MTAT.03.094
Software Engineering

Lecture 07: Architecture and Design

Dietmar Pfahl
email: dietmar.pfahl@ut.ee

Fall 2018
Schedule of Lectures

Week 01: Introduction to SE
Week 02: Requirements Engineering I
Week 03: Requirements Engineering II
Week 04: Analysis
Week 05: Development Infrastructure I
Week 06: Development Infrastructure II
Week 07: Architecture and Design
Week 08: Verification and Validation I
Week 09: Verification and Validation II
Week 10: Continuous Development and Integration
Week 11: Refactoring (and TDD)
Week 12: Agile/Lean Methods
Week 13: no lecture
Week 14: Software Craftsmanship
Week 15: Course wrap-up, review and exam preparation
Acknowledgements

Textbooks/Slides:


• Stefan Zörner: Softwarearchitekturen dokumentieren und kommunizieren (in German), 2013

• Hans van Vliet: Software Architecture, Free University of Amsterdam, Lecture 2008

• Richard Taylor et al.: Software Architecture, University of California at Irvine, Lecture 2011

• Alexander Serebrenik: Software architecture: Domain-Specific Software Architecture and Architectural Patterns, TU Eindhoven, Lecture 2013

• George Fairbanks: Just Enough Software Architecture, 2012 (Video: https://www.youtube.com/watch?v=x30DcBfCJRI)

• Tutorials by Derek Banas (on YouTube)
Structure of Lecture 07

• Why Architecture?
• Terminology: Architect, Architecting, Architecture
• Viewpoints and View Models
• Notation
• Architecture & Design Patterns
Two Telephone Systems

**Plain Old Telephone System**
- Feature: Call subscriber
- Architecture: Centralized switchboard
- Good qualities

**Skype**
- Feature: Call subscriber
- Architecture: Peer-to-peer software
- Good qualities

Architects pay more attention to qualities that arise from architecture choices.
Two Telephone Systems

**Plain Old Telephone System**
- **Feature:**
  - Call subscriber
- **Architecture:**
  - Centralized switchboard
- **Good qualities**
  - Works during power outages
  - Reliable
  - Emergency calls get location information

**Skype**
- **Feature:**
  - Call subscriber
- **Architecture:**
  - Peer-to-peer software
- **Good qualities**
  - Scales without central hardware changes
  - Easy to add new features (e.g., video calling)

Architects pay more attention to **qualities** that arise from architecture choices.
### Two Telephone Systems – Pro’s and Con’s

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Landline Phone</th>
<th>VoIP (Skype)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Outage Tolerant</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Reliable</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Scalable</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Extendable (new features)</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

**Which one is better?**
Two Telephone Systems – Pro’s and Con’s

Which one is better?

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Landline Phone</th>
<th>VoIP (Skype)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Outage Tolerant</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>Reliable</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Scalable</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>Extendable (new features)</td>
<td>-</td>
<td>++</td>
</tr>
</tbody>
</table>
Trade-Offs and Decision-Making

Telephone system for a Fire Brigade Station:

Which one is better?

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Landline Phone</th>
<th>VoIP (Skype)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Outage Tolerant</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>Reliable</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Scalable</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>Extendable (new features)</td>
<td>-</td>
<td>++</td>
</tr>
</tbody>
</table>
Trade-Offs and Decision-Making – Template

Telephone system for a Fire Brigade Station:

- Because <Power Outage Tolerance> is more important than <Scalability> for this system, we choose a <Landline Phone>, accepting <a higher cost for adding new subscribers>.

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Landline Phone</th>
<th>VoIP (Skype)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Outage Tolerant</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>Reliable</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Scalable</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>Extendable (new features)</td>
<td>-</td>
<td>++</td>
</tr>
</tbody>
</table>
Trade-Offs and Decision-Making – Template for capturing design rationales:

Telephone system for a Fire Brigade Station:

- Because <Quality Attribute 1> is more important than <Quality Attribute 2> for this system, we choose <technical (design/architecture) option>, accepting <drawback>

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Landline Phone</th>
<th>VoIP (Skype)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Outage Tolerant</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>Reliable</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Scalable</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>Extendable (new features)</td>
<td>-</td>
<td>++</td>
</tr>
</tbody>
</table>
Trade-Offs and Decision-Making – Template for capturing design rationales:

Telephone system for a University:

- Because <Quality Attribute 1> is more important than <Quality Attribute 2> for this system, we choose <technical (design/architecture) option>, accepting <drawback>

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Landline Phone</th>
<th>VolP (Skype)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Outage Tolerant</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>Reliable</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Scalable</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>Extendable (new features)</td>
<td>-</td>
<td>++</td>
</tr>
</tbody>
</table>
Trade-Offs and Decision-Making – Template Architecture for the CSR system:

• Because Scalability is more important than Data freshness for this system, we choose V3, accepting lower ad hoc query ease.

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Freshness</td>
<td>+</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Scalability</td>
<td>--</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Ad-hoc query ease</td>
<td>+</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

Mind Map for Capturing Architecture/Design Rationale

(1) Question
- What exactly is the problem?
- Why is it relevant for the architecture?
- What effects does the decision have?

(2) Influencing factors
- What fixed constraints do we have to consider?
- What quality goals do we have to consider?
- What risks do we have to consider?

(3) Assumptions
- What assumptions have we made?
- What assumptions can be checked upfront?
- What new risk may emerge?

(4) Alternatives considered
- What are promising solution alternatives?
- How do we assess each alternative?
- Is there a solution option that we discard explicitly?

(5) Decision
- What alternative has been chosen?
- How is the decision justified?
- When and by whom was the decision made?

What exactly is the problem?
Why is it relevant for the architecture?
What effects does the decision have?

What problems?
- P1: What network topology should be used?
- P2: What transmission technology should be used
- P3: ...?

Why relevant?
- P1-R1: Topology choice defines skills needed by the development team
- P1-R2: Topology choice defines what suppliers for 3rd party components are available
- P2-R1: Transmission technology defines the transmission speed
- P2-...

What effects?
- P1-P2: Choice of topology (P1) may interfere with choice made regarding transmission technology (P2)
- ...
Mind Map for Capturing Architecture/Design Rational

(1) Question
- What exactly is the problem?
- Why is it relevant for the architecture?
- What effects does the decision have?

(2) Influencing factors
- What fixed constraints do we have to consider?
- What quality goals do we have to consider?
- What risks do we have to consider?

Capture Design Rationale

(4) Alternatives considered
- What are promising solution alternatives?
- How do we assess each alternative?
- Is there a solution option that we discard explicitly?
- What alternative has been chosen?
- How is the decision justified?
- When and by whom was the decision made?

(5) Decision

What constraints?
- P1-C1: available skills in staff
- P1-C2: stability of power supply
- P2-C1: ...

What quality goals?
- P1-Q1: Power outage tolerance
- P1-Q2: Reliability
- P1-Q3: Scalability
- P1-Q4: Evolvability/Maintainability (-> adding new features)

What risks?
- P1-R1: ...
- ...

Quelle: Stefan Zörner, Softwarearchitekturen dokumentieren und kommunizieren
Mind Map for Capturing Architecture/Design Rational

(1) Question
- What exactly is the problem?
- Why is it relevant for the architecture?
- What effects does the decision have?

(2) Influencing factors
- What fixed constraints do we have to consider?
- What quality goals do we have to consider?
- What risks do we have to consider?

(3) Assumptions
- What assumptions have we made?
- What assumptions can be checked upfront?
- What new risk may emerge?

What assumptions?
- P1-C1-A1: skills can be learned if needed
- P2- ...

What assumptions can be checked?
- P1-C2-AC1: stability of power supply
- P2- ...

What new risks may emerge?
- PX-NR1: subscribers are more and more mobile
- ...

What are promising solution alternatives?

How do we assess each alternative?

Is there a solution option that we discard explicitly?

What alternative has been chosen?

How is the decision justified?

When and by whom was the decision made?

What assumptions?
- P1-C1-A1: skills can be learned if needed
- P2- ...

What assumptions can be checked?
- P1-C2-AC1: stability of power supply
- P2- ...

What new risks may emerge?
- PX-NR1: subscribers are more and more mobile
- ...
Mind Map for Capturing Architecture/Design

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Landline Phone</th>
<th>VoIP (Skype)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Outage Tolerant</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>Reliable</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Scalable</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>Extendable (new features)</td>
<td>-</td>
<td>++</td>
</tr>
</tbody>
</table>

What exactly is the problem?
What promising solutions?
- P1-S1: Centralized switch
- P1-S2: Peer-to-peer software
- P2-...
How assess alternative?
- P1-SX: cross-table against quality attributes
- P2-SX: ...
Solution explicitly discarded?
- P1-EX1: Ring topology
- ...

What are promising solution alternatives?
How do we assess each alternative?
Is there a solution option that we discard explicitly?
What alternative has been chosen?
How is the decision justified?
When and by whom was the decision made?

Trade-Offs and Decision-Making – Template

Architecture for the CSR system:

- Because <Scalability> is more important than <Data freshness> for this system, we choose <V3>, accepting <lower ad hoc query ease>.

**Quality Attribute** | **Landline Phone** | **VoIP (Skype)**
--- | --- | ---
Power Outage Tolerant | ++ | –
Reliable | + | –
Scalable | – | ++
Extendable (new features) | – | ++

Telephone system for a Fire Brigade Station: Because <Power Outage Tolerance> is more important than <Scalability> for this system, we choose a <Landline Phone>, accepting <a higher cost for adding new subscribers>.

**Capture Design Rationale**

**What choice?**
- P1-Choice: Centralized switch
- P2- ...

**How justified?**
- P1-Choice-J: Cross table + Application domain
- P2- ...

**Choice made by whom/when?**
- P1-Choice-WW: ...
- ...

**What exactly is the problem?**
**Why is it relevant for the architecture?**
**What effects does the decision have?**
**What fixed constraints do we have to consider?**
**What quality goals do we have to consider?**

**What alternative has been chosen?**
**How is the decision justified?**
**When and by whom was the decision made?**

Why is documenting architecture/design decisions important?

- Explains why this is a good (= suitable) architecture
  - Emphasizes qualities and criticality with regards to requirements/goals
- Provides context and background
- Prevents repeating (expensive) past steps

Yet another design rationale example:

Since avoiding vendor lock-in is a high priority, we choose to use a standard industry framework with multiple vendor implementations, even though using vendor-specific extensions would give us greater performance.
What if you don’t think architecturally?

- Developers optimize locally, miss the big picture
  - Lousy choice of frameworks, languages, ...

- Project success depends on having virtuosos in the team
  - But how many James Goslings and Jeff Deans are there?

- Poor communication
  - Idiosyncratic notations, fuzzy semantics

- Shallow (or no) analysis of design options
  - Ad hoc; no use of best practices
  - From first principles, therefore high effort
  - Little attention to tradeoffs and rationale

- Architectural patterns ignored
  - ... or incorrectly chosen
  - Squandering known-good designs

Remember:
All programs have an architecture
...
But not every architecture suits the problem to be solved by the program!
Structure of Lecture 07

• Why Architecture?
• Terminology: Architect, Architecting, Architecture
• Viewpoints and View Models
• Notation
• Architecture & Design Patterns
Terminology

- Architect – Person
- Architecting – Process
- Architecture – Product
The Role of the Architect

client, users → requirements

architect → creates

solutions → assess

designer, developers → assess

visualises

architectural design

creates

prescribes

construction, co-operation

appearance, behaviour → assess

requirements
Now, what is ‘Architecting’?
Non-Architecture-Driven Life-Cycle

Characteristics:

- Iteration mainly on functional requirements
- Few stakeholders involved
- No balancing of functional and quality requirements

Architecture?
Adding Architecture – The “Easy” Way

stakeholders (few)

functionality

quality

agreement

architecture
detailed design
implementation
development
Adding Architecture – The “Easy” Way

stakeholders (few) ↔ functionality ↔ quality

architecture development

agreement

few stakeholders
Architecture in the Life-Cycle

Characteristics:

- Iteration on both functional and quality requirements
- Many stakeholders involved
- Balancing of functional and quality requirements
Architecture Iterations Example

Top-level:

- **usability** $\Rightarrow$ separate UI $\Rightarrow$ 3-tier architecture

Iteration 1
Architecture Iterations Example

Top-level:
- **usability** $\Rightarrow$ separate UI $\Rightarrow$ 3-tier architecture

Lower-level, within **user interface**:
- **security** $\Rightarrow$ authenticate users

Lower-level, within **data layer**:
- **availability** $\Rightarrow$ active redundancy
Attribute-Driven Design (ADD)

An architect faces many design issues

These are sub-problems of the overall design problem.

Each issue usually has several alternative solutions (or design options)

The designer makes a design decision to resolve each issue.

Process of ‘Architecting’ = choosing the best options from among the alternatives

ADD = focus on quality attributes and do ‘Architecting’ in iterations
Example: The type and level of **Security** in a system

- Security can be decomposed into
  - authentication (user recognition),
  - authorization (user access to data),
  - privacy (encryption of data exchanged on a public network).
ADD: Design Issues, Options and Decisions

Example: The type and level of Security in a system

- Security can be decomposed into
  - authentication (user recognition),
  - authorization (user access to data),
  - privacy (encryption of data exchanged on a public network).

If the architecture is for a medical system, then all security sub-issues must be addressed.
ADD: Design Issues, Options and Decisions

Example: The type and level of **Security** in a system

- Security can be decomposed into
  - authentication (user recognition),
  - authorization (user access to data),
  - privacy (encryption of data exchanged on a public network).

If the architecture is for **gaming applications**, probably not all of them are important, and could be dropped in favor of, e.g., higher performance.
Now, what is ‘Architecture’?
Architecture in Construction of Buildings
Software Architecture

- Architecture is *conceptual*.

- Architecture is about *fundamental* things.

- Architecture exists in some *context*.

Architectural descriptions are concrete, but the architecture itself is inherently conceptual, and cannot be captured in any (set of) views – nor in the code.

Abstraction !!!

NB: We can only understand qualities in context.
Software Architecture – Definition (1)

The architecture of a software system defines that system in terms of computational components and interactions among those components.

Software Architecture – Definition (2)

The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

Software Architecture – Definition (3)

Architecture is the fundamental organization of a system embodied in its **components**, their **relationships** to each other and to the **environment** and the **principles** guiding its design and evolution

NB: ‘Principles’ includes the explicit identification and mentioning of properties (-> behaviour)

Structure of Lecture 07

- Why Architecture?
- Terminology: Architect, Architecting, Architecture
- Viewpoints and View Models
- Notation
- Architecture & Design Patterns
Analogy with Building Architecture

- Overall picture of building (client)
- Front view (client, municipal “beauty” committee)
- Separate picture for water supply (plumber)
- Separate picture for electrical wiring (electrician)
- etc
IEEE Model for Architectural Descriptions

System stakeholder: an individual, team, or organization (or classes hereof) with interests in, or concerns relative to, a system.

View: a representation of a whole system from the perspective of a related set of concerns.

Viewpoint: A viewpoint establishes the purposes (concerns) and audience (stakeholders) for a view and the techniques or methods employed in constructing a view.

Standard:
Most Common Architectural Views

Typically, two different views are used:

**Structural Spec.**
- AlarmUR1
- User1
- Check1
- AlarmUR2
- User2
- Check2
- AckRU1
- AckRU2
- Timer

**Behavioral Spec.**
- Posets, pre-post conditions
- Process algebras
- Labeled transition systems, IO Automata, Buchi automata
- Statechart, UML state machines

- Component-based languages
- UML notations
Philippe Kruchten’s 4+1 View Model

**Logical View**
- Object Model of Design
- End-user Functionality

**Implementation (Development) View**
- Static Organization of the Software
- Programmers Software management

**Process View**
- Concurrency and Synchronization
- System integrators Performance Scalability Throughput

**Deployment View**
- System engineering System topology Delivery, installation Communication
- Software Mapping To Hw

**Use Case View** (Scenarios)

**Conceptual**

**Physical**
Philippe Kruchten’s 4+1 View Model

Logical View:

• Addresses the end user's concerns: captures the functional requirements, i.e., the services the system should provide to its end users.

• Typically, it shows the key abstractions (e.g., classes and associations amongst them).
  
  • In OO systems, this is often at the class level. In complex systems, you may need a package view and decompose the packages into multiple class diagrams. In other paradigms, you may be interested in representing modules and the functions they provide.

• The end result should be a mapping of the required functionality to components that provide that functionality.

Suggested UML notations:

• class diagrams, object diagrams, state charts, and composite structures
Philippe Kruchten’s 4+1 View Model

Logical View:
- Addresses the end user’s concerns: captures the functional requirements, i.e., the services the system should provide to its end users.
- Typically, it shows the key abstractions (e.g., classes and associations amongst them).
- In OO systems, this is often at the class level. In complex systems, you may need a package view and decompose the packages into multiple class diagrams. In other paradigms, you may be interested in representing modules and the functions they provide.
- The end result should be a mapping of the required functionality to components that provide that functionality.

Suggested UML notations:
- class diagrams, object diagrams, state charts, and composite structures
Philippe Kruchten’s 4+1 View Model

Process View:

- Focuses on behaviour. Addresses needs of system integrators, i.e., people designing the whole system and then integrating the subsystems or the system into a system of systems.
- This view shows tasks and processes that the system has, interfaces to the outside world and/or between components within the system, the messages sent and received, and how performance, availability, fault-tolerance, and integrity are being addressed. The process view also specifies which thread of control executes each operation of each class identified in the logical view.

Suggested UML notations:

- sequence diagrams, communication diagrams, activity diagrams, timing diagrams, interaction overview diagrams
Philippe Kruchten’s 4+1 View Model

Process View:
- Focuses on behavior. Addresses needs of system integrators, i.e., people designing the whole system and then integrating the subsystems or the system into a system of systems.
- This view shows tasks and processes that the system has, interfaces to the outside world and/or between components within the system, the messages sent and received, and how performance, availability, fault-tolerance, and integrity are being addressed. The process view also specifies which thread of control executes each operation.

Suggested UML notations:
- sequence diagrams, communication diagrams, activity diagrams, timing diagrams, interaction overview diagrams...
Philippe Kruchten’s 4+1 View Model

Implementation View:

• Focuses on the organization of the actual software modules in the software-development environment.

• The software is packaged in small chunks (program libraries or subsystems) that can be developed by one or more developers.

Suggested UML notations:

• component diagrams, package diagrams
Philippe Kruchten’s 4+1 View Model

Implementation View:
- Focuses on the organization of the actual software modules in the software-development environment.
- The software is packaged in small chunks (program libraries or subsystems) that can be developed by one or more developers.

Suggested UML notations:
- component diagrams, package diagrams
Philippe Kruchten’s 4+1 View Model

Deployment View:

• Defines how the various elements identified in the logical, process, and implementation views (networks, processes, tasks, and objects) must be mapped onto the various nodes. (=> Physical View)

• Takes into account the system's non-functional requirements such as system availability, reliability (fault-tolerance), performance (throughput), and scalability.

• This view is primarily for system designers and administrators who need to understand the physical locations of the software, physical connections between nodes, deployment and installation, and scalability.

Suggested UML notations:

• deployment diagrams
Philippe Kruchten’s 4+1 View Model

Deployment View:

- Defines how the various elements identified in the logical, process, and implementation views (networks, processes, tasks, and objects) are mapped onto the various nodes. (=> Physical View)
- Takes into account the system’s non-functional requirements such as system availability, reliability (fault-tolerance), performance (throughput), and scalability.
- This view is primarily for system designers and administrators who need to understand the physical location of software, physical connections between nodes, deployment and installation, and scalability.

Suggested UML notations:

- deployment diagrams
Philippe Kruchten’s 4+1 View Model

Scenarios:

- The scenarios help to capture the requirements so that all the stakeholders understand how the system is intended to be used.

- This view is redundant with the other ones (hence the "+1"), but it plays two critical roles:
  - Acts as a driver to help designers discover architectural elements during the architecture design;
  - Validates and illustrates the architecture design, both on paper and as the starting point for the tests of an architectural prototype.

Suggested UML notations:

- Use case diagrams, activity diagrams
Philippe Kruchten’s 4+1 View Model

Scenarios:
- The scenarios help to capture the requirements so that all the stakeholders understand how the system is intended to be used.
- This view is redundant with the other ones (hence the "+1"), but it plays two critical roles:
  - Acts as a driver to help designers discover architectural elements during the architecture design;
  - Validates and illustrates the architecture design, both on paper and as the starting point for the tests of an architectural prototype.

Suggested UML notations:
- Use case diagrams, activity diagrams
Structure of Lecture 07

• Why Architecture?
• Terminology: Architect, Architecting, Architecture
• Viewpoints and View Models
• Notation
• Architecture & Design Patterns
Three Flavours of Architectural Notations

- Informal
  - Hand-drawn figures on the whiteboard
  - Plain text
- Semi-Formal: **UML**

- Formal
  - Architecture Description Languages (ADLs)
    
Conceptual View → Customer, Users
More technical view → Developers (same system as on previous slide)
A University Course Catalogue System
(see full report on course wiki)

Logical & Implementation View

Deployment View

Process View
Formal Architecture Description Languages (ADLs)

Darwin Example

```plaintext
component filter{
    provide output<stream char>;
    require input<stream char>;
}

component pipeline(int n){
    provide output;
    require input;
    array F[n]: filter;
    forall k:0..n-1{
        inst F[k] @ k+1;
        when k < n-1
            bind F[k+1].input -- F[k].output;
    }
    bind
        F[0].input -- input;
        output -- F[n-1].output;
}
```
Comparison of Notations

<table>
<thead>
<tr>
<th>Informal</th>
<th>UML-based</th>
<th>Formal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pro:</strong></td>
<td><strong>Pro:</strong></td>
<td><strong>Pro:</strong></td>
</tr>
<tr>
<td>of immediate use</td>
<td>not too difficult</td>
<td>formal semantics</td>
</tr>
<tr>
<td>perfect for sketching</td>
<td>same notation for SA and design modeling</td>
<td>computable</td>
</tr>
<tr>
<td>communicative</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cons:</strong></td>
<td><strong>Cons:</strong></td>
<td><strong>Cons:</strong></td>
</tr>
<tr>
<td>Ambiguous</td>
<td>not a 100% fit</td>
<td>difficult to learn</td>
</tr>
<tr>
<td>non automated</td>
<td>tool investment</td>
<td>general lack of industrial tools</td>
</tr>
<tr>
<td></td>
<td></td>
<td>proliferation</td>
</tr>
</tbody>
</table>
Structure of Lecture 07

• Why Architecture?
• Terminology: Architect, Architecting, Architecture
• Viewpoints and View Models
• Notation
• Architecture & Design Patterns
Learning from Others: Patterns, Styles, and DSSAs

Experience is conveyed in the form of:
- Guidelines
- Practices
- Rules
- Patterns
- Styles

How to solve a problem?

- Solve the problem (design architecture) **from scratch**
  - Unexpected solutions can be found
  - Labor-intensive and error-prone
- Apply a **generic solution/strategy (style/pattern)** and adapt it to the problem at hand
  - Reuse, less work and less errors
  - Generic solution might be ill-fitting or too generic, requiring rework
- Apply a solution **specific for your domain (DSSA)**
  - Highest amount of reuse
  - What if such solution does not exist?
Examples of Domains

- Compilers for programming languages
- Consumer electronics
- Electronic commerce system/Web stores
- Video game
- Business applications
  - Basic/Standard/“Pro”

We can subdivide, too:
  - Avionics systems -> Boeing Jets -> Boeing 747-400
Domain-Specific Software Architectures

A DSSA comprises:

- A **reference architecture**, which describes a general computational framework for a significant domain of applications
- A **component library**, which contains reusable chunks of domain expertise, and
- An **application configuration method** for selecting and configuring components within the architecture to meet particular application requirements

Examples:

- ADAGE for avionics, AIS for adaptive intelligent systems, and MetaH for missile guidance, navigation, and control systems
Reference Architecture – Example

- Structural view of Lunar Lander DSSA
- Invariant with explicit points of variation
  - Satellite relay
  - Sensors
Reference Architecture

Reference architectures is the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation.

MURA:
Microsoft Upstream Reference Architecture (Oil & Gas Industry)

Data Integration and Business Process Management

Which models exactly, what integration mechanisms...

DSSAs also include ...

A **component library** contains reusable chunks of domain expertise.

An **application configuration method** for selecting and configuring components within the architecture to meet particular application requirements.

**MURA:**
Microsoft Upstream Reference Architecture (Oil & Gas Industry)

---

**MURA Guiding Principles to Microsoft Technology**

**BizTalk (integration), SQL Server (data store), ...**
Patterns, Styles, and DSSAs

General constraints: Usually there is one dominant style for each architecture.

- Pipes and filters
- Data abstraction and object-oriented organization
- Layered systems
- Repositories
- Event-based, implicit invocation
- ... and many more

Patterns, Styles, and DSSAs
Architectural Patterns

• An **architectural pattern** is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears.

• Similar to DSSAs but applied “at a lower level” and within a much narrower scope.

• Examples:
  • State-Logic-Display: Three-Tiered Pattern
  • Model-View-Controller
State-Logic-Display (a.k.a. Three-Tier Pattern)

- **Fundamental rule:**
  - No direct communication between Display and State

- **Display, Logic and State**
  - are developed and maintained as independent modules,
  - most often on separate platforms,
  - often using different technologies
State-Logic-Display (a.k.a. Three-Tier Pattern)

- “Business logic”
  - Tax calculation rules
  - Game rules
  - ...

- Application Examples
  - Business applications
  - Multi-player games
  - Web-based applications
State-Logic-Display in Web Development

Static or cached dynamic content rendered by the browser.
JavaScript, Ajax, Flash, jQuery…

Dynamic content processing and generation level application server
Java, .NET, ColdFusion, PHP, Perl, Rails…

Database + connection (e.g., ORM like Hibernate, Java Persistence API, …)
Model-View-Controller (MVC)

- **Objective:** Separation between information, presentation and user interaction.
Model-View-Controller (MVC)

- **Objective**: Separation between information, presentation and user interaction.

- When a **model** object value changes, a notification is sent to the **view** and to the **controller**.
  - view updates itself
  - controller modifies the view if its logic so requires.

- User input is sent to the controller
  - If a change is required, the controller updates the model.
MVC Code Example

Simple Model View Controller (MVC) JavaFX example

by Almas Baimagambetov

University of Brighton

Copyright © 2018 Almas Baimagambetov

https://github.com/AlmasB/FXTutorials/tree/master/src/com/almasb/calc
package com.almasb.calc;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class App extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("ui.fxml"));
        primaryStage.setScene(new Scene(root));
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
package com.almasb.calc;

public class Model {
    public long calculate(long number1, long number2, String operator) {
        switch (operator) {
            case "+":
                return number1 + number2;
            case "-":
                return number1 - number2;
            case "*":
                return number1 * number2;
            case "/":
                if (number2 == 0)
                    return 0;
                return number1 / number2;
        }

        System.out.println("Unknown operator - " + operator);
        return 0;
    }
}

Model:
- Contains the business logic
- Doesn’t know about view
- Doesn’t know about controller
MVC Code Example
ui.fxml (=View)

<?xml version="1.0" encoding="UTF-8"?>
<?import java.net.*?>
<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.beans.property.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.shape.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.image.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.text.*?>
<?import javafx.scene.paint.Color?>

<VBox spacing="10" alignment="CENTER" prefWidth="300" prefHeight="300"
    <fx:define>
        <Font fx:id="FONT" size="18" />
    </fx:define>

    <StackPane alignment="CENTER">
        <Rectangle fill="TRANSPARENT" stroke="GRAY" width="230" height="50" />
        <Text fx:id="output" font="$FONT" />
    </StackPane>
</VBox>
MVC Code Example

ui.fxml (=View)

```xml
<HBox spacing="10" alignment="CENTER">
    <Button text="7" prefWidth="50" font="$FONT" onAction="#processNumpad" />
    <Button text="8" prefWidth="50" font="$FONT" onAction="#processNumpad" />
    <Button text="9" prefWidth="50" font="$FONT" onAction="#processNumpad" />
    <Button text="/" prefWidth="50" font="$FONT" onAction="#processOperator" />
</HBox>

<HBox spacing="10" alignment="CENTER">
    <Button text="4" prefWidth="50" font="$FONT" onAction="#processNumpad" />
    <Button text="5" prefWidth="50" font="$FONT" onAction="#processNumpad" />
    <Button text="6" prefWidth="50" font="$FONT" onAction="#processNumpad" />
    <Button text="*" prefWidth="50" font="$FONT" onAction="#processOperator" />
</HBox>

<HBox spacing="10" alignment="CENTER">
    <Button text="1" prefWidth="50" font="$FONT" onAction="#processNumpad" />
    <Button text="2" prefWidth="50" font="$FONT" onAction="#processNumpad" />
    <Button text="3" prefWidth="50" font="$FONT" onAction="#processNumpad" />
    <Button text="-" prefWidth="50" font="$FONT" onAction="#processOperator" />
</HBox>

<HBox spacing="10" alignment="CENTER">
    <Button text="0" prefWidth="110" font="$FONT" onAction="#processNumpad" />
    <Button text="=" prefWidth="50" font="$FONT" onAction="#processOperator" />
    <Button text="+" prefWidth="50" font="$FONT" onAction="#processOperator" />
</HBox>
</VBox>
```
MVC Code Example
Controller.java

```java
package com.almasb.calc;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.text.Text;

public class Controller {

    @FXML
    private Text output;

    private long number1 = 0;
    private String operator = "";
    private boolean start = true; // flag that signals whether a new calculation may start

    private Model model = new Model();

    @FXML
    private void processNumpad(ActionEvent event) {
        if (start) {
            output.setText("" + value);
            start = false;
        }
    }

    Controller:
    - Receives user input from view (-> event)
    - Modifies view if needed (-> output)

    Processes numbers (operands)
    [operators see next slide]
```
MVC Code Example
Controller.java

```java
@FXML
private void processOperator(ActionEvent event) {
    String value = ((Button)event.getSource()).getText();
    if (!"=".equals(value)) {
        if (!operator.isEmpty())
            return;

        operator = value;
        number1 = Long.parseLong(output.getText());
        output.setText(" ");
    } else {
        if (operator.isEmpty())
            return;

        output.setText(String.valueOf(model.calculate(number1,
            Long.parseLong(output.getText()), operator)));
        operator = " ";
        start = true;
    }
}
```

Controller:
- Receives user input from view
  (-> event)
- Modifies view if needed
  (-> output)
- Updates model if needed
  (-> calculate)
MVC Code Example

Controller.java

```java
@FXML
private void processOperator(ActionEvent event) {
    String value = ((Button)event.getSource()).getText();

    if (!"=".equals(value)) {
        if (!operator.isEmpty())
            return;

        operator = value;
        number1 = Long.parseLong(output.getText());
        output.setText("");
    } else {
        if (operator.isEmpty())
            return;

        output.setText(String.valueOf(model.calculate(number1,
                                                 Long.parseLong(output.getText()), operator)));
        operator = "";
        start = true;
    }
}
```

If operator is not "=" and (previous) operator is not empty (i.e., was +, -, * or /) then do nothing (=return)

If operator is not "=" and (previous) operator is empty then
- operator is +, -, * or /
- number1 is last entered number
- output display is cleared
@FXML
private void processOperator(ActionEvent event) {
    String value = ((Button)event.getSource()).getText();

    if (!"=".equals(value)) {
        if (!operator.isEmpty())
            return;

        operator = value;
        number1 = Long.parseLong(output.getText());
        output.setText("");
    }
    else {
        if (operator.isEmpty())
            return;

        output.setText(String.valueOf(model.calculate(number1,
            Long.parseLong(output.getText()), operator)));
        operator = "";
        start = true;
    }
}
Benefits and Liabilities of MVC

**Benefits**
- Supports *multiple* views
  - Users can individually change the appearance of the web-pages based on the same model
- Well-suited for *evolution*
  - User interface requirements change faster than the models
  - Changes are limited to the views only

**Liabilities**
- **Complexity**
  - new levels of indirection
  - behavior becomes more event-driven complicating debugging
- **Communication**
  - If model is frequently updated, it could flood the views with update requests.
Design Patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution.

- A pattern is a description of the problem and the essence of its solution.

- It should be sufficiently abstract to be reused in different settings.

- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

**ELEMENTS:**

Name

A meaningful pattern identifier.

Problem description.

Solution description (might have an example)

Not a concrete design but a template for a design solution that can be instantiated in different ways.

Benefits and Consequences

The results and trade-offs of applying the pattern.
The Observer Pattern

Name: Observer

Problem description

• Situations where multiple displays of state are needed.

Solution description

• Separates the display of object state from the object itself.
• See UML description.

Consequences

• Optimisations to enhance display performance are difficult.
The Observer Pattern – Inbuilt in JDK

Inbuilt Observer Pattern in Java:

Java has an inbuilt Observer pattern implementation using:

- Class `java.util.Observable` (represents Subject [=publisher])
- Interface `java.util.Observer` (represents an Observer [=subscriber])

Concrete Observers in Java need to implement the Observer interface, whereas concrete Subject needs to extend Observable to provide its own notification logic.

Source:
http://www.javabrahman.com/design-patterns/observer-design-pattern-in-java/
The Observer Pattern – Example Implementation (Class Diagram)

Source: http://www.javabrahman.com/design-patterns/observer-design-pattern-in-java/
The Observer Pattern – Example Implementation (Code)

```
import java.util.Observable;
public class Publisher extends Observable{
    public void changeStateTo(String newStateName){
        this.setChanged();
        this.notifyObservers(newStateName);
    }
}

public class Subscriber1 implements Observer{
    String currentPublisherState;
    @Override
    public void update(Observable o, Object arg){
        System.out.println("New state received by Subscriber 1:");
        this.currentPublisherState=(String)arg;
    }
}

public class Subscriber2 implements Observer{
    String currentPublisherState;
    @Override
    public void update(Observable o, Object arg){
        System.out.println("New state received by Subscriber 2:");
        this.currentPublisherState=(String)arg;
    }
}

public class Client {
    public static void main(String args[]){
        Publisher publisher=new Publisher();
        publisher.addObserver(new Subscriber1());
        publisher.addObserver(new Subscriber2());
        publisher.changeStateTo("assigned A");
        publisher.changeStateTo("assigned B");
        publisher.changeStateTo("assigned C");
    }
}
```

Source:
http://www.javabrahman.com/design-patterns/observer-design-pattern-in-java/
The Observer Pattern – Example Implementation (Code)

import java.util.Observable;
public class Publisher extends Observable{
    public void changeStateTo(String newStateName){
        this.setChanged();
        this.notifyObservers(newStateName);
    }
}

public class Client {
    public static void main(String args[]){
        Publisher publisher=new Publisher();
        publisher.addObserver(new Subscriber1());
        publisher.addObserver(new Subscriber2());
        publisher.changeStateTo("assigned A");
        publisher.changeStateTo("assigned B");
        publisher.changeStateTo("assigned C");
    }
}

Output:
New state received by Subscriber 2:assigned A
New state received by Subscriber 1:assigned A
New state received by Subscriber 2:assigned B
New state received by Subscriber 1:assigned B
New state received by Subscriber 2:assigned C
New state received by Subscriber 1:assigned C
The Observer Pattern – Example Implementation (Code)

Lets quickly go through what’s there in Java’s example’s class diagram & corresponding code –

• Publisher is the Subject. It extends java.util.Observable.

• Subscriber1 & Subscriber2 are the Observers. They implement java.util.Observer.

• Client first initiates Publisher. It then adds one instance each of Subscriber1 & Subscriber2 to Publisher’s list of Observers.

• Client then invokes method changeStateTo() with new state value as “assigned”. Internally, Publisher then initiates notifyObservers() with this new state value. Before notifying, Publisher calls setStateChanged() which is a requirement of the Java’s observer pattern implementation.

• update() methods of Subscriber1 and Subscriber 2 are called internally by the notifyObservers() method and the new state value received by both in the parameter arg is printed.
Three Types of Patterns

**Creational patterns:**
- Deal with initializing and configuring classes and objects

**Structural patterns:**
- Deal with decoupling interface and implementation of classes and objects
- Composition of classes or objects

**Behavioral patterns:**
- Deal with dynamic interactions among societies of classes and objects
- How they distribute responsibility

<table>
<thead>
<tr>
<th>Scope of pattern</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class-level patterns</td>
<td>Factory method</td>
<td>Adapter</td>
<td>Interpreter Template Method</td>
</tr>
<tr>
<td>Object-level patterns</td>
<td>Abstract Factory Builder Prototype Singleton</td>
<td>Adapter Bridge Composite Decorator Façade Proxy</td>
<td>Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor</td>
</tr>
</tbody>
</table>
Benefits of Design Patterns

• Design patterns enable large-scale reuse of software architectures and also help document systems
• Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
• Patterns help improve developer communication
• Pattern names form a common vocabulary

More on Design Patterns: Mini-Tutorials by Derek Banas
https://www.youtube.com/playlist?list=PLF206E906175C7E07
Further Reading

- **George Fairbanks: Just Enough Software Architecture, 2012.**
- Erich Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, 1995.
- Mary Shaw and David Garlan, Software Architecture; Perspectives of an Emerging Discipline, 1995.
- **Richard Taylor et al.: Software Architecture, University of California at Irvine, Lecture 2011.**
- **Hans van Vliet: Software Architecture, Free University of Amsterdam, Lecture 2008.**
Next Lecture

• Date/Time:
  • Friday, 26-Oct, 10:15-12:00
• Topic:
  • Verification and Validation (Testing) I
• For you to do:
  • Go to Labs next week!
  • Submit Homework 3
  • Start working on Homework 4