Karlsruhe Institute of Technology

Softwaretechnik II

Oliver Hummel, IPD

Topic 7
Software Architecture

SOFTWARE DESIGN AND QUALITY GROUP
INSTITUTE FOR PROGRAM STRUCTURES AND DATA ORGANIZATION, FACULTY OF INFORMATICS

sdq.ipd.kit.edu

Coarse Course Schedule (1)

Date	Tentative Content
Mo. 21.10.	Warm-Up
Di. 22.10.	Software Processes
Mo. 28.10.	cont.
Di. 29.10.	Agile Development
Mo. 04.11.	*Guest Lecture by Andrena Objects*
Di. 05.11.	Requirements Elicitation
Mo. 11.11.	cont. + Use Cases
Di. 12.11.	cont.
Mo. 18.11.	Requirements Analysis
Di. 19.11.	cont.
Mo. 25.11.	*Software Architecture*
Di. 26.11.	cont. + Component-Based Architectures
Mo. 02.12.	cont.
Di. 03.12.	Persistence Patterns

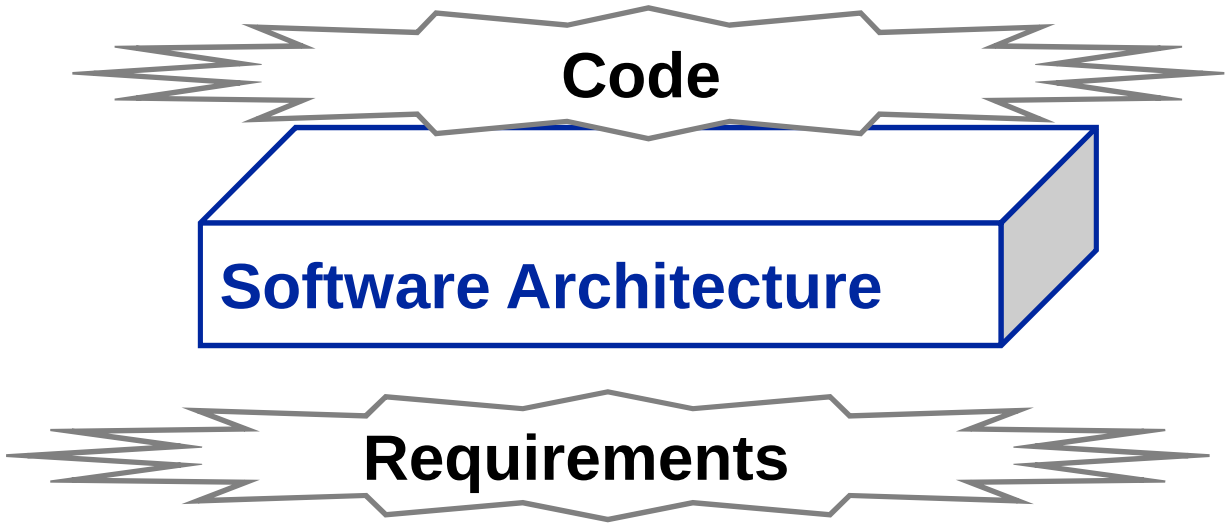
Overview on Today's Lecture

- Content
  - Introduction to Software Architecture
    - Context and Motivation
    - Some Terminology
    - Intro to Architectural Patterns
    - Layered Architecture
    - Some more Patterns
- Learning Goals
  - get acquainted with fundamental concepts of software architecture
  - understand the principles and patterns applied in layered OO architectures
  - be able to design simple layered architectures

What is an Architecture?

Several definitions exist:

- A software architecture defines the coarse-grained structure of the system
  - or more pragmatically: A software architecture captures design decisions which are hard to revert or which have to be made early



Architectural Design Decisions (1)

- 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?
  - What approach will be used to structure the system?
  - How will the system be decomposed into subsystems (modules, components)?
  - What management and evolution strategy should be used?
  - How will the architectural design be evaluated?
  - What are realistic evolution scenarios?
  - …>

Architectural Design Decisions (2)

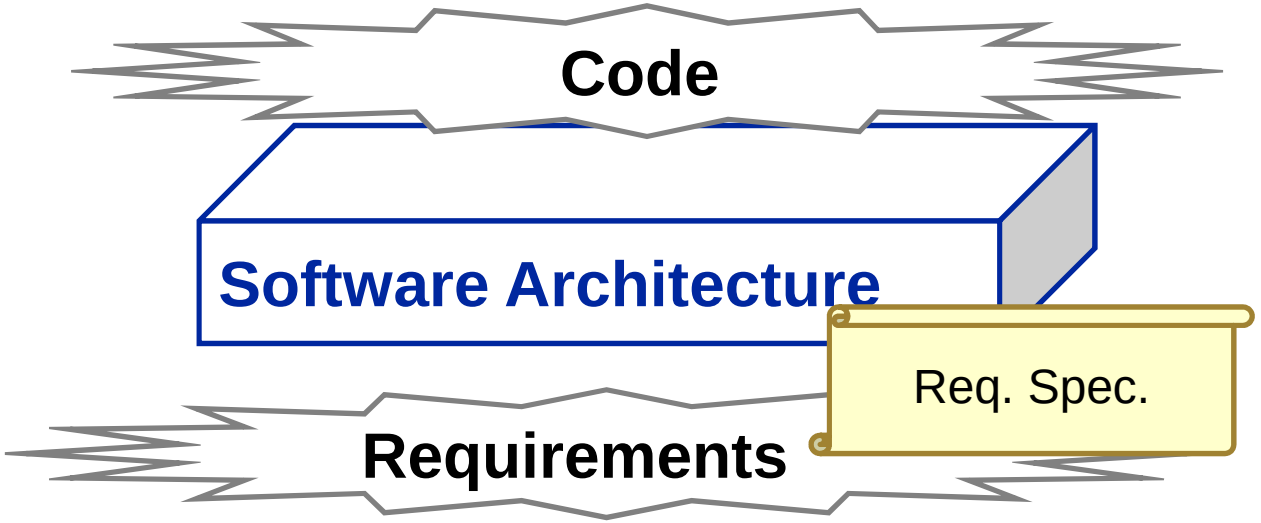
- architectural design decisions continued –
  - How should the architecture be documented?
  - Which components can or must be bought?
  - How to include legacy software?
  - How to communicate with existing software?
  - How to access existing data?
  - How does the architecture fit into the existing portfolio?
  - What can be re-used from older projects?
  - What should be re-used in the next project?
  - Is a product-line architecture appropriate?
  - …

➔ *Common reference architectures can simplify answering this question catalog*

- *as well as the usage of proven architectural patterns*

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

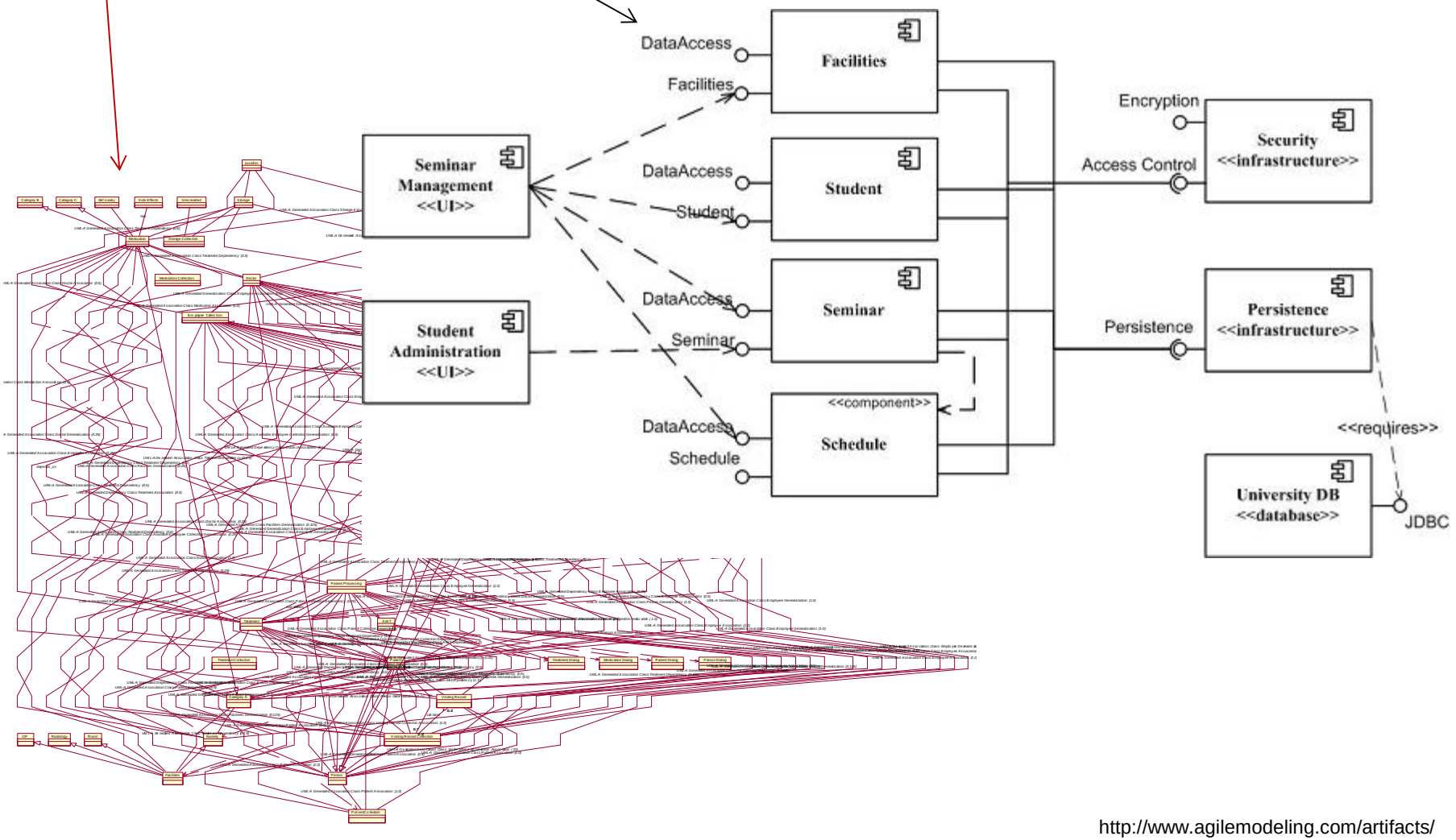


➔ It involves identifying major system components, their communications and mapping to hardware resources

What Constitutes a Software Architecture?

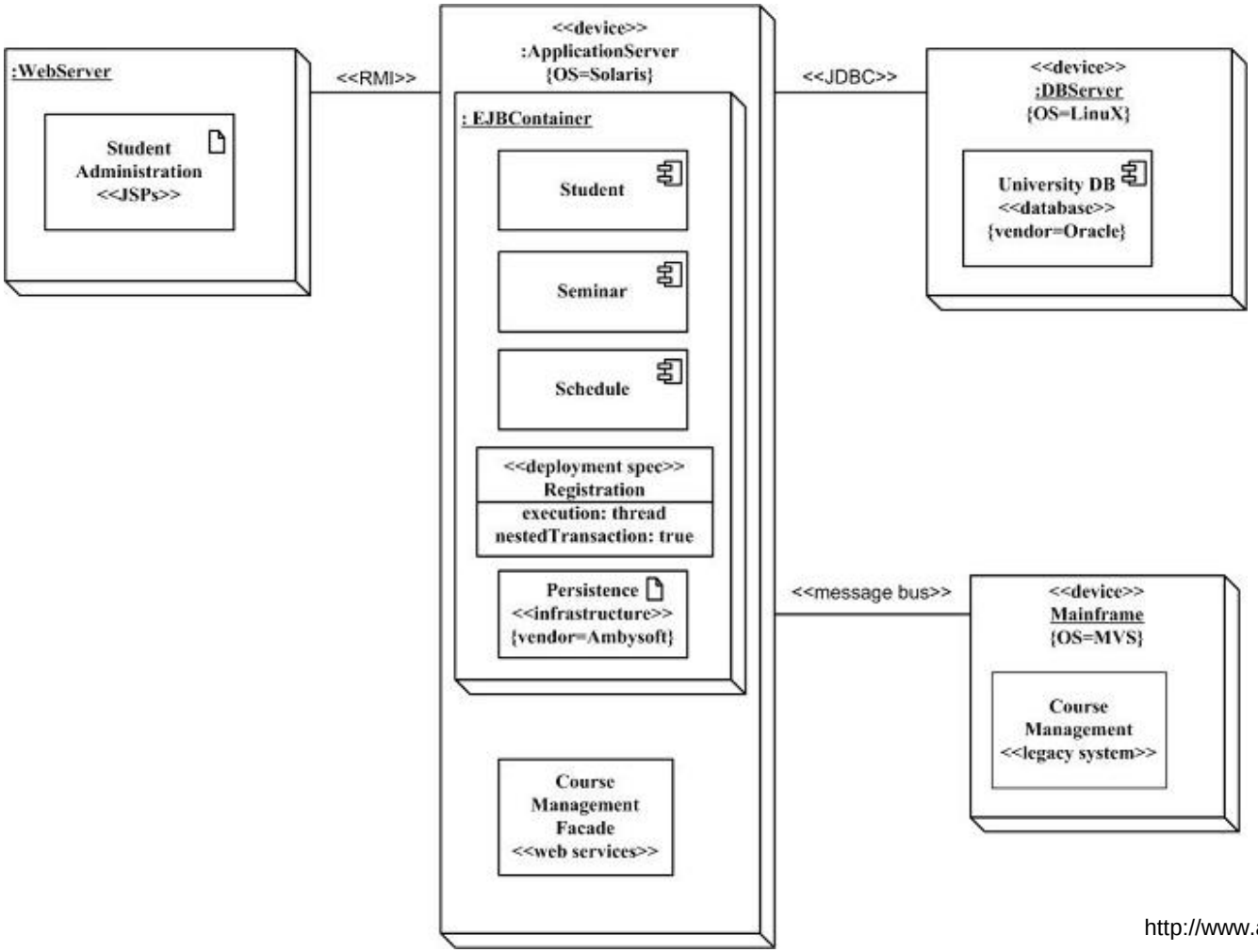
- *Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment...*
  - excerpt of the IEEE standard 1471
- However, the **logical architecture** is only one aspect
  - the **deployment architecture** is another one
    - ➔ which software component is deployed on which hardware node?
- In other words, the logical architecture of a system describes its logical partitioning into **layers, subsystems and packages**
  - and how they communicate with each other
    - e.g. in terms of sequence diagrams illustrating communication between components
      - roughly on the detail level of system sequence diagrams
  - ➔ dynamic view on the architecture

Design vs. Logical Architecture



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

Deployment View



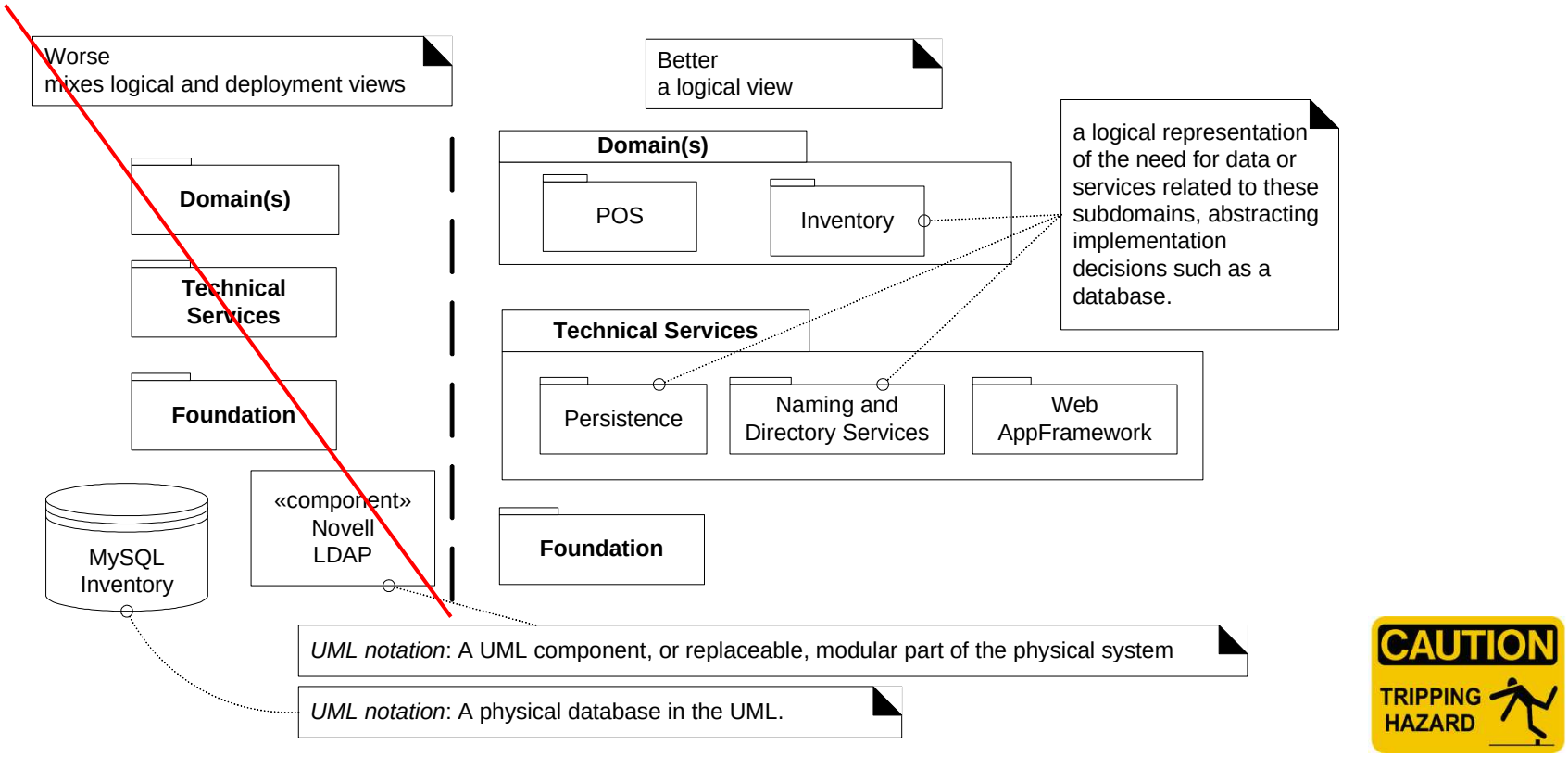
http://www.agilemodeling.com/artifacts

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

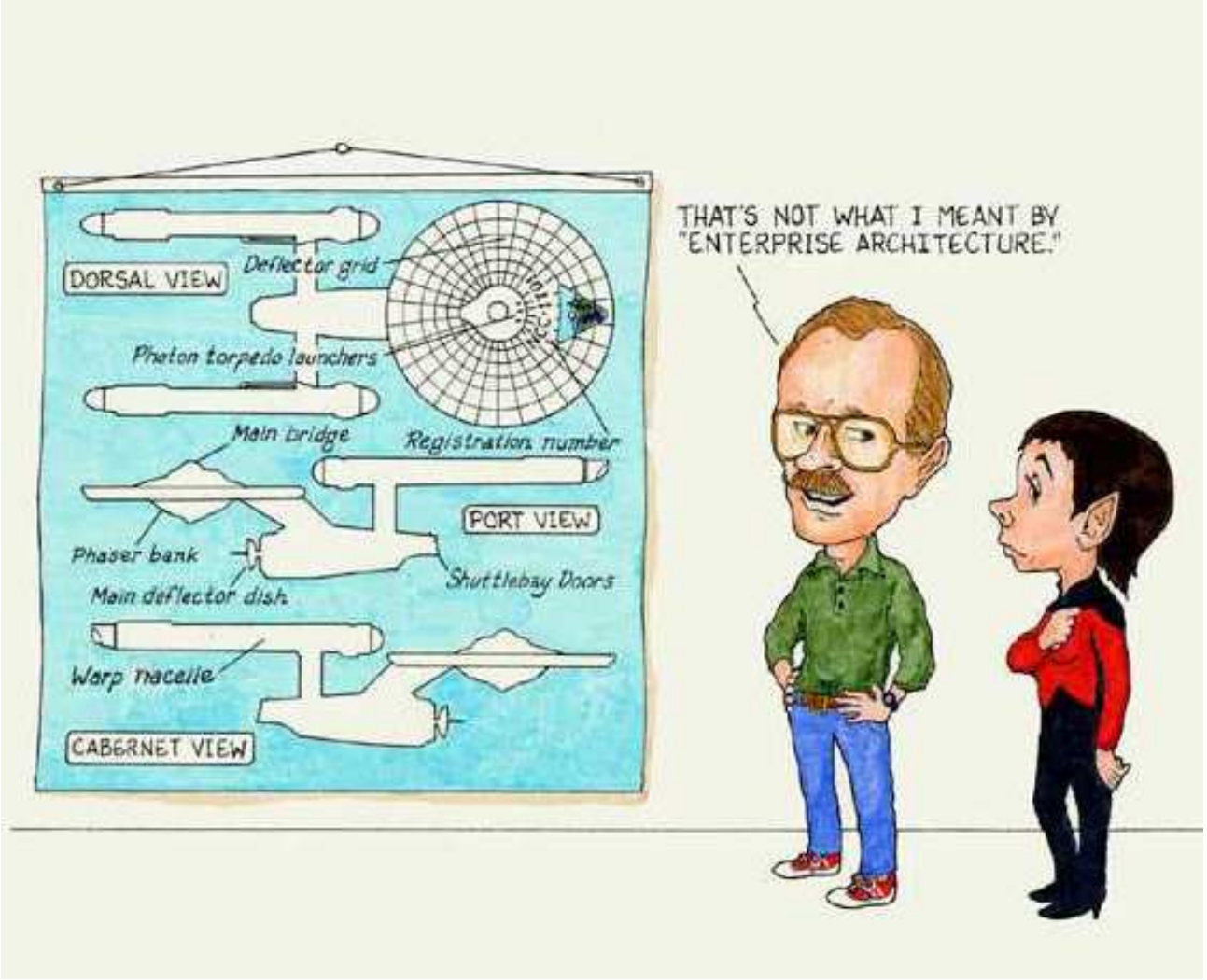
Guideline

- Do not mix logical and deployment architecture
  - although many systems require external resources…
    - such as a database

    … do not show them as part of the logical architecture [Larman05]



Relax for a Moment



Advantages of an Explicit Architecture

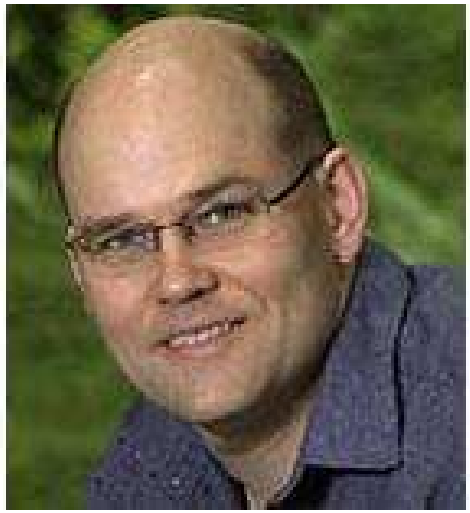
- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders
- System analysis
  - Analysis of whether the system can meet its non-functional requirements
- Large-scale reuse
  - The architecture may be reusable across a range of systems
  - Existing components 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.

Impact of the System Architecture

- Identify the most important goals of your system's architecture (non-functional requirements), e.g. –
  - Performance
    - localise operations in large-grained components to minimise sub-system communication
  - Security
    - use a layered architecture with critical assets in inner layers
  - Safety
    - isolate safety-critical components
  - Availability
    - include redundant components in the architecture
  - Maintainability
    - use fine-grained, self-contained components
  - Scalability
    - consider concurrency effects in case you need to distribute the system
  - ➔ *They may often contradict with each other and thus need to be balanced*

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]

➔ *Remember: A system (usually) reflects the organizational structure that built it*
known as Conway's law [Endres/Rombach03]

Some Further Terminology

- **A word of warning!**
  Software Architecture is a relatively new field (~20 years) where many terms are still overloaded

- Pattern
  - a proven solution to a recurring problem
- Architectural pattern
  - solution to recurring situation where several forces have to be balanced
- Architectural Style
  a) [Reussner] Cross-cutting principles (object-oriented style, modular style), independent of application, should not be mixed (within category)
     - like in buildings: baroque-style, classicist-style
  b) often synonymously (and incorrectly) used for Architectural Pattern
- Reference Architecture
  - defines domain concepts, components and subsystems that can be used by concrete instances

(Different) Architectural Styles Examples

- The 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 can simplify the problem of defining system architectures
  - however, most large systems are heterogeneous and do not follow a single architectural style
- Architectural styles may have different application areas

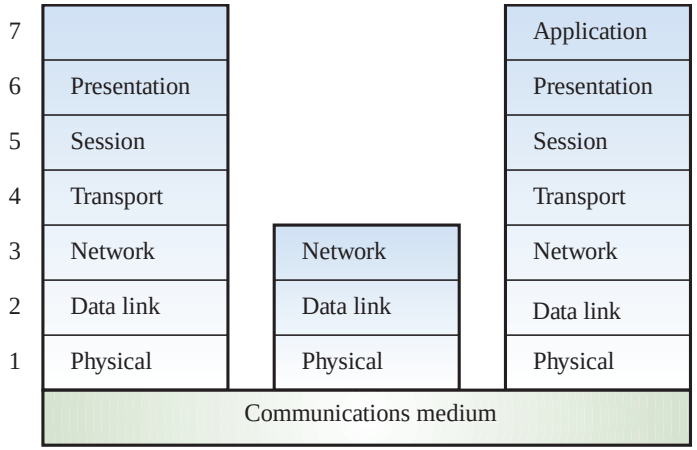
Category	Styles
Communication	Service-Oriented Architecture (SOA), Message Bus
Deployment	Client/Server, N-Tier, 3-Tier
Structure	Component-Based, (Object-Oriented), Layered Architecture

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

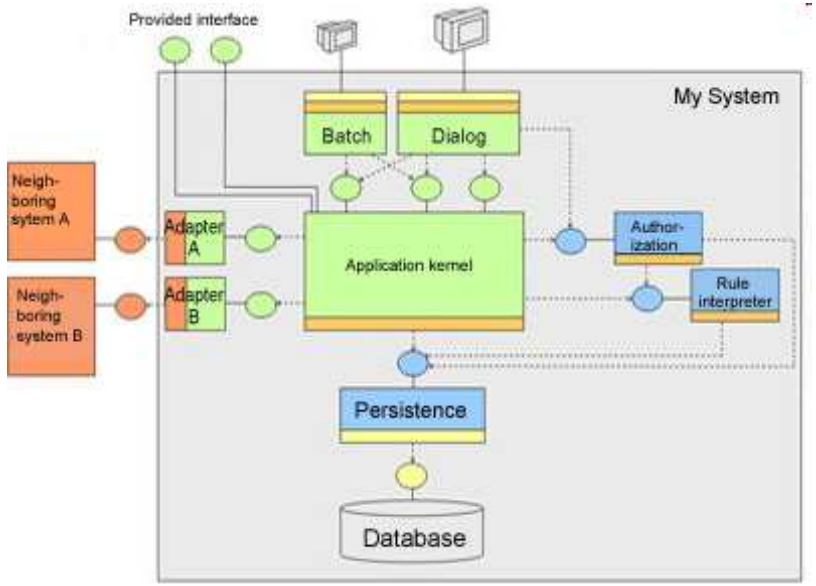
Reference Architectures

- 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

Herzliche Einladung

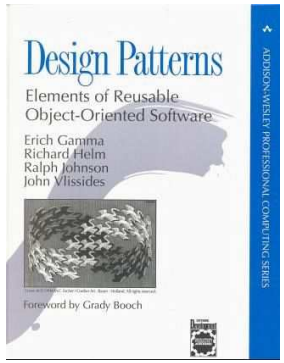
Software-Cluster

- Zum Cloud-Workshop des Software-Cluster
  - Aktuelles Cloud Computing aus Wissenschaft und Praxis
  - 9:30 bis 17:30 Uhr, am 03.02.14 im FZI House of Living Lab
  - Teilnahme für Studierende kostenfrei (Anmeldung erforderlich)
- Mit hochkarätigen Referenten
  - *Prof. Dr. Stefan Tai, Karlsruhe Institute of Technology (KIT)*
    Thema: "Cloud Systems“
  - *Dr. Vasilios Andrikopoulos, Universität Stuttgart*
    Thema: "Migrating to the Cloud: Making the Right Decision“
  - *Dr. Markus Bauer, CAS Software AG, Karlsruhe*
    Thema: "Konstruktion von Cloud-Lösungen – ein Praxisbericht”
  - *Dr. Jan Schaffner, SAP AG*
    Thema: "SAP HANA and In Memory Column Databases: Technical Fundamentals and Operation in the Cloud“
  - *Dr. Thomas King, audriga GmbH*
    Thema: "Praxisbericht zur Nutzung von Public-Clouds anhand von Amazon AWS„
  - *Thomas von Bülow, 1&1 Internet AG*
    Thema: Cloud-Forschung und Entwicklung

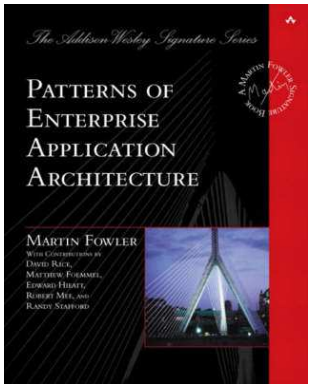
http://www.cyberforum.de/news-termine/termine/cyberforum-termine/details/veranstaltung/software-cluster-workshop-cloud-computing

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



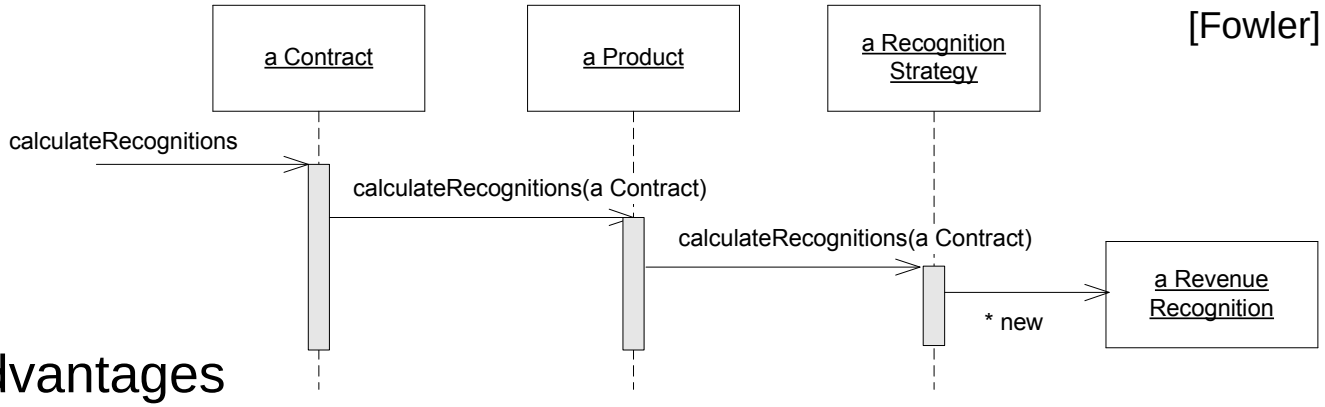
- Groups of architectural patterns are related to –
  - domain/business logic
  - data sources and O/R mapping
  - (web) presentation and session handling
  - distribution and concurrency
  - basic issues



Pattern Example

- We have met an architectural pattern already: Domain Model
  - facilitates object-oriented thinking
- Objects collaborate to do a transaction

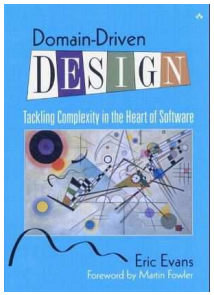
“An object model of the domain that incorporates both behavior and data.”



[Fowler]

- Advantages
  - organizes complex domain functionality in a “natural” way
- Problems
  - data persistence more complex
  - *steep learning curve*

also see



The Logical Architecture... [Larman, Fowler, Evans]

- … deals with the large-scale organization of classes into packages and subsystems
  - *makes no decision how to distribute these elements on physical machines*
- 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 the layer directly below in a strictly layered architecture
      - in order to limit coupling between layers
    - in practice relaxed layered architectures are 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*

Problems with the JFrameExample?

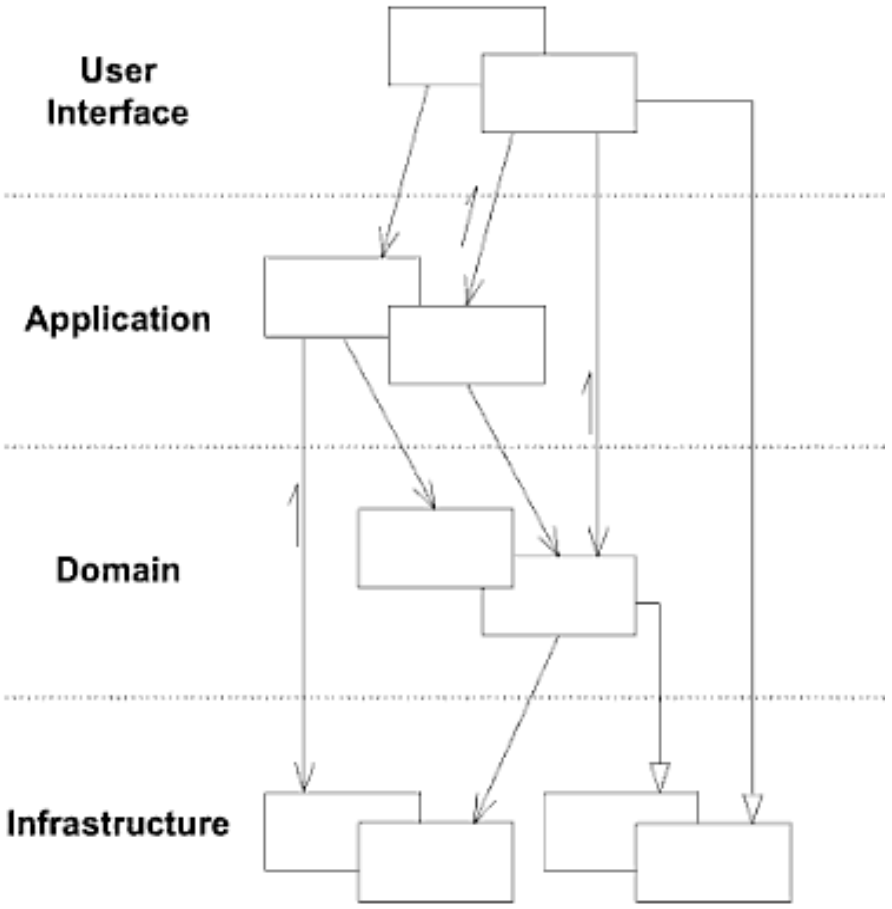
```
public class JFrameExample extends JFrame implements ActionListener {
…
public void actionPerformed(ActionEvent ae) {
      if (ae.getActionCommand().equals(jb2.getText())) {
        JOptionPane.showMessageDialog (this, "Juuuuhuu, bis bald!");
        System.exit(0);
      } else if (ae.getActionCommand().equals(jb1.getText())) {
        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 + "?");
   }
} // eof
```

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 –
  - 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

Layered (Reference) Architecture I

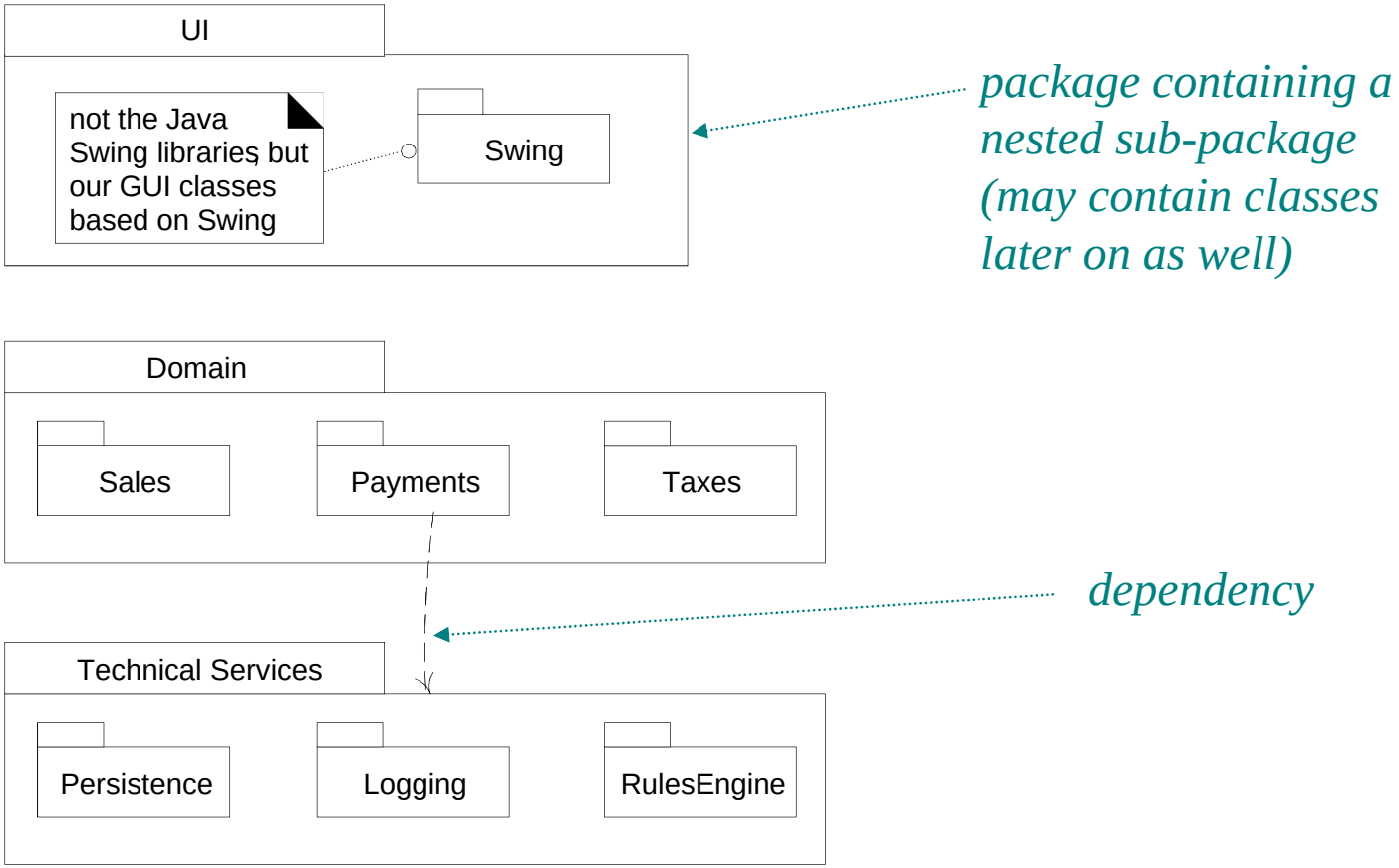
- Benefits:
  - reduces “accidental” complexity
  - improves modifiability
  - clear separation of concerns
  - independent exchangeability
  - compatible to SOAs
  - simplified testing
- Drawbacks:
  - usually increases the amount of classes
    - through facades or data transfer objects
    - *however, these are patterns in their own right that help to better deal with complexity*



[Evans]

Package Diagrams…

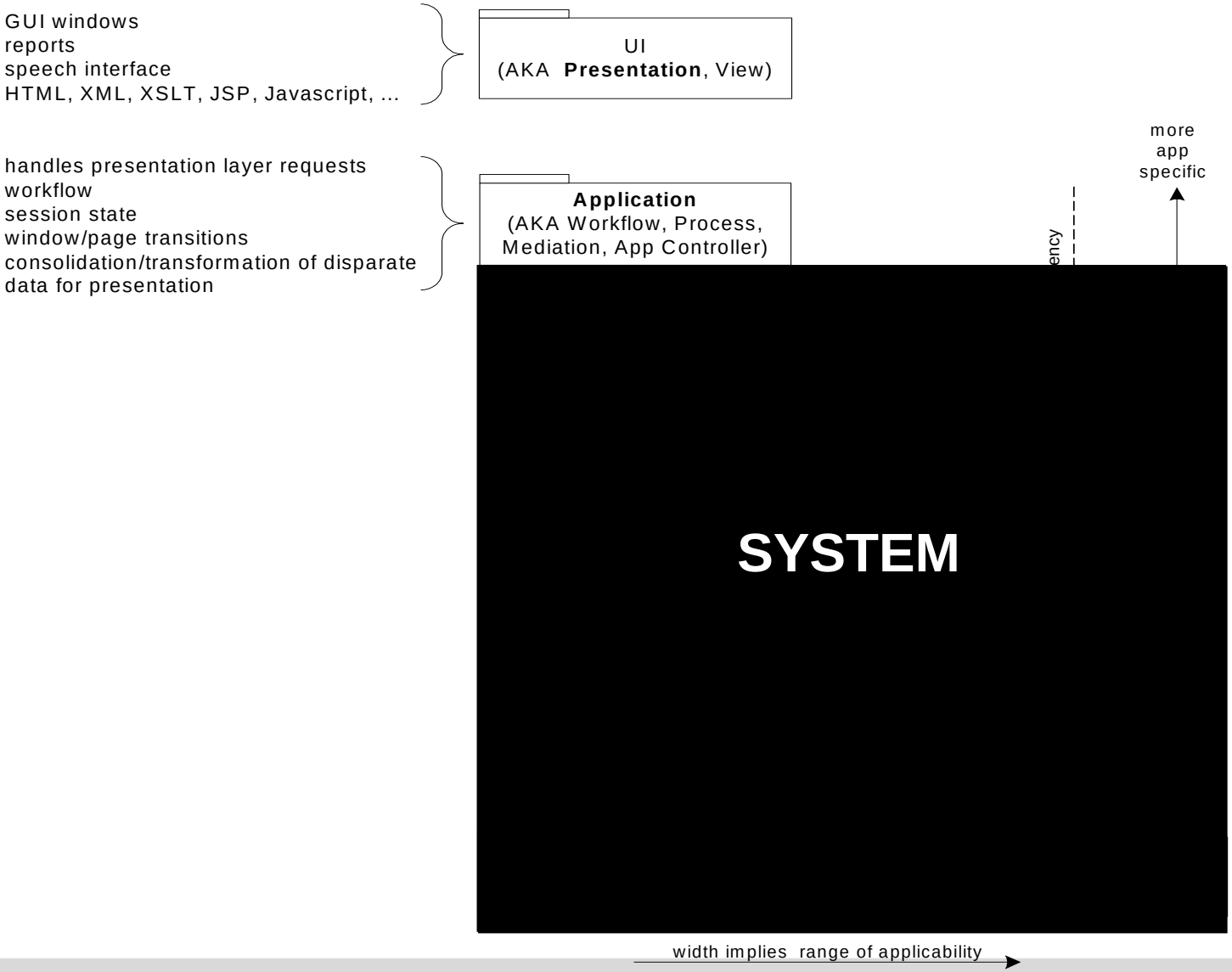
- … are UML diagrams helpful for illustrating system architectures
- … can be drawn on various abstraction levels



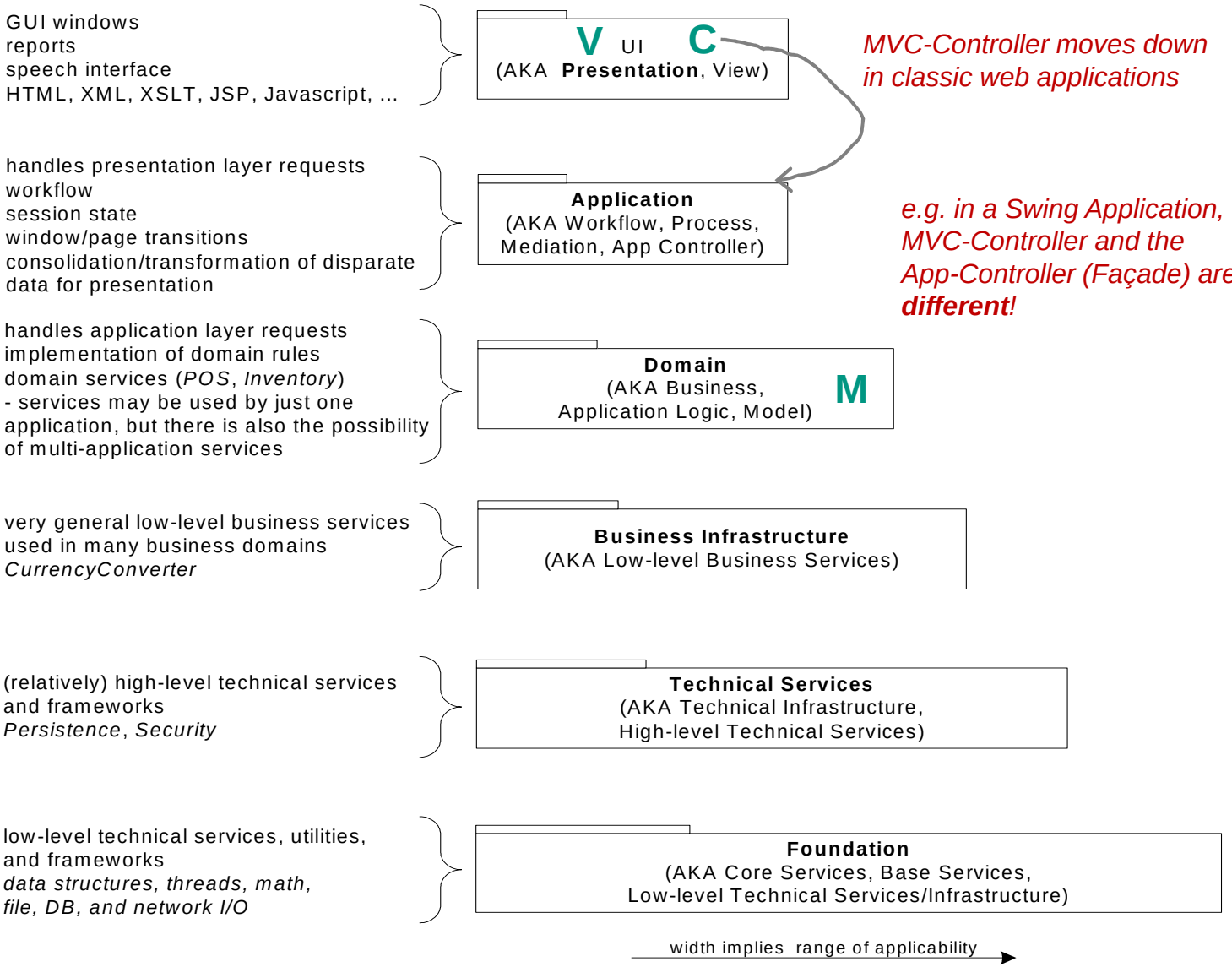
Some (Architectural) Design Principles

- **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**
  - You ain’t gonna need it! (YAGNI) vs. Design for Extensibility/Reusability
  - ➔ Refactoring

Layered Architecture II [Larman]



MVC in a Layered Architecture



The User Interface Layer…

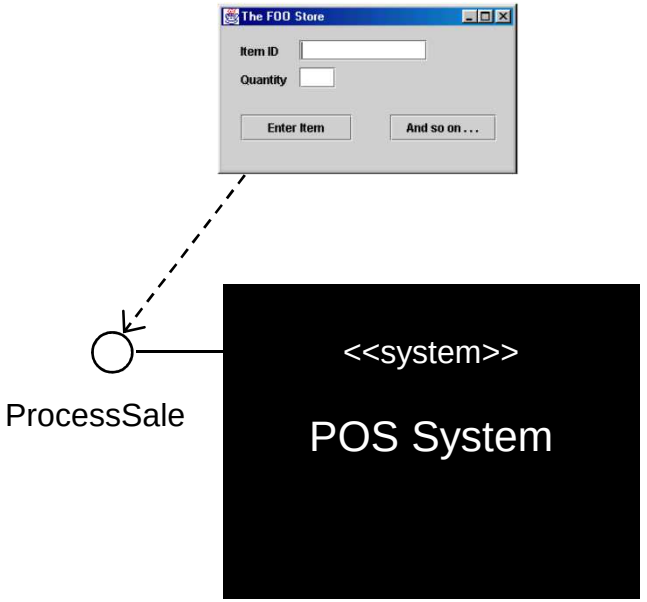
- … is responsible for the presentation of data to the user
- … and for managing the interaction with the user
  - i.e. the screen flow

- *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*

  ➔ 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. ProcessSale in our POS example
    - or SessionBeans in EJB-based systems
      - a.k.a Session Facades

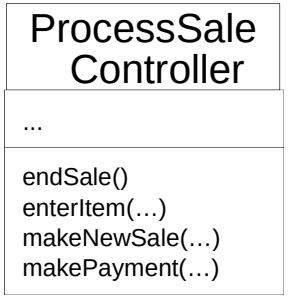


- In very small systems it is optional to have an application layer
  - i.e. domain objects can be directly called by UI
  - however, for multi-tier architectures it is usually mandatory
    - due to more complex session handling
- A good rule of thumb is to have one controller/facade per use case

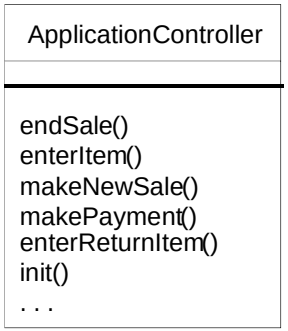
Modelling Alternatives

- As said before, various strategies exsist for accomodating the system operations –

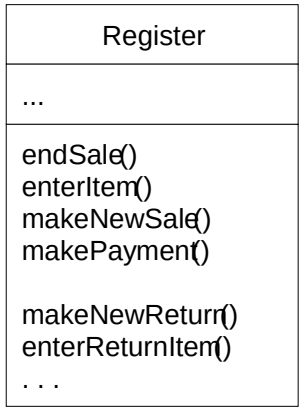
1. one controller class per use case
   - works well for systems with many use cases
   - *per CRUD use case -> SessionBeans in JEE*



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



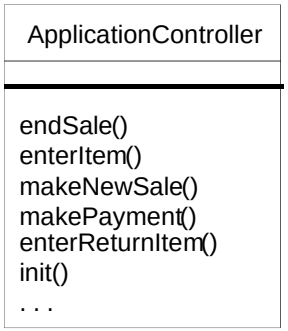
3. direct access to appropriate domain objects
   - reduces passing through of parameters
   - but easily brings control flow logic into the domain model

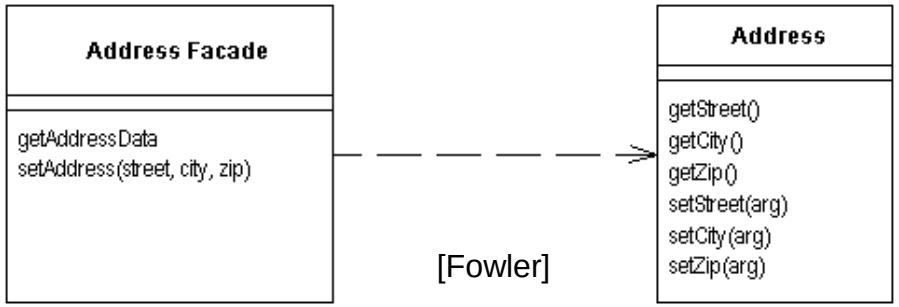


Facades

- Packages (and layers) sometimes contain a large number of classes
  - but are intended to offer relatively little functionality to the world
    - this may be considered a **subsystem**
- In Java, however, a package itself does not support having an interface
  - ➔ thus, it makes sense to use **façade classes** where method calls need to cross subsystem boundaries
    - in order to reduce coupling
  - e.g. the black-box system with its operation is such a façade
    - hiding implementation details
    - and controlling the flow inside the blackbox

also compare [GoF]

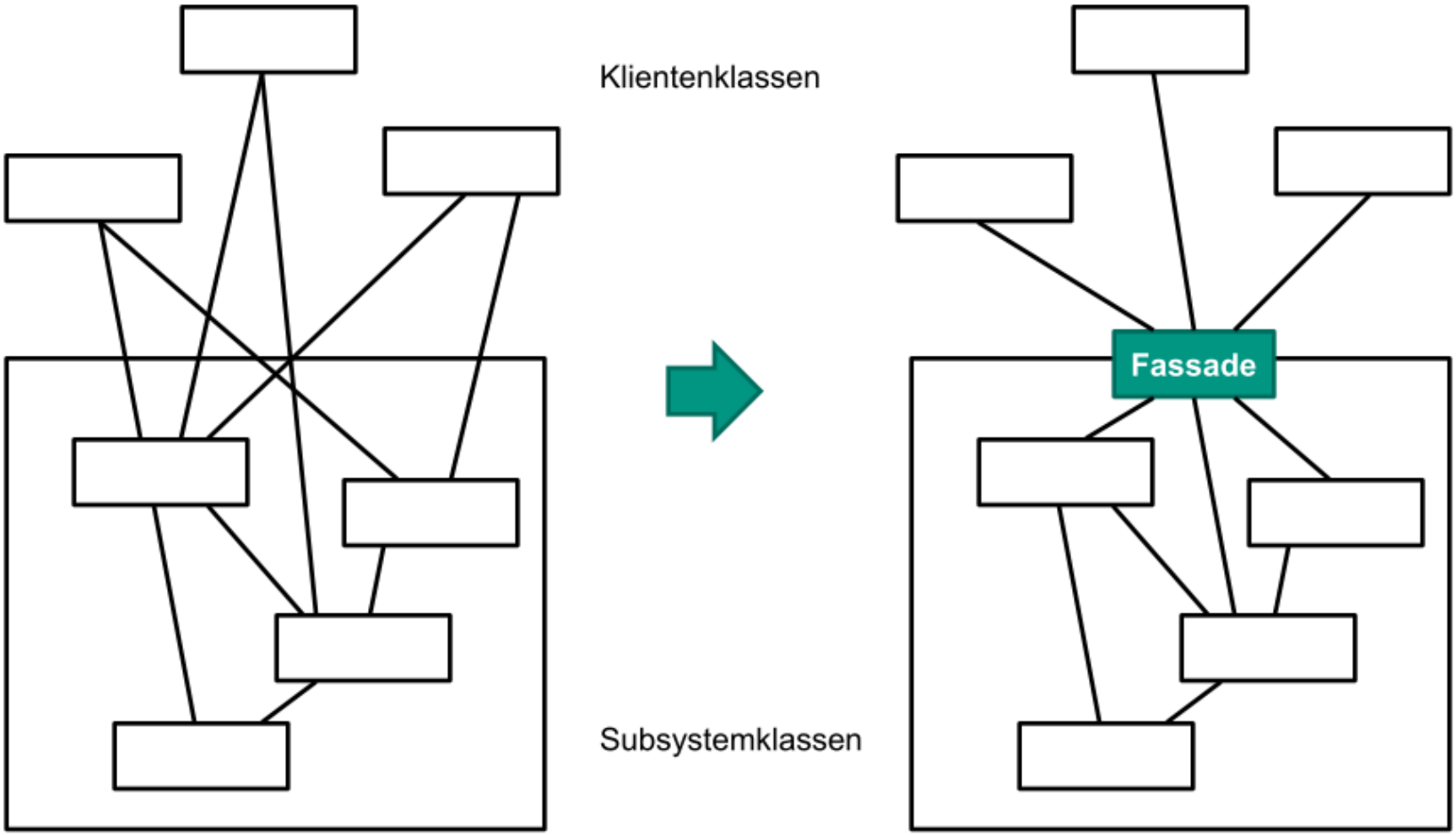




[Fowler]

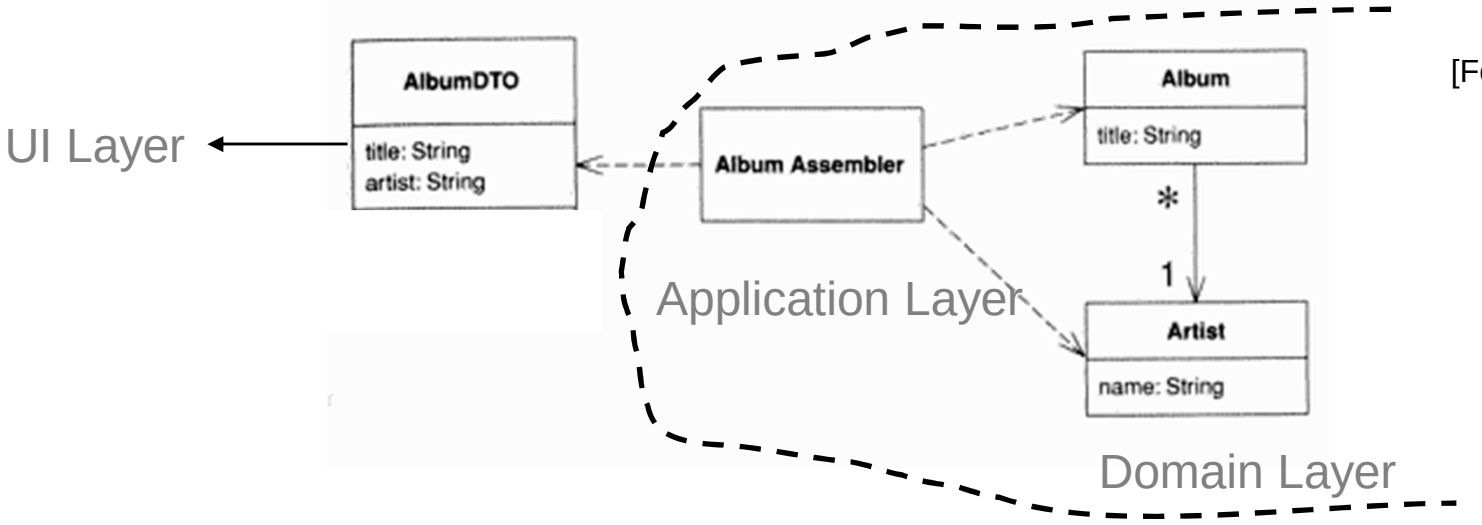
Facades Example

- [SWT1]



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 method calls
    - i.e. to reduce coupling
    - it usually contains only attributes and getters and setters for them
    - it may contain methods for serialization in distributed systems

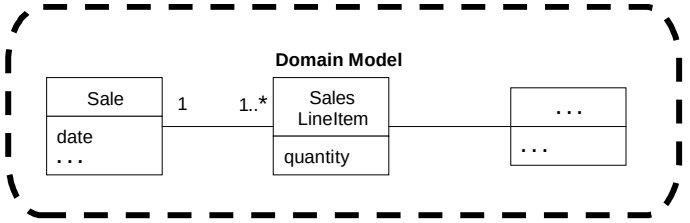


[Fowler]

- 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)

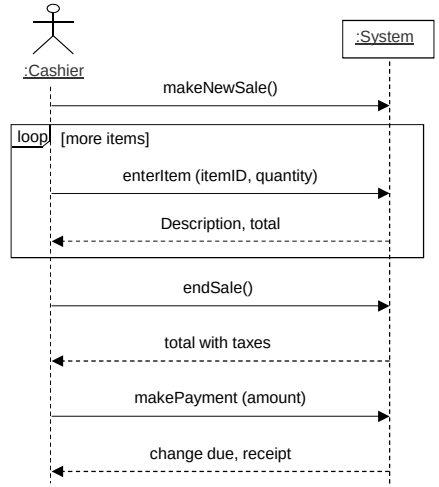
➔ Thus, it is typically very application specific

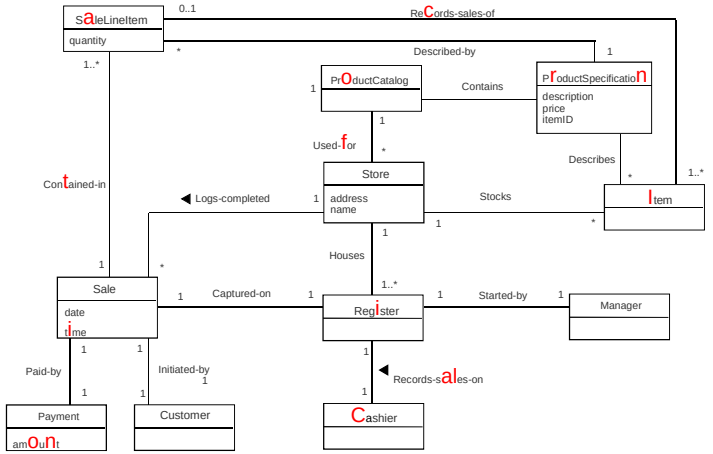
➔ domain objects should be kept in the domain layer

- in single-process desktop applications (without serialization) they may be passed to application, UI or persistence layer
  - in order to avoid the creation of additional DTOs

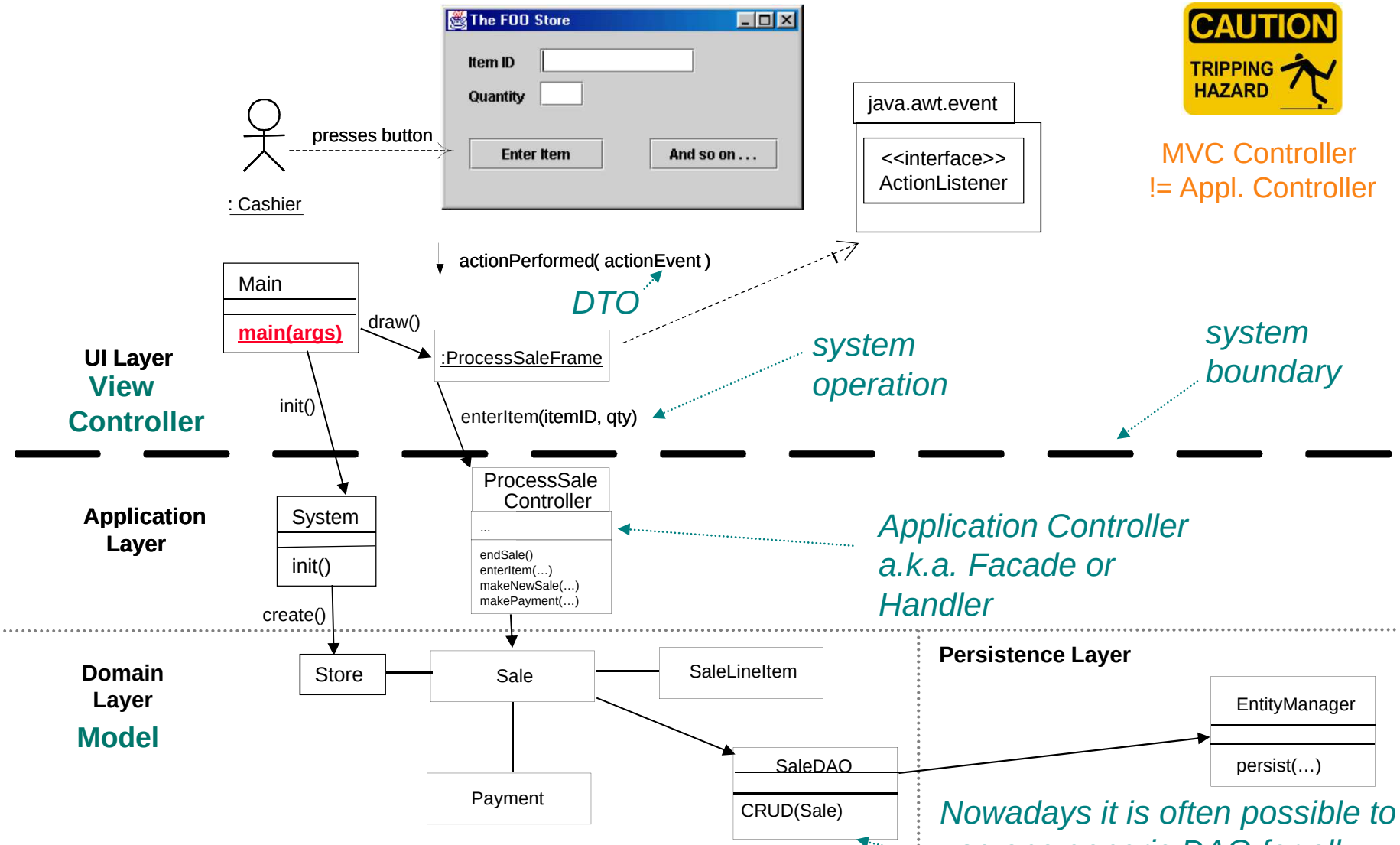
Hands on Architectural Modeling

- 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



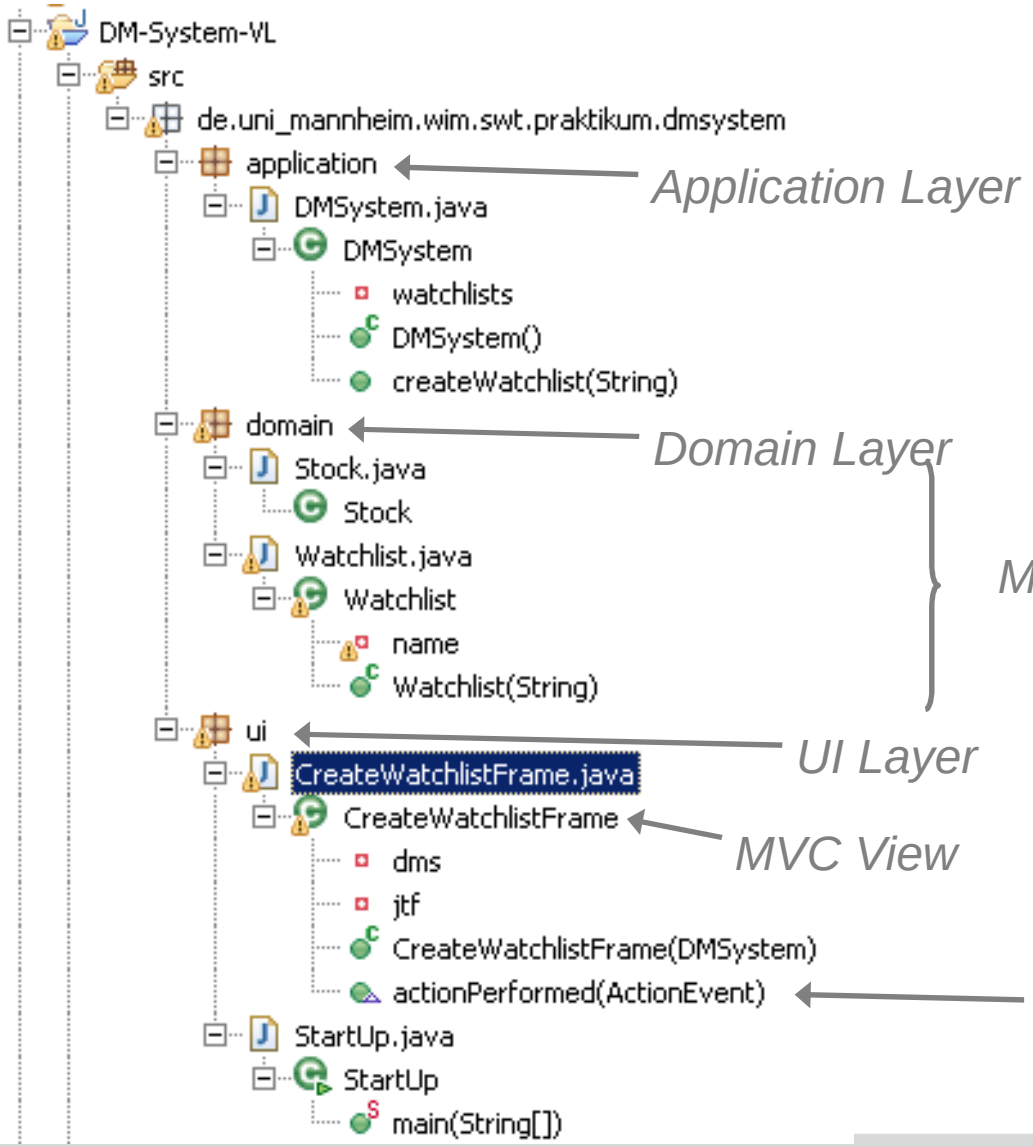


Architecture Example [Larman + some own extensions]



MVC Controller
!= Appl. Controller

Layers + MVC Example



Application Layer

Domain Layer

MVC Model

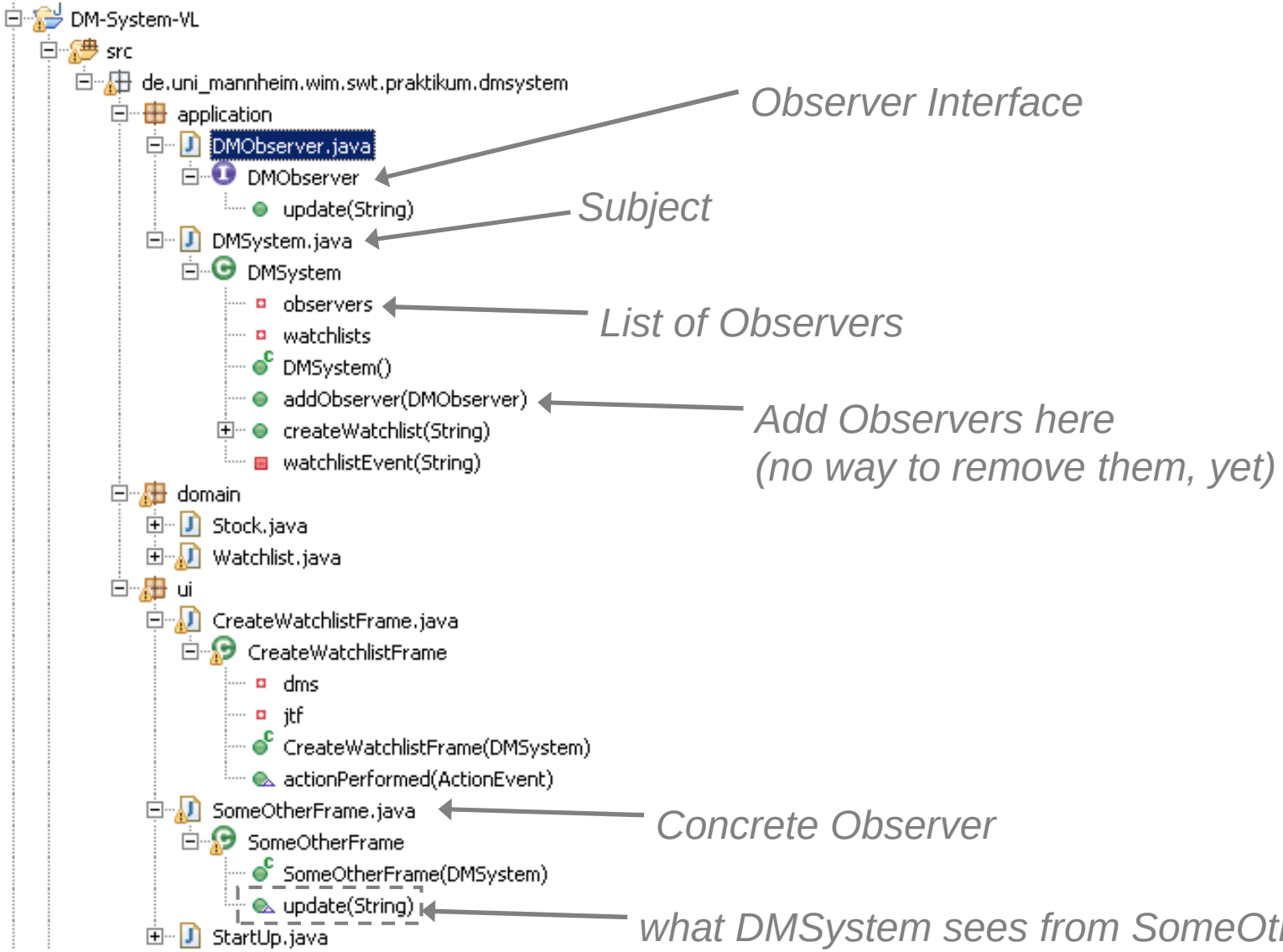
UI Layer

MVC View

MVC Controller – *can be placed in a separate class as well; somewhat depends on personal taste*

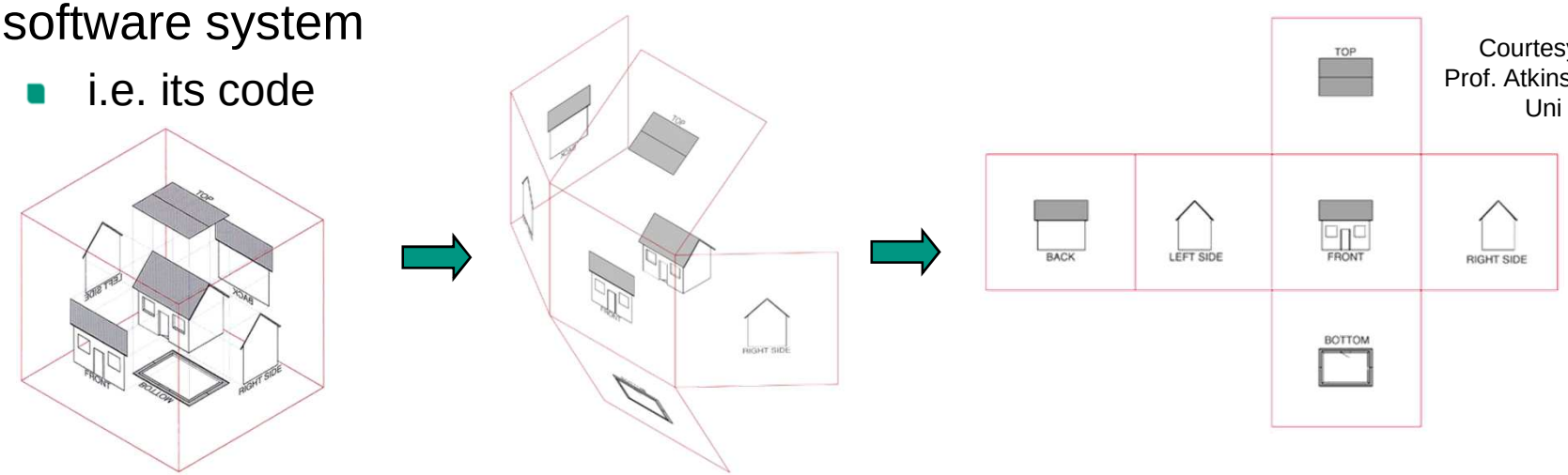
Open question: where do we place watchlist functionality and how do we implement "upward" calls to the UI if necessary?

Solution with Observer

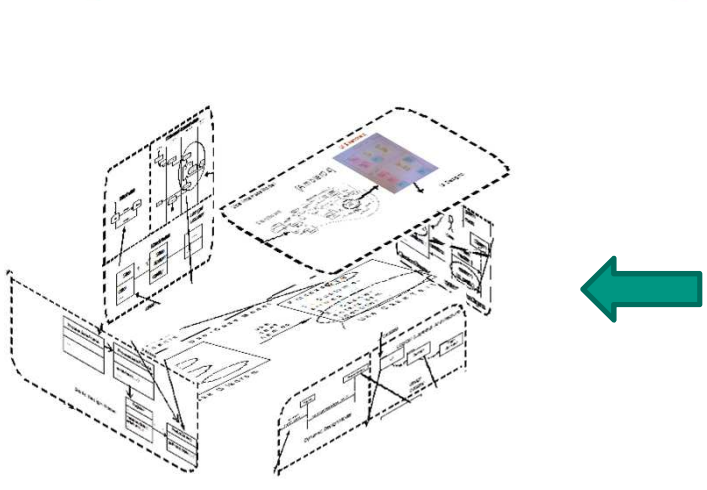


Architectural Analogies I

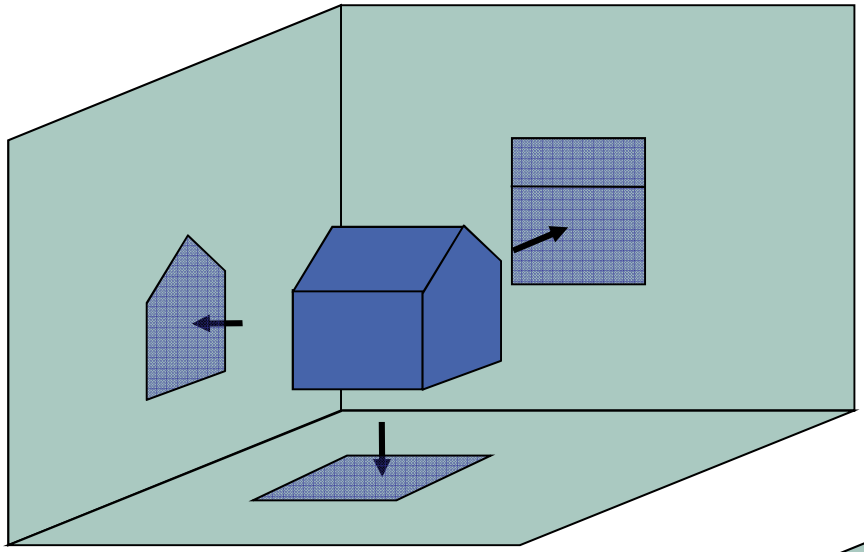
- Software models can be seen as orthographical projections of a software system
  - i.e. its code



Courtesy of Prof. Atkinson, Uni MA

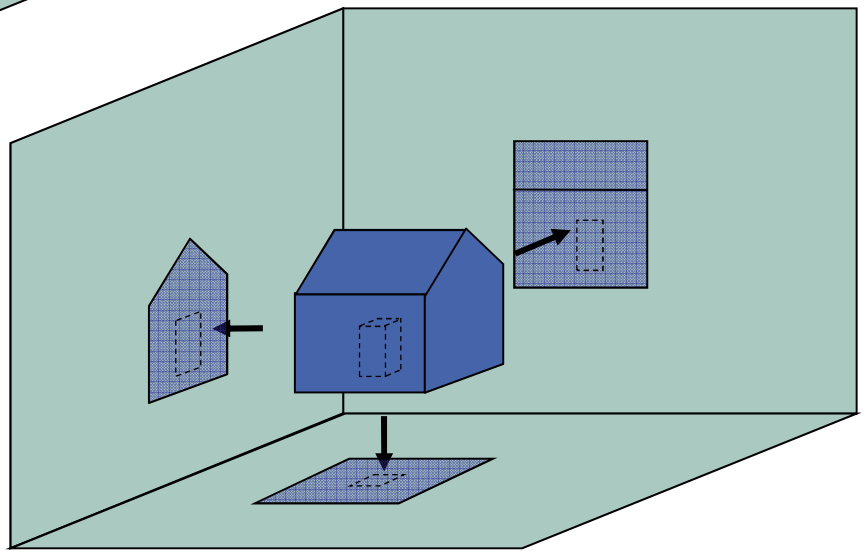


Architectural Analogies II



➔ black box view

- only surface features are displayed



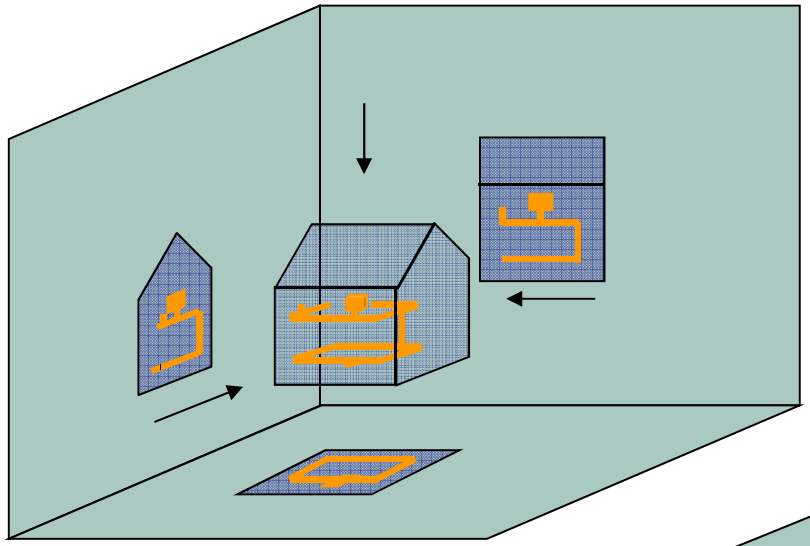
➔ white box view

- internal (hidden) features displayed
- rooms ~ packages/components

Courtesy of Prof. Atkinson, Uni MA

Architectural Analogies III

- Ever heard about Aspect-Oriented Programming (AOP)?

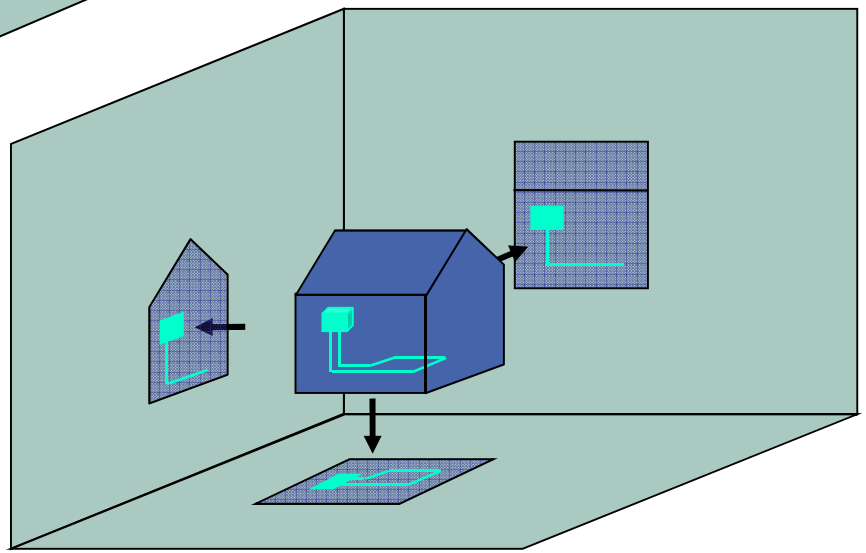


➔ models describing one particular aspect of an object's realization

- only "orange" concerns depicted

➔ models describing a different aspect

- only "turquoise" concerns depicted

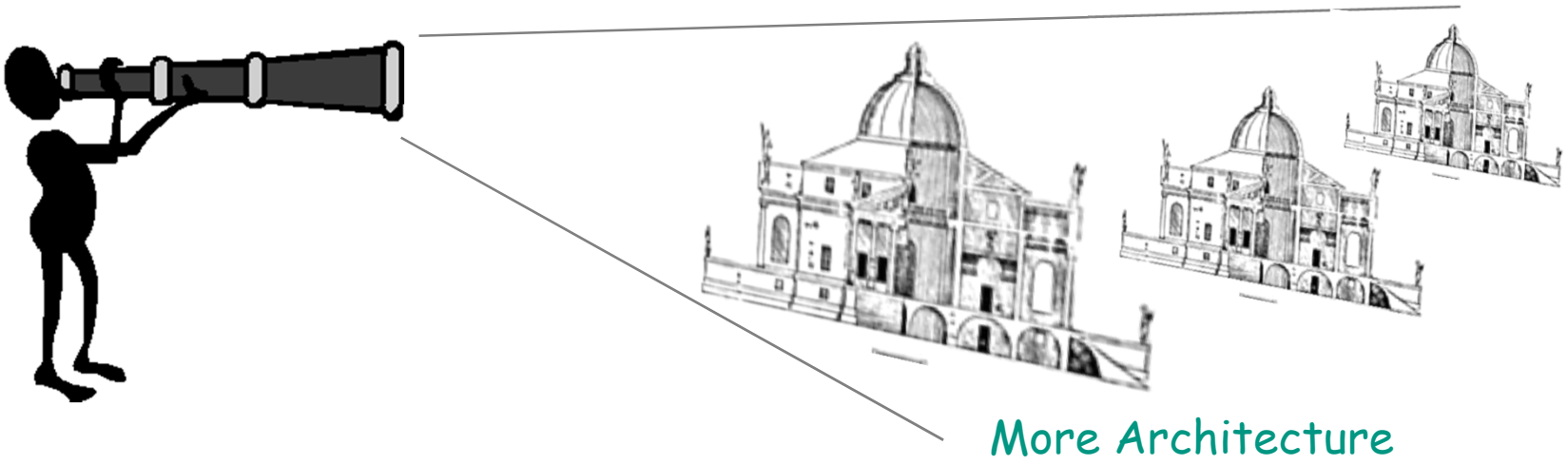


Courtesy of Prof. Atkinson, Uni MA

Conclusion

- Architectures facilitate model-centric development
  - instead of code-centric development
  - in order to better manage complexity
    - and to benefit from re-use and other ammenities

- Thank you for your attention!



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.
  - ➔ Probekapitel: http://files.hanser.de/hanser/docs/20051107_2012053191536103_978-3-446-40837-1_Kap01.pdf

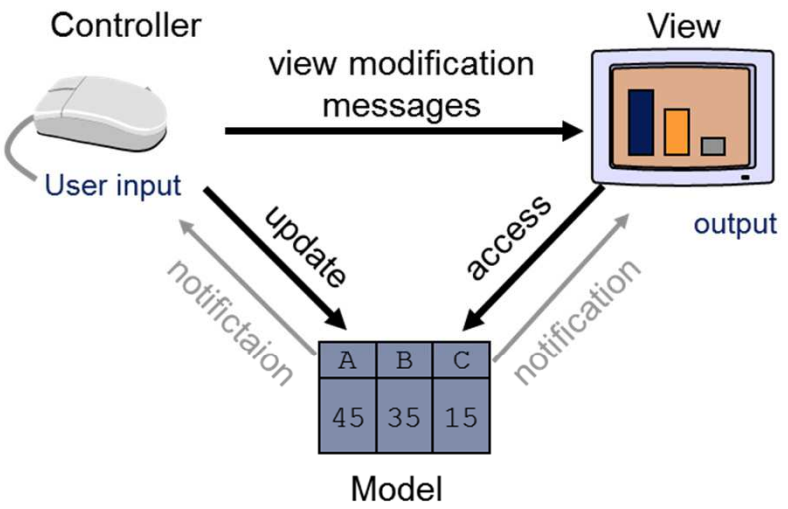
Ralf Reussner, Wilhelm Hasselbring
Handbuch der Software-Architektur
2. Auflage, dPunkt-Verlag, Heidelberg, 2008

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 interface.
Problem	User interfaces are particularly prone to change requests.
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

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.

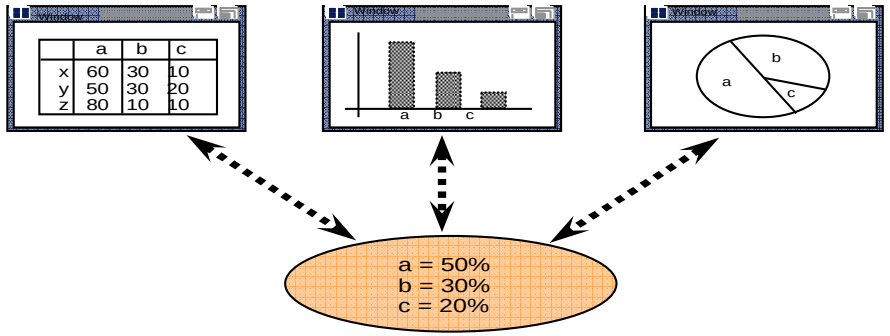
A.K.A. Dependents, Publish-Subscribe

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



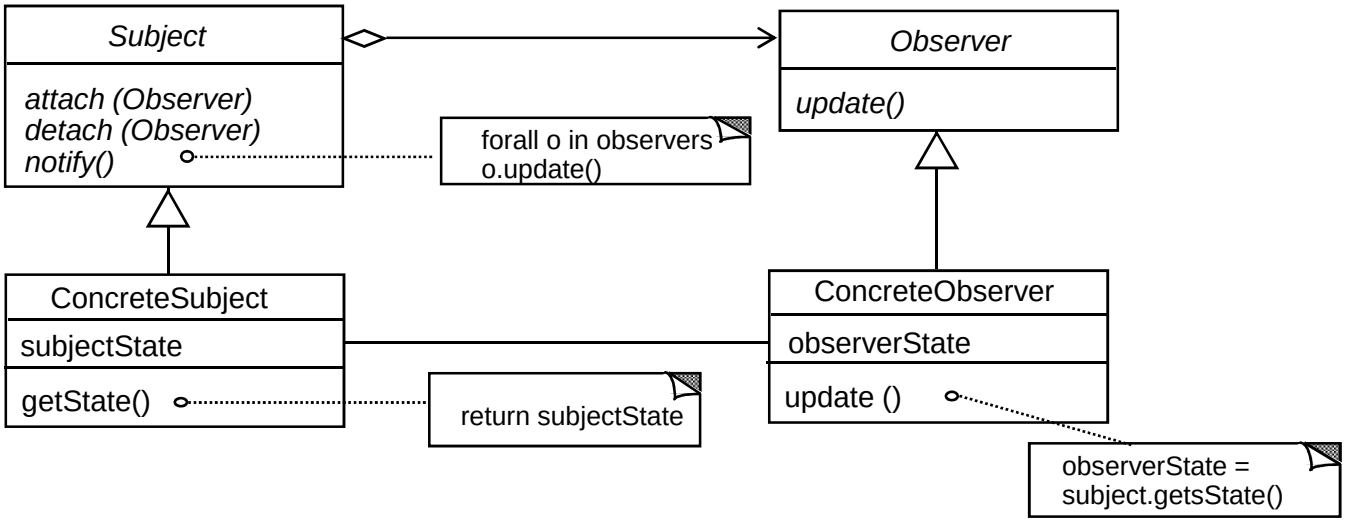
UI Layer

Domain Layer

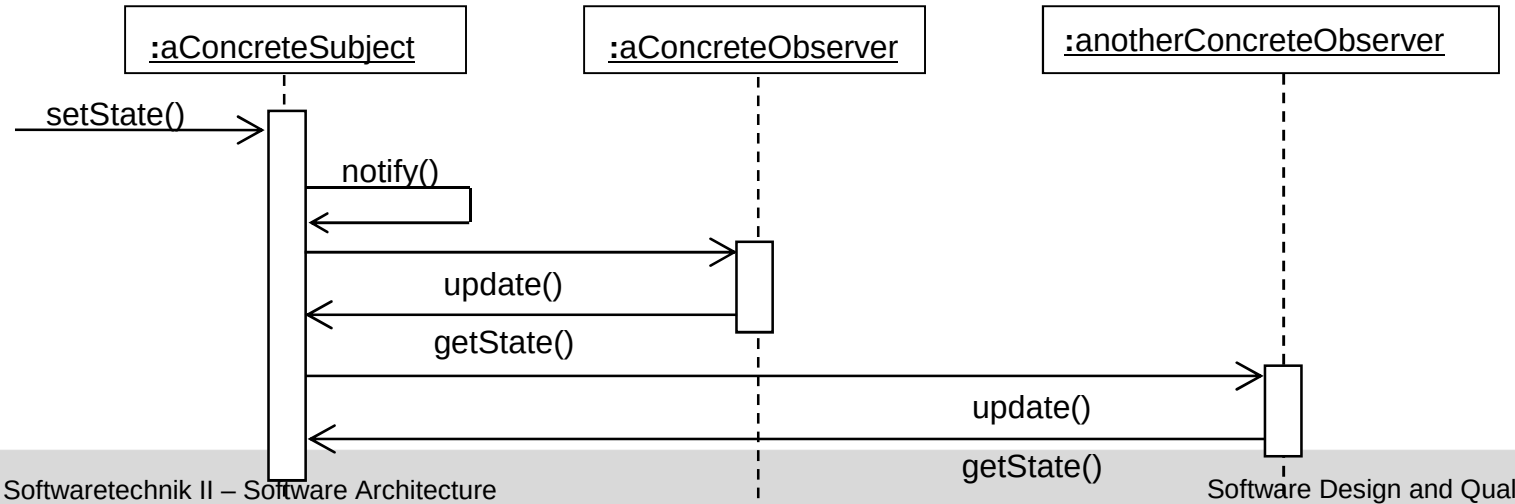
Observer Pattern Continued

Participants Subjects, Observer, ConcreteSubject, ConcreteObserver

Structure



Collaborations

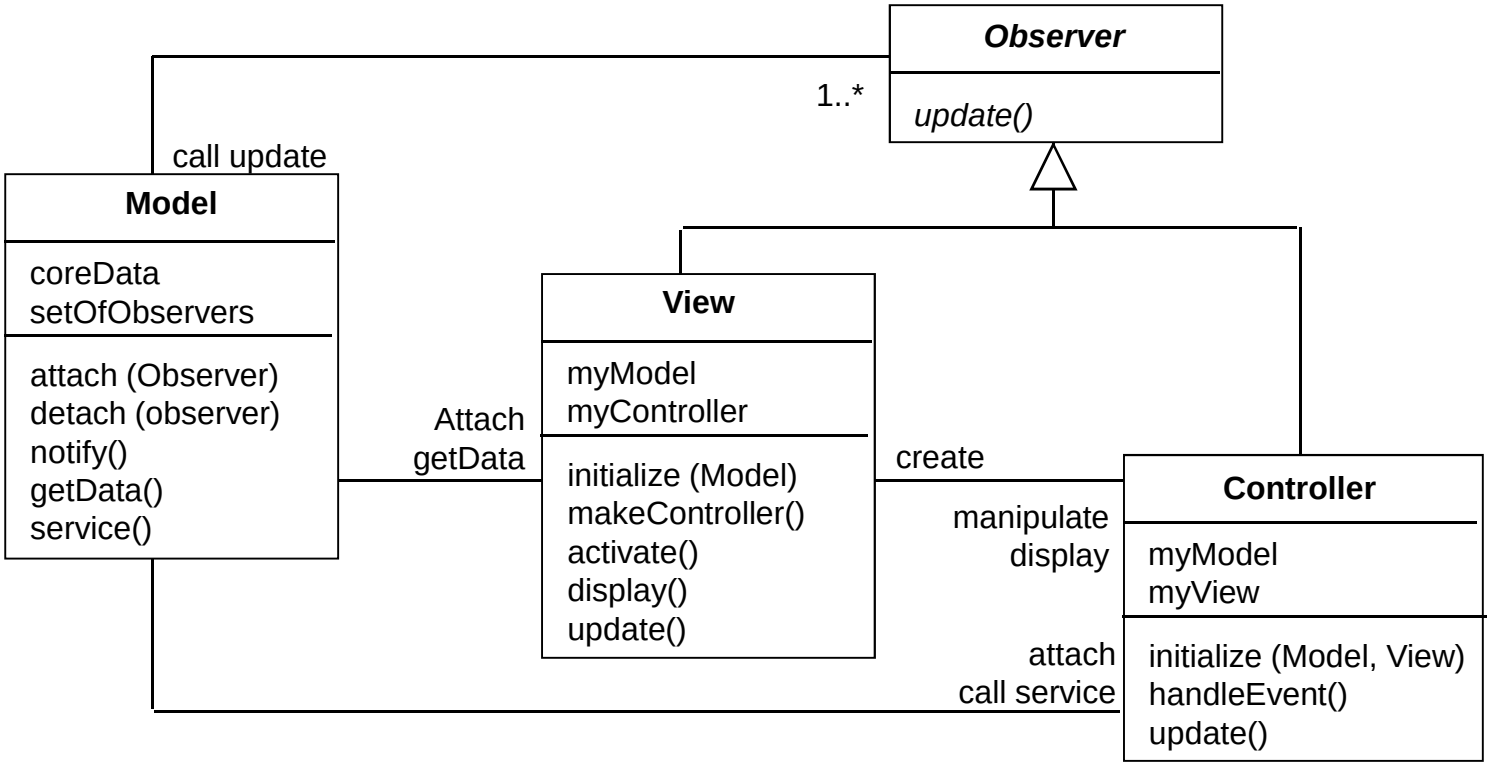


Model View Controller & Observer

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.



Appendix: Additional Literature (1)

- R. N. Taylor, N. Medvidovic, and E. M. Dashofy:
  “Software Architecture: Foundations, Theory, and Practice”,
  Wiley, Hoboken, 2009

- 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.:

- *"Documenting Software Architectures: Views and Beyond"*,
  Addison-Wesley, Boston, 2005
- *"Software Product Lines: Practices and Patterns"*,
  Addison-Wesley, Boston, 2002
- *"Evaluating Software Architectures: Methods and Case Studies"*,
  Addison-Wesley, Boston, 2002