Karlsruhe Institute of Technology

Software Engineering II

Prof. Dr. Ralf H. Reussner

Topic 3

Software Architecture - Foundations

DSIS – DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
KASTEL – INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY

dsis.kastel.kit.edu

Course Topics

Designing High Quality Systems

- Clean Coding
- Software Architecture & Components
  - Clean Architecture
  - Enterprise Application Architecture
  - Microservices
  - Cloud Architecture
- Domain-driven design

Processes and Requierments

- Development processes
- Agile development
- Requirements engineering
- Use cases

Ensuring Software Quality

- Continuous Integration
- Software Reliabilty
  - Real-time Systems
  - Real-time design patterns
- Software Security

Overview on This and Next Lecture

Content: Introduction to Software Architecture

- Context and Motivation
- Foundations and Terminology
- Architectural Patterns and Styles
  - Layered Architecture
  - Some more Patterns

Learning Goals: Be able to

- reproduce and describe the definitions of software architecture
- explain the difference between software architecture and software architecture documentation
- describe the advantages of explicit architecture and the influences on architecture decisions
- assign design decisions and elements to architectural layers
- explain the architectural style layered architecture
- be able to apply the patterns MVC and Observer to examples

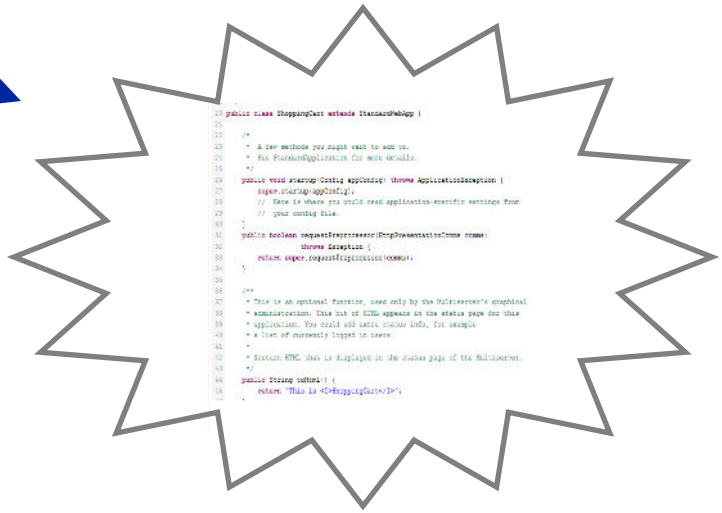
MOTIVATION

A Kind of Magic?

- How can we bridge the gap between requirements and code?
  - in a systematic manner?

Problem / opportunity in the real world

???

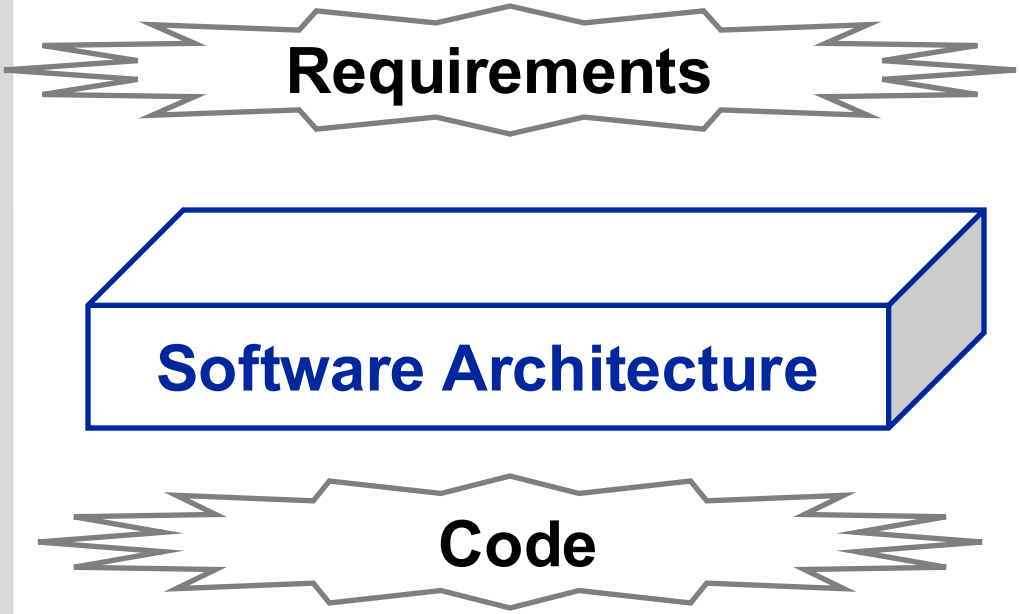


Code

Software Architecture

Idea:

- A software architecture is the important structure(s) of the system
  - or more pragmatically: A software architecture is the set of design decisions which are hard to revert or which have to be made early

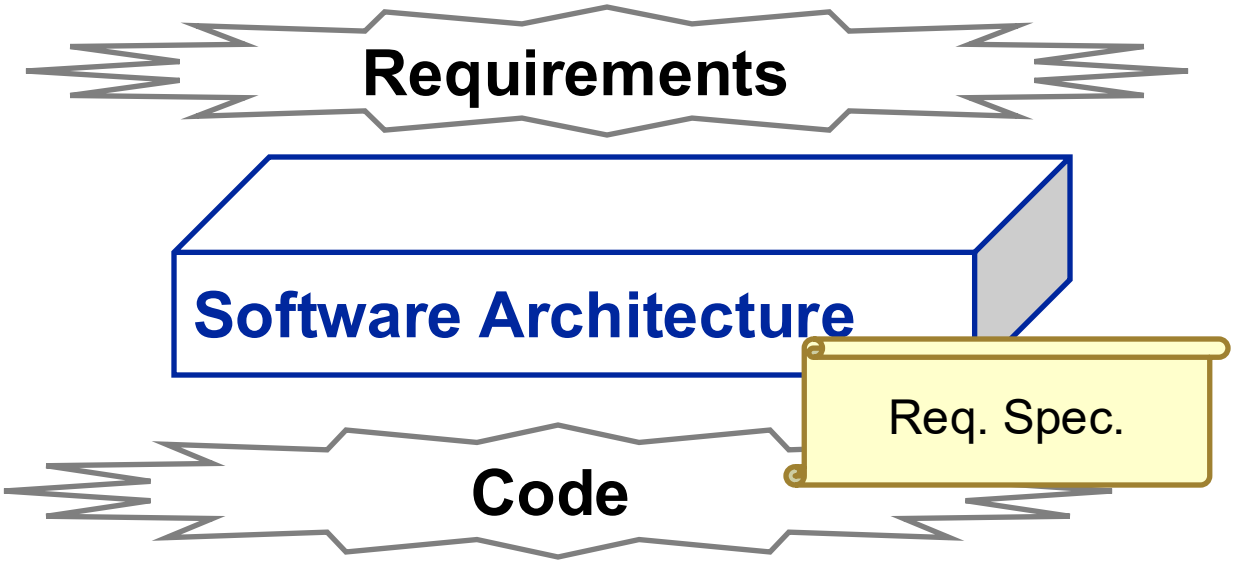


Advantages of explicit architecture:

- Stakeholder communication
- System analysis
- Large-scale-reuse
- Project planning

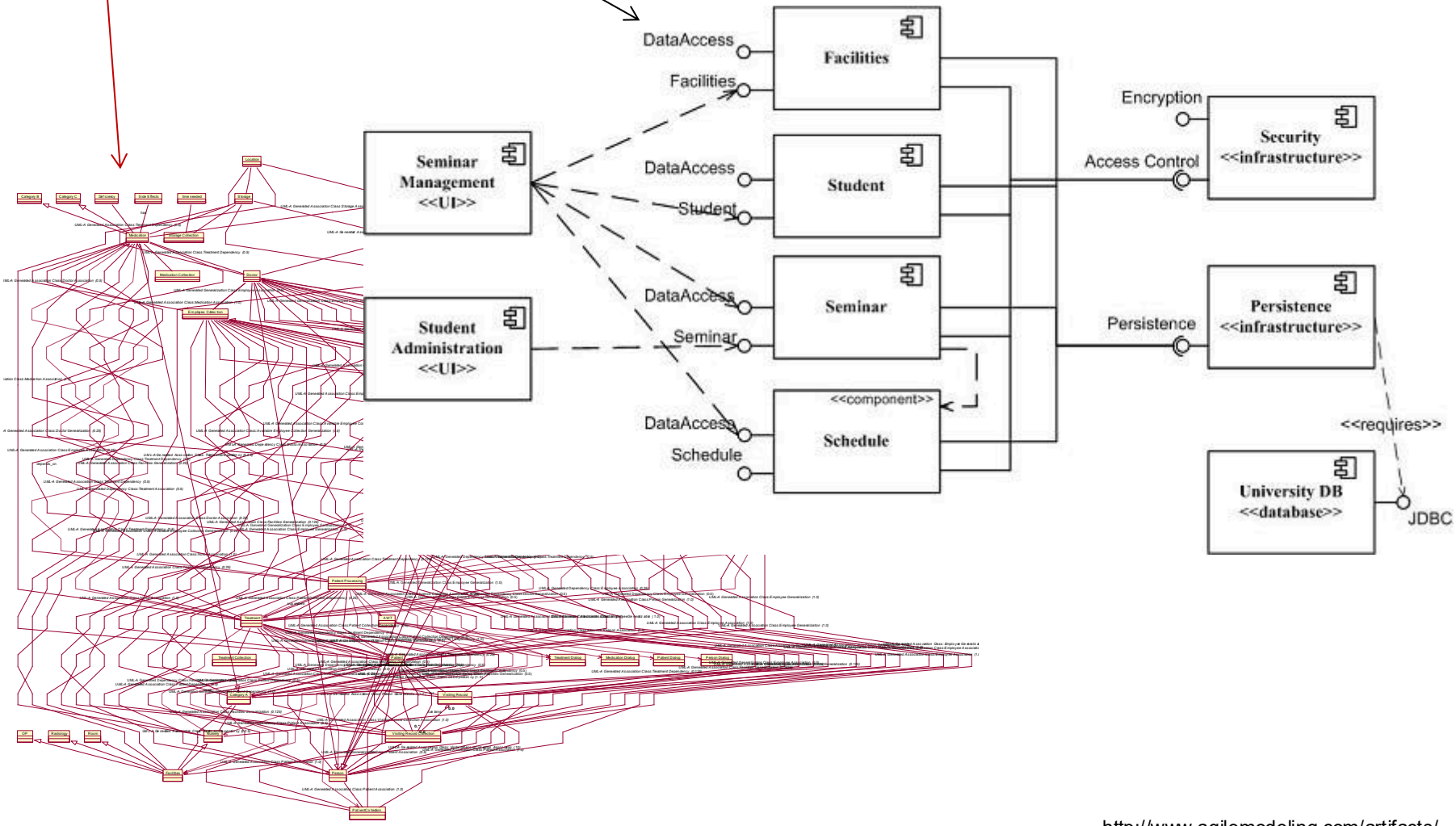
Architectural Design

- An early stage of the system design process
- Represents the link between specification and design
  - often carried out in parallel with some specification activities



➔ It involves identifying major system components, their communications and mapping to hardware resources

Design vs. Architecture



http://www.agilemodeling.com/artifacts/

FOUNDATIONS AND TERMINOLOGY

Various Definitions: Software Architecture

- Architecture: fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution
  [ISO/IEC 42010, cf. http://en.wikipedia.org/wiki/ISO/IEC_42010]
  - Focus on important questions
  - What are the parts the system is built from
  - How are these connected and how do they interact
  - Further major decisions
  - Also consider evolution

- A software system's architecture is the set of principal design decisions made about the system
  [Taylor et al., 2009, p.58]
- The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both
  [Bass et al., 2013]

Our Definition

A software architecture is

- the result of a set of design decisions — As in definition by Taylor et al.
- comprising the structure of the system — As in definition by Bass et al.
  - with components and
  - their relationships and
  - their mapping to execution environments.

[Reussner 2016]

Software Architecture

- A **software architecture** is the result of a set of **design decisions** relating to the structure of a system with components and their relationships and their mapping to execution environments. [Reussner]
- **Architectural design decisions** are made in
  - early phases of the software development process (requirement-oriented)
  - later phases, when the influence of the execution environment becomes clearer (code-oriented)
- Any design decision of a software system can be deferred
  - but this is a decision again
  - not all decisions can be deferred together

→ Software architecture is not an implicit internal structure of the system.

- Software architectures should be documented with dedicated languages.

→ Programming languages were never designed to document architectural decisions.

Architectural Design

- **Architectural design** is a creative process depending on the type of system being developed.
- A number of common decisions span all design processes:
  - Is there a generic application architecture that can be used?
  - Which kinds of distribution are possible, appropriate, and necessary?
  - What architectural styles are appropriate?

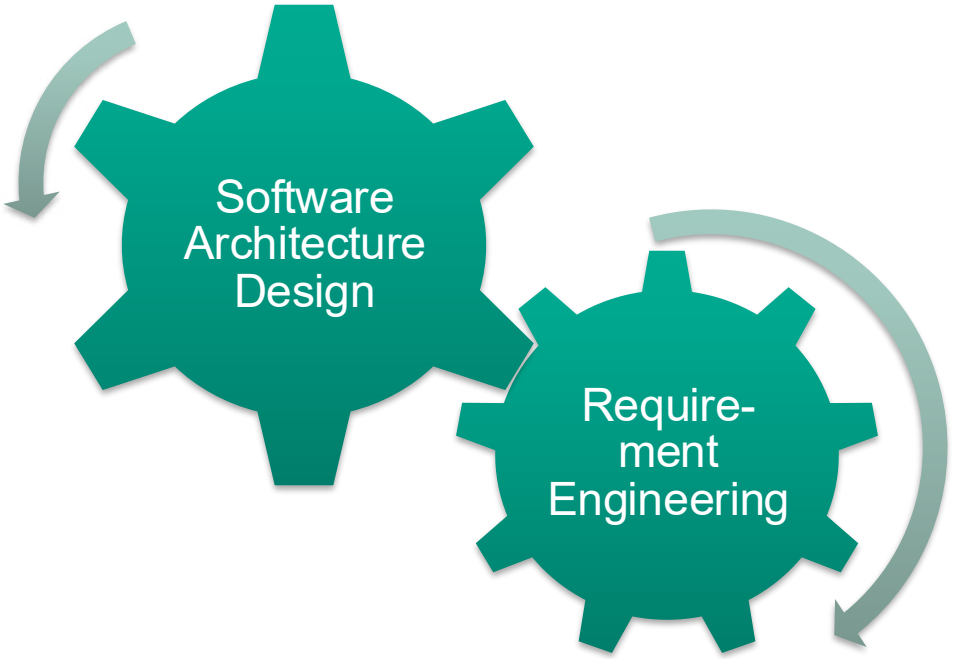
  - How will the system be decomposed into subsystems (components)?
  - Which components can or must be bought?
  - Which components can be re-used?
  - Which components should be re-usable on future projects?

  - What management and evolution strategy should be used?
  - How will the architectural design be evaluated?
  - How should the architecture be documented?

  - What are realistic evolution scenarios?
  - …

Requirement Engineering

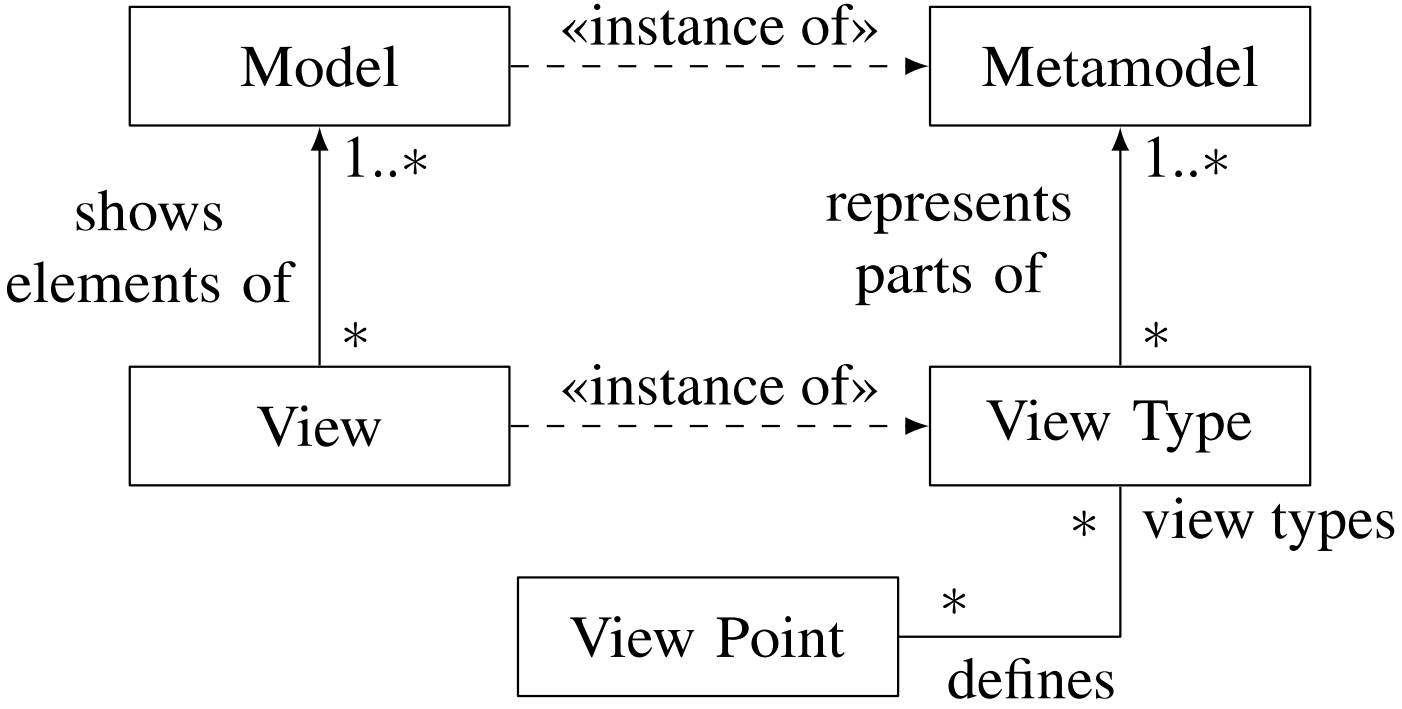
- An early stage of the system design process
- Requirement engineering process and software design process are interleaved.



- Requirement engineering and design involves identifying **major system components**, their **communications** and **mapping to hardware** resources.
- There is no sharp distinction between what and how, due to mutual dependencies, different view points (customer, user, architect, programmer, tester, …).

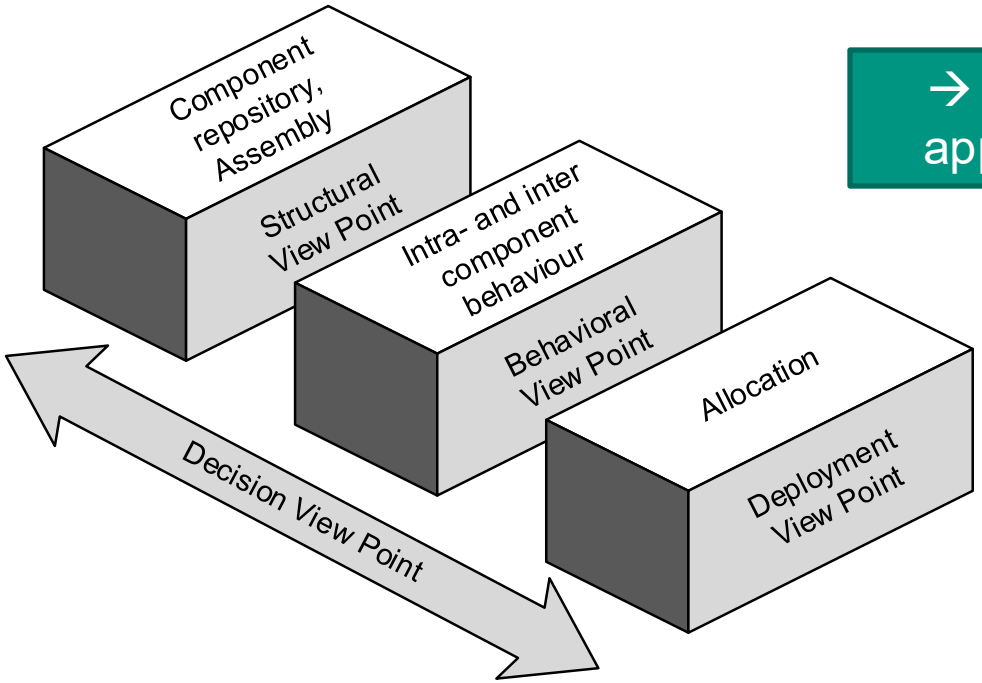
Relation of Views, View Types and View Points

[Burger 2014]



Views and View Points Examples

- Views and view points in Palladio (more later)



→ Choose the set of views appropriate for your project

- Examples of other view (point) models
  - UP [Larman]: Logical, Process, Deployment, Data, Use Case, Implementation
  - 4+1 [Kruchten]: Logical, Development, Process, Physical + Scenarios

View Points in Palladio

- The ***structural*** view types contain information about the static properties of a system. They can be differentiated into system-specific and system-independent types. The *repository* view type is the only system-independent view type; it shows all components and interfaces that may be re-used within multiple systems. The *assembly* view type shows how components are instantiated in a given system and how these instances are connected.
- The ***behavioural*** view types contain information about the functional *(sequence diagrams)* and extra-functional *(SEFF)* execution semantics of the systems. Furthermore, the behaviour of users or other systems which interact with the system is characterized using the *usage model* view types.
- Finally, the ***deployment*** view point is characterised using the *allocation* view type, which contains information on which component instances of the assembly view type are allocated on which containers of the environment view type, and the *resource environment* view type, which depicts all containers and the links between them.

[Reussner 2016]

4+1 Architectural Views [Kruchten 1995]

- **Logical view :** The logical view is concerned with the functionality that the system provides to end-users. UML Diagrams used to represent the logical view include Class diagram, Communication diagram, Sequence diagram.
- **Development view :** The development view illustrates a system from a programmer's perspective and is concerned with software management. This view is also known as the **implementation** view. It uses the UML Component diagram to describe system components. UML Diagrams used to represent the development view include the Package diagram.
- **Process view :** The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behaviour of the system. The process view addresses concurrency, distribution, integrators, performance, and scalability, etc. UML Diagrams to represent process view include the Activity diagram.
- **Physical view :** The physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer, as well as the physical connections between these components. This view is also known as the **deployment** view. UML Diagrams used to represent physical view include the Deployment diagram.
- **Scenarios :** The description of an architecture is illustrated using a small set of use cases, or scenarios which become a fifth view. The scenarios describe sequences of interactions between objects, and between processes. They are used to identify architectural elements and to illustrate and validate the architecture design. They also serve as a starting point for tests of an architecture prototype. This view is also known as **use case** view.
- Used in the RUP

Some UP Architectural Views [Larman, p.658]

1. **Logical**
   - Conceptual organization of the software in terms of the most important layers, subsystems, packages, frameworks, classes, and interfaces. Also summarizes the functionality of the major software elements, such as each subsystem.
   - Shows outstanding use-case realization scenarios (as interaction diagrams) that illustrate key aspects of the system.
   - A view onto the UP Design Model, visualized with UML package, class, and interaction diagrams.
2. **Process**
   - Processes and threads. Their responsibilities, collaborations, and the allocation of logical elements (layers, subsystems, classes, ...) to them.
   - A view onto the UP Design Model, visualized with UML class and interaction diagrams, using the UML process and thread notation
3. **Deployment**
   - Physical deployment of processes and components to processing nodes, and the physical network configuration between nodes.
   - A view onto the UP Deployment Model, visualized with UML deployment diagrams. Normally, the "view" is simply the entire model rather than a subset, as all of it is noteworthy.
4. **Data**
   - Overview of the persistent data schema, the schema mapping from objects to persistent data (usually in a relational database), the mechanism of mapping from objects to a database, database stored procedures and triggers.
   - A view onto the UP Data Model, visualized with UML class diagrams used to describe a data model.

Some UP Architectural Views [Larman, p.658]

5. **Security**

6. **Implementation**

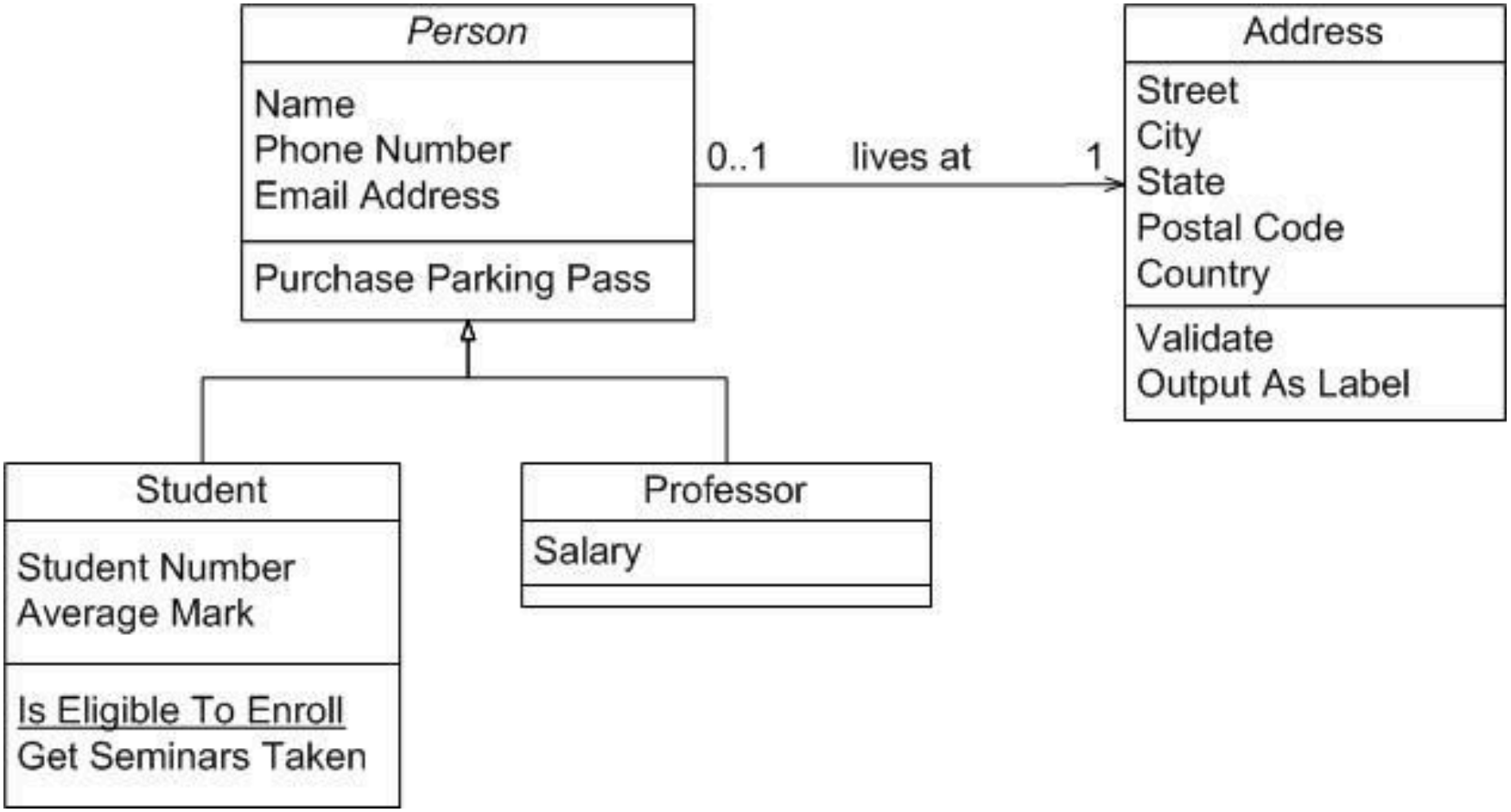
- First, a definition of the Implementation Model: In contrast to the other UP models, which are text and diagrams, this "model" is the actual source code, executables, and so forth. It has two parts: 1) deliverables, and 2) things that create deliverables (such as source code and graphics). The Implementation Model is all of this stuff, including web pages, DLLs, executables, source code, and so forth, and their organization—such as source code in Java packages, and bytecode organized into JAR files.
- The implementation view is a summary description of the noteworthy organization of deliverables and the things that create deliverables (such as the source code)

7. **Development**

8. **Use case**

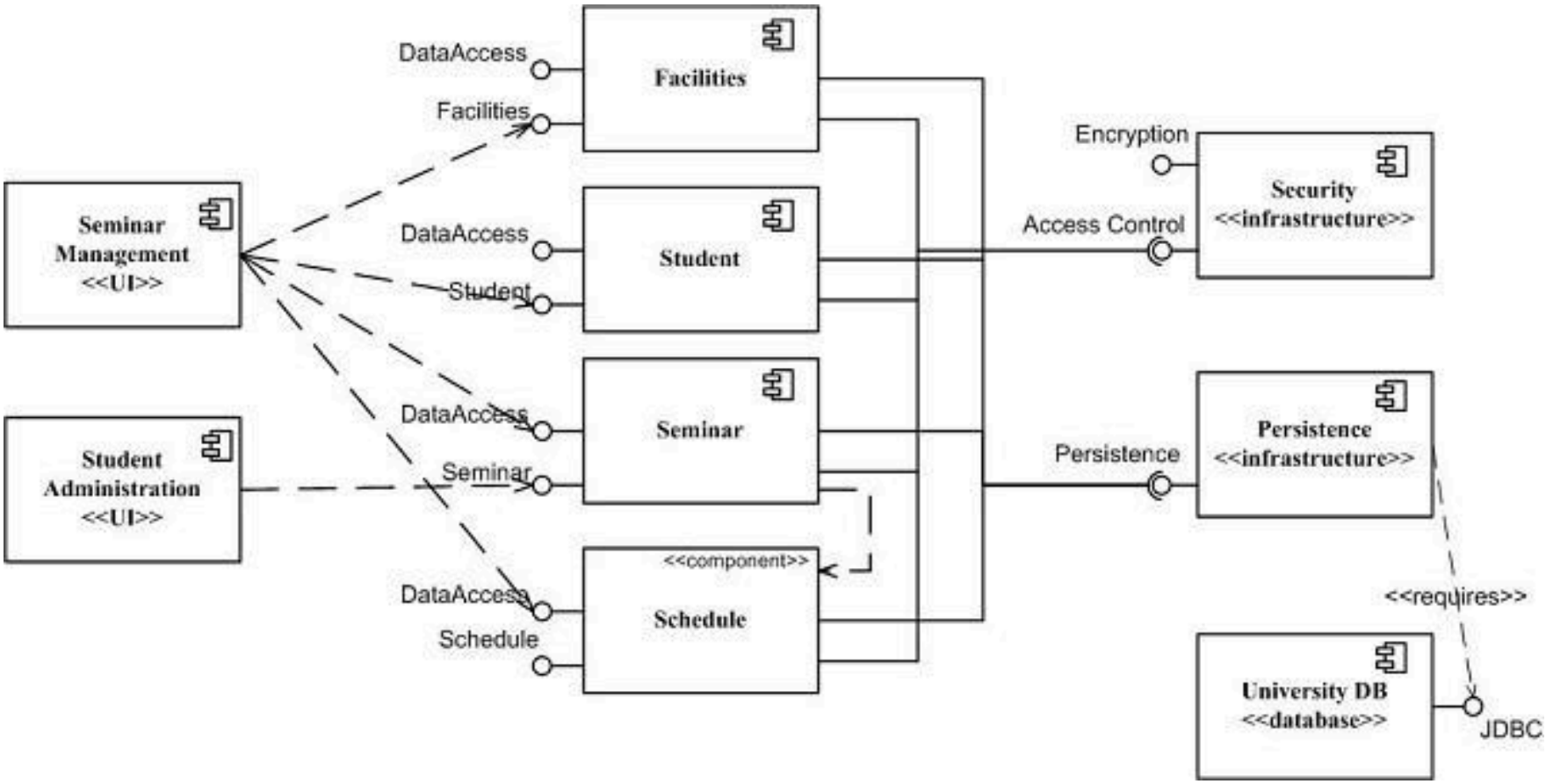
- Summary of the most architecturally significant use cases and their non-functional requirements. That is, those use cases that, by their implementation, illustrate significant architectural coverage or that exercise many architectural elements. For example, the Process Sale use case, when fully implemented, has these qualities.
- A view onto the UP Use-Case Model, expressed in text and visualized with UML use case diagrams.

A Structural View (Data Objects)



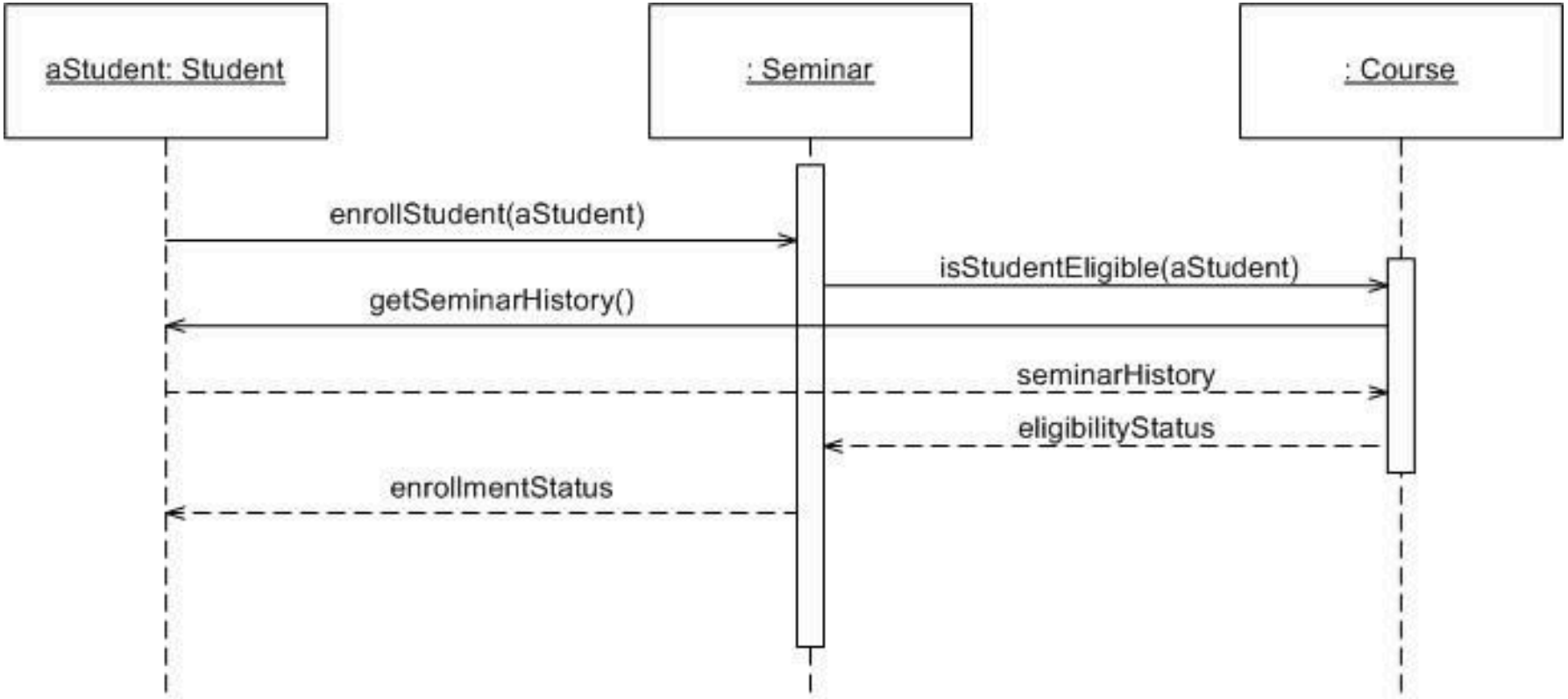
http://www.agilemodeling.com/artifacts/

A Structural View (Architecture)



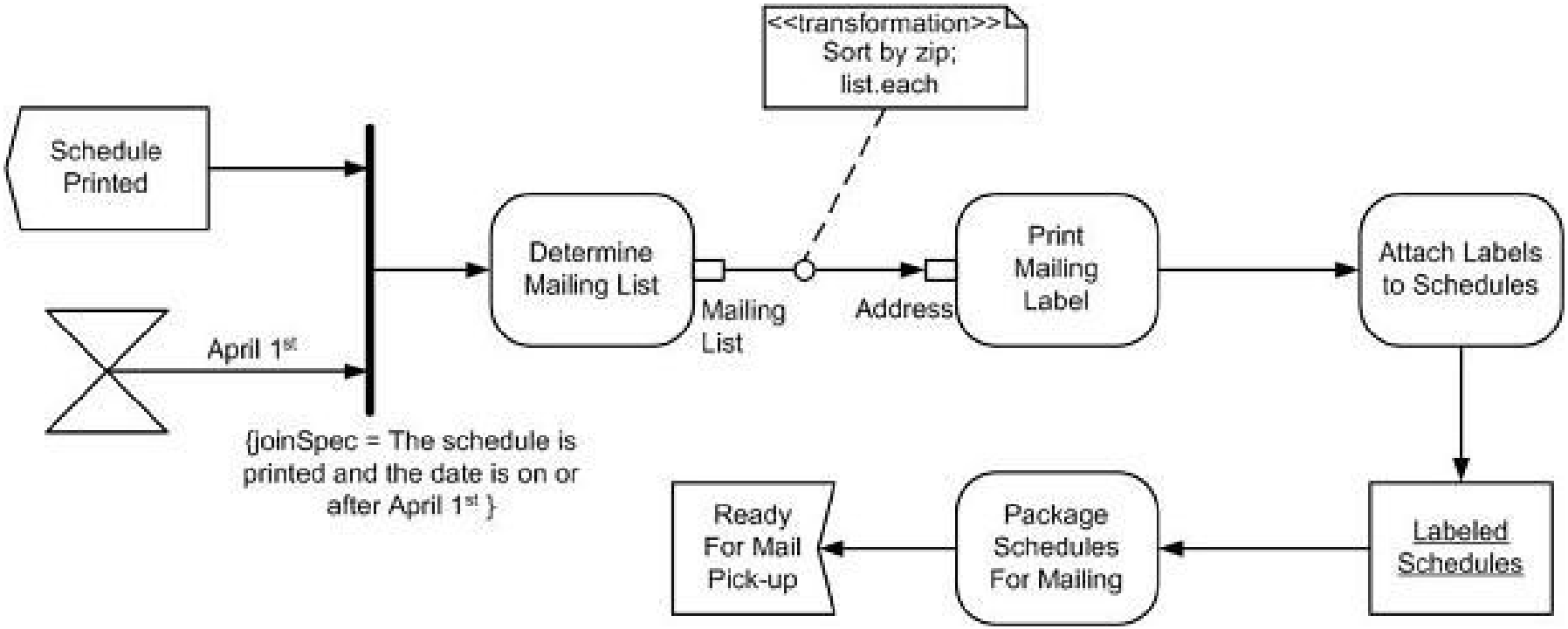
http://www.agilemodeling.com/artifacts/

A Behavioural View (inter component)



http://www.agilemodeling.com/artifacts/

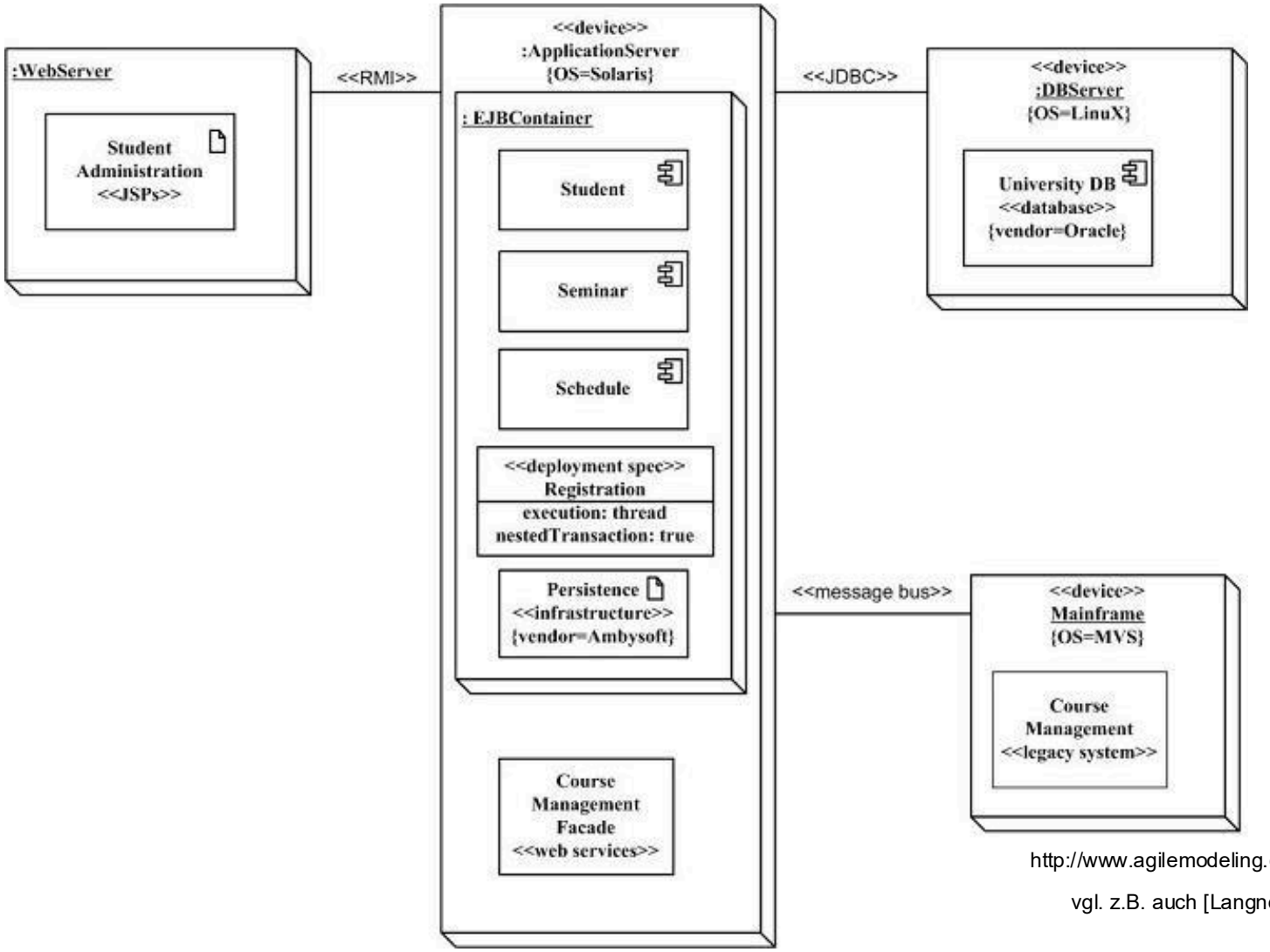
A Behavioural View (intra component)



„Distributing schedules“
as an activity diagram

http://www.agilemodeling.com/artifacts/

A Deployment View



http://www.agilemodeling.com/artifacts

vgl. z.B. auch [Langner & Reiberg]

Advantages of an Explicit Architecture

- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders
- System analysis
  - Analysis of whether the system can meet its non-functional requirements
- Large-scale reuse
  - The architecture may be reusable across a range of systems
  - Existing components can be considered during design
    - COTS, in-house components, commissioned / off-shore
- Project planning
  - Cost-estimation, milestone organisation, dependency analysis, change analysis, staffing

Predicting the quality attributes of an artefact during design is a core property of any engineering discipline.

Agile Architecture

Scott Ambler

- The primary goal of architectural modeling should be to come to a common vision or understanding with respect to how you intend to build your system(s). In other words, you will model to understand.
- My experience is that 99.999% of all software project teams need to invest some time modeling the architecture of their system, and that this is true even of Scrum/XP teams that rely on a metaphor to guide their development efforts.

[http://www.agilemodeling.com/essays/agileArchitecture.htm]

Integration with iterative development in UP:
cf. [Larman, p. 542, p.556ff, p.669 and Ch 33 in general]

Recording Architecture Decisions

- Record
  - Influential factors
  - Decisions and why chosen
  - Alternative solutions and why not chosen

- Important to record **why** a decision has been made (rationale)
  - So that others can understand later
  - So that you can understand later
  - So that non-existance decisions are documented
  - Be able to quickly recall reasoning later
    - During later design
    - During evolution
    - Do not loose time re-inventing the solution or even understanding the reasoning!

[Larman p. 549-551]

Factors Influencing the Architecture

- Requirements
  - In particular quality requirements and constraints
  - Architecturally-significant requirements

[Bass 2013], also discussed in [Larman, Ch 33]

- Re-Use
  - Architectures
  - Subsystems / Components
  - Styles, Patterns, Guidelines

[Reussner 2008, Ch 4 and Part V] [Reussner 2016, Chapter 4]

- Organisation
  - *"A system (usually) reflects the organizational structure that built it"*, known as Conway's law [Endres/Rombach03]
  - Team size, team number, experience, organisation structure

Influential Factor: Quality Requirements

Identify the most important goals of your system's architecture (quality requirements), e.g. –

- Performance
  - e.g. localize operations in large-grained components to minimise sub-system communication
- Security
  - e.g. use a layered architecture with critical assets in inner layers
- Safety
  - e.g. analyse possible failures and design architecture to prevent underlying faults
- Availability
  - e.g. include redundant components in the architecture
- Maintainability
  - e.g. use fine-grained, self-contained components
- Scalability
  - e.g. consider concurrency effects in case you need to distribute the system

➔ *They may often contradict with each other and thus need to be balanced*

cf. [Reussner 2008, Ch. 13] and [Reussner 2016, Ch. 10]

Reuse of (Architectural) Design Principles

- **Separation of concerns**
  Minimize coupling, maximize cohesion
- **Single Responsibility principle**
  one responsibility per module/component…
- **Information Hiding**
  only what is hidden can be changed without risk (Parnas)
- **Principle of Least Knowledge**
  a.k.a. Law of Demeter: Don’t talk to strangers!
- **Don’t repeat yourself (DRY)**
  nomen est omen
- **Minimize upfront design (as much as reasonable)**
  You ain’t gonna need it! (YAGNI) vs. Design for Extensibility/Reusability
  Use refactoring when needed

ARCHITECTURAL PATTERNS AND STYLES

Reuse Terminology

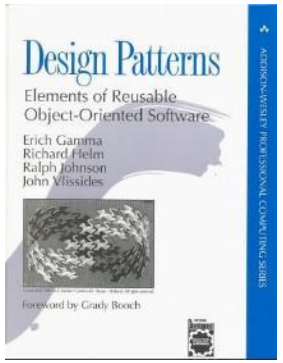
[Reussner 2008]

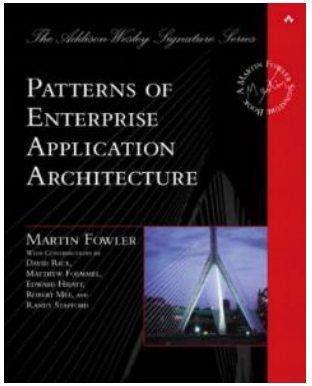
- **A word of warning!**
  Software Architecture is a relatively new field (~25 years) where many terms are still overloaded

- Pattern
  - a proven solution to a recurring problem
- Architectural Pattern
  - solution to recurring problem where several forces have to be balanced at the architecture level
- Architectural Style
  - solution principles (object-oriented style, modular style), independent of application, should be used throughout the architecture
  - (often also synonymously used for architectural pattern)
- Reference Architecture
  - defines domain concepts, components and subsystems that can be used by concrete instances

Architectural Patterns

- Many of the architectural aspects discussed so far have been codified in patterns
- The boundary between architectural and design patterns is sometimes blurry
  - as many ideas can be used on both levels
  - as a rule of thumb:
    - as soon as a pattern crosses the boundaries of architectural elements it can be seen as an architectural pattern
      e.g. MVC
- Groups of architectural patterns are related to –
  - domain/business logic
  - data sources and O/R mapping
  - (web) presentation and session handling
  - distribution and concurrency
  - basic issues
- Examples: Model-View-Controler, Client-Server, Blackboard (→ SWAQ)





Architectural Styles Examples

- Architectural model of a system may conform to a generic architectural style

  A style is a set of constraints which apply system-wide

- Awareness of these styles simplifies defining system architectures
  - adherence to styles eases system understanding, maintenance and evolution
  - Do not mix styles on the same dimension
- Architectural styles act in different dimensions

Dimension	Example Styles
Communication	Service-Oriented Architecture (SOA), Message Bus, REST, …
Deployment	Client/Server, N-Tier, 3-Tier, Microservice
Structure: Primary / Secondary	Component-Based, (Object-Oriented), Layered Architecture, Microservice

http://msdn.microsoft.com/en-us/library/ee658117.aspx

Style Example

- Modern OO systems are usually grouped in layers
  - each consisting of one or more subsystems having a cohesive responsibility
  - higher layers are supposed to call lower layers
    - only layer directly below in a *strictly* layered architecture
      - in order to limit coupling between layers
    - in practice relaxed layered architectures widely used
      - as dependencies to foundation classes (such as in java.util) may appear from all layers
- Layers are not the same as tiers!
  - *layer → conceptual separation of software*
  - *tier → physical separation on servers*
  - *But in practice often used inter-changeably*

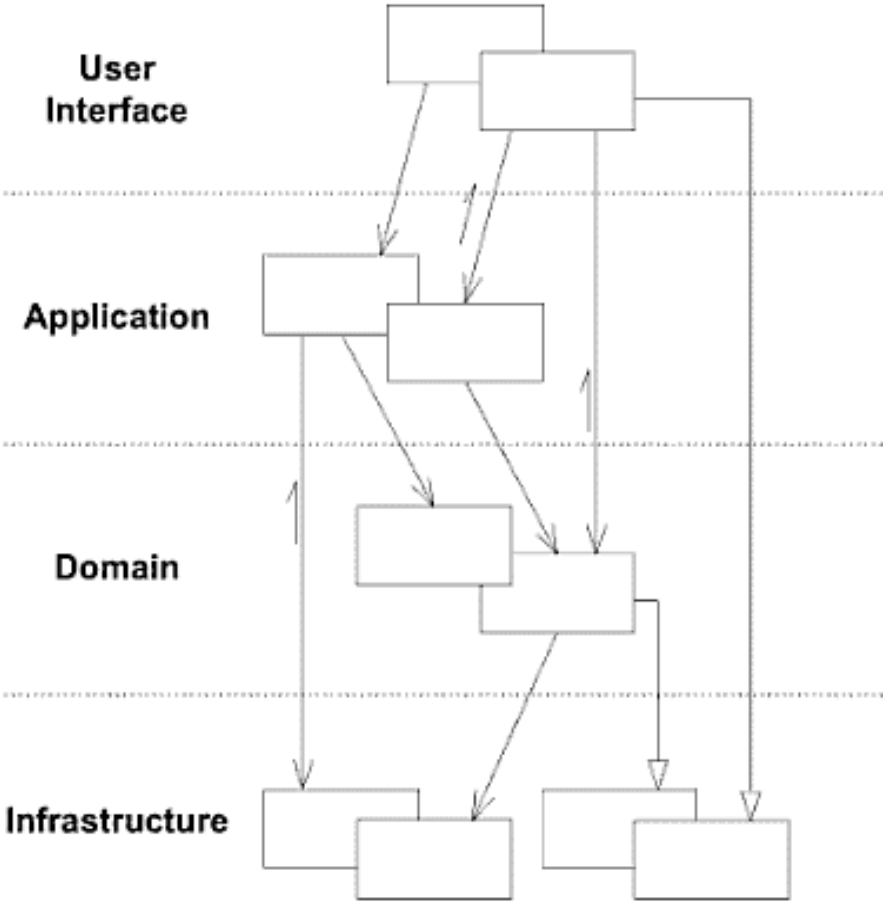
Layered Architecture I

Benefits:

- reduces “accidental” complexity
- improves modifiability
- clear separation of concerns
- independent exchangeability
- simplified testing

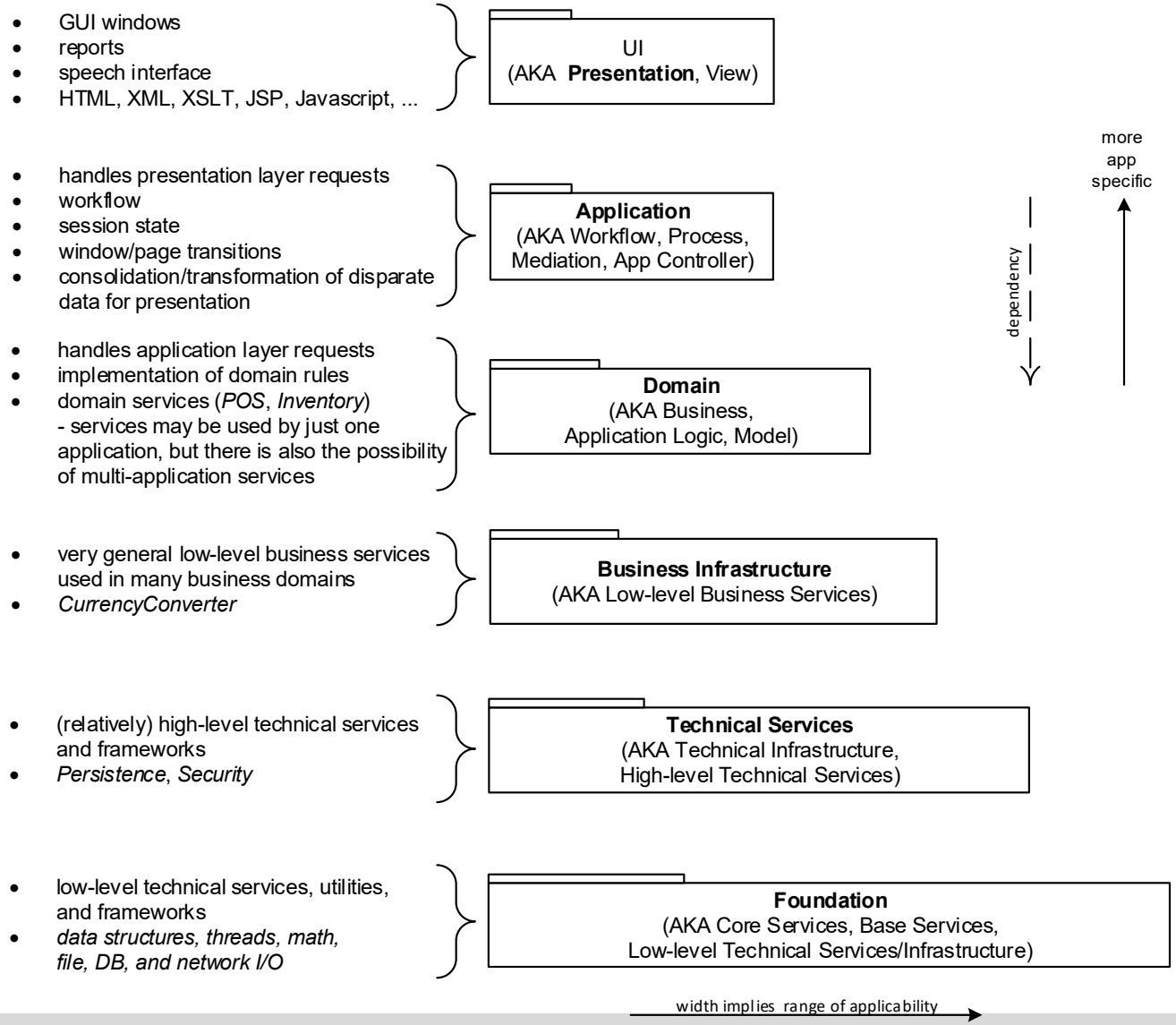
Drawbacks:

- usually increases the amount of classes
  - through facades or data transfer objects
  - *however, these are patterns in their own right that help to better deal with complexity*

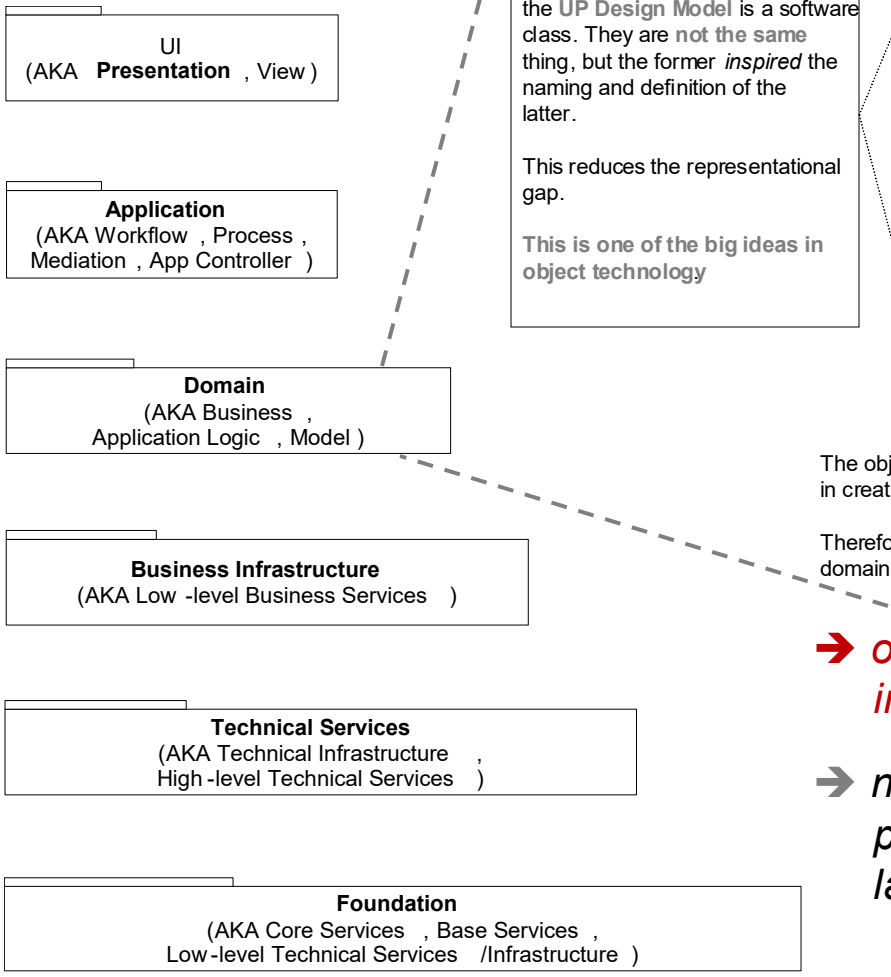


[Evans]

Layered Architecture II [Larman , Ch 13]



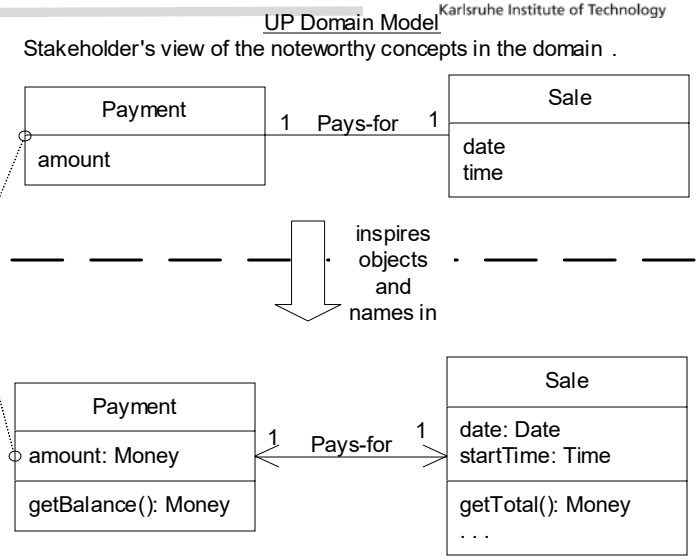
Domain Layer vs. Domain Model



Payment in the **UP Domain Model** is a concept, but a Payment in the **UP Design Model** is a software class. They are **not the same** thing, but the former *inspired* the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology



Domain layer of the architecture in the UP Design Model

The object-oriented developer has taken inspiration from the real world domain in creating software classes .

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software , has been lowered.

➔ *object design happens primarily in the Domain layer*

➔ *nevertheless the GRASP / SOLID patterns can be applied on other layers as well*

[Larman]

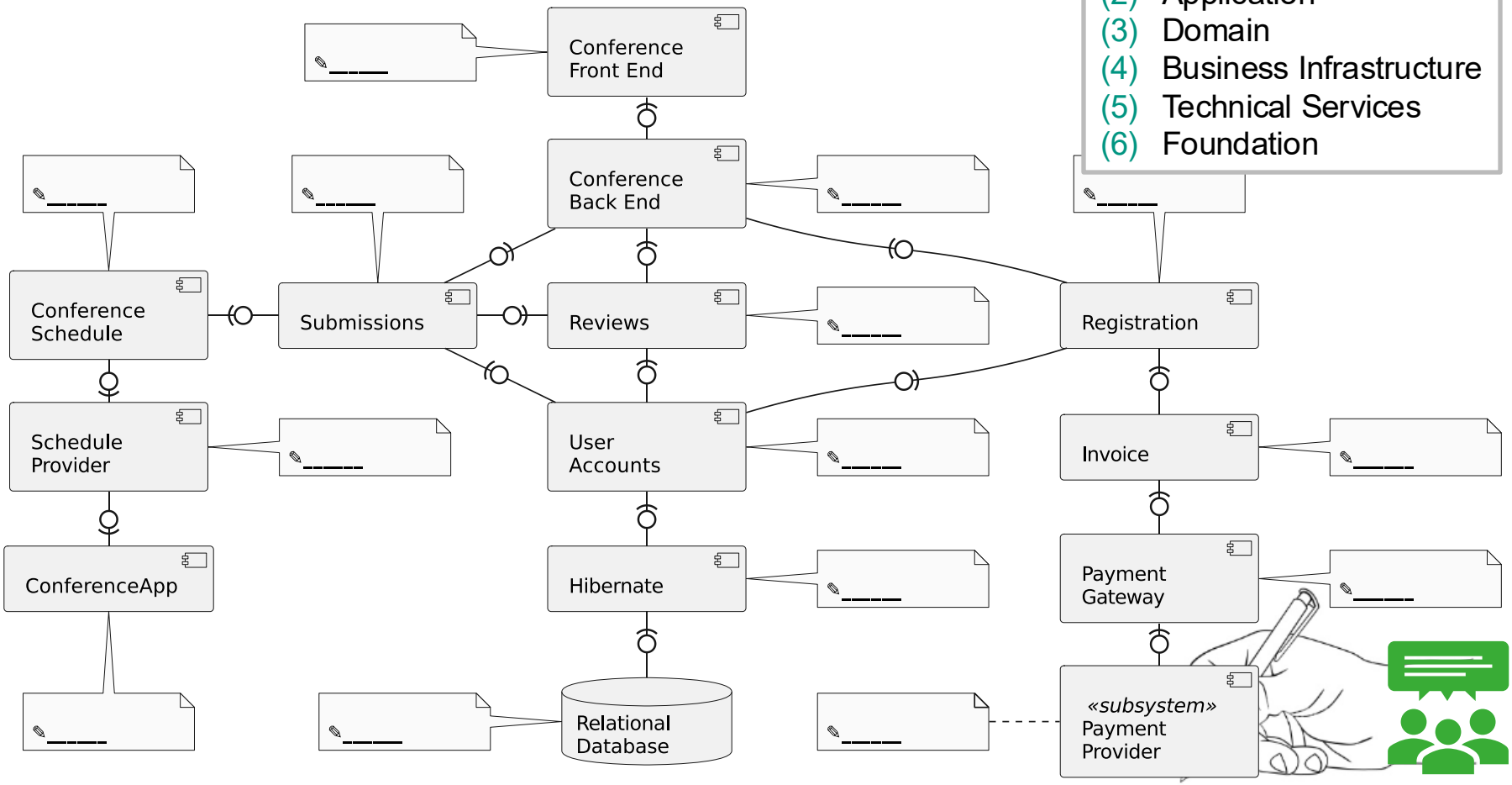
Caveat: OO Analysis and OO Design Model not the same!

- OO Analysis (OOA) (Domain Model)
  - modelling concepts of the application world
- OO Design (OOD) (e.g., in the Domain Layer)
  - prescription of implementation
  - OOD classes are inspired by OOA classes
  - same name, OOD class represents OOA concept
  - Additional OOD classes (from technical domain, most likely absent in requirements), e.g., scheduler, access control, data conversion, etc
  - OOD avoids cyclic dependencies
    - identify root class from where you can navigate to all dependent classes
      - (convert cycle into tree)
    - common pittfall to forget removal of cyclic OOA dependencies in OOD!

Break-Out Discussion

- On which layer does each component reside?

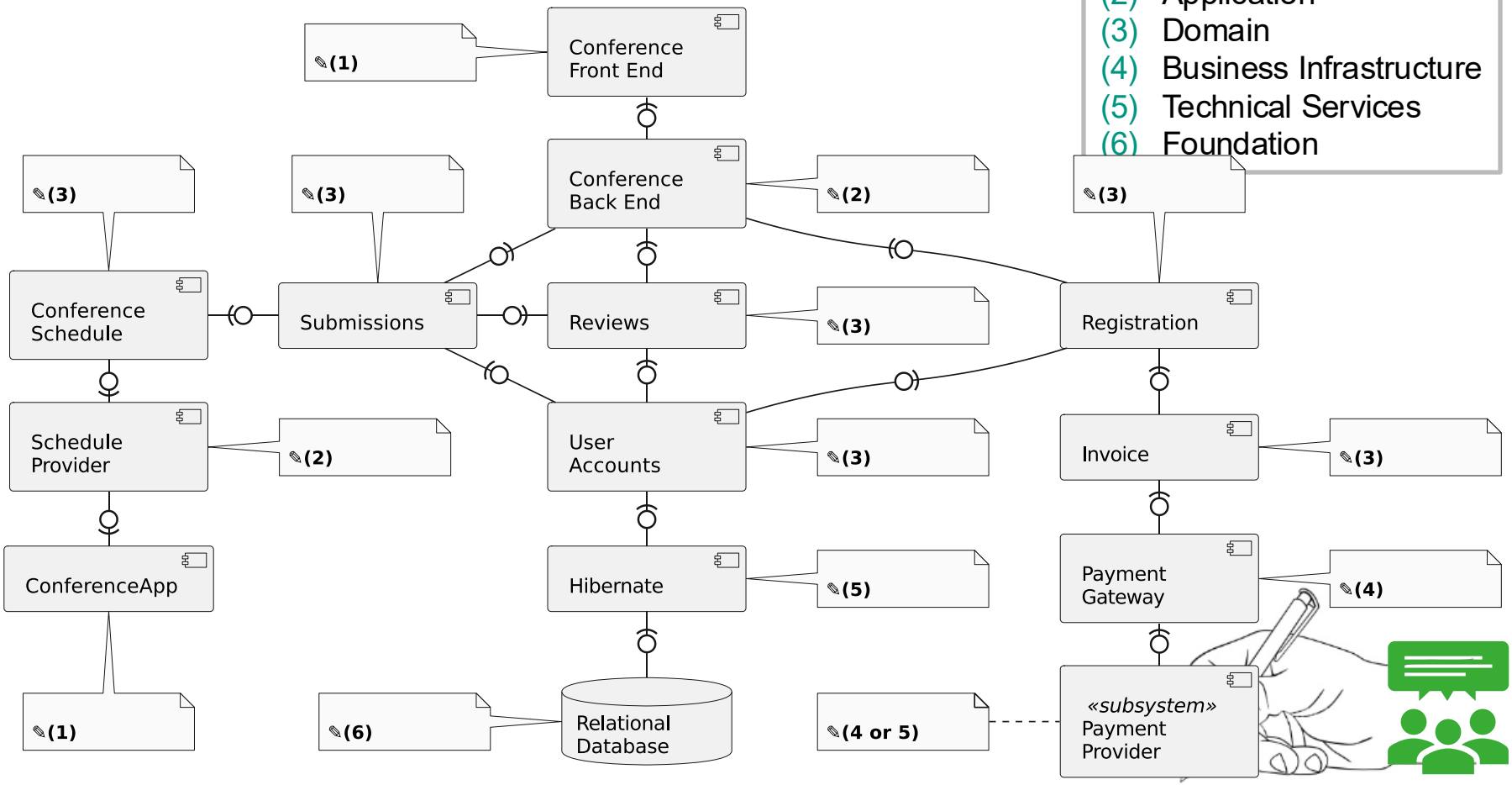
(1) User Interface
(2) Application
(3) Domain
(4) Business Infrastructure
(5) Technical Services
(6) Foundation



Break-Out Discussion

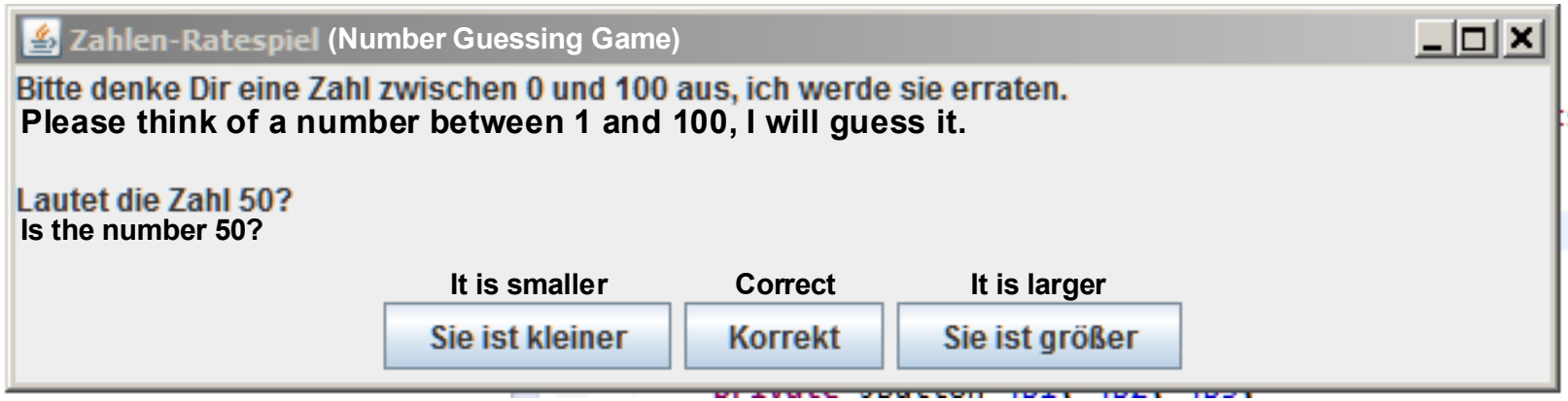
- On which layer does each component reside?

(1) User Interface
(2) Application
(3) Domain
(4) Business Infrastructure
(5) Technical Services
(6) Foundation



MODEL VIEW CONTROLLER IN A LAYERED ARCHITECTURE

SmartUI



What may be wrong with the following code from an architectural perspective?

SmartUI – Source Code

```
public class JFrameExample extends JFrame implements ActionListener {
  /*...*/
  public void actionPerformed(ActionEvent ae) {
    if (ae.getActionCommand().equals(jb2.getText())) {
      JOptionPane.showMessageDialog (this, "Juuuuhuu, bis bald!");
      System.exit(0);
    } else if (ae.getActionCommand().equals(jb1.getText())) {
      upper = number;
      number = number - (number - lower) / 2;
    } else if (ae.getActionCommand().equals(jb3.getText())) {
      lower = number;
      number = number + (int) ((upper - number) / 2.0 + 0.5);
    }
    if (lower == number || upper == number)
      question.setText("Ja, was denn nun?");
    else
      question.setText("Lautet die Zahl " + number + "?");
  }
}
```

Separation of Concerns

- ➔ “Smart UIs” are widely known as an “anti-pattern”
  - *i.e.* ***something to avoid***

- One of the core tenets of good software engineering is separating presentation and domain logic (i.e **model-view separation**), since – [Larman, p.209]
  - they deal with different concerns
    - use different libraries, skills etc.
  - it allows to create different views for an application
    - e.g. HTML, command line, WAP...
  - testing UI objects is usually hard
    - model-view separation facilitates testing of the application core

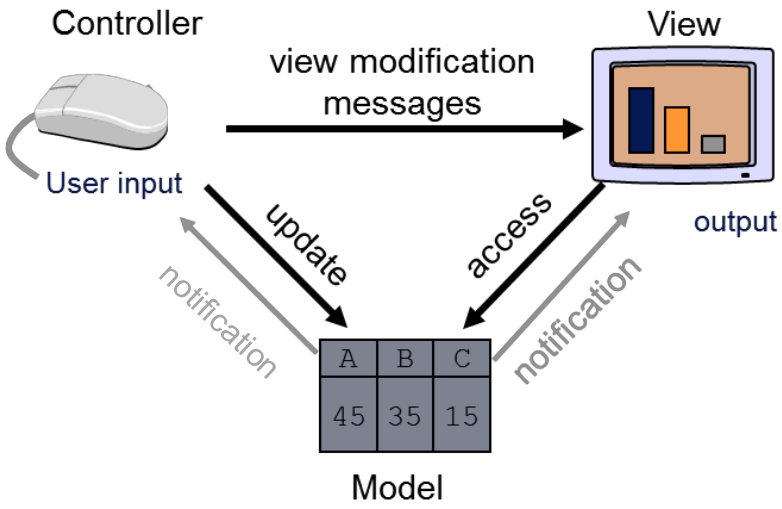
- Separation of control logic and UI is also recommended
  - ➔ Model-View-Controller
    - although often not as easy (and obvious) with common UI frameworks
    - e.g. Java seduces to have both in the same class
      - which is fine, however, in most cases

Arch. Pattern: Model View Controller (MVC)

Summary	Divides an interactive application into three elements. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.
Context	Interactive applications with a flexible number of user interfaces.
Problem	**Several views need to be kept consistent. New views may be added dynamically.**
Solution	MVC divides an interactive application into three areas: processing, outputs and input. The model component encapsulates core data and functionality. The model is independent of specific output representations or input behaviour. View components display information to the user. A view obtains the data from the model. There can be multiple views for one model. Each view has an associated controller. Controllers receive input events and translate these into service requests for the model or the view. The user interacts with the system solely through controllers.

Model View Controller Continued

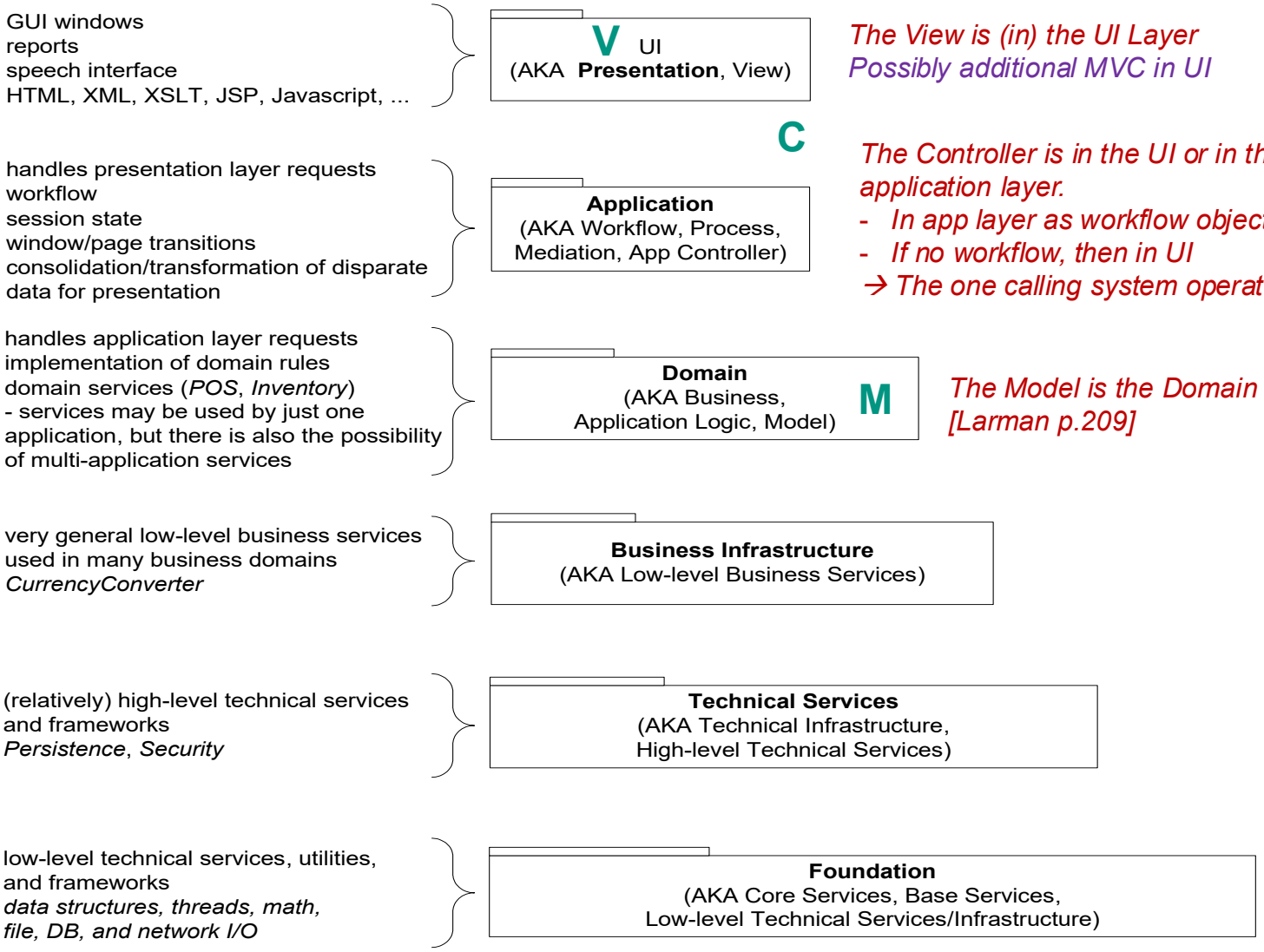
Class: Model	Collaborators
Responsibility • Provides functional core of the application • Registers dependent views and controllers • Notifies dependent components about data changes	• View • Controller



Class: Controller	Collaborators
Responsibility • Accepts user input as events • Translates events to service requests for the model or display requests for the view • Implements the update procedure if required	• View • Model

Class: View	Collaborators
Responsibility • Creates and initializes its associated controller • Displays information to the user • Implements the update procedure • Retrieves data from model	• Controller • Model

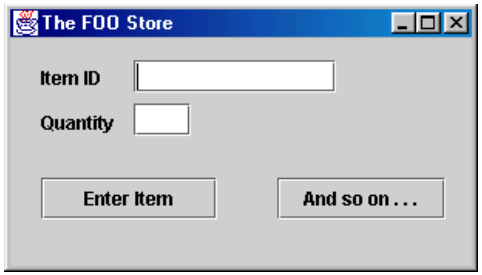
Architectural applications of MVC



[Larman, p.308]

The User Interface Layer…

- … is responsible for the presentation of data to the user
- … and for managing the interaction with the user
  - i.e. the screen flow

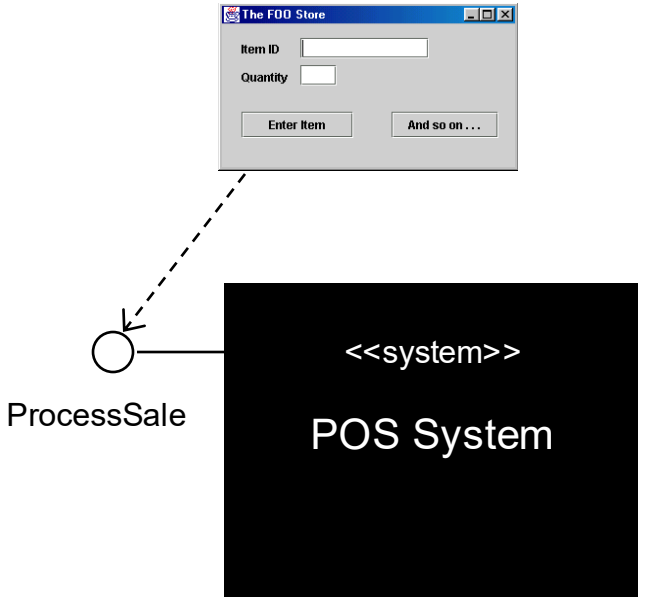


- *In Java: once the control flow is handed to the Swing UI it remains there*
  - until the user triggers an ActionEvent
    - and the UI "calls back"
  - known as the "Hollywood Principle": *don't call us, we call you*

  ➔ UI event handlers **should not process** system events directly
    - they forward the UI event to the application facade
      - i.e. to system operations in the facade
        - that trigger the processing in the domain layer

Application Layer *(Service Layer in [Fowler])*

- The part of the system that distributes the incoming requests, responsible for –
  - remembering session state and controlling flow of work
    - i.e. by controlling the order of windows (or web pages)
  - implementing the system operations
    - e.g. of Process Sale in our POS example
    - or Session Beans in EJB-based systems
      - a.k.a Session Facades



- In very small systems it is optional to have an application layer
  - i.e. domain objects can be directly called by UI
  - however, for multi-tier architectures it is usually mandatory
    - due to more complex session handling [Larman, p.572]
- A good rule of thumb is to have one controller/facade per use case

Modelling Alternatives for Controller Layer

- As said before, various strategies exist for accommodating the system operations –

1. one controller class per use case
   - works well for systems with many use cases
   - *per CRUD use case -> Session Beans in JEE*

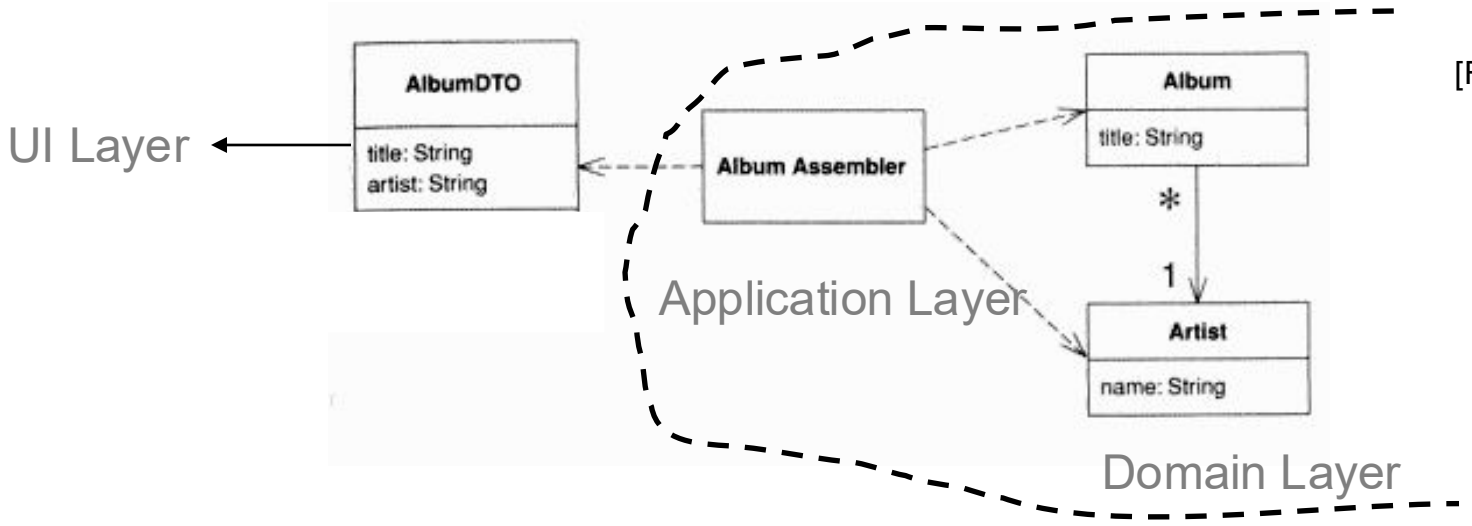
2. one controller class per application/system
   - feasible for smaller systems with ~ <12 system operations
   - Façade controller

3. direct access to appropriate domain objects
   - reduces passing through of parameters
   - but easily brings control flow logic into the domain model

[Larman, Sec. 17.13]

Data Transfer Objects

- A DTO is a (serializable) object that carries data between processes or architectural elements (i.e. layers, components, tiers etc.)
  - in order to reduce the number of remote method calls
    - it usually contains only attributes and getters and setters for them
    - it may contain methods for serialization in distributed systems

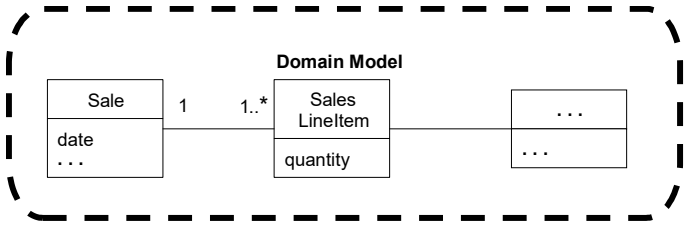


[Fowler]

- Domain objects usually cannot be transferred due to complex dependencies

The Domain Layer

- The domain layer contains the domain model and the business logic
  - inspired by real-life objects
  - this is where object design will become necessary
    - and object-oriented programming is showing its full usefulness



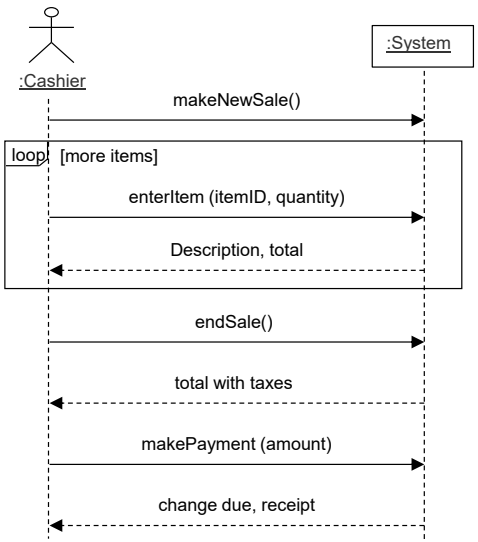
handles application layer requests
implementation of domain rules
domain services (*POS*, *Inventory*)
- services may be used by just one application, but there is also the possibility of multi-application services

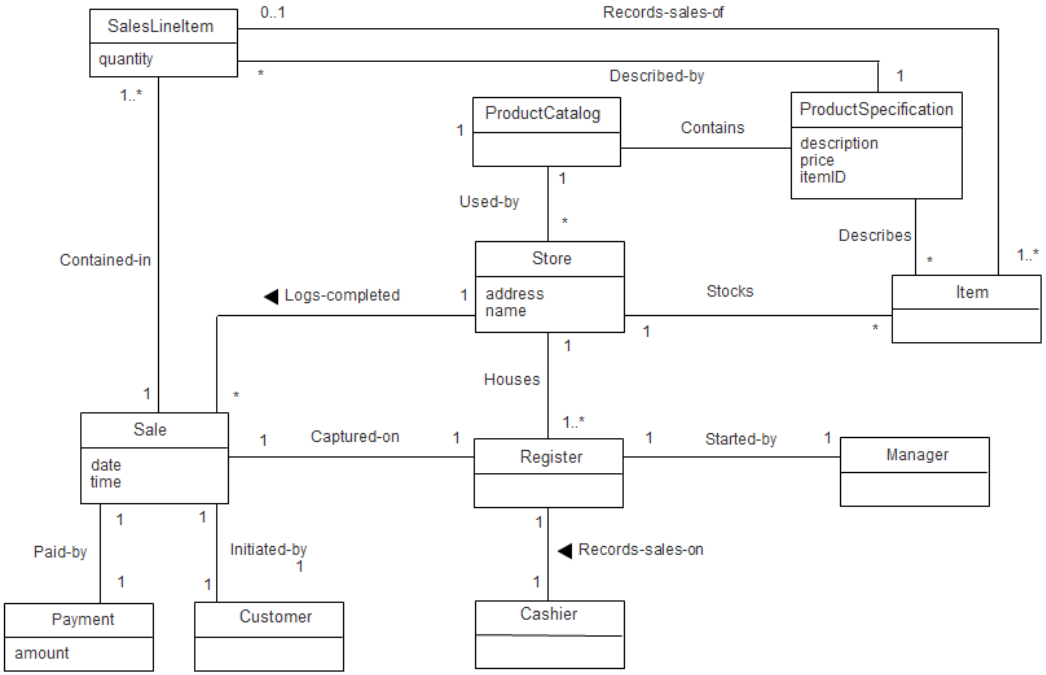
Domain
(AKA Business,
Application Logic, Model)

➔ Thus, it is typically very application specific
  ➔ domain objects should be kept in the domain layer
    - in single-process desktop applications (without serialization) they may be passed to application, UI or persistence layer

Hands on Architectural Modelling

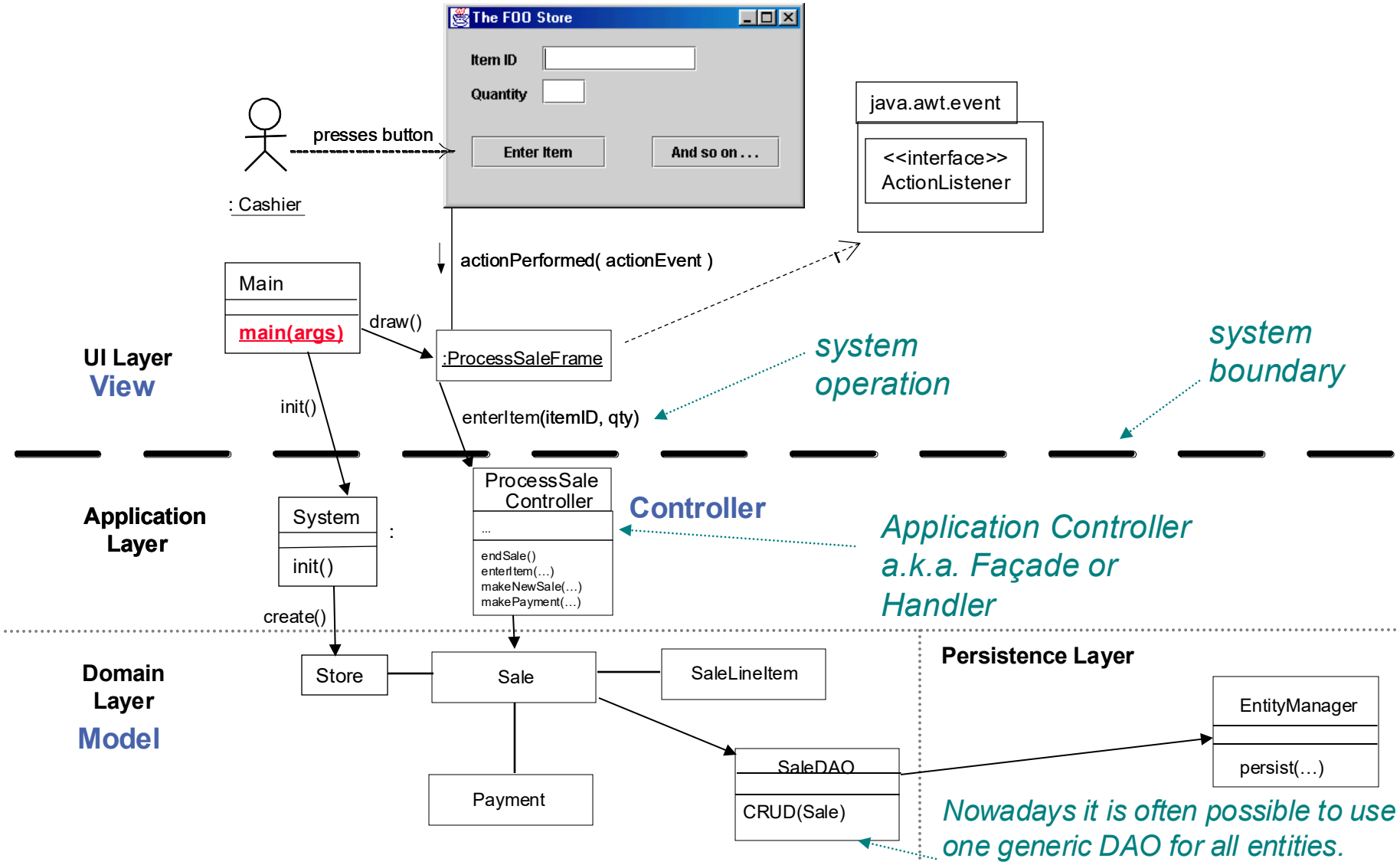
- Let‘s design an architecture for the POS example
  - based on the previous models of the system
- Focus should be on a layered class diagram
  - but you may want to add „method call arrows“ for the sake of the example





Architecture Example

[Larman p.304 + extensions by Hummel]



Observer Pattern

Intent Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

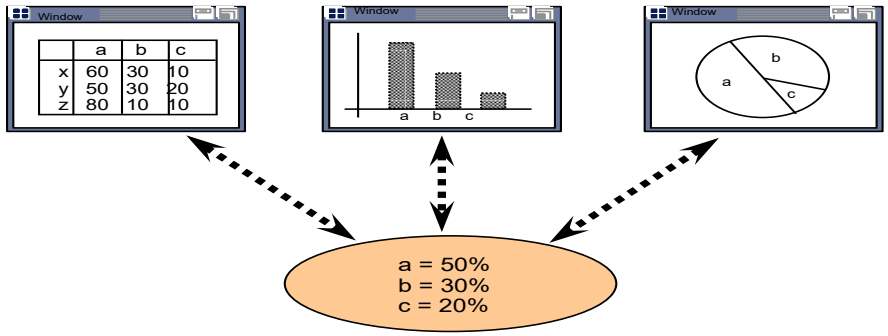
A.K.A. Dependents, Publish-Subscribe

Applicability When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects into separate objects lets you vary and use them independently

When a change to one object requires changing others, and you don't know how many objects need to be changed.

When an object should be able to notify other objects without making assumptions about who these objects are

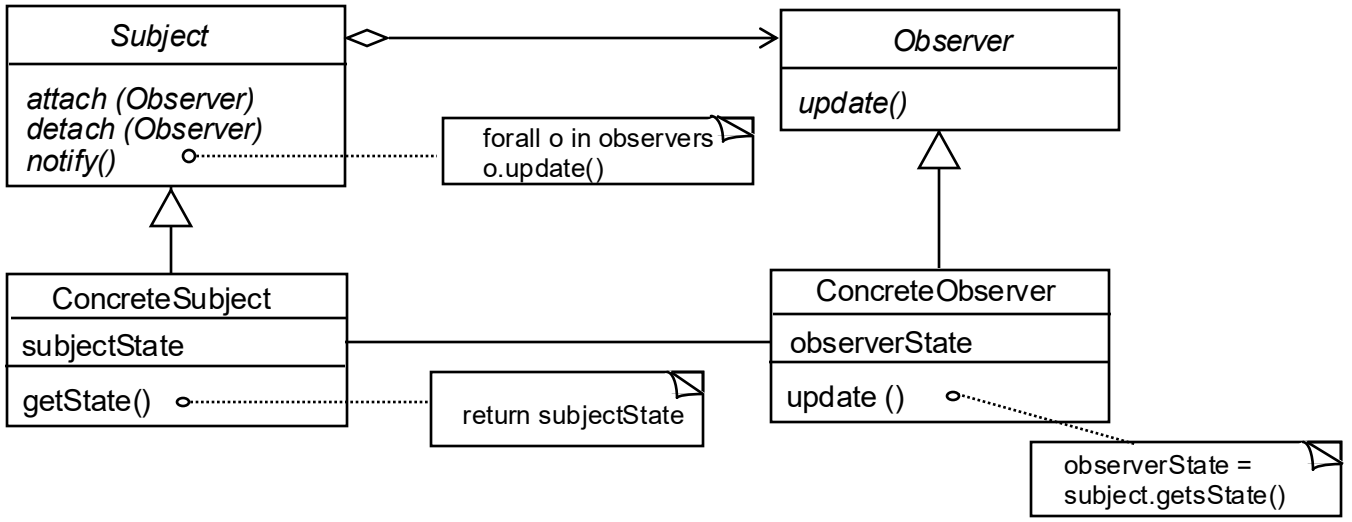
Motivation



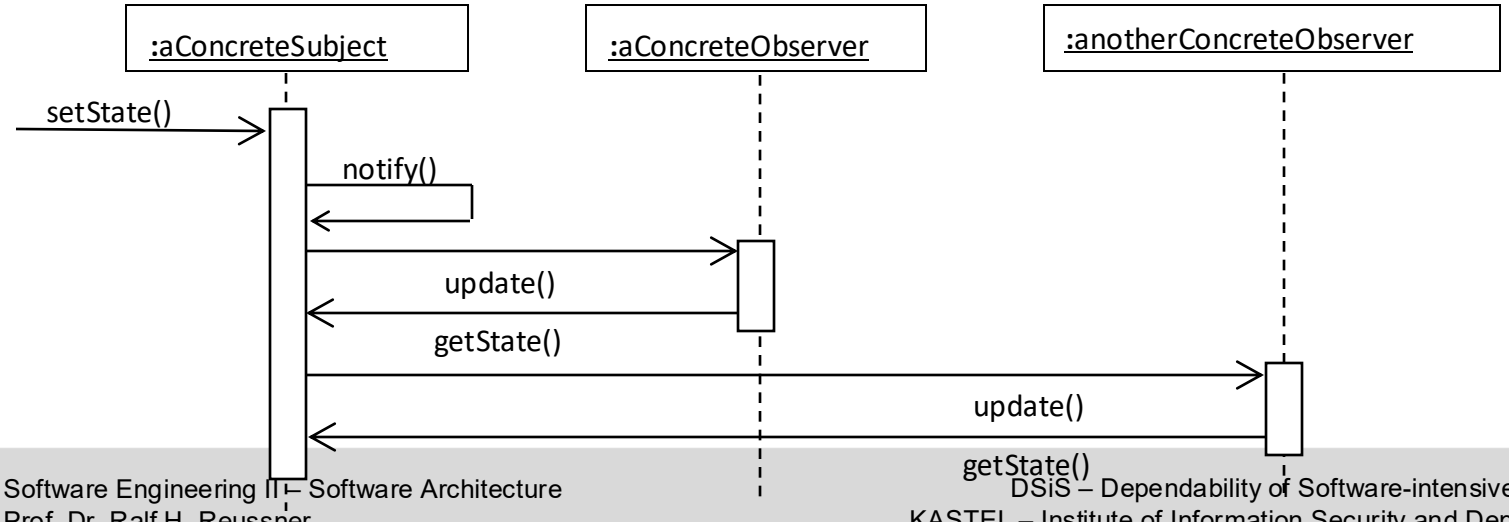
Observer Pattern Continued

Participants Subjects, Observer, ConcreteSubject, ConcreteObserver

Structure



Collaborations

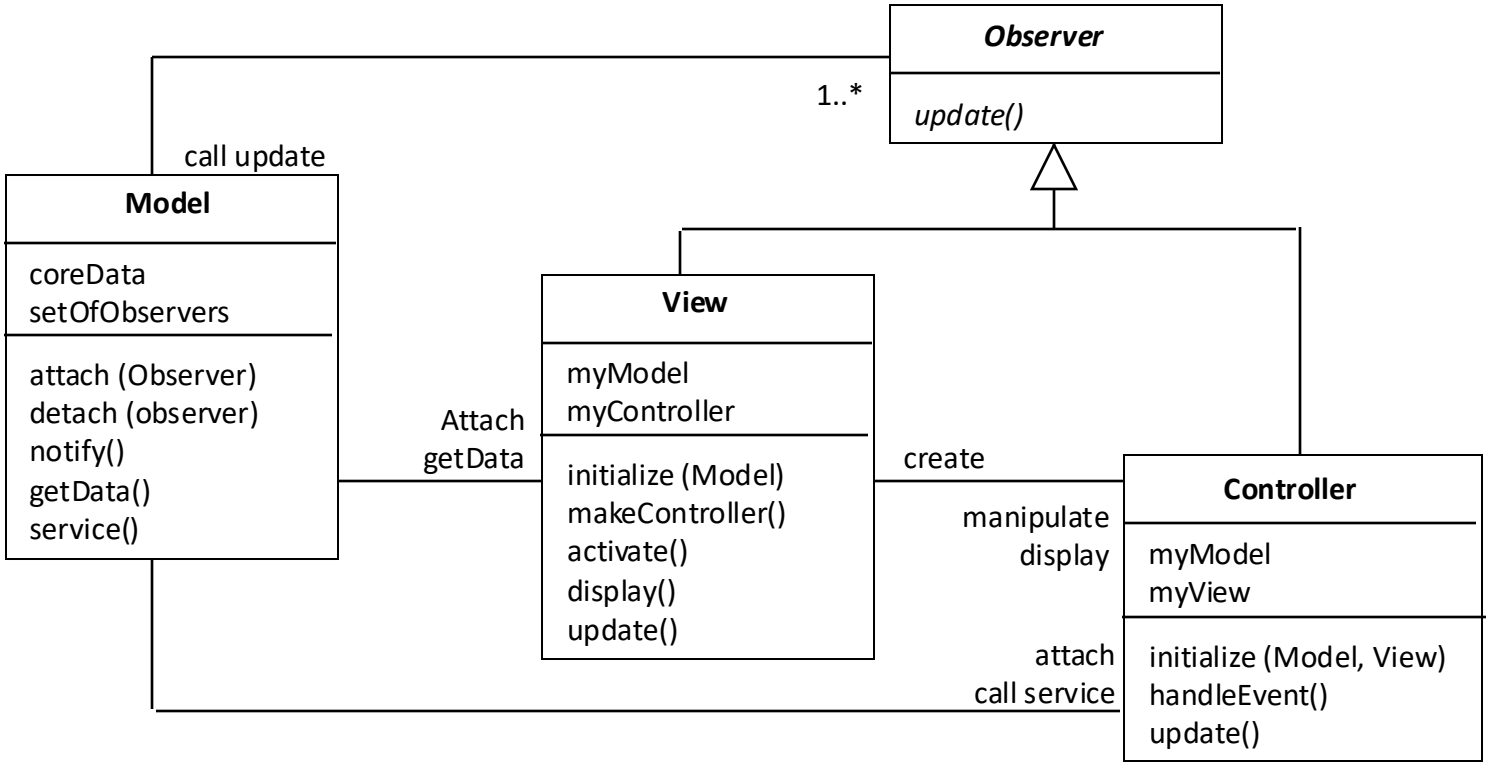


Model View Controller & Observer

Structure

The model component contains the functional core of the application (the business logic).

In order to allow “upward” calls, usually implementing an observer is required.



How to implement this in the POS example: See [Larman], Chapter 26

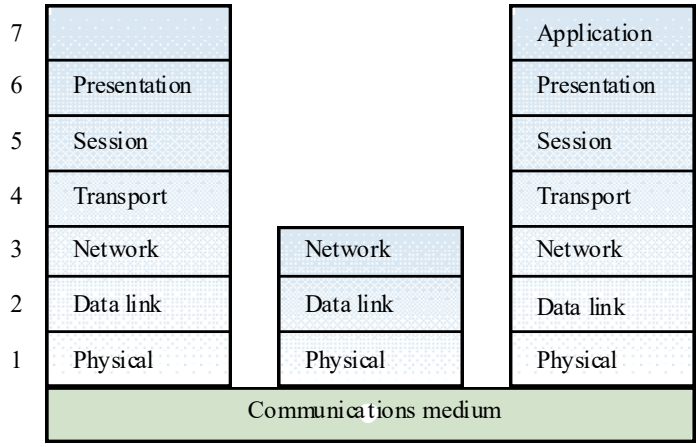
Reference Architectures

[Reussner 2008]

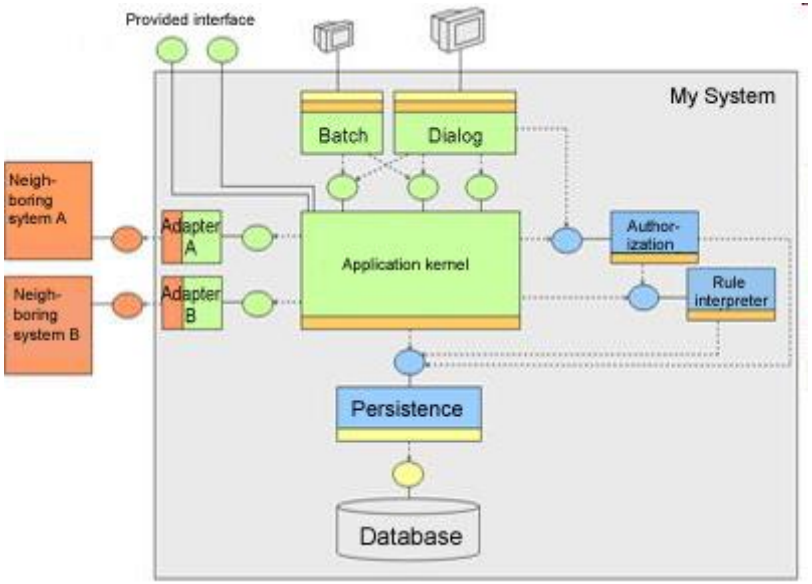
- The architectural model of a system may conform to a generic reference architecture
  - reference models are derived from a study of the application domain
  - it acts as a standard against which systems can be built and evaluated

➔ An awareness of these can simplify the problem of defining system architectures

- as they can be used as templates



OSI 7 Layer model

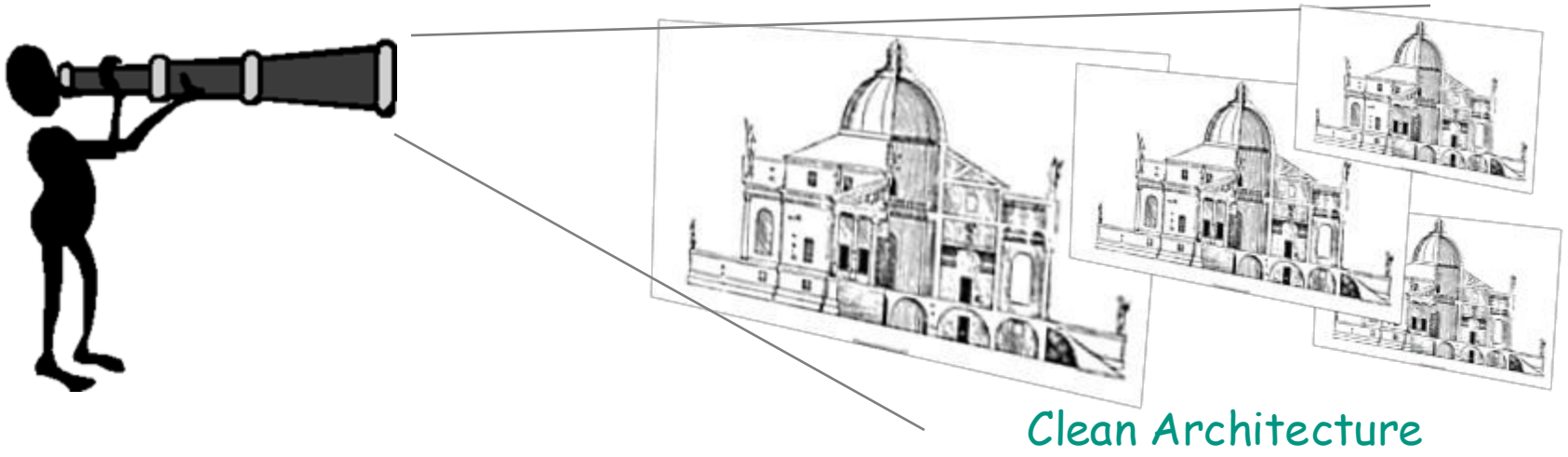


sd&m's Quasar

- other example: clean architecture (next lecture)

Conclusion

- Architectures ease to shift decisions on an architectural level
  - instead of code-centric development
  - in order to better manage complexity
    - and to benefit from re-use and other amenities
- Architecture view points examples: Structural, behavioural, deployment
- Different ways of reuse
- Layered reference architecture, MVC, Façade, Observer
- References



References

- S. Ambler
  The Object Primer: Agile Model-Driven Development with UML 2.0
  Cambridge University Press, 2004

- E. Evans
  Domain-Driven Design
  Addison-Wesley, 2004

- Martin Fowler
  Patterns of Enterprise Application Architecture, Addison-Wesley, 2003

- Gang of Four (E. Gamma et al.)
  Design Patterns
  Pearson Education, 1995

- C. Larman
  Applying UML and Patterns (3rd ed.)
  Prentice Hall, 2004

- T. Langner & D. Reiberg
  J2EE und JBoss: Grundlagen und Profiwissen
  Hanser, 2006.
  ➔ **Sample Chapter**: http://files.hanser.de/hanser/docs/20051107_2012053191536103_978-3-446-40837-1_Kap01.pdf

Ralf Reussner, Wilhelm Hasselbring
Handbuch der Software-Architektur
2. Auflage, dPunkt-Verlag, Heidelberg, 2008

Ralf H. Reussner et al.
Modeling and Simulating Software Architectures — The Palladio Approach
MIT Press, 2016

Some slides from:
Ian Sommerville,
"Software Engineering",
7th Ed., Addison-Wesley,

References

- R. N. Taylor, N. Medvidovic, and E. M. Dashofy: *“Software Architecture: Foundations, Theory, and Practice”*, Wiley, Hoboken, 2009
- Len Bass, Paul Clements, and Rick Kazman. "Software Architecture in Practice”. Addison-Wesley, 3rd edition, 2013.
- Paul Clements et al. *"Documenting Software Architectures: Views and Beyond"*, 2nd edition, Addison-Wesley, Boston, 2010
- Erik Burger: *“Flexible Views for View-based Model-driven Development”,* phd thesis, Karlsruher Institut of Technologie (KIT), 2014

Appendix: Additional Literature (1)

- Richard Monson-Haefel:
  "97 Things Every Software Architect Should Know",
  O'Reilly, Sebastopol, 2009

- Oliver Vogel et al.:
  "Software-Architektur: Grundlagen - Konzepte - Praxis",
  2. Aufl., Spektrum Akadem. Verlag, Heidelberg, 2009

- Gregor Engels et al.:
  „Quasar Enterprise“,
  dPunkt-Verlag, Heidelberg, 2008

Additional Literature (2)

- Ian Gorton:
  "Essential Software Architecture",
  Springer, Berlin, 2006
- Markus Völter and Thomas Stahl:
  "Model-Driven Software Development",
  Wiley, New York, 2006
- Johannes Siedersleben:
  "Moderne Software-Architektur",
  dPunkt-Verlag, Heidelberg, 2004
- Torsten Posch et al.:
  "Basiswissen Software-Architektur",
  dPunkt-Verlag, Heidelberg, 2004

Additional Literature (3)

- Stephen J. Mellor
  "MDA Distilled",
  Addison-Wesley, Boston, 2004

- Martin Fowler:
  "Patterns of Enterprise Application Architecture",
  Addison-Wesley, 2003

- Christine Hofmeister et al.:
  "Applied Software Architecture",
  Addison-Wesley, 2000

- Jan Bosch:
  "Design & Use of Software Architectures",
  Addison-Wesley, 2000

Additional Literature (4)

- Frank Buschmann et al.:
  "Pattern-oriented Software Architecture",
  Wiley, New York, 1996-2007 (Vol. 1-5)

Paul Clements et al.:

- *"Software Product Lines: Practices and Patterns"*,
  Addison-Wesley, Boston, 2002
- *"Evaluating Software Architectures: Methods and Case Studies"*,
  Addison-Wesley, Boston, 2002