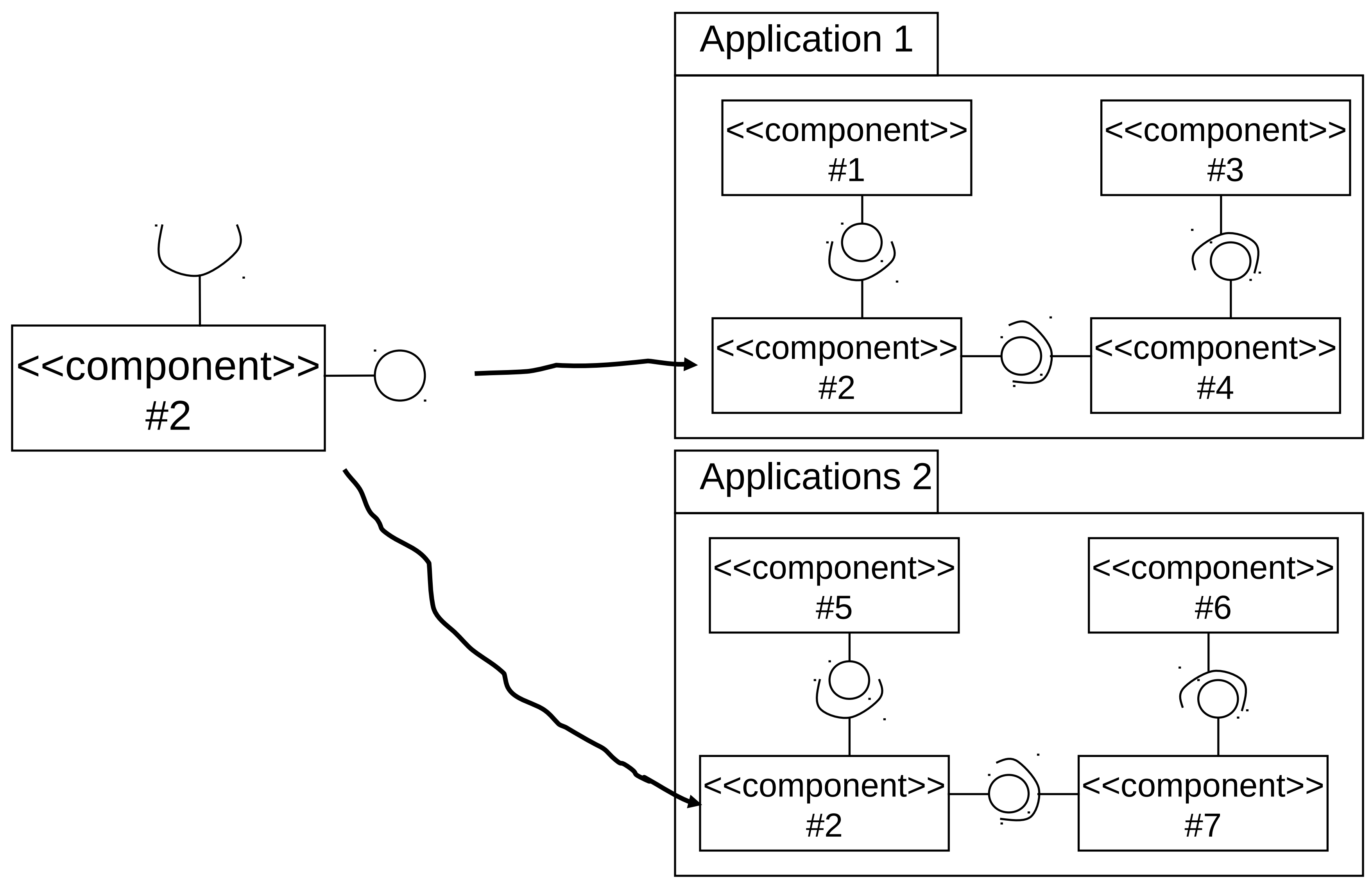



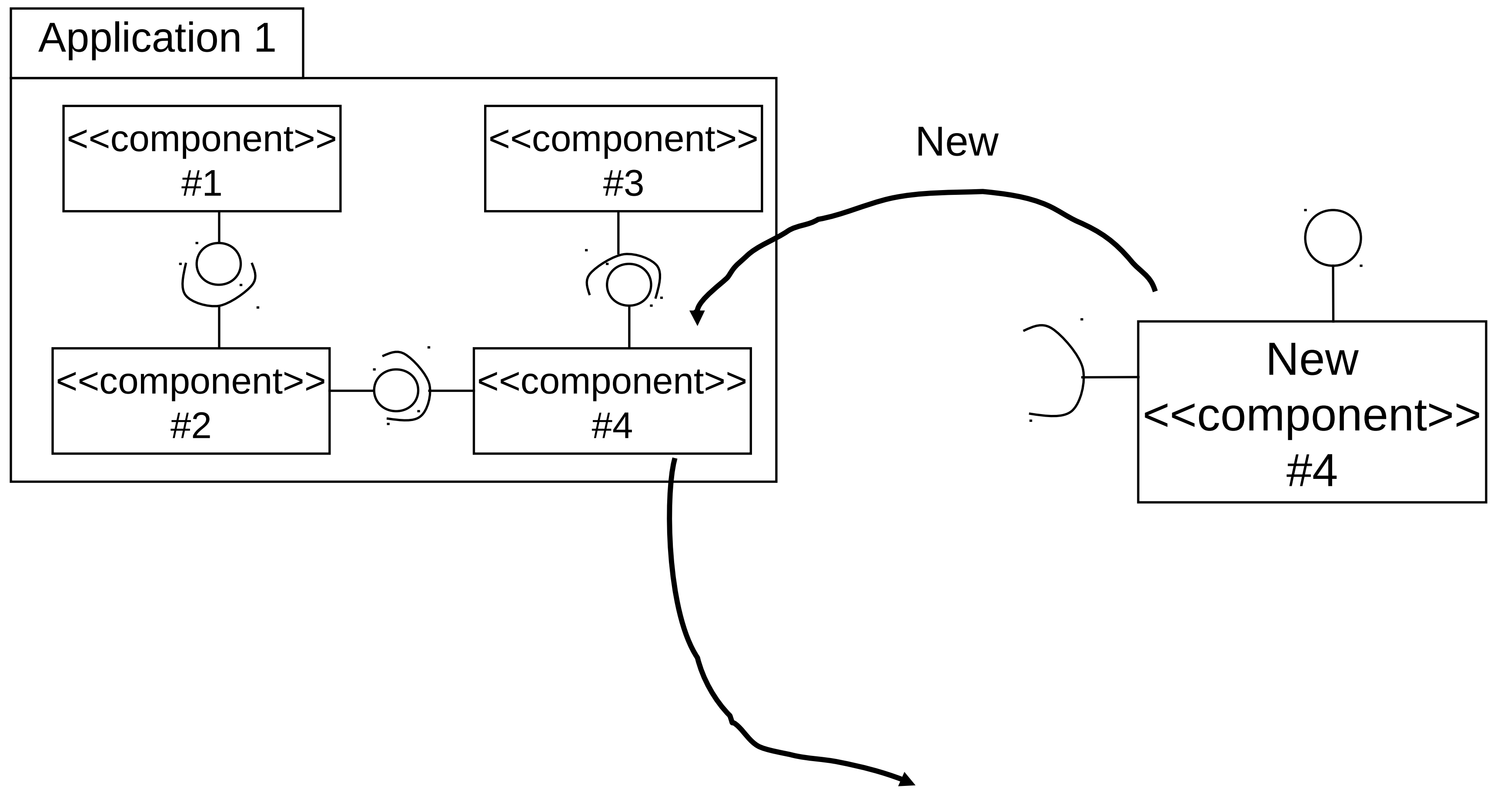
# Application



# Reuse of Components



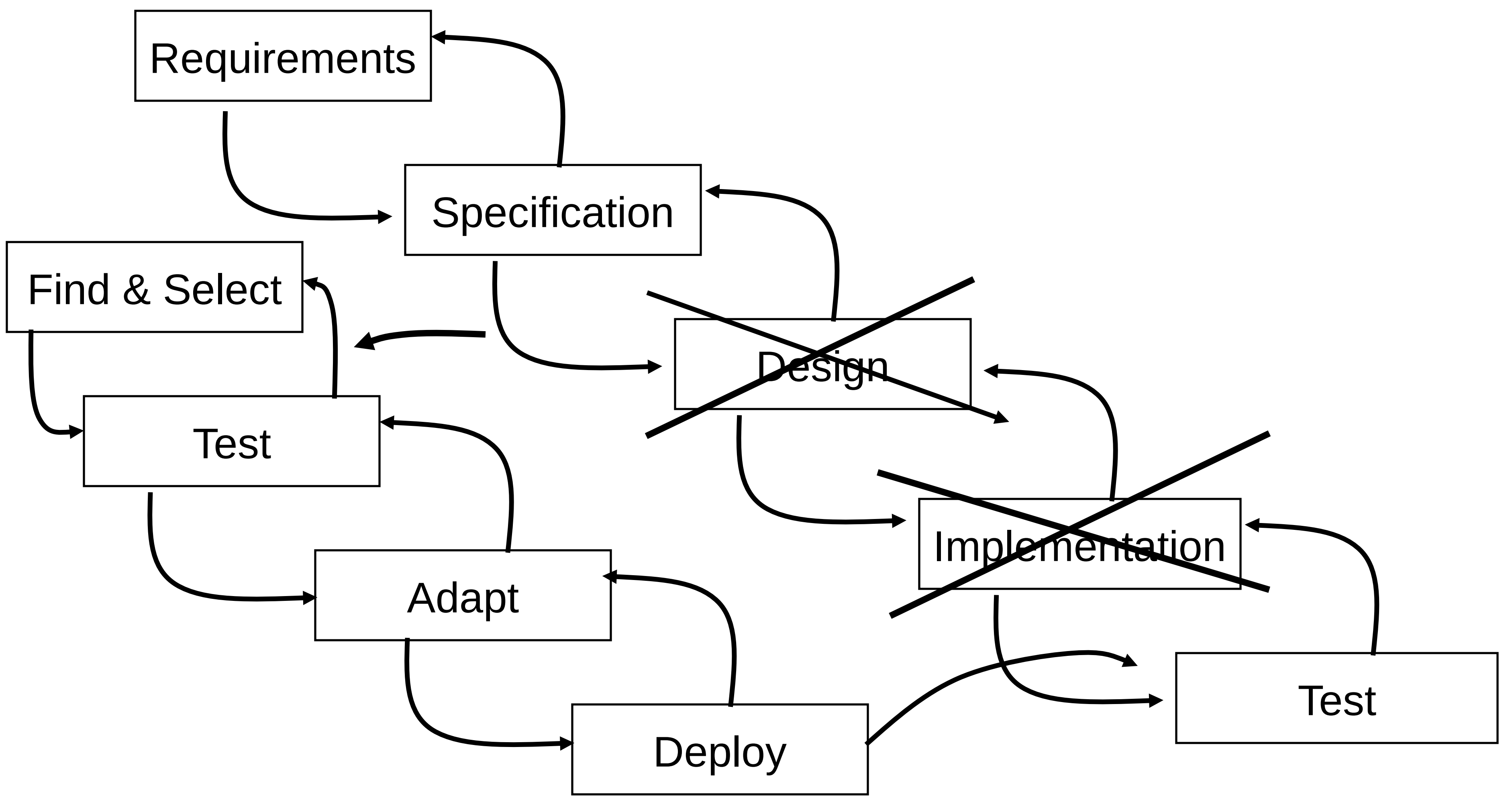
# Change a Component



# Components

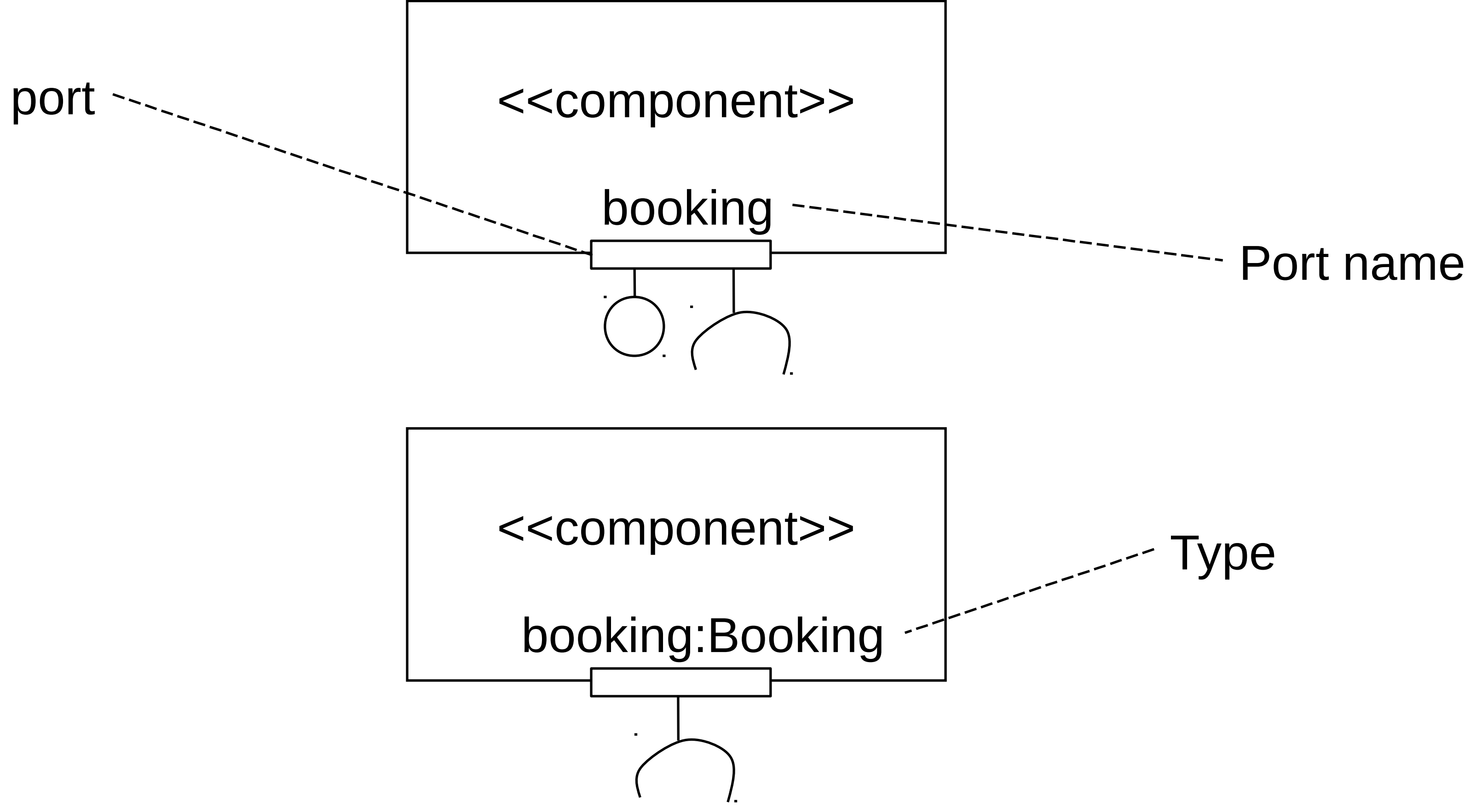
- Parnas’s laws:
  - Only what is hidden can be changed without risk.

# Different Development Process



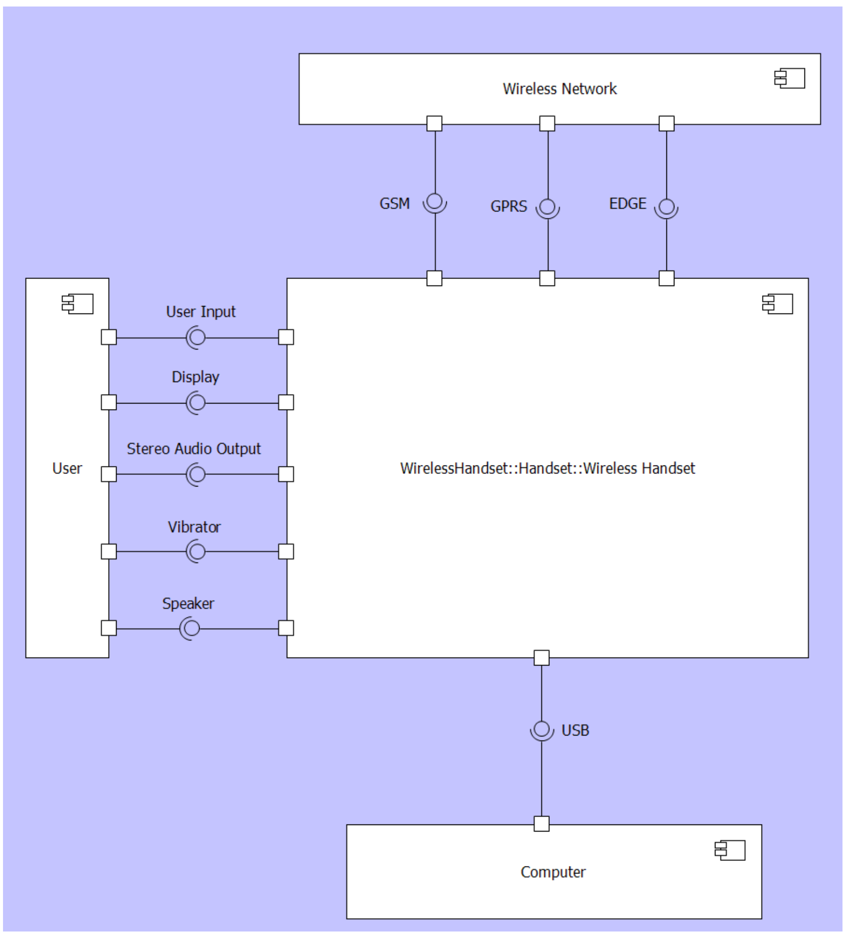
# Port

- Semantically cohesive set of provided and required interfaces.

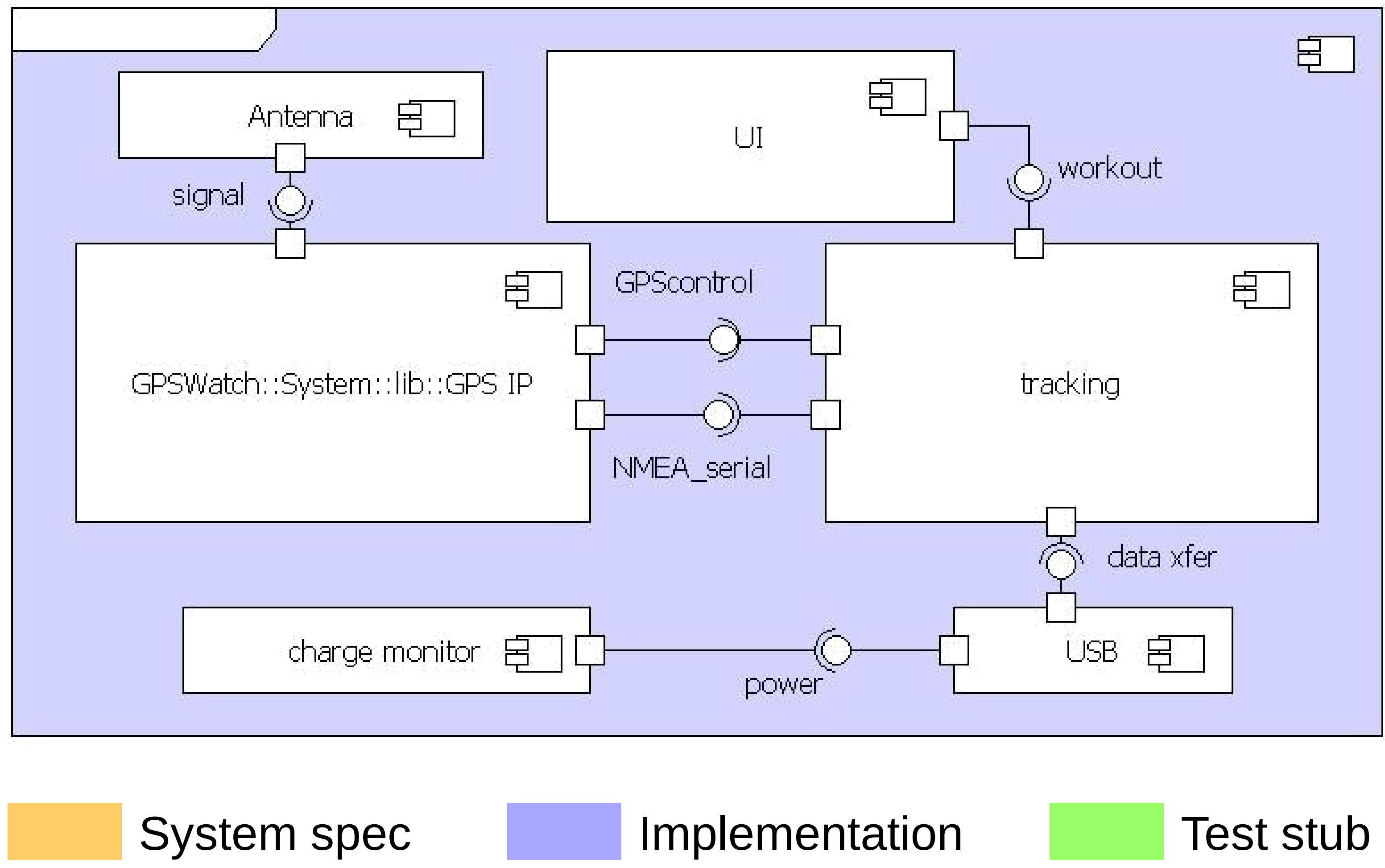


# Getting Started

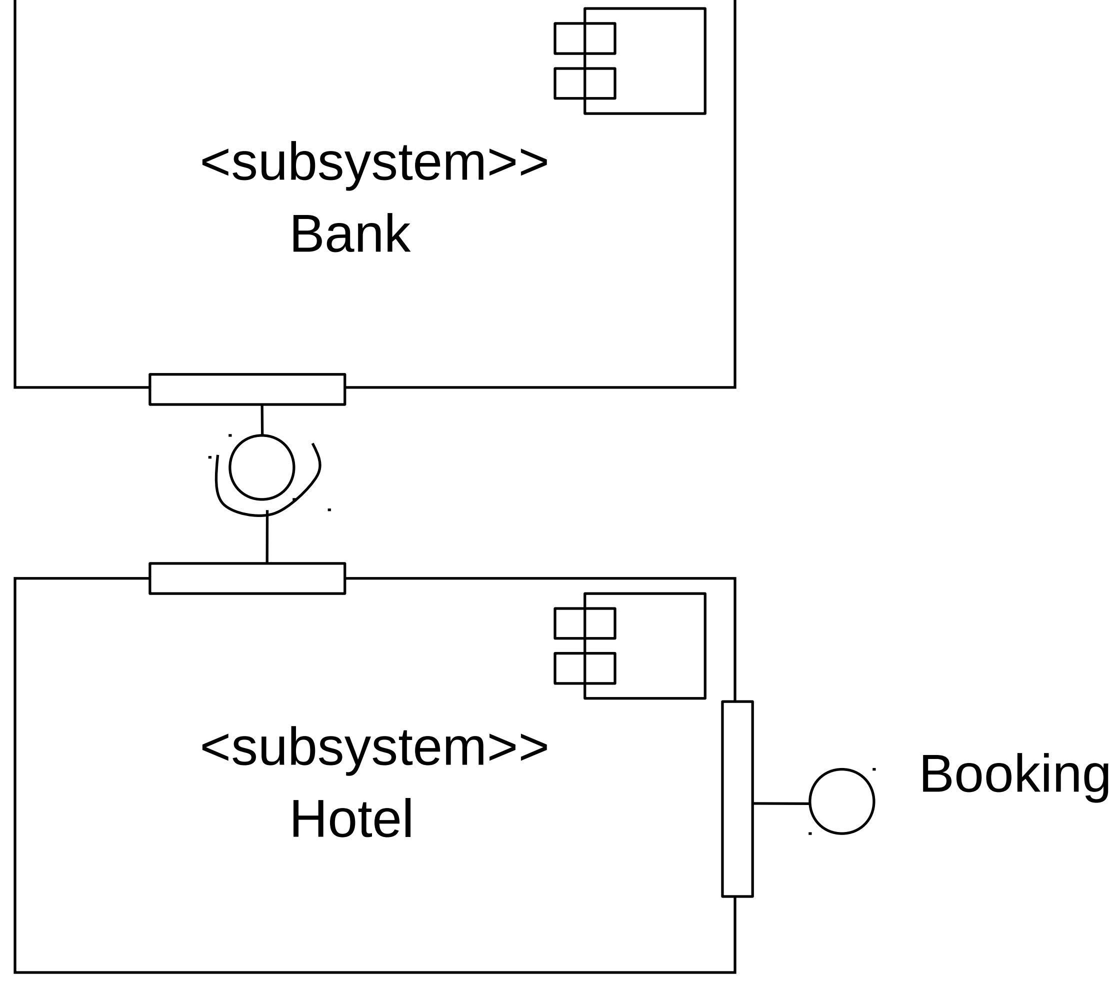
- Divide and conquer
  - Any boundary
  - Hierarchically nesting
- Define interfaces
  - Operations and signals
- Connect components



# Test stub



# Part of the Hotel System



# Challenge

- Who owns the components?
- Can one trust components?
- Hard to make general programs
- Few programming languages support components
- Hard to find good interfaces
- Can be hard to combine
- Performance
  - …
  - Active research area! OCL can be important for making the contracts of the interfaces.

# Deployment model

Instance:

<<device>>
Server:IBM AS/400

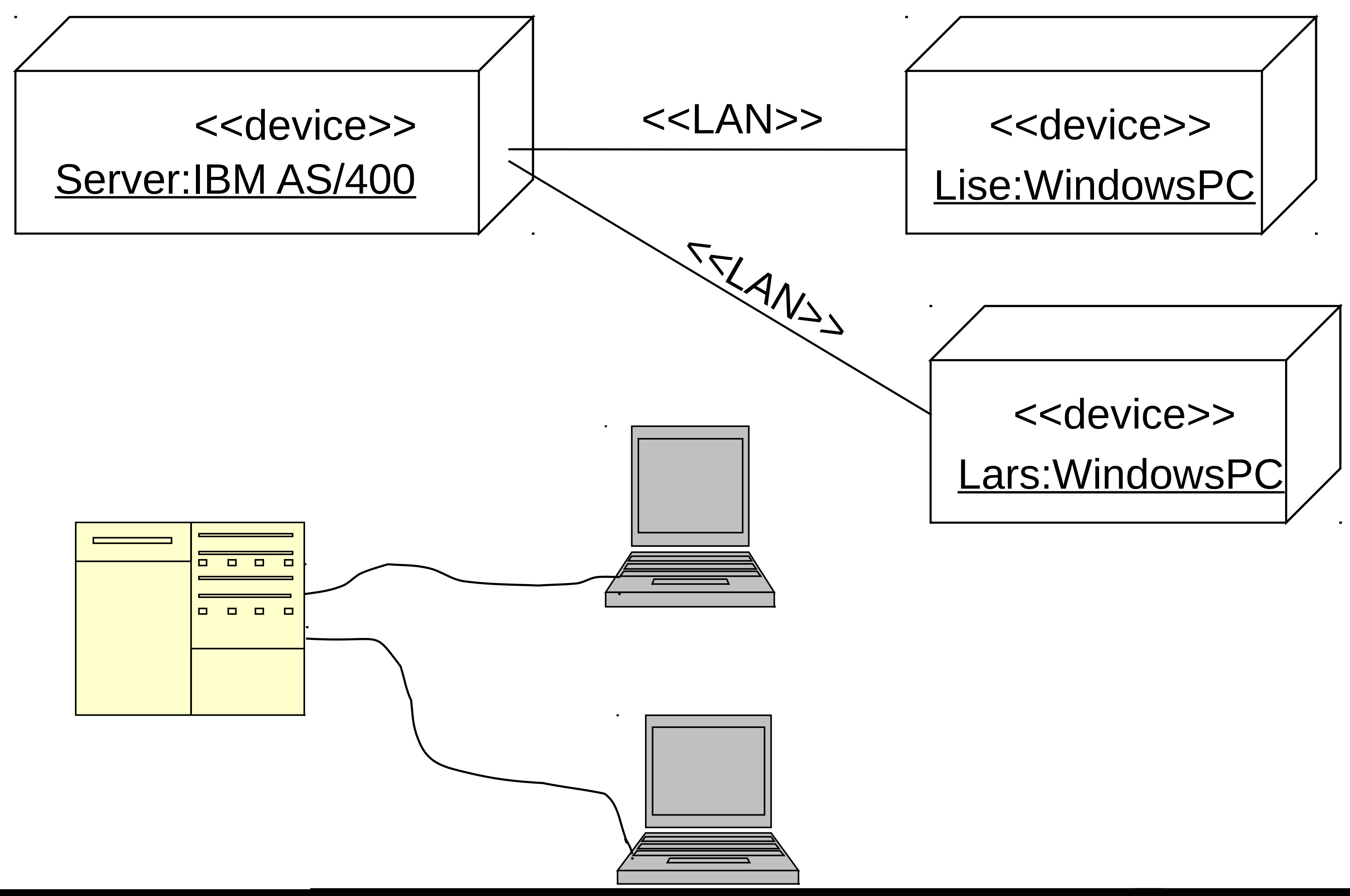
<<LAN>>

<<device>>
Lise:WindowsPC

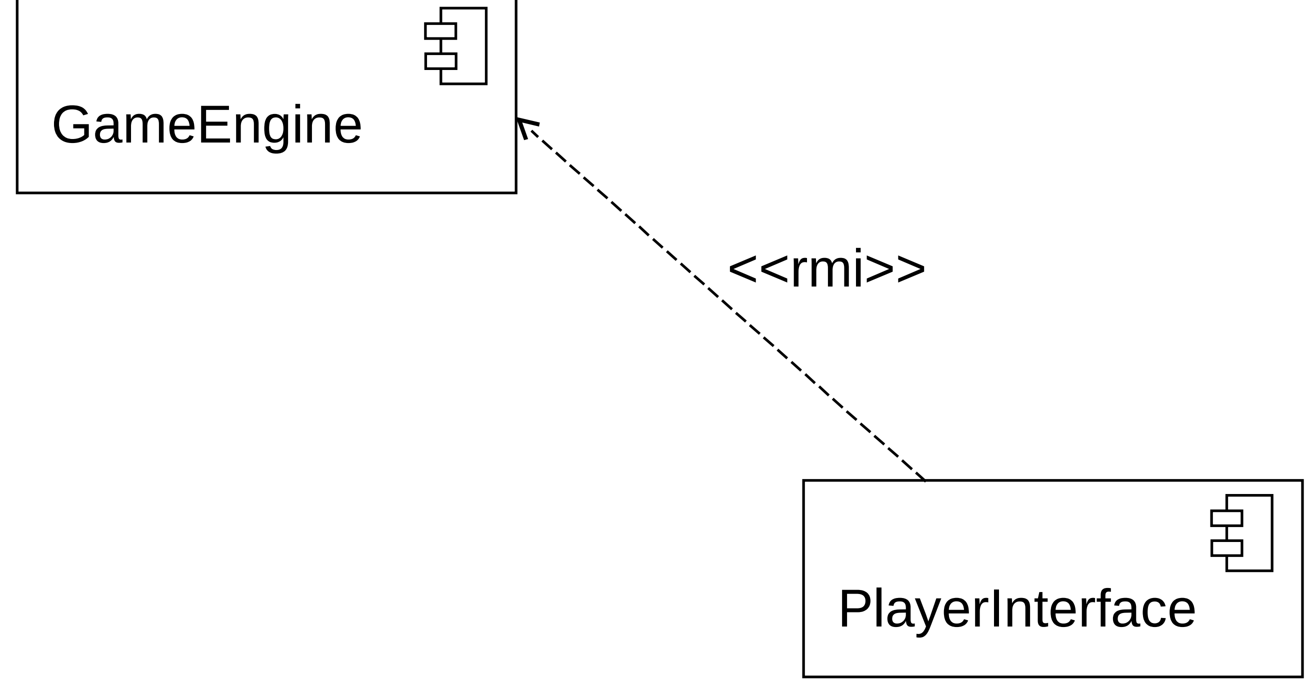
<<LAN>>

<<device>>
Lars:WindowsPC

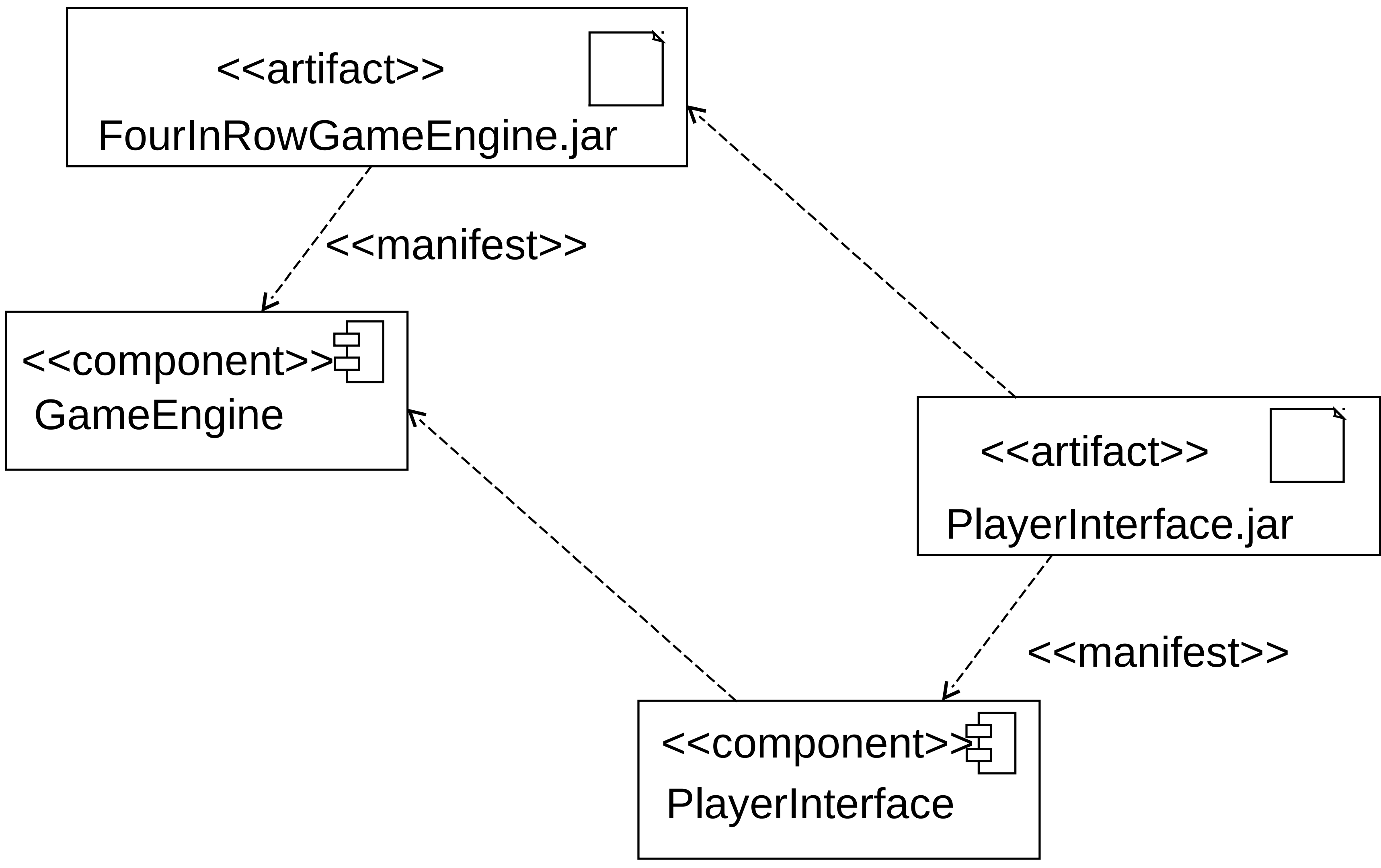
# Real world



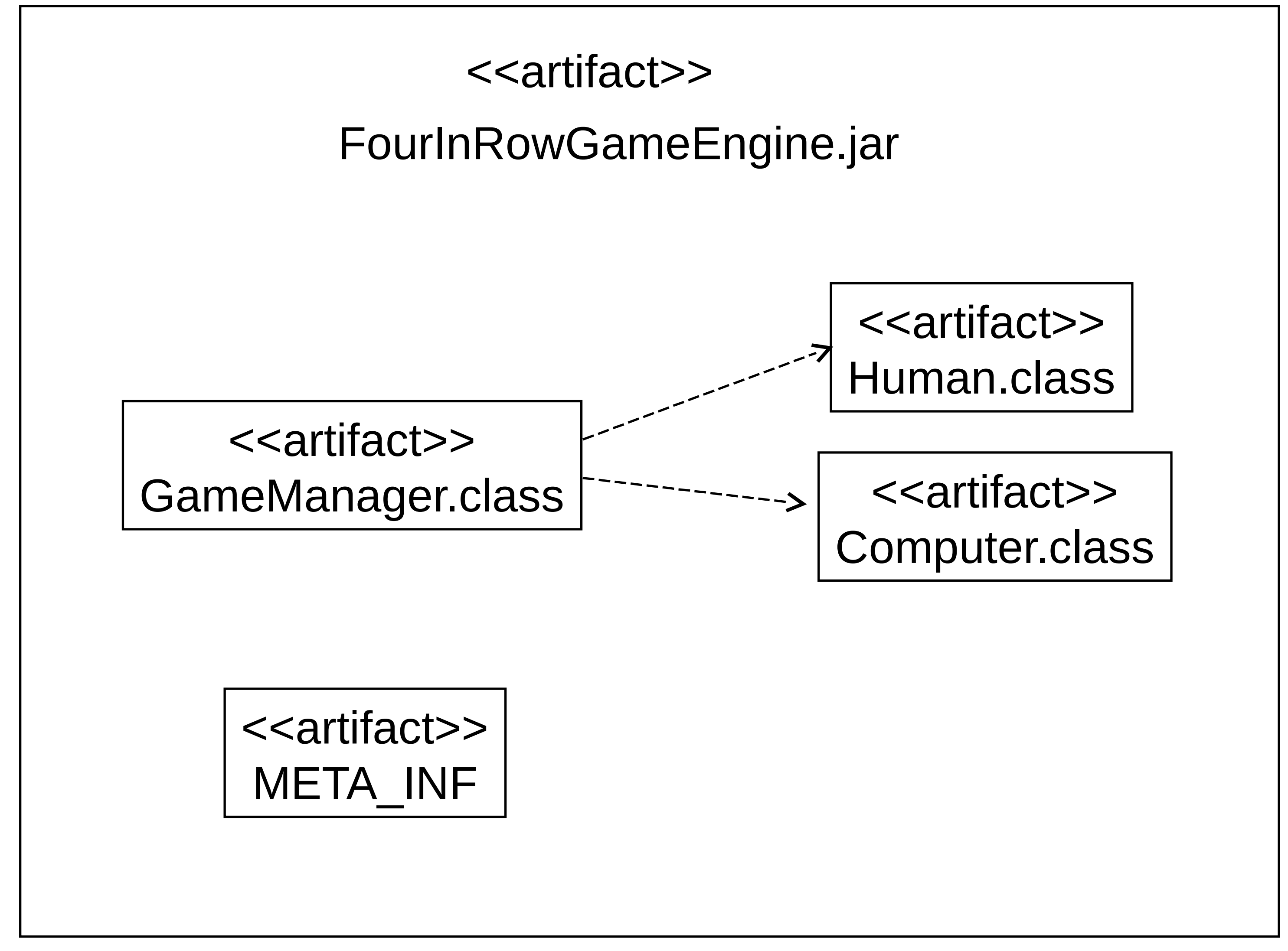
# Example components



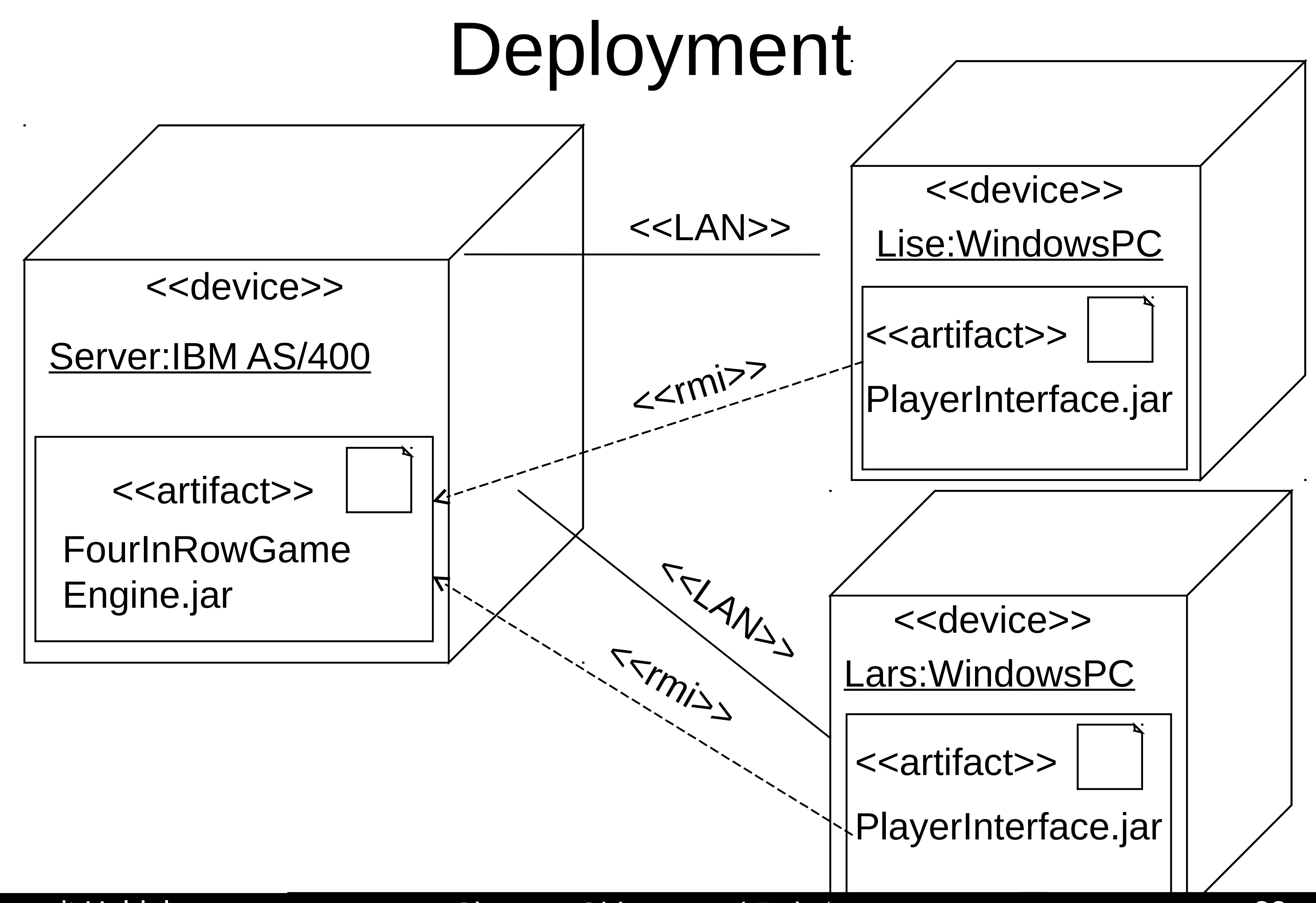
# Artifact



# FourInRowGameEngine.jar



# Deployment



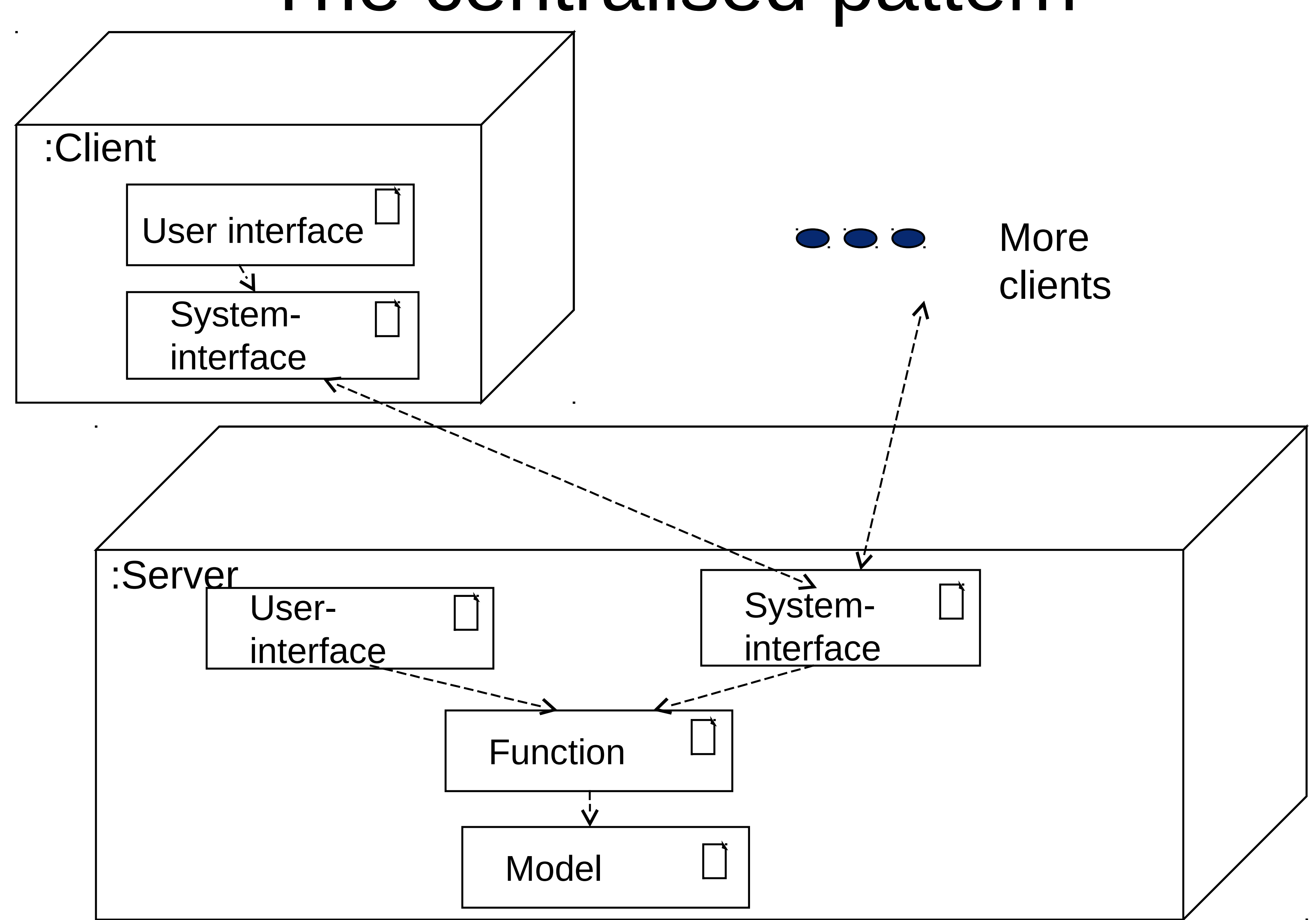
# Artifacts

- Artifacts are deployed on nodes. Some examples of artifacts are:
  - Scripts
  - Source files
  - Database tables
  - Documents
  - Components

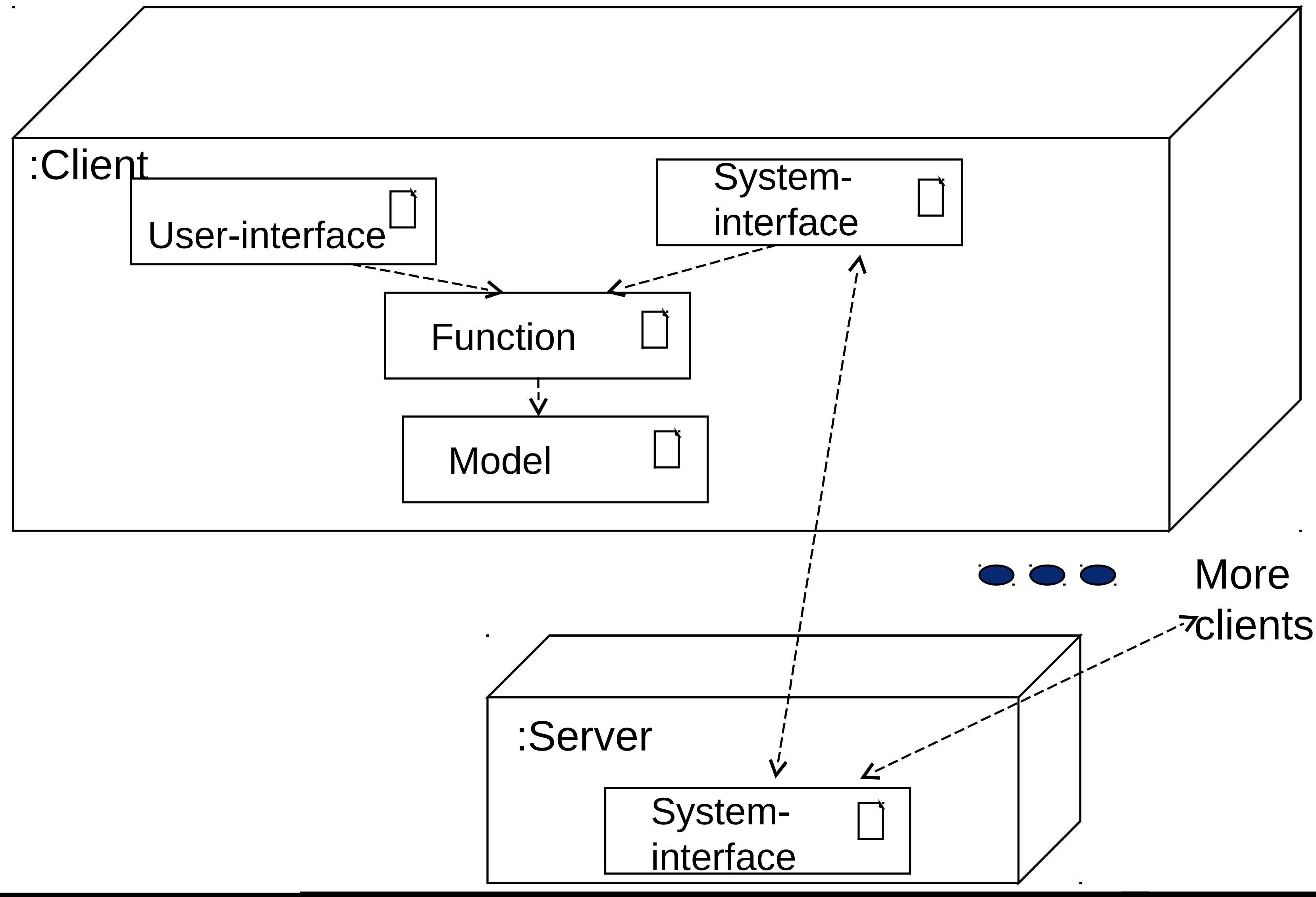
# Patterns for splitting up the work on hardware

- The centralised pattern
- The distributed pattern
- The decentralised pattern

# The centralised pattern



# The distributed pattern



# The decentralised pattern