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
Week 06: Continuous Development and Integration
Week 07: Project Estimation / Architecture and Design I
Week 08: Architecture and Design II

Week 09: Verification and Validation I
Week 10: Verification and Validation II
Week 11: Refactoring (and TDD)
Week 12: Agile/Lean Methods
Week 13: Industry Guest Lecture
Week 14: Course wrap-up, review and exam preparation
Week 15: Reserve time slot (no lecture scheduled as of today)
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
- George Fairbanks: Just Enough Software Architecture, 2012 (Video: https://www.youtube.com/watch?v=x30DcBfCJRI)
- Tutorials by Derek Banas (on YouTube)
- Lecture: Advanced Software Design – Software Architecture by Dave Clarke
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)

Architecture and Design – Bridging the Gap

Requirements

Architecture (high-level Design)

Design (low-level Design)

Code
Architecture and Design – Bridging the Gap

Requirements

Architecture (high-level Design)

Design (low-level Design)

Code

Architectural design
- Overall structure: main components and their connections; determining which sub-systems you need (e.g., web server, DB...)

Detailed design
- Inner structure of main components
- Take programming language into account
Structure of Lecture 08

• Why Architecture?
• Terminology: Architect, Architecting, Architecture
• Viewpoints and View Models
• Notation
• Architecture & Design Patterns
How to see the SW Architecture?
How to see the SW Architecture?
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
Most Common Architectural Views

Typically, two different views are used:

**Structural**
- Component-based languages
- UML notations

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

Philippe Kruchten’s 4+1 View Model

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

- **Implementation 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 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 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) 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

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

UNIVERSITY OF TARTU
INSTITUTE OF COMPUTER SCIENCE
Structure of Lecture 08

- 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)
Logical & Implementation View

Deployment View

A University Course Catalogue System
(see full report on course wiki)

Process View

Logical & Implementation View

Deployment View

A University Course Catalogue System
(see full report on course wiki)

Process View
Formal Architecture Description Languages (ADLs)

Darwin Example [Darwin]

```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;
}
```

Diagram of Darwin Example
## 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>
</tbody>
</table>
Questions?
Structure of Lecture 08

• Why Architecture?
• Terminology: Architect, Architecting, Architecture
• Viewpoints and View Models
• Notation
• Architecture & Design Patterns
How to solve a problem?

• Solve **from scratch**
  • Unexpected solutions might be found
  • Labor-intensive and error-prone

• Apply a **generic solution/strategy** and adapt to problem at hand (= reuse styles/patterns)
  • Less work and errors due to reuse
  • Generic solution might be ill-fitting or too generic, requiring rework

• Apply a **solution specific to your domain** (= full reuse & configuration)
  • Highest degree of reuse
  • Such a solution might not (yet) exist
Learning from Others: Patterns, Styles, and DSSAs

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

Learning from Others: Patterns, Styles, and DSSAs

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

Examples of Domains

- Compilers for programming languages
- Consumer electronics
- Electronic commerce system/Web stores
- Video games
- 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
- Explicit points of variation
  - Satellite relay
  - Sensors
Reference Architecture – Example

- Structural view of Lunar Lander DSSA
- Explicit points of variation
  - Satellite relay
  - Sensors
Domain-Specific Software Architecture – Example from Oil and Gas Industry

MURA
Microsoft Upstream Reference Architecture
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)

BizTalk (integration), SQL Server (data store), ...

Mapping MURA Guiding Principles to Microsoft Technology
Extreme Case of DSSA ...

• What happens when the domain becomes narrower?
  • Consumer Electronics ⇒ Sony WEGA TVs
  • Avionics ⇒ Boeing 747 Family
  • ...

• **Engineering Product Line**: a set of products that have substantial commonality from a technical/engineering perspective
Product Line Architecture

- A **product-line architecture** captures the architectures of many related products simultaneously
  - Explicit **variation points**

- **Common**: features common to all products
- **A**: features specific to product A
  - Product A = Common + A
- **B**: features specific to product B
  - Product B = Common + B
Product Line Architecture – Why?

Traditional engineering

Product-line-based engineering
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

Architecture – Styles (and Patterns)

- Model View Controller
- Blackboard
- Client/Server
- Layered
- Pipe-and-Filter
Pipe-and-Filter

Components: filters that read input data stream and convert it into output data stream

Connectors: pipes that provide output of filter as input to other filter

Advantages: simple, no complex interaction, high reusability, portability

Disadvantages: require common data format, no shared state, redundancy in (un)parsing

Example: unix shell (ls –l | grep key | more …)
Client/Server

Components: server subsystem provides services to multiple instances of client sub-system

Connectors: network; client typically requests services from server

Advantages: distribution, scalability

Disadvantages: responsiveness (if network slow), robustness (if server goes down)

Example: web architecture
Web Architecture (Client/Server)
Google App Engine (Client/Server)

Javascript + HTML

Web Browser

Web Application Server (Google App Engine)

Java (servlets)

Web Services

RPC

Data

JDO

RPC = Remote Procedure Call
JDO = Java Data Objects
Layered Components:
- **presentation** – displays info to users
- **business logic** – core system functionality
- **persistence** – stores data

Connectors: **messaging mechanism** (between adjacent layers)

Advantages: clear separation of concerns

Disadvantages: might be too restrictive (in the strict form)

Example: Could be split into several layers
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…
… React, CodeKit, Vue.js, Angular.js

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)

Components:
- **model** contains core functionality and data;
- **views** display information to the user;
- **controllers** handle user input

Connectors: **change-propagation mechanism** (observer)

Advantages: interactivity, expandability due to separation of functionality (model) from presentation (view and controller)

Disadvantages: small scale solution but can become complex (heavy design)

Example: ?
Model-View-Controller (MVC)

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

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

- If user input (an event) happens, the view notifies the controller

- If the logic so requires the controller:
  - modifies the view
  - updates the model

- If a model object data changes, this propagates to the view, the view accesses the model, and updates itself
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
MVC Code Example

App.java

```java
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);
    }
}
```

Model: simple calculator

View: uses JavaFX event handler

Controller:
- Receives user input from view
- Modifies view if needed
- Updates model if needed
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
<?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)

```
<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("");
            start = false;
        }

        String value = ((Button) event.getSource()).getText();
        output.setText(output.getText() + value);
    }
}
```

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;
    }
}
```
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;
    }
}
Two Flavors of MVC: Passive Model

- The passive model is employed when one controller manipulates the model exclusively.
- The controller modifies the model and then informs the view that the model has changed and should be refreshed.
- The model is completely independent of the view and the controller, i.e. there is no means for the model to report changes in its state.
- The HTTP protocol is an example of this. There is no simple way in the browser to get asynchronous updates from the server. The browser displays the view and responds to user input, but it does not detect changes in the data on the server.
- Only when the user explicitly requests a refresh is the server interrogated for changes.
Two Flavors of MVC: Passive Model

How does this translate to our example:

- 'handleEvent' corresponds to the clicking of the 'Calc' button
- 'service' corresponds to
  'theModel.addTwoNumbers(firstNumber, secondNumber)'
- 'update' corresponds to
  'theView.setCalcSolution(...)'
- 'getData' corresponds to
  'theModel.getCalculationValue()'

However, we use already the Observer pattern, so it's a bit more complex.
Two Flavors of MVC: Active Model

- The active model is employed when the view is automatically updated whenever the model changes.

- If user input (an event) happens, the view notifies the controller.

- If the logic so requires the controller:
  - modifies the view
  - updates the model

- If a model object data changes, this propagates to the view, the view accesses the model, and updates itself.

Solution: Observer Design Pattern!
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, Styles, and DSSAs
Design Patterns

- Design Patterns – Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994 (1st ed.)
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)

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

import java.util.Observer;
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;
    }
}

import java.util.Observer;
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;
    }
}

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)

```java
import java.util.Observable;
import java.util.Observer;

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");
    }
}
```

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.
Note on Observer Pattern in JDK

Observer interface is deprecated since Java 9

- One of its cons is that Observable isn't an interface but a class, that's why subclasses can't be used as observables.
- Also, a developer could override some of the Observable's synchronized methods and disrupt their thread-safety.

Use instead `PropertyChangeListener` interface

- see https://www.baeldung.com/java-observer-pattern
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>
# Three Types of Patterns

<table>
<thead>
<tr>
<th>Scope of pattern</th>
<th>Type of pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Creational</td>
</tr>
<tr>
<td>Class-level patterns</td>
<td>Factory method</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Object-level patterns</td>
<td>Abstract Factory</td>
</tr>
<tr>
<td></td>
<td>Builder</td>
</tr>
<tr>
<td></td>
<td>Prototype</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Singleton**
Singleton

• Singleton pattern is one of the simplest design patterns in Java

• This pattern involves a single class which is responsible to create an object while making sure that only a single object gets created.

• This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class
Singleton

Useful for:

• Access to hardware resources such as printers (print spooler implemented as singleton multiple concurrent access and deadlocks)

• Producing logs (logger utility is implemented as singleton; again, avoids potential confusion due to multiple concurrent access to logger)

• Configuration file (if implemented as singleton allows only one access, loading the config data into the object once → performance advantage because it avoids repeated reading from config file)

• Cache (similar to config file; if cache object is created as a singleton object, it can have a global point of reference and for all future calls to the cache object the client application will use the in-memory object)
Singleton

Implementation:

Step 1

Create a Singleton Class.

*Singleton.java*

```java
public class Singleton {

    // create an object of Singleton
    private static Singleton instance = new Singleton();

    // make the constructor private so that this class cannot be
    // instantiated
    private Singleton(){}

    // Get the only object available
    public static Singleton getInstance(){
        return instance;
    }

    public void showMessage(){
        System.out.println("Hello World!");
    }
}
```
Singleton

Implementation:

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

```java
public class SingletonPatternDemo {
    public static void main(String[] args) {

        //illegal construct
        //Compile Time Error: The constructor SingleObject() is not visible
        //SingleObject object = new SingleObject();

        //Get the only object available
        SingleObject object = SingleObject.getInstance();

        //show the message
        object.showMessage();
    }
}
```
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
In my Design Patterns Video Tutorial I will cover all of the most common design patterns. I'll also explain when to use them and other topics on OOP design principles.
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.**
Next Lecture

- Date/Time:
  - Friday, 30-Oct, 10:15-12:00
- Topic:
  - Verification and Validation (Testing) I
- For you to do:
  - Continue working on Homework 4
  - Go to Assessment Labs next week!