TDT4240 - Design Patterns

Carl-Fredrik Sørensen
Senior Advisor – ICT Architecture
mailto:carlfrs@ieee.org

NTNU
Norwegian University of
Science and Technology

Design is hard. One way to* avoid *the act of design is to* reuse *existing designs.

—Dr. Ralph Johnson, University of Illinois
(April, 1995)

NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Content

NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Patterns history

- Origin by the architect Christopher Alexander during the late 1970s.
- The pattern movement in software: OOPSLA 1987
- In 1995: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides published: *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma95].

- Later: A lot of books and papers, some specific to Programming Languages

NTNU
Norwegian University of
Science and Technology

www.ntnu.no

What is Design?

- What is **design** ?
  - Design is a *creational* and *intentional* act.
  - Conception and construction of a structure on purpose for an purpose.
  - Takes the range of detail from **analysis** and broad **architecture** to the fine granularity of *programming languages*.
- Meet functional and non-functional requirements!

NTNU
Norwegian University of Science and Technology

Purpose of Design

- Other forces for design:
  - Create reusable code?
  - Create maintainable and extensible code?
  - Portability, Flexibility, Transparency?
- Keywords:
  - high cohesion, loose coupling
  - delegation
  - abstraction/generalisation

NTNU
Norwegian University of Science and Technology

What is a pattern?

- Many problems have structurally **similar** *solutions*; with **common** *issues* and *rationale*. Experienced developers learn to *recognise* problems and *apply* solutions.
- A pattern *captures* and *communicates* the experience in a **reusable** form.

NTNU
Norwegian University of Science and Technology

Pattern Definition (1)

*A pattern describes a **problem** which **occurs over and over again** in our environment, and then describes the core of the **solution** to the problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*

Christopher Alexander et al., Authors

A Pattern Language (Oxford, 1977)

*Design patterns are **descriptions** of communicating objects and classes that are customised to **solve** a **general design problem** in a particular **context**. One person's pattern can be another person's building block.*

Erich Gamma et al., Authors

Design Patterns (Addison-Wesley, 1994)

NTNU
Norwegian University of Science and Technology

Social Example

- Context:
  - You are in a pub and want to get a beer
- Problem:
  - Other people are waiting at the bar
- Solution:
  - You enter the line

NTNU
Norwegian University of Science and Technology

Design Example

- Context:
  - A class library providing limited functionality contains a lot of classes
- Problem:
  - The user is exposed to the internal complexity of the library
- Solution:
  - Create a new façade class that interacts with the user and hide all the details

NTNU
Norwegian University of Science and Technology

Types of Patterns (1)

- **Architectural** styles (Patterns)
  - Expresses a fundamental **structural** organisation or schema for software systems.
  - Provides a set of **predefined** subsystems, specifies their **responsibilities**, and includes **rules** and **guidelines** for organising the relationships between them.
  - Examples: Client-server, Pipe-and-filter, SOA

NTNU
Norwegian University of Science and Technology

Types of Patterns (2)

- **Design** Patterns
  - Provides a scheme for **refining** the subsystems or components of a software system, or the **relationships** between them.
  - Describes commonly **recurring** structure of communicating components that solves a general design problem within a particular context.
- **Idioms**
  - Low-level pattern **specific** to a programming language.
  - Describes how to implement **particular** aspects of components or the relationships between them using the features of the **given** language.

NTNU
Norwegian University of Science and Technology

GRASP Patterns (1)

[Craig Larman; Applying UML and Patterns.]

- Describe **fundamental** principles of assigning responsibilities to objects, expressed as patterns.
- Examples:
  - **Expert**: Assign a responsibility to the information expert – the class that has the information necessary to fulfil the responsibility.
  - **Creator**: Assign class B the responsibility to create an instance of class A if B aggregates, contains, records instances of, or closely uses A Objects, or has the initialising data that will be passed A when it is created.

GRASP Patterns (2)

[Craig Larman; Applying UML and Patterns.]

- Examples:
  - **High Cohesion**: Assign a responsibility so that cohesion remains high
  - **Loose Coupling**; Assign a responsibility so that coupling remains low.
  - **Controller**; Assign the responsibility for handling a system event message to a class. (Façade, role, use-case controller)

Design Pattern Types – GOF

- **Creational** (e.g. Factory, Singleton)
  - Encapsulates **creational** knowledge for an object in a method, a class or another object.
- **Structural** (e.g. Composite, Proxy, Façade)
  - Concerned with how *classes* and *objects* are **composed** to form larger structures.
  - Structural class patterns use inheritance to compose interfaces or implementations.
- **Behavioural** (e.g. Observer, State, Strategy)
  - Are concerned with **algorithms** and the assignment of **responsibilities** between objects.
  - Describes not just patterns of objects or classes but also the patterns of **communication** between them.

Singleton (1)

- Context:
  - A class represents a concept that requires a single instance
- Problem:
  - Clients could use this class in an inappropriate way
  - It must be accessible to clients from a well-known access point

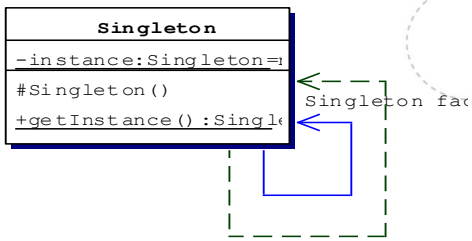
Singleton (2)

- Solution:
  - Make the class itself **responsible** for keeping track of its sole instance.
  - The class can ensure that **no other instance** can be created (by intercepting requests to create new objects), and it can provide a way to **access** the instance.
- Rationale and Motivation:
  - The singleton pattern applies to the many situations in which there needs to be a **single instance** of a class, a single object.
  - It is often left up to the programmer to insure that the important consideration in implementing this pattern is how to make this single instance easily accessible by many other objects.

Singleton Code and Structure

```
public class Singleton {
   protected Singleton(){}
   public static Singleton getInstance()
   {
       if (instance == null) {
          instance = new Singleton();
       }
       return instance;
   }

   private static Singleton instance = null;
}
```



Singleton Advantages

- Single **instance** is controlled absolutely.
- The class have **direct control** of how *many* instances can be created.
- *Reduced name space*: Improvement of global variable
- Easily extended to allow a controlled number of "singleton" objects to be created.
  - The most important modification in the operator that has control over access to the instances: The getInstance() function would need to be changed.

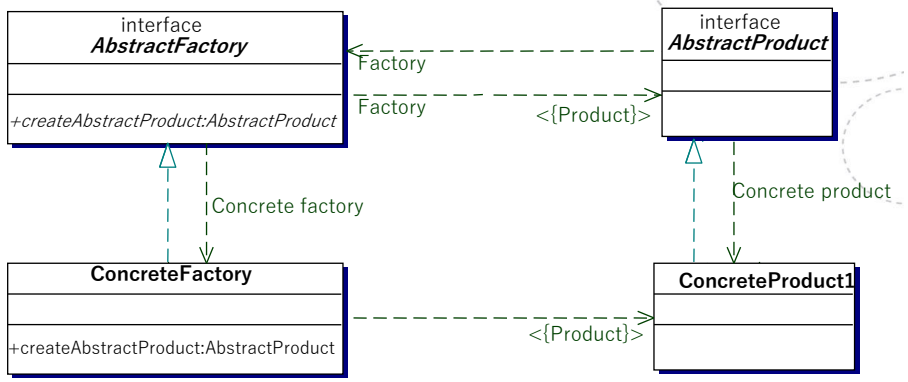
Singleton Applied

- Managing log objects, factory, connection pools…
  - The **Abstract Factory**, **Builder**, and **Prototype** patterns can use Singletons in their implementation.
  - **Facade** objects are often Singletons because only one Facade object **is required.**
  - **State objects** are often Singletons.
- Beware:
  - Singleton is frequently misused, is sometimes seen as a legitimate way of doing “OO global variables”
  - Complicates unit testing if stateful.
  - Consider threading
  - Instance may be garbage collected in Java

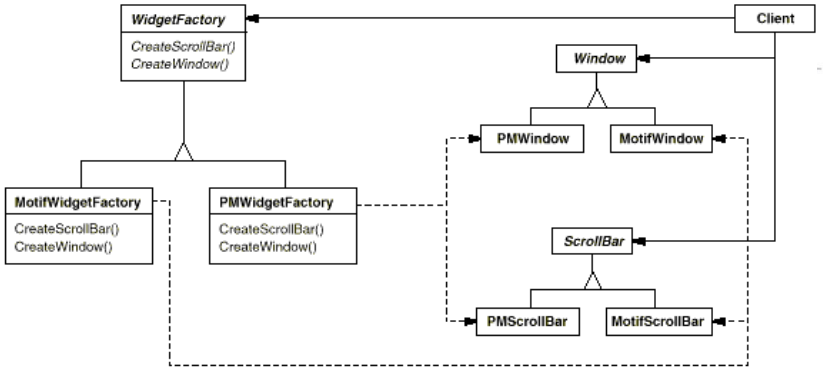
Abstract Factory

- Context
  - A family of related classes can have different implementation details
- Problem
  - The client should not know anything about which variant they are using

Abstract Factory Structure



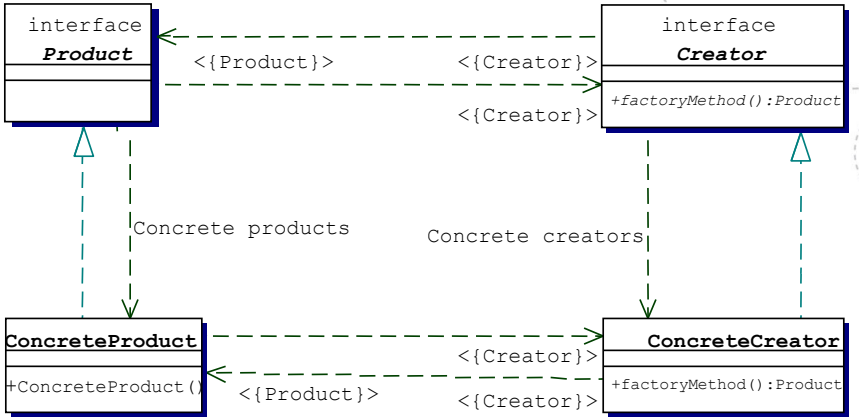
Abstract Factory Example



Factory method

- Context
  - a class *cannot* **anticipate** the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localise the knowledge of which helper subclass is the delegate.
- Problem
  - How to create an object without knowing its concrete class ?
- Solution
  - Provide a **method** for **creation** at the interface level
  - **Defer** the **actual creation** responsibility to subclasses

Factory method Structure



Factory method usage

- Factory methods are common in **toolkits** and **frameworks**.
- Parallel class hierarchies often require objects from one hierarchy to be able to create appropriate objects from another.
- Factory methods are used in **test-driven development** to allow classes to be put under test

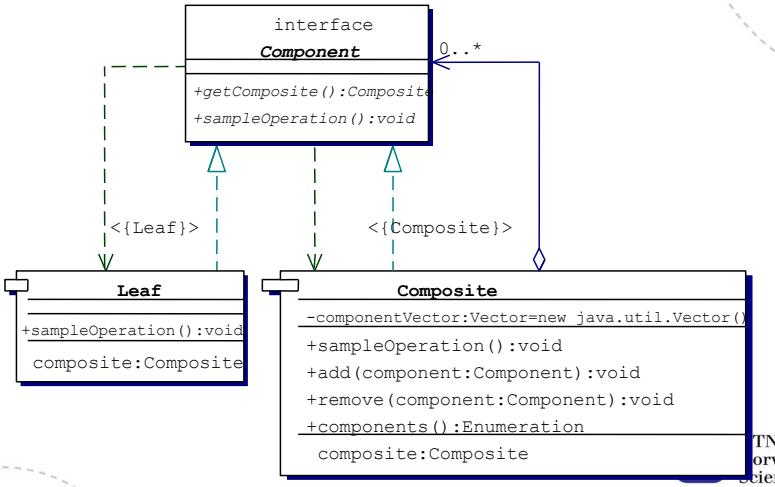
Composite (1)

- Context:
  - You need to represent part-whole hierarchies of objects
  - A client manipulating target objects either individually or grouped together.
- Problem:
  - Transparent treatment of single and multiple objects would simplify client code
  - Not all differences between single and multiple object manipulation can be ignored
  - Difference between composition objects and individual objects.
  - Allow arbitrary grouping of elements and uniform treatment of groups and primitives.

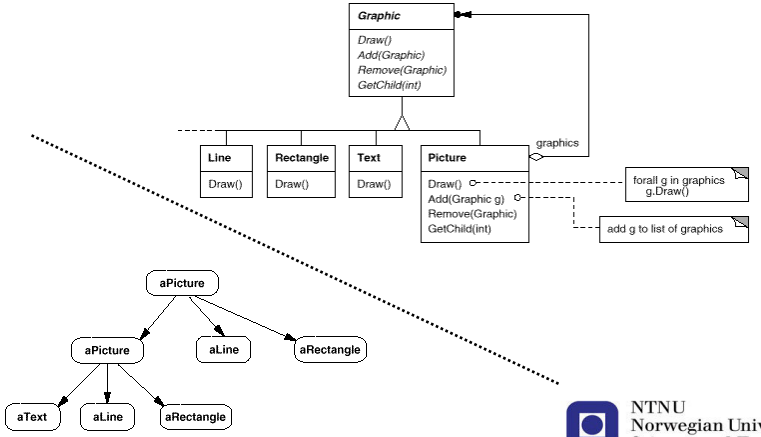
Composite (2)

- Solution:
  - Common interface for a group and primitives
  - Introduce a composite object that implements the same interface as a leaf object through a common component interface or abstract class
  - The composite object holds links to many component objects, each of which may be either leaf objects or further composite objects, and forwards its operation to each of them
- Consequences
  - Composite and leaf objects are treated uniformly
  - The component interface may include operations specific to the composite that hold no meaning for the leaves

Composite Structure



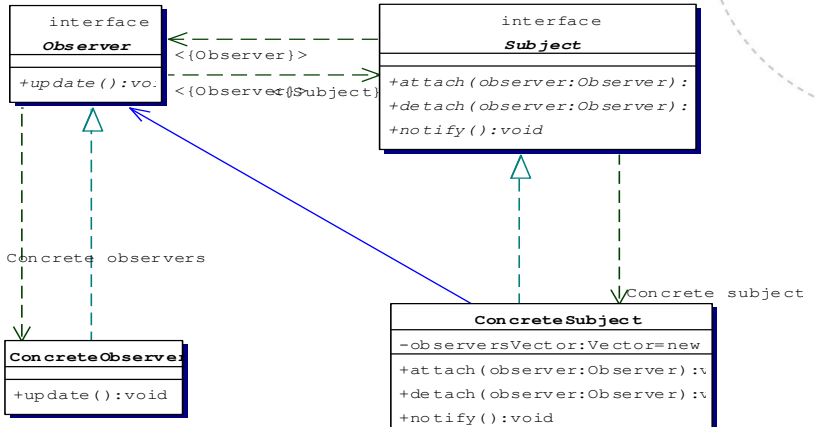
Composite Example



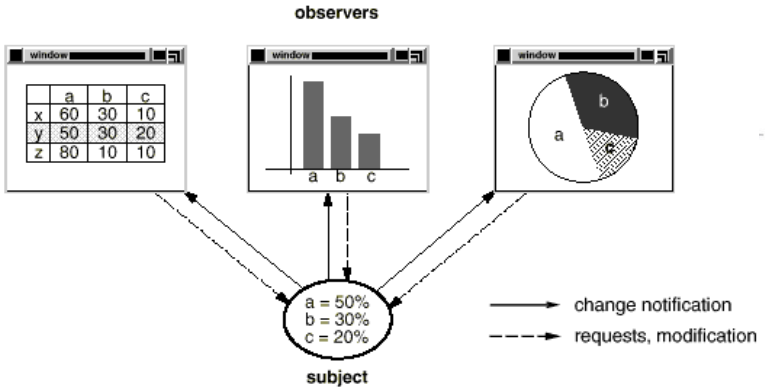
Observer (1)

- Context:
  - The change in one object may influence one or more other objects
- Problem:
  - High coupling
  - Number and type of objects to be notified may not be known in advance

Observer (2)



Observer Example

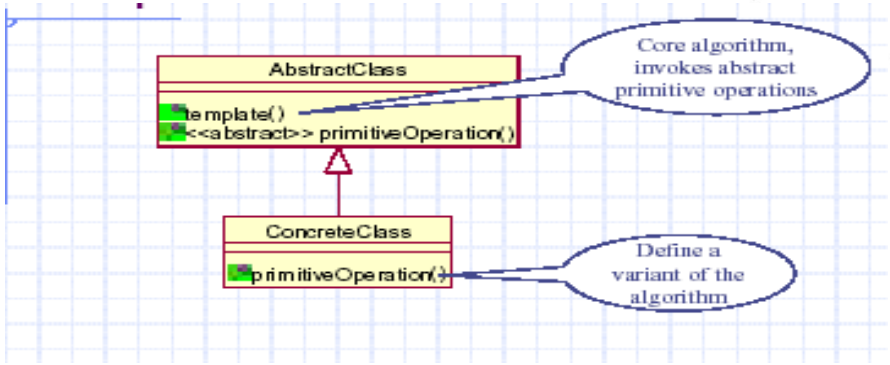


The observer pattern is implemented in numerous *programming libraries* and systems, including almost all GUI toolkits.

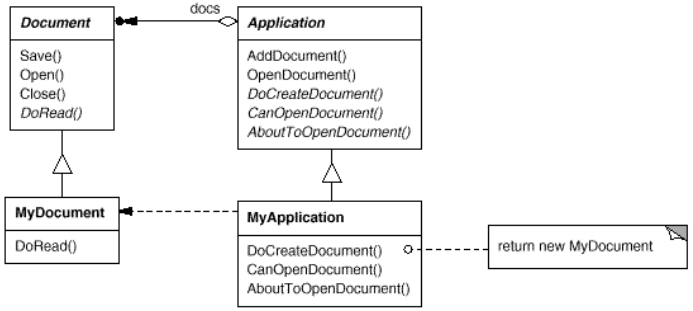
Template Method

- Context:
  - An algorithm/behaviour has a stable core and several variation at given points
- Problem
  - You have to implement/maintain several almost identical pieces of code

Template Method Structure



Template Method Example



Template Method Usage

```
// A template method
void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
        // cannot handle this document
        return;
    }
    Document* doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}
```


Template Method Usage

- Let subclasses implement (through *method overriding*) behaviour that can vary
- Avoid duplication in the code: the general workflow structure is implemented once in the abstract class's algorithm, and necessary variations are implemented in each of the subclasses.
- Control at what point(s) *subclassing* is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change

Pattern Languages

- Some patterns may be applied in a **sequence** from one to another
  - Context and resulting context describe a predecessor/successor relationship
- A pattern language **connects** patterns together
  - Intent of a language is to generate a particular kind of system
- A **pattern language** is a *collection* of patterns that build on each other to generate a **system**.
  - A pattern in isolation solves an isolated design problem;
  - A pattern language builds a system.
  - It is through pattern languages that patterns achieve their fullest power.

  [Coplien 1996]

Programming languages as Context for Design

- Support of certain mechanisms in languages and technology impacts how to think about a problem and also how to design an application.
- Examples:
  - Sub-classing, Interfaces, Polymorphism, Type checking, Packages, Reference-based objects
- A consequence is that design **cannot** be performed independent of technology.

Pattern Definition in short

A reusable **solution**
to a known **problem**
in a well-defined **context**

Characteristics of a Design Patterns

- The **name**:
  - Good names form a **vocabulary** for discussing conceptual abstractions
- The **problem**:
  - The **motivation** or **context** that this pattern applies to.
  - **Prerequisites** that should be satisfied before deciding to use a pattern.
- The **solution**:
  - A **description** of the program structure that the pattern will define.
  - A list of the **participants** needed to complete a pattern.
- The **consequences** of using the pattern...both ***positive*** and ***negative***.

Description of a Design Patterns

- Pattern Name and Classification
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Summary

- Patterns make us able to
  - **communicate/perform** designs in a more **precise** and **efficient** way.
  - reuse **best practice** solutions on recurring problems in all stages of product development.
- *Thinking* in and *use* of **patterns** makes us able to
  - create **better** designs without using already known patterns simply by giving us good examples, descriptions and solutions of “general” problems.

Conclusion

- Design patterns are
  - A Common Design Vocabulary
  - A Documentation and Learning Aid
  - An Adjunct to Existing Methods
  - A Target for **Refactoring**

NTNU
Norwegian University of Science and Technology

Patterns references

- http://en.wikipedia.org/wiki/Design_pattern_(computer_science)
- http://hillside.net/patterns/
- http://www.sw-technologies.com/dpattern/
- Books:
  - GOF
  - Patterns in Java (Mark Grand)
  - POSA (Buschmann et.al.)
  - PLoPD books
  - Analysis Patterns (Martin Fowler)
  - Design patterns (W.Pree)

NTNU
Norwegian University of Science and Technology

www.ntnu.no

NTNU
Innovation and Creativity

TDT4240 Software Architecture

Professor Alf Inge Wang

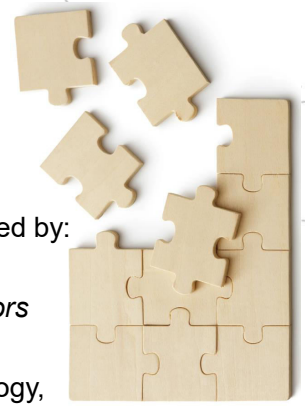
Architectural Patterns from "Software Architecture in Practice 3rd edition"

www.ntnu.no

What is an Architectural Pattern?

An architectural pattern establishes a relationship between:

- *A context*: A recurring, common situation in the world that gives rise to a problem.
- *A problem:* The problem, appropriately generalized, that arises in the given context.
- *A solution:* A successful architectural solution to the problem, appropriately abstracted. The solution for a pattern is determined and described by:
  - A set of *element types*
  - A set of *interaction mechanisms* or *connectors*
  - A *topological layout* of the components
  - A *set of semantic constraints* covering topology, element behavior, and interaction mechanisms



NTNU
Innovation and Creativity

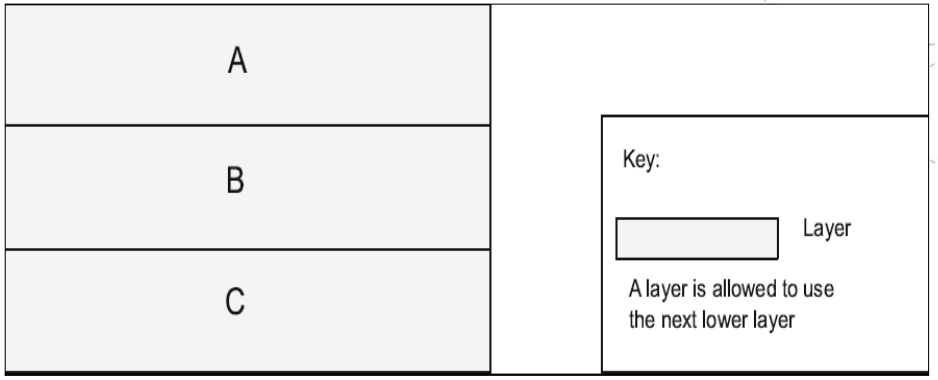
www.ntnu.no

The Layered Pattern

- Divide the software into *layers*.
- The usage must be unidirectional (a layer only uses & accesses the layer below).
- Each layer is exposed through a public interface.

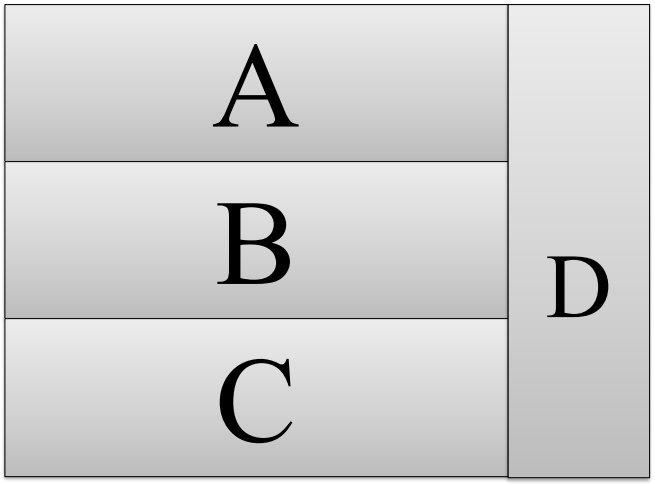
NTNU
Innovation and Creativity

Layer Pattern Example



NTNU
Innovation and Creativity

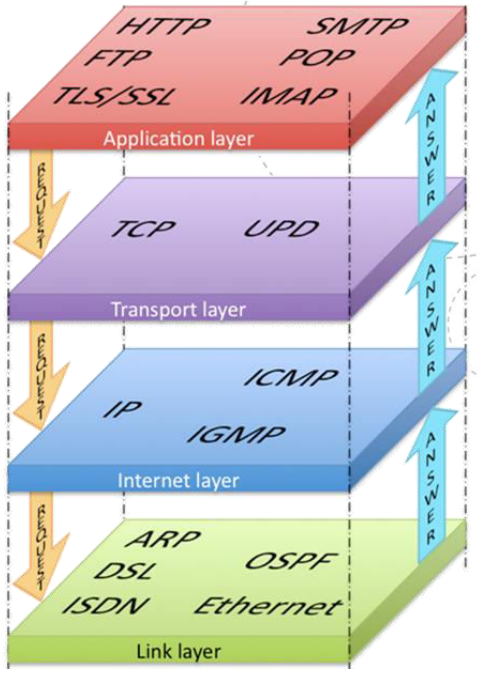
Layers with a sidecar



NTNU
Innovation and Creativity

Example:

Internet Protocol Stack

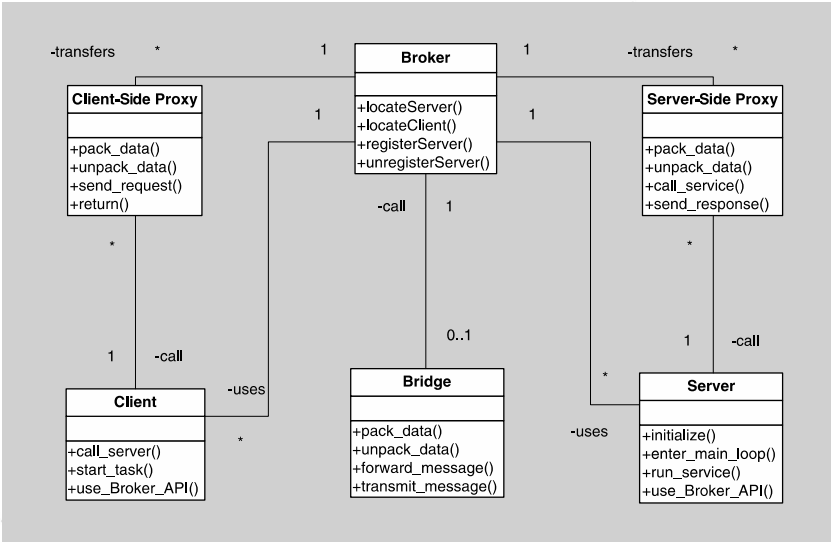


Broker Pattern

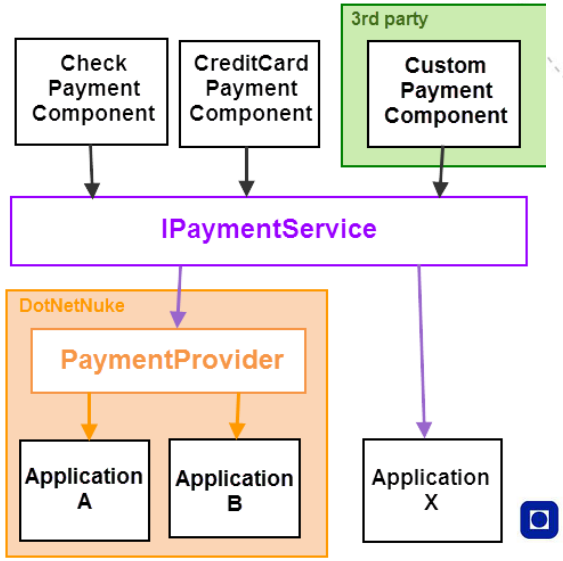
- The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a ***broker***.

NTNU
Innovation and Creativity

Broker Example



Example: Broker pattern



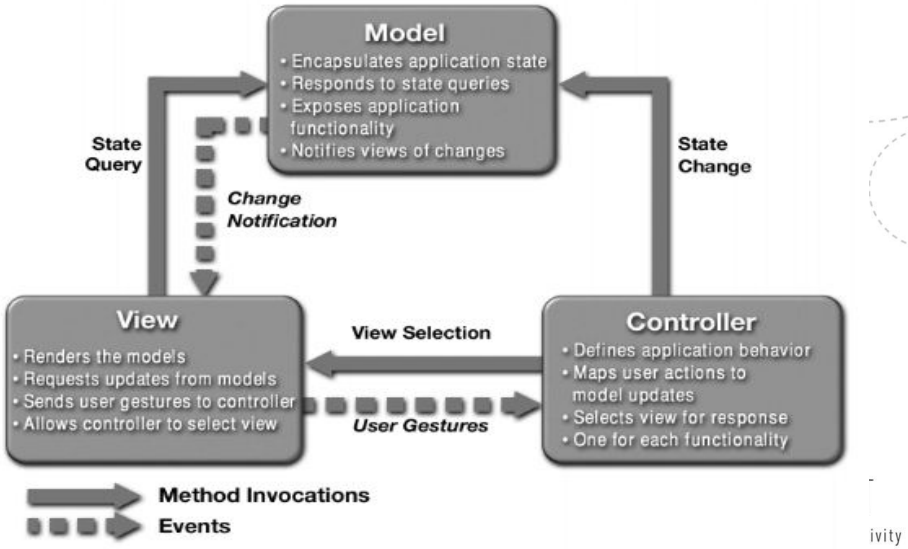
NTNU
Innovation and Creativity

Model-view Controller

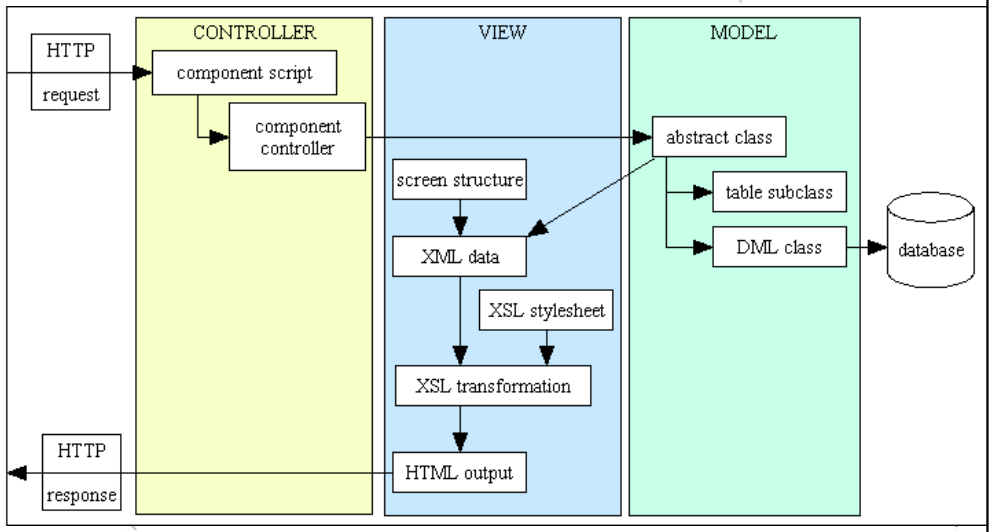
- The model-view-controller (MVC) pattern separates application functionality into three kinds of components:
  - A *model*, which contains the *application's data*
  - A *view*, which *displays* some portion of the *underlying data* and *interacts with the user*
  - A *controller*, which *mediates between the model and the view* and manages the notifications of state changes



MVC Example



Example: Model-view controller

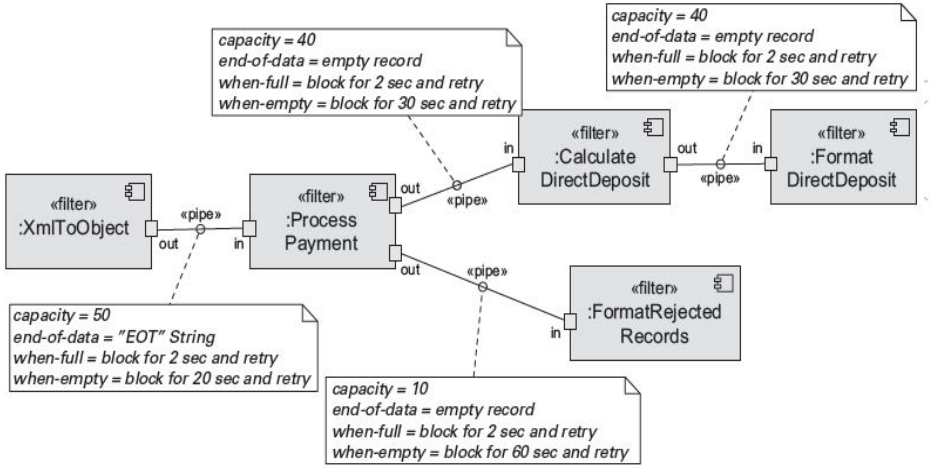


Pipe & Filter

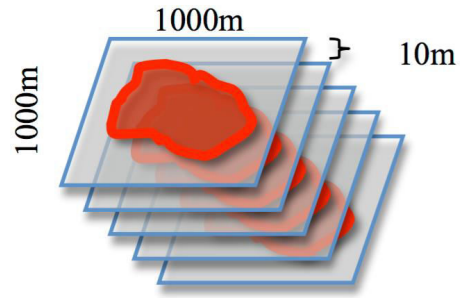
- The pipe-and-filter pattern involves successive transformations of data streams.
- Data enters through a filter's input port, is processed, and then exits through its output port to flow through a pipe to the next filter.

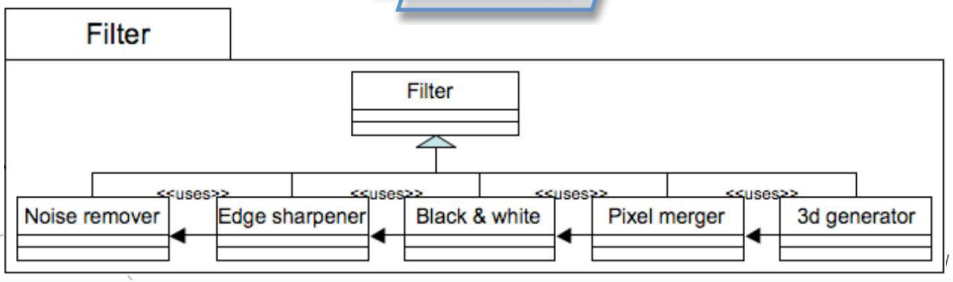
NTNU
Innovation and Creativity

Pipe and Filter Example



Pipe and Filter Example from Exam 2008: Oil Reservoir





Client-Server

- Clients interact by requesting services of servers, which provide a set of services.
- Some components may act as both clients and servers.
- There may be one central server or multiple distributed ones.

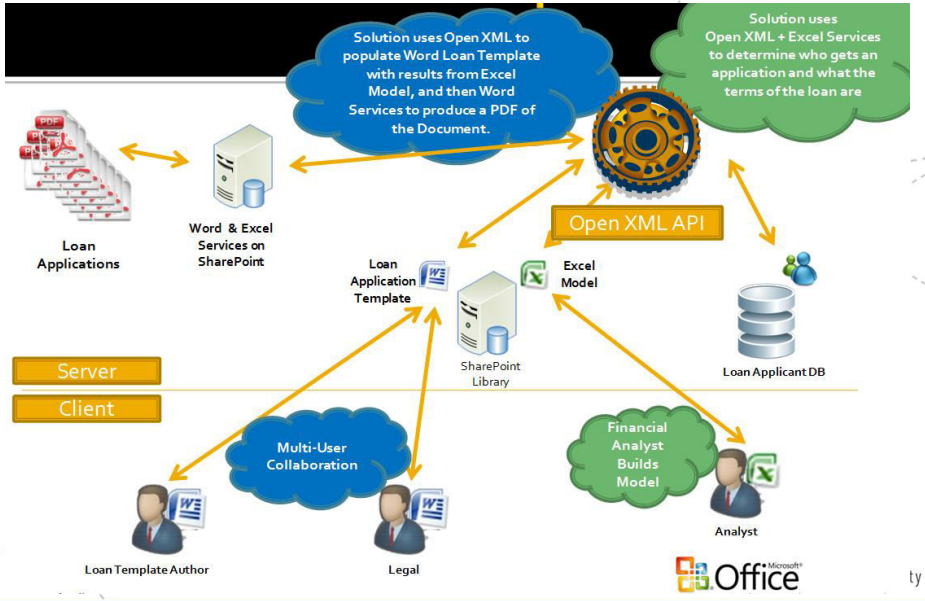
Innovation and Creativity

Client-Server pattern

- **Context:** Centralized management of shared resources ensures controlled access and quality of service for distributed clients.
- **Problem:** How to enhance modifiability and reuse by centralizing common services for easier updates?
- **Solution:** Clients request services from servers, which manage and provide shared resources.

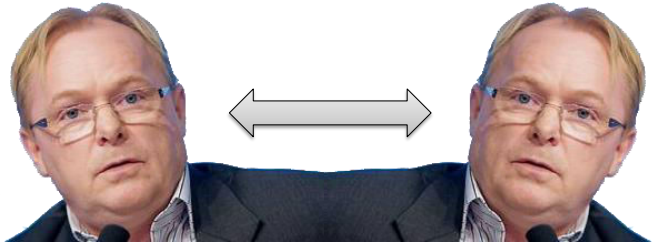
NTNU
Innovation and Creativity

Client-Server Example

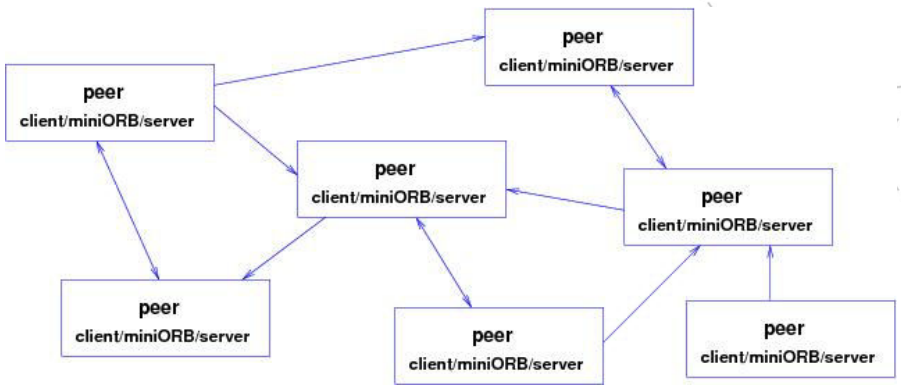


Peer-to-peer

- Components act as both clients and servers, directly interacting as equals.
- No peer or group is critical to system health, ensuring resilience.
- Communication follows a request/reply model without the asymmetry of client-server patterns.



Peer-to-Peer Example



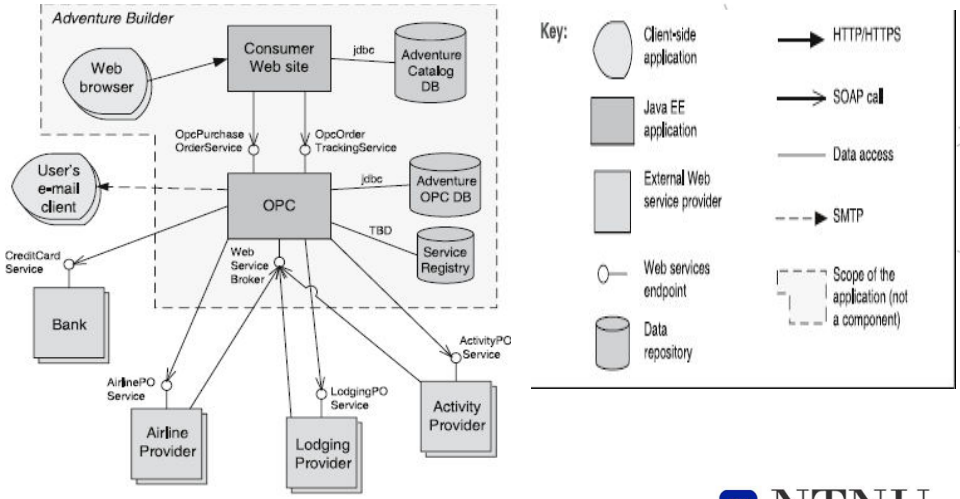
NTNU
Innovation and Creativity

Service-oriented architecture

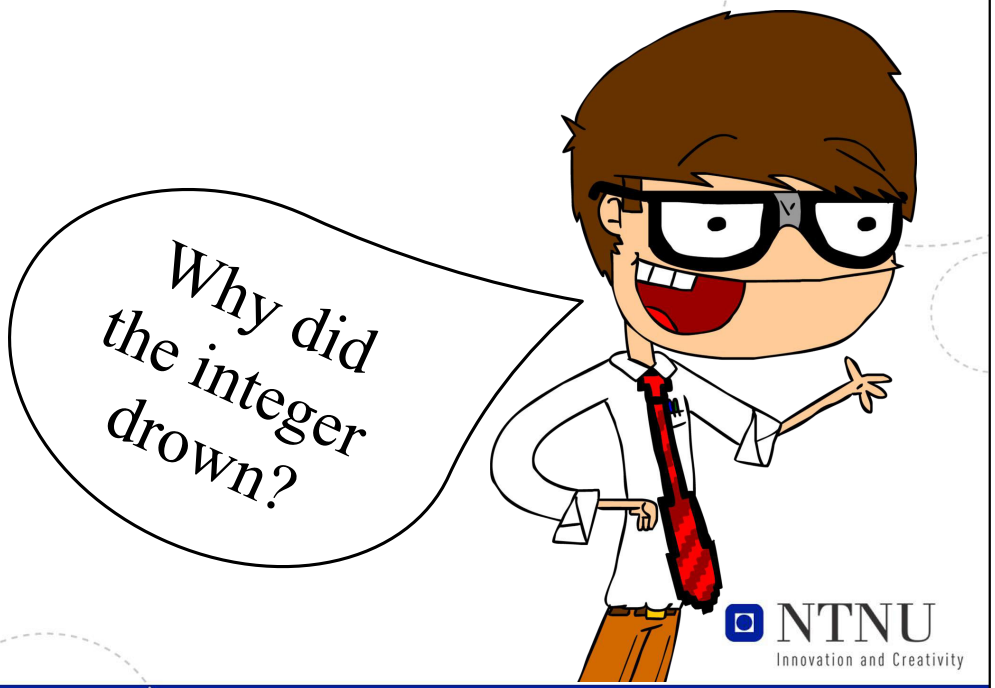
- SOA consists of distributed components that provide or consume services via a service bus.
- Each service can be independently developed on different platforms.

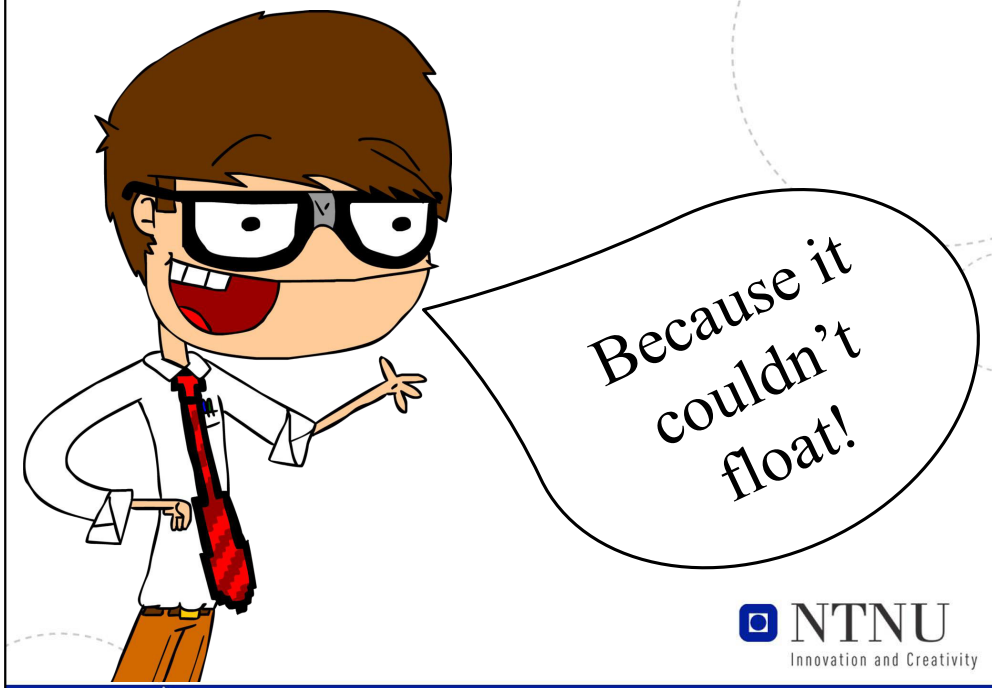
NTNU
Innovation and Creativity

Service Oriented Architecture Example



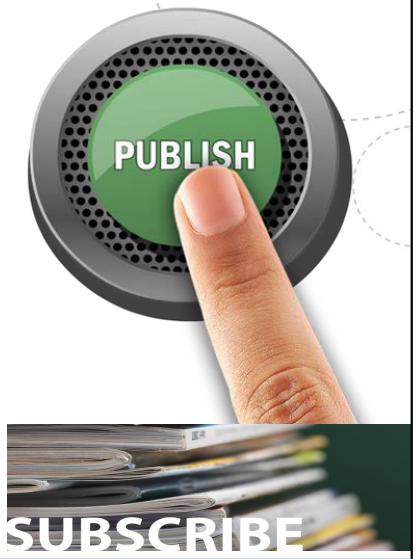
NTNU
Innovation and Creativity



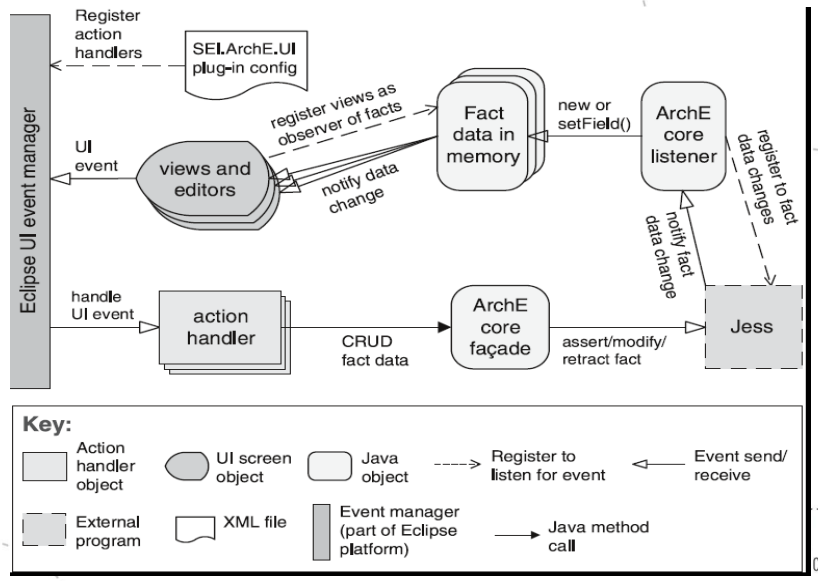


Publish-Subscribe

- Components interact through events (messages).
- Publishers announce events on a bus, which delivers them to subscribers registered for those events.

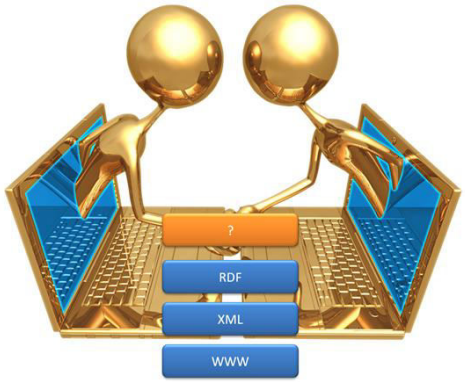


Publish-Subscribe example

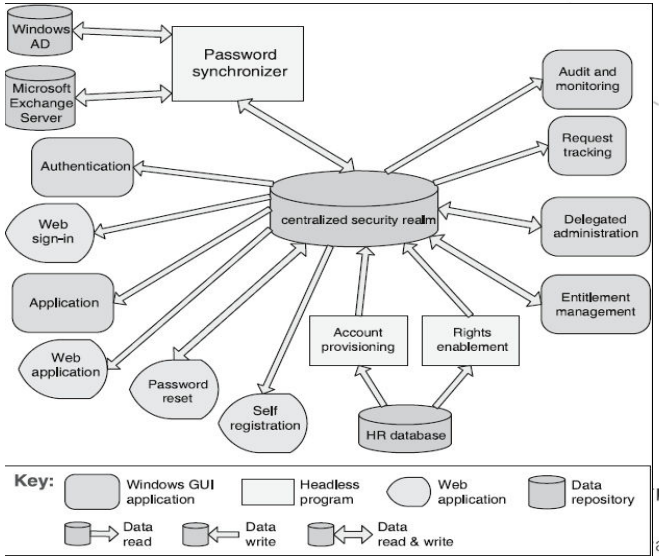


Shared-Data

- Interaction revolves around persistent data exchange between multiple accessors and a shared data store (database).
- Accessors or the data store (database) initiate data reading and writing.
- Also known to be an anti-pattern (bad).

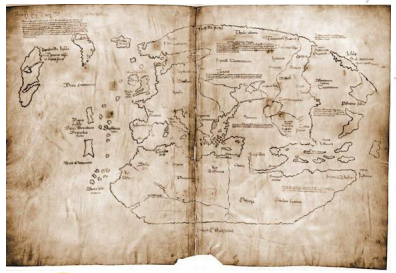


Shared Data Example



Map-Reduce

- This pattern involves three components:
  1. **Infrastructure** handles task allocation, data sorting, and parallel computing across hardware nodes.
  2. **Map** filters data to identify relevant items.
  3. **Reduce** combines the filtered results.

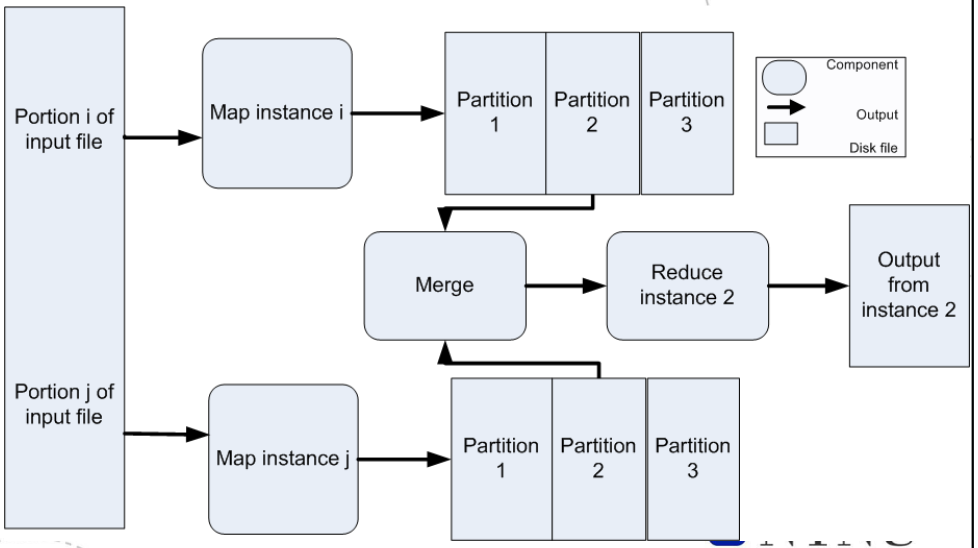


reduce

What is MapReduce?

https://www.youtube.com/watch?v=ztbNRNEuvsU

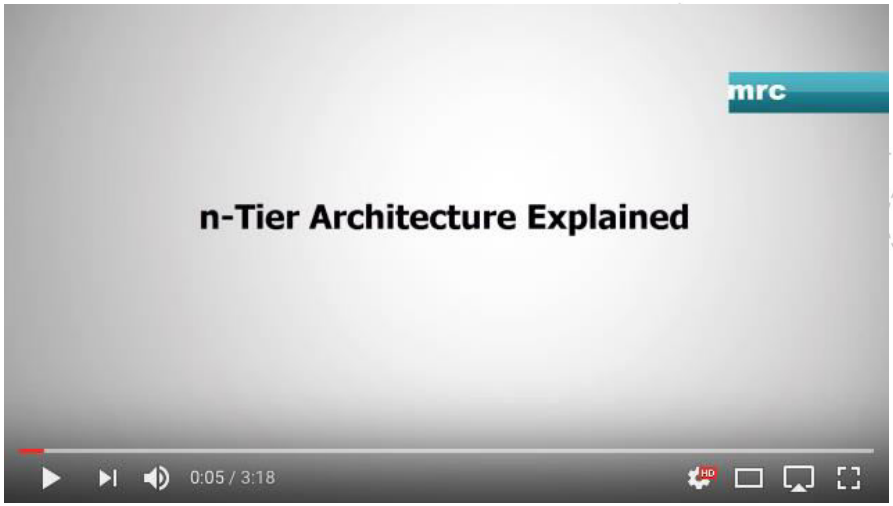
Map-Reduce Example



Multi-Tier

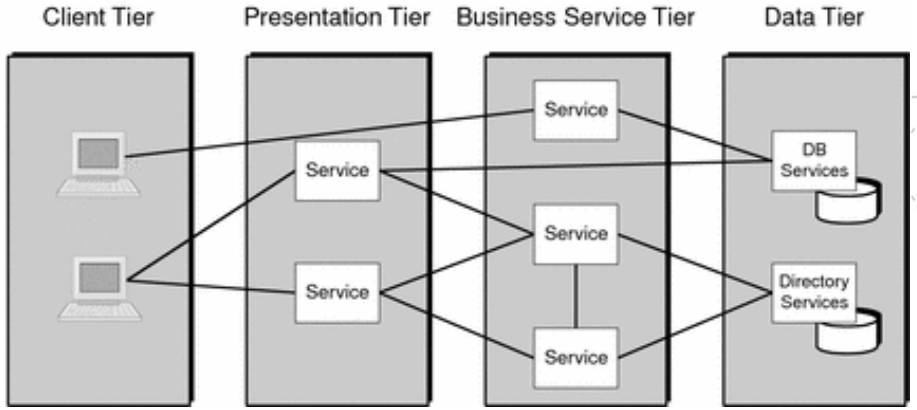
- Systems, often servers, are organized into logical tiers, each grouping with related components.
- Common tiers include:
  1. User Interface
  2. Application logic
  3. Data (e.g., database)

n-tier Architecture



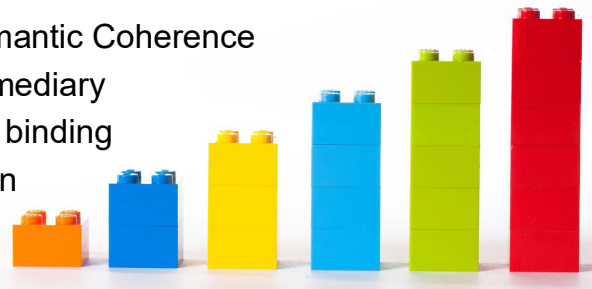
https://www.youtube.com/watch?v=KlHvRKSH4pk

Multi-Tier example



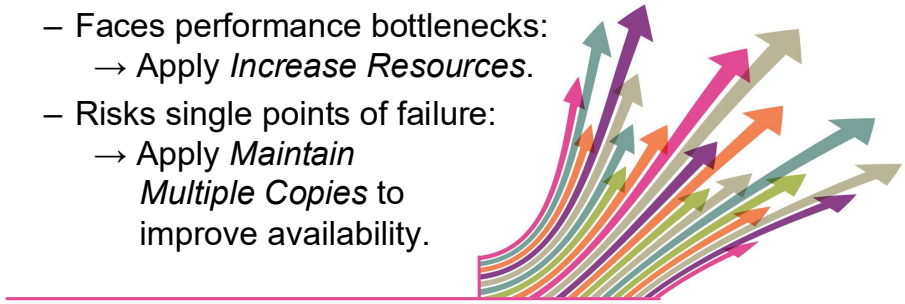
Relationship between Architectural Tactics and Patterns

- Tactics are the building blocks of patterns; if a pattern is a molecule, a tactic is its atom.
- For example, the Model-View Control pattern incorporates these tactics:
  - Increase Semantic Coherence
  - Use an intermediary
  - Use run time binding
  - Encapsulation



Tactics boost patterns

- Architectural patterns address specific problems but may have weaknesses in other areas.
- For example, the **Broker Pattern**:
  - Faces performance bottlenecks:
    → Apply *Increase Resources*.
  - Risks single points of failure:
    → Apply *Maintain Multiple Copies* to improve availability.



NTNU
Innovation and Creativity

NTNU | Norwegian University of Science and Technology

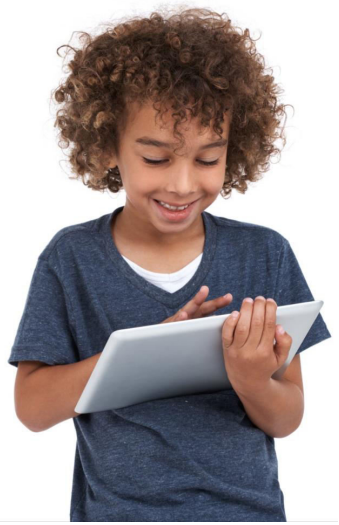
TDT4240 Software Architecture

Chapter 20: Designing an Architecture

Professor Alf Inge Wang

Learning Outcome

- Knowledge on Attribute-Driven Design
- Know the Attribute-Driven Design process and steps
- Know how to use the Attribute-Driven Design method



If .....

$$\lim_{x \to 8} \frac{1}{x-8} =$$

Equals .....

$$\lim_{x \to 8} \frac{1}{x-8} = \infty$$

What is .....

$$\lim_{x \to 5} \frac{1}{x-5} =$$

Solution ☺

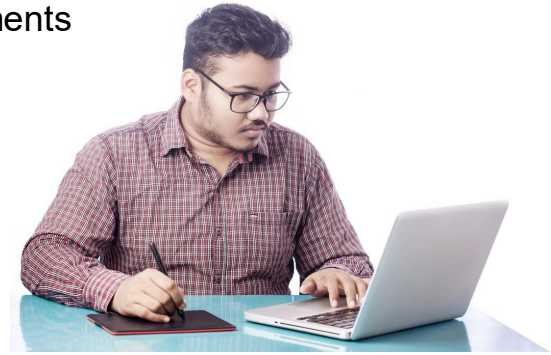
$$\lim_{x \to 5} \frac{1}{x-5} = 5$$

Why Attribute-Driven Design (ADD)?

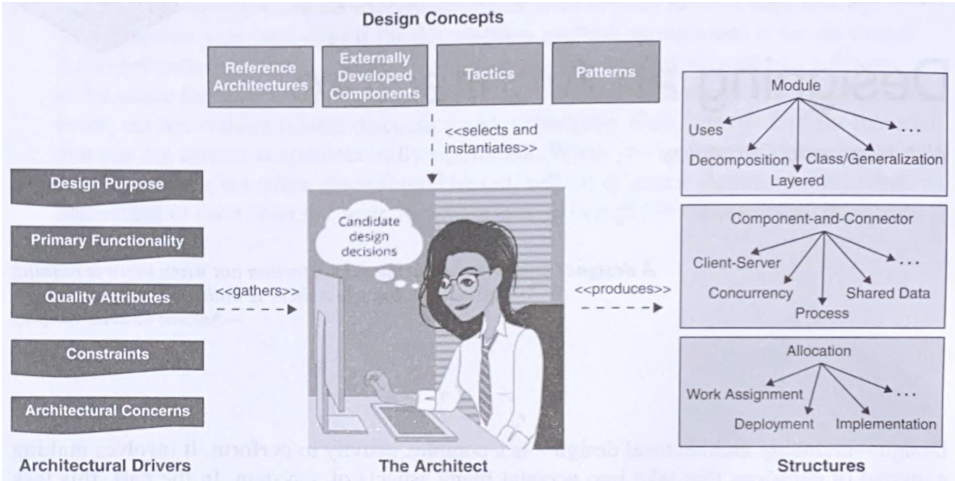
- "A systematic method that provides guidance in performing the complex design activity, so it can be learned and capably performed by *mere mortals*."
- Allows architecture to be designed systematically, repeatably, and cost-effectively.

Attribute-Driven Design

- Making decisions using available materials and skills, to meet requirements and constraints.
- **Turn decisions about architectural drivers into structures.**
- Architectural driver: A key factor shaping the architecture.

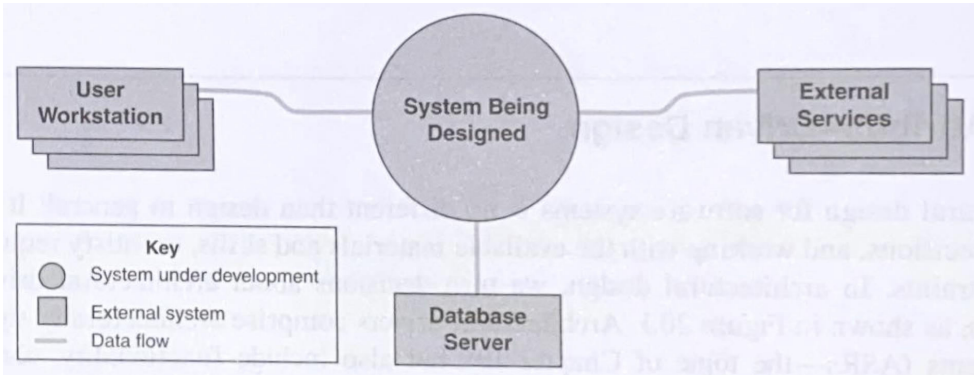


Overview of the architecture design activity

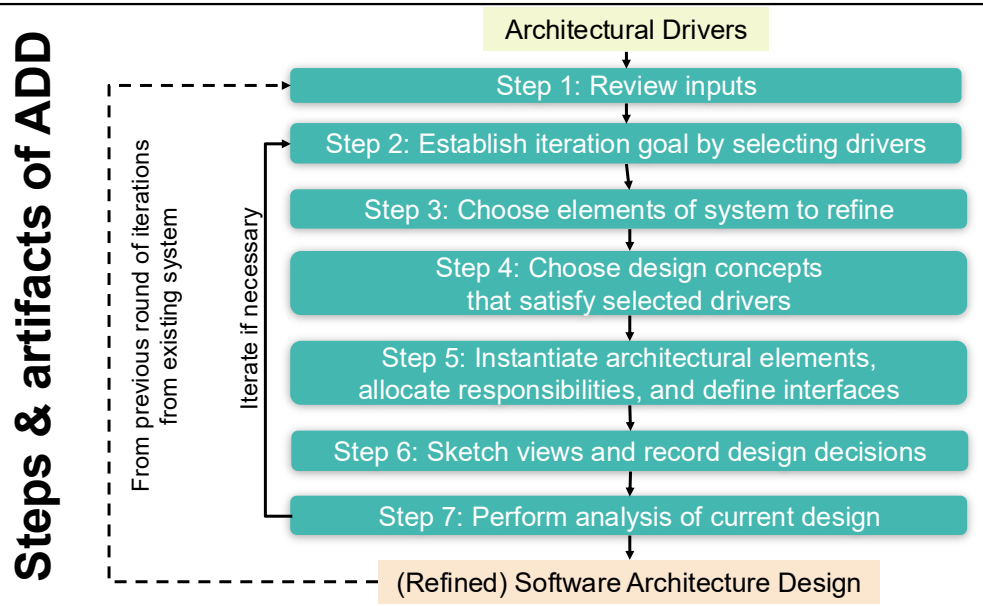


Before starting architectural design, the scope of the system must be established

– Example of a system context diagram –



Steps & artifacts of ADD

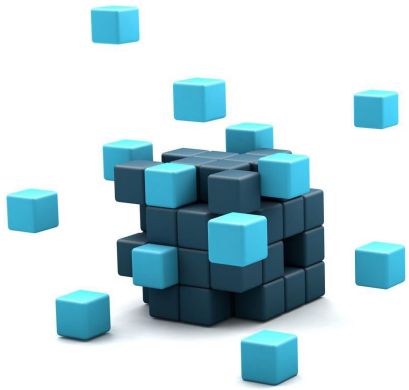


Step 1: Review Inputs

- Before starting a design round, ensure that the architectural drivers are available and correct.
- Architectural drivers:
  - The purpose of the design round
  - The primary functional requirements
  - The primary quality attribute (QA) scenarios
  - Any constraints
  - Any concerns

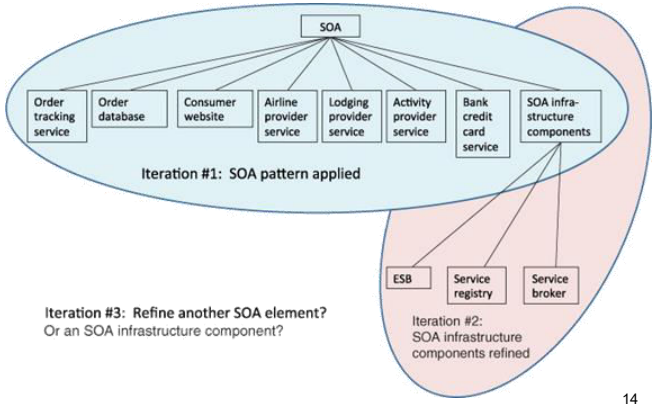
Step 2: Establish Iteration Goal by Selecting Drivers

- Each design iteration targets a specific goal:
  - Focuses on satisfying a subset of drivers.
  - Example: Design structures to support a performance scenario or use case.



Step 3: Choose One or More Elements of the System to Refine

- Approaches:
  - Top-down
  - Bottom-up
  - Improve previously identified elements



Step 4: Choose One or More Design Concepts That Satisfy the Selected Drivers

- Probably the most difficult decision in the design process
- Design concepts include:
  - Architectural tactics
  - Reference architectures
  - Design and architectural patterns
  - External developed components

Step 4: Choose Design Concepts That Satisfy the Selected Drivers

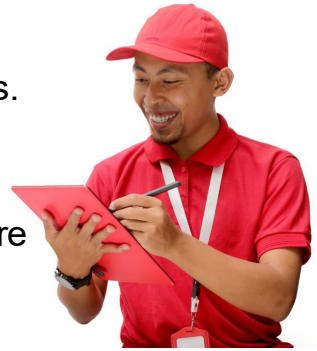
- Identification of Design Concepts (list of candidates):
  - Utilize existing best (documented) practices.
  - Utilize your own knowledge and experience.
  - Utilize the knowledge and experience of others.
- Selection of Design Concepts
  - Select alternative most appropriate to solve problem at hand.
- Creation of Prototypes
  - Test new technology, drivers add risk, lack of info, configuration options, unclear about integration, selection of external COTS.

Step 5: Instantiate Architectural Elements, Allocate Responsibilities and Define Interfaces

- **Instantiate architectural elements**:
  - How will elements specifically be used, e.g., how many layers.
- **Allocate Responsibilities**:
  - Assign functionality to elements, such as presentation, business, and data layers.
- **Define Interfaces**:
  - Specify relationships and information flow between elements.

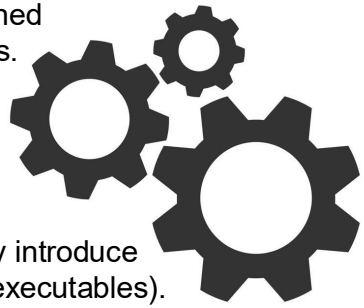
Step 5: Instantiate Architectural Elements, Allocate Responsibilities and Define Interfaces

- Producing structures:
  - **Module:** Development-time structures.
  - **Component & Connector:** Run-time structures.
  - **Allocation:** Software and non-software structures at development or run-time (e.g., file systems, hardware, teams).



Step 5: Instantiate Architectural Elements, Allocate Responsibilities and Define Interfaces

- Instantiating Elements
  - **Reference architectures:** Modify predefined structures by adding or removing elements.
  - **Patterns:** Tailor generic structures to fit specific problem needs.
  - **Tactics:** Adapt or reuse existing design concepts to implement a tactic.
  - **Externally developed components:** May introduce new elements (e.g., libraries vs. external executables).

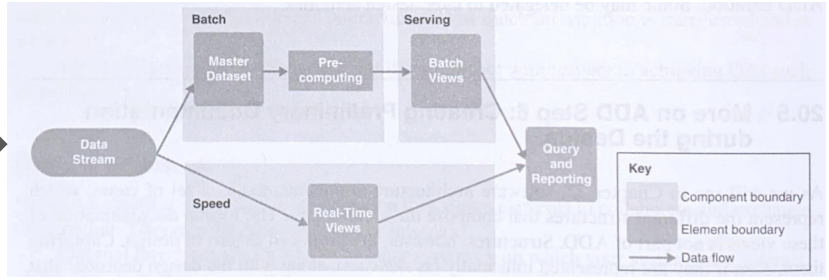


Step 6: Sketch Views and Record Design Decisions

- **Record Structures**:
  - Document views taking whiteboard photos or use design tools.
- **Record Design Decisions**:
  - Capture rationale, including chosen elements, relationships, and reasoning.

Step 6: Sketch Views and Record Design Decisions

Example preliminary documentation

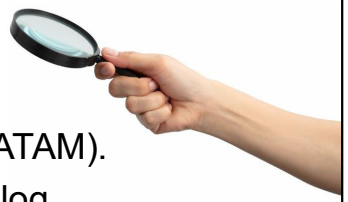


Elements and Responsibilities

Element	Responsibility
Data Stream	This element collects data from all data sources in real time, and dispatches it to both the Batch Component and the Speed Component for processing.
Batch	This is responsible for storing raw data and pre-computing the Batch Views to be stored in the Serving Component.
...	...

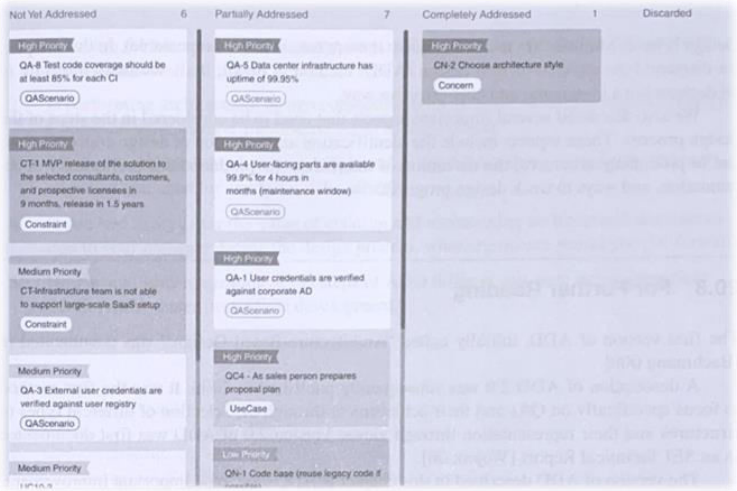
Step 7: Perform Analysis of Current Design

- **Evaluate Design:** Assess if it meets goals and requirements (quality, functionality, etc.).
- **Review Process:** Best done externally, though architects can review.
- **Methods:** Use techniques like Architecture Trade-off Analysis Method (ATAM).
- **Tracking:** Maintain an architectural backlog to monitor progress.



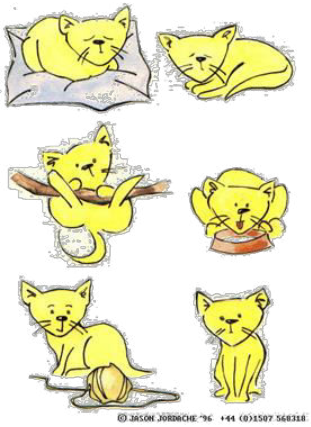
Step 7: Perform Analysis of Current Design

A Kaban board used to track design process



Forming the Team Structure

- Team structure mirrors the module decomposition:
  - Information hiding
  - Modules can be viewed as mini-domains.
  - Well-defined interfaces/protocols



Creating a Skeletal System

- **Build Core System:** Start once architecture and teams are set.
- **Implement Core Functions:** Focus on execution and component interaction first.
- **Add Features Gradually:** Expand functionality incrementally.
- **Test Regularly:** Perform periodic integration testing.

Summary

- Designing architecture involves:
  - Identifying architectural drivers
  - Selecting design concepts to meet drivers
  - Applying concepts to create structures across multiple views
  - Analyzing the structures
  - Iterating until requirements are met

NTNU | Norwegian University of Science and Technology

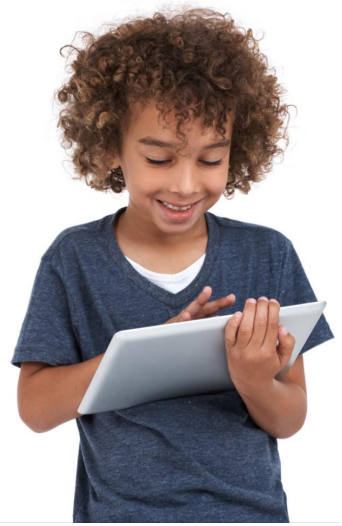
TDT4240 Software Architecture

Chapter 22:
Documenting an Architecture &
4+1 View Model (paper)

Professor Alf Inge Wang

Learning Outcome

- Knowledge and understanding of the 4+1 view model
- Know the how to use the 4+1 view model
- Understand the relationships between the views



Solve

$$\frac{1}{n} \sin x = ?$$

Solution

$$\frac{1}{n} \sin x = ?$$

$$\text{six} = 6$$

Some important definitions

- **Software Architecture: "**The set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both".
- **Architectural Description:** "A collection of products to document an architecture".
- **Architecture View:** "A representation of a whole system from the perspective of a related set of concerns".
  - Focus on some specific parts of the system and not on other parts.

Architecture Description – ISO/IEC/IEEE 42010

- Prescribe the least common multiple on what should be included in a complete architectural description:
  - Architectural documentation: AD ID, version, overview information
  - Stakeholders and their concerns: Who are involved and their interests
  - Architectural Viewpoints: What *views* will be included, what *concerns* addressed, which *stakeholders* addressed, and *model kinds* (language, notations, conventions, modelling techniques, analytical methods etc.). Architectural views: The actual architectural design through multiple views.
  - Consistency among architectural views: Inconsistencies among the views
  - Architectural rationale: A motivation/reason for the design decisions.
- Does **not** prescribe **what notation** or other presentation format that should be applied.

Table of Contents

Example of Table of Contents from an Architecture description in the TDT4240 project according to the ISO / IEC / IEEE 42010 standard

Views as presented in the textbook

- Module views:
  - Documents modules (class, package, layer) and relations
  - Focus on how responsibilities are allocated to modules
  - Relations: "is-part-of", "depends-on", "is-a"
  - Notation: Class and Component Diagram (UML), Layered
- Component-and-connector views:
  - Document runtime components and connections
  - Focus on what the system looks like when it is running
  - Relations: "attachment"
  - Notation (UML): Component, Collaboration, State, Sequence

Views as presented in the textbook

- Allocation views:
  - Documents software elements in one or more non-software environment (hardware, file structure, team, organization etc.)
  - Relations: “allocated-to”
  - Notation: UML Deployment and specialized ones
- Critique of architectural documentation in textbook:
  - The approach is unclear and unfocused
  - Therefore the 4+1 view model is used in this course (paper)

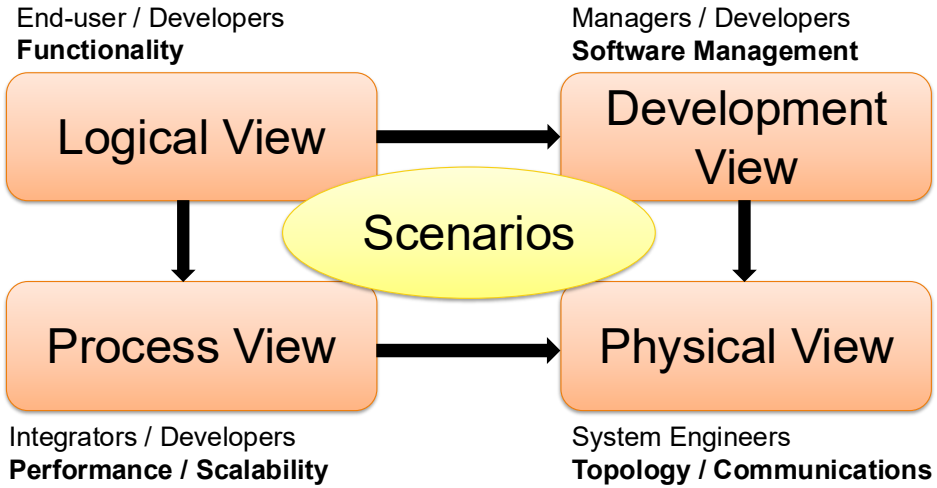
The 4+1 View Model (architectural description)

- Describe software architecture using *five related views*.
- Views represents the position, where the system stakeholders see the architecture.
- Views defines the concerns covered.

The 4+1 Views

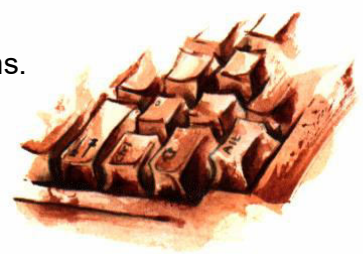
- **Logical**: Assigns functionality to modules (static model) and defines their relationships.
- **Process**: Captures runtime aspects.
- **Physical**: Maps software to hardware.
- **Development**: Maps software to development environment.
- **Scenario**: Highlights key use cases for the system.

4+1 View model



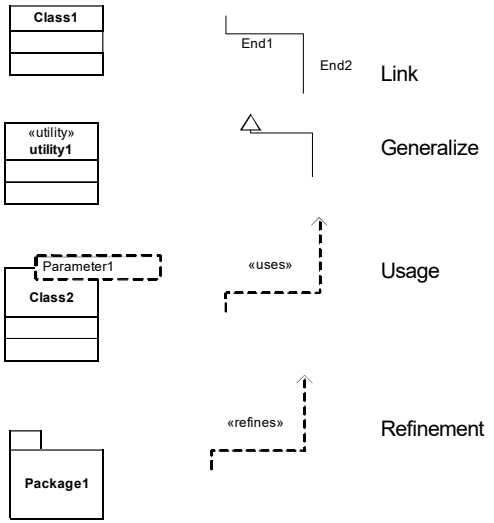
Logical view: End user / Developer

- Focus on Modifiability, Integrability, Testability
- System decomposed into a set of key abstractions
- Corresponds to *Module view* in the textbook
- Object-oriented approach:
  - Model with class or component diagrams.
  - Set of classes (packages), Association, Usage, Composition, Inheritance.
- Data-driven approach:
  - Model with ER-diagrams.

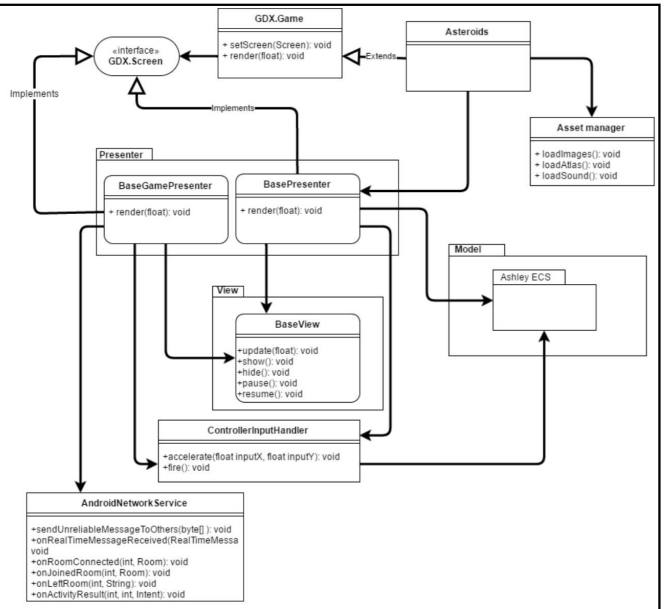


Notation for Logical view

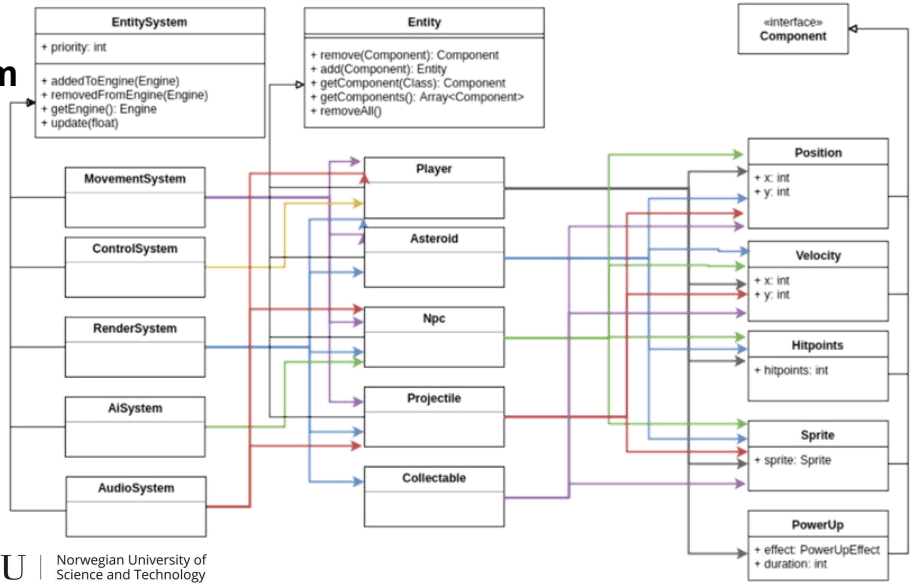
- UML Class Diagram
- Elements:
  - Package, Class (standard, utility, parameterized, etc.)
- Relations:
  - Associations (link), Generalization, Usage, Refinement etc.



Logical view from Game Project

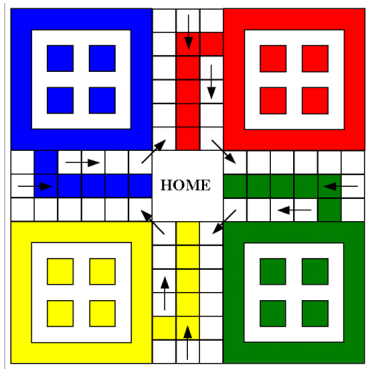


Logical view from Game Project

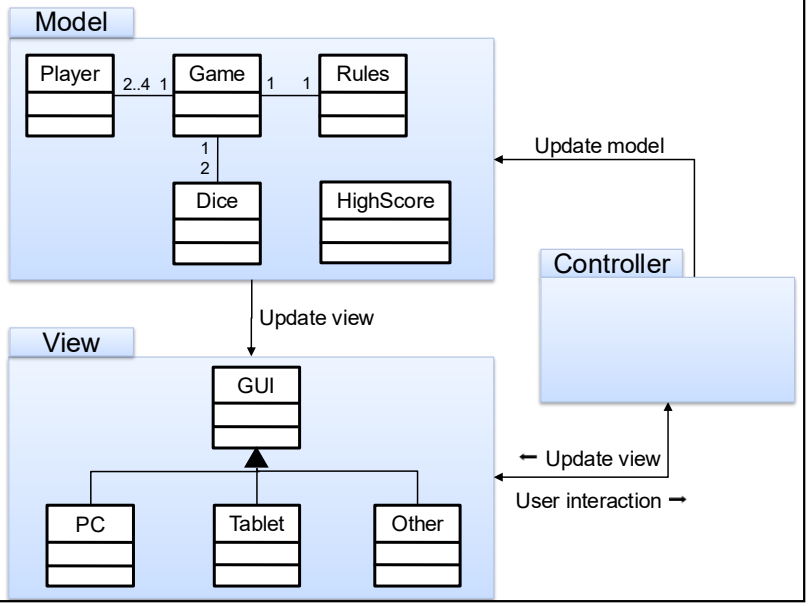


Exercise: Create Logic view [10 min]

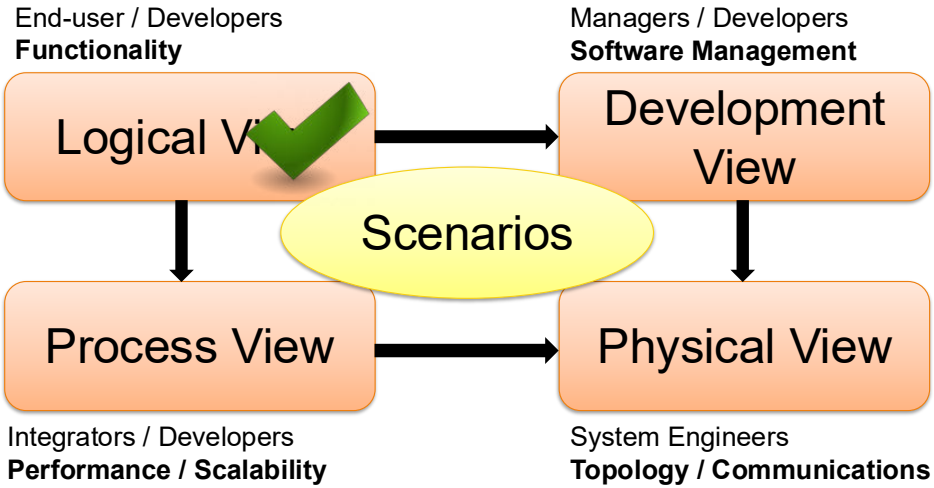
- Create a logic architectural view (class diagram) of a web-based version of the LUDO game:
  - 1-4 players (online)
  - Each player has four pieces of same color, One dice
  - High score list for players with fewest moves
  - Clients on PC, tablets and possibly smart phones
  - Use known tactics, patterns



Possible solution: Logic view Ludo



4+1 View model

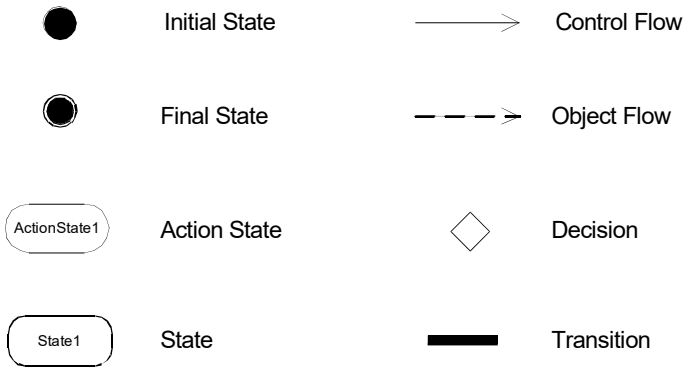


Process View: Integrators / Developers

- Focus on Performance, Availability, Safety, Energy efficiency
- Corresponds to Component & Connector views in the book
- Addresses issues like:
  - Concurrency and distribution
  - System integrity / fault tolerance
  - Execution threads
  - System states
  - System interaction when running the system

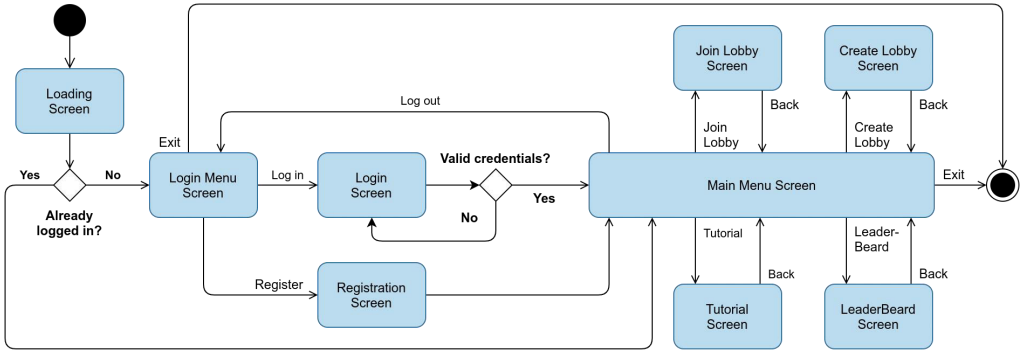
Notation for Processing View

- Activity diagram (UML)
- Can also use following UML diagrams: State, sequence, collaboration, component



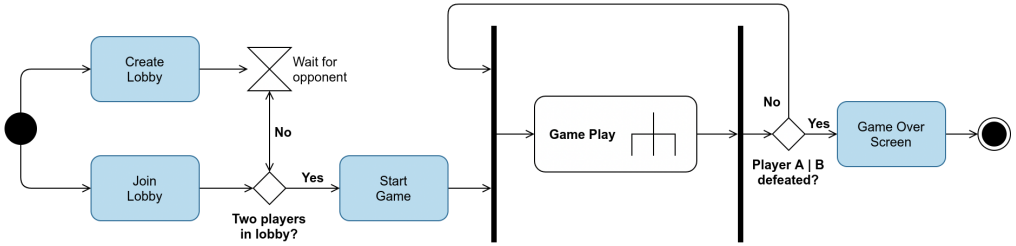
Process view from Game Project

Activity Diagram



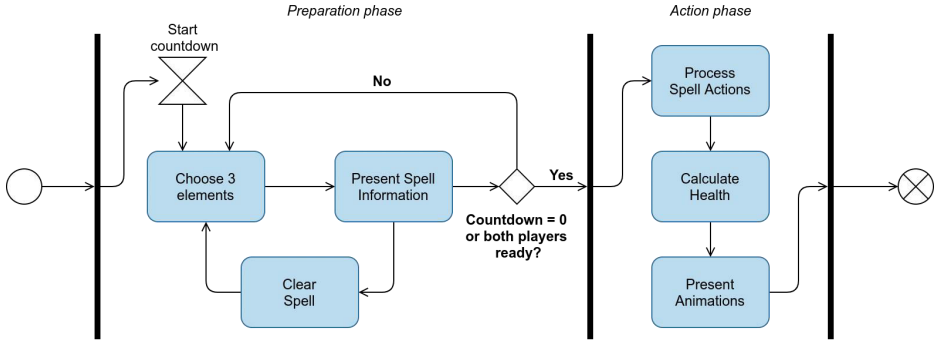
Process view from Game Project

Activity Diagram



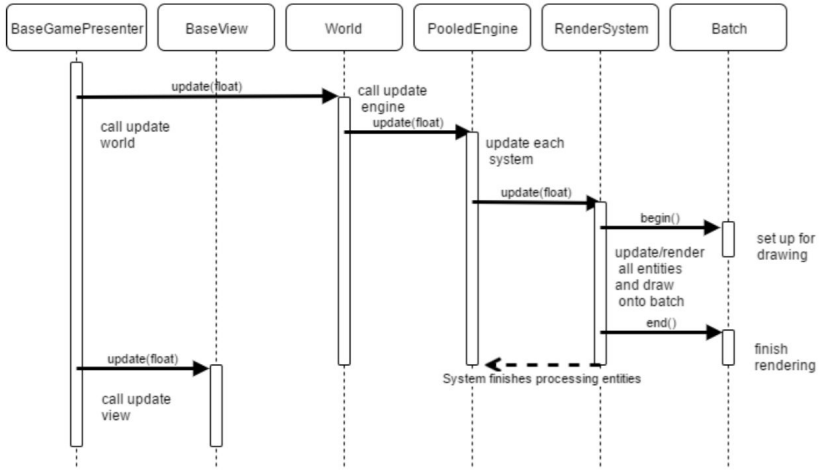
Process view from Game Project

Activity Diagram



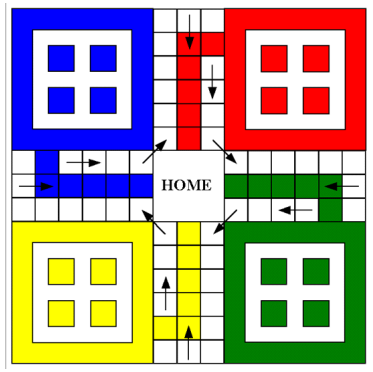
Process view from Game Project

Sequence Diagram

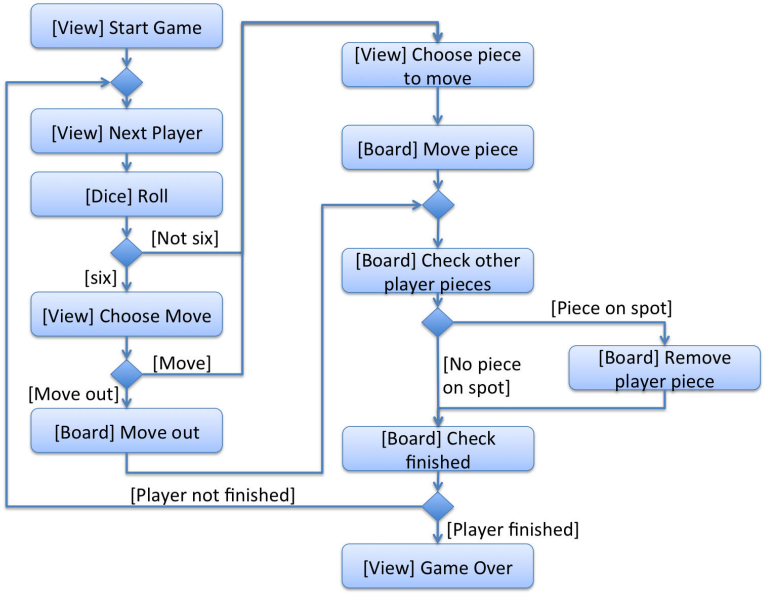


Exercise: Create Process view [5 min]

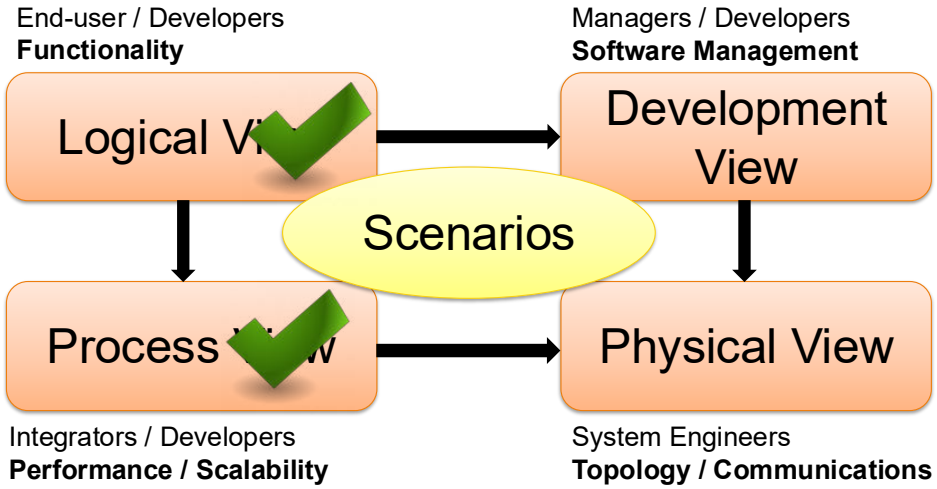
- Create a process architectural view of a web-based version of the LUDO game:
  - Focus on game states
  - 1-4 players (online)
  - Each player has four pieces of same color, One dice
  - High score list for players with fewest moves
  - Clients on PC, tablets and possibly smart phones



Possible solution: Process view Ludo

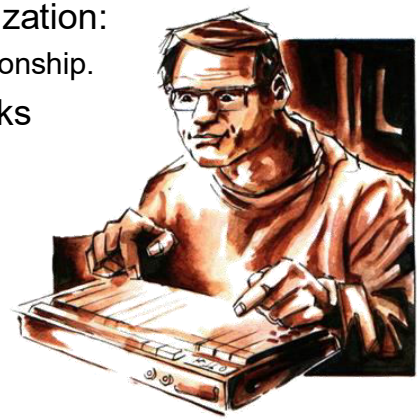


4+1 View model



Development View: Manager/Developer

- Focus on software module organization:
  - Subsystems with export/import relationship.
- Software packages in small chunks
  - Subsystems can be developed by one or few developers.
  - Subsystems can be organized in hierarchy of layers.
  - Each layer provides well-defined interface to layers above.
  - Can also organize in other ways.

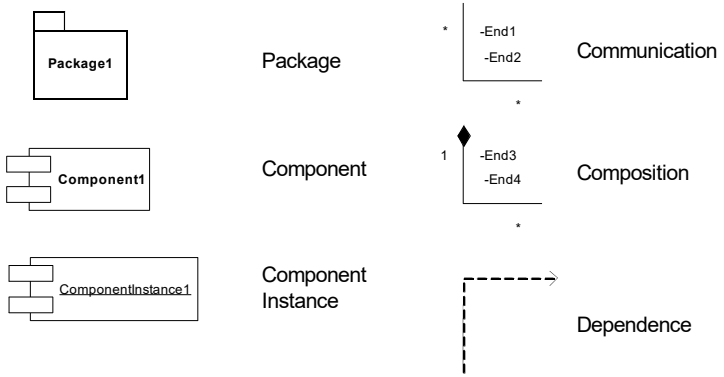


Development View: Manager/Developer

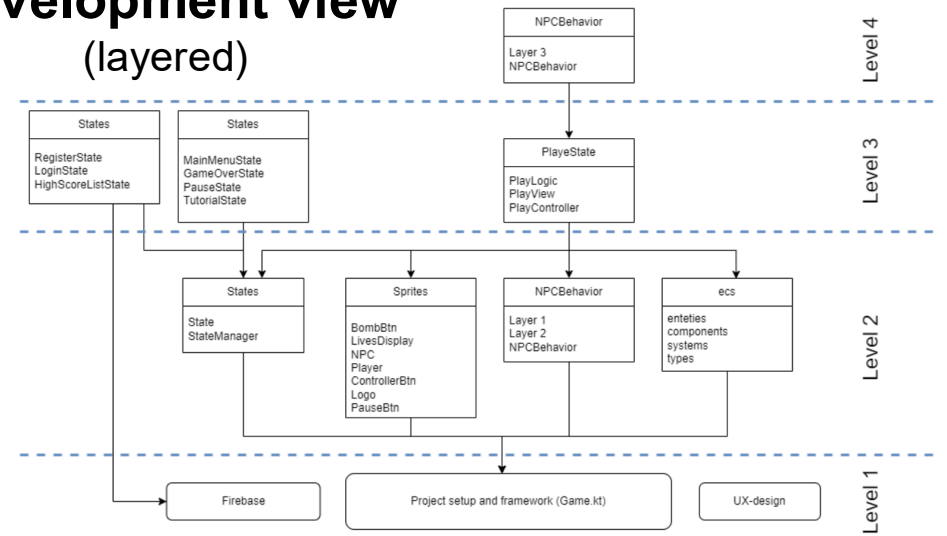
- Rules for Development view:
  - Partitioning, grouping, visibility.
- Development view should ease development:
  - Software management, reuse, commonalities, constraints imposed by toolset or programming language.
  - Foundation for organization, cost planning, monitoring etc.

Notation for Development view (UML)

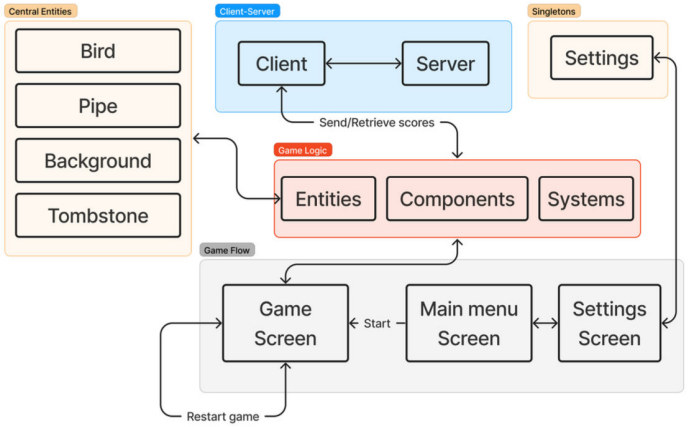
Package diagram (UML)



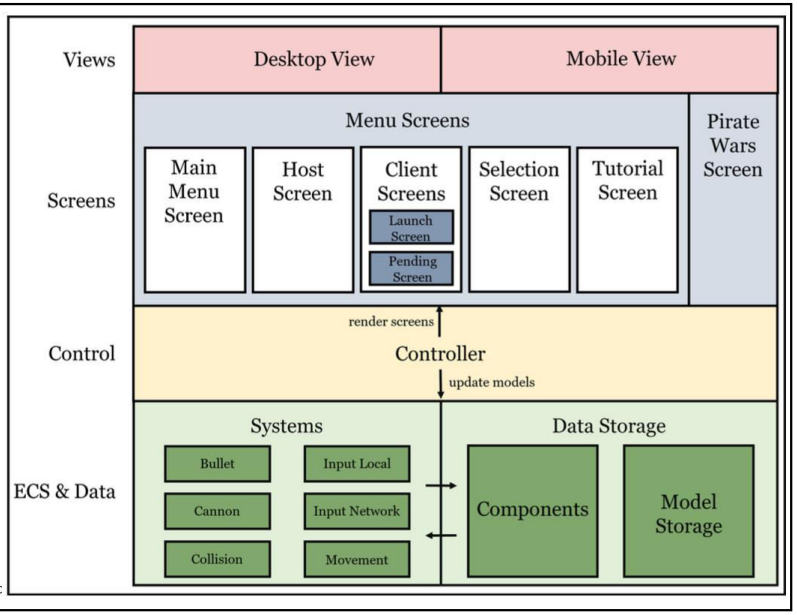
Development view (layered)



Development view (packages)



High-level layered development view



Project timeline and modules

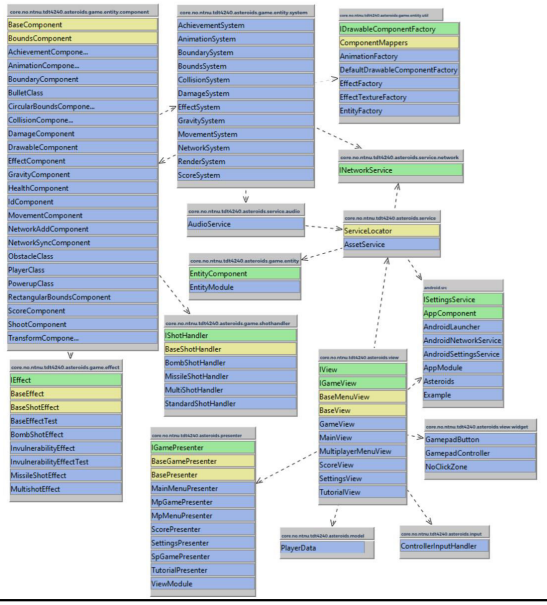
			Views		Screens						
					Main Menu	Hosts Screens	Client Screens		Selection Screen	Tutorial Screen	Game Screen
Week #	Start Date	End Date	Desktop View	Mobile View		Host Launch Screen	Client Launch Screen	Waiting Screen			
Week 10	07.03.2023	13.03.2023	H, LB: Implement desktop view		H, LB: Add black screen						H, LB, SO: Add initial screen
Week 11	14.03.2023	20.03.2023			F: Add main menu buttons + functionality	F: Add screen + buttons	F: Add screen + buttons + textfield				SO: Add map designs
Week 12	21.03.2023	27.03.2023		SO: Implement mobile view + joystick							SO: Add format scaling to screen
Week 13	28.03.2023	03.04.2023		SO: Implement firing button		S: Implement IP display	H: Implement IP input				SI: Add final ship/ bullet skins + multi color support
Week 14	04.04.2023	10.04.2023	LB: Implement independent cannon control	LB: Implement independent cannon control	SO: Improve menu design						
Week 15	11.04.2023	17.04.2023		L: Fix input recognition				LB: Add screen	SO: Add screen + buttons	H: Add screen + buttons	
Week 16	18.04.2023	23.04.2023			SO: Customize design	SO: Customize design; J: Add player connection monitoring functionality	SO: Customize design; H, SO: Add invalid IP address error message with screen persistence	SO: Customize design, Add back button	SO, LB: Implement map choice; customize design	SO: Improve and customize design	SO: Customize design, Add end-game overlay message

Project timeline and modules (2)

			Control	ECS & Data							
				Systems						Data Storage	
Week #	Start Date	End Date	Controller	Input Local	Input Network	Movement	Collision	Cannon	Bullet	Components	Model Storage
Week 10	07.03.2023	13.03.2023	H, LB: Add basic game functionality	H, LB: Add button inputs			H, LB: Implement ship shore collision				L, SI, LB: Add entity/ component creation
Week 11	14.03.2023	20.03.2023		SI, LB: Add local multiplayer input		L, J, SI, LB: Refactor movement to system	SO: Implement map collisions			J, L, SI, LB: Create initial components	SO: Add random spawn functionality; SI, LB: Multi spawn
Week 12	21.03.2023	27.03.2023	F: Add screen switching functionality	SO: Add input of joystick	SI, H: Implement network ship control		SO, SI, L: Update collision + refactor	J, L: Add weapon firing functionality	J, L: Add riccochet + damage functionality	L, J: Add components for bullets	
Week 13	28.03.2023	03.04.2023					SO: Update system for different obstacles		J: Fix bullet riccochet	LB: Implement network update functionality	
Week 14	04.04.2023	10.04.2023		LB: Implement cannon position input	LB: Implement cannon position input						
Week 15	11.04.2023	17.04.2023					SO: Differ between obstacles for ships and bullets	LB: Add aim helper	J, L: Add bullet hits bullet functionality		
Week 16	18.04.2023	23.04.2023	SO: Add map switching functionality		H: Add max player limitation, J: Fix Role Designation				J, LB: Fix random bullet kills	J: Fix health component	

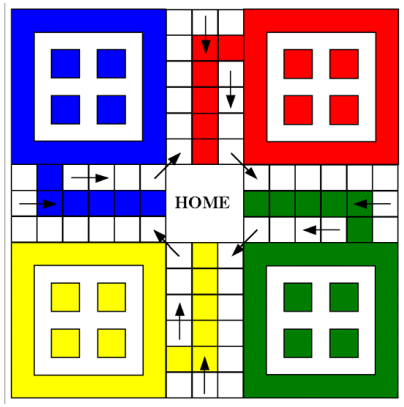
Development view from Game Project

Packages + interfaces

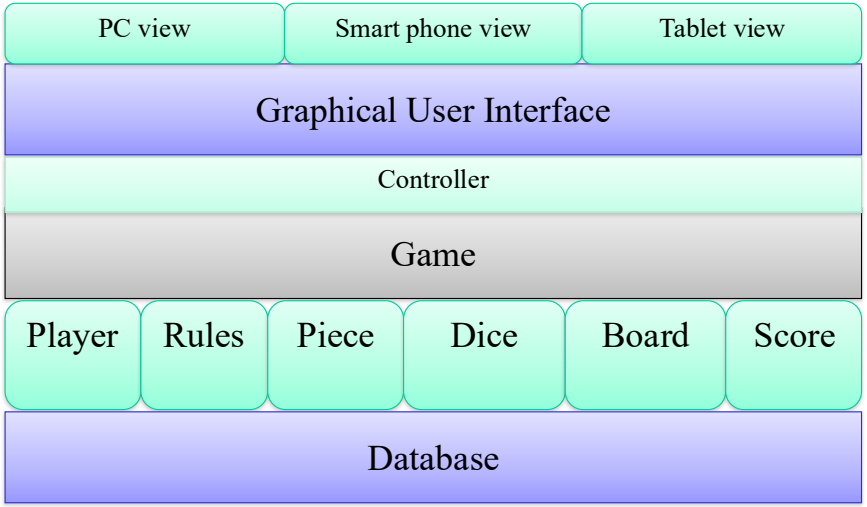


Exercise: Create Development view [5 min]

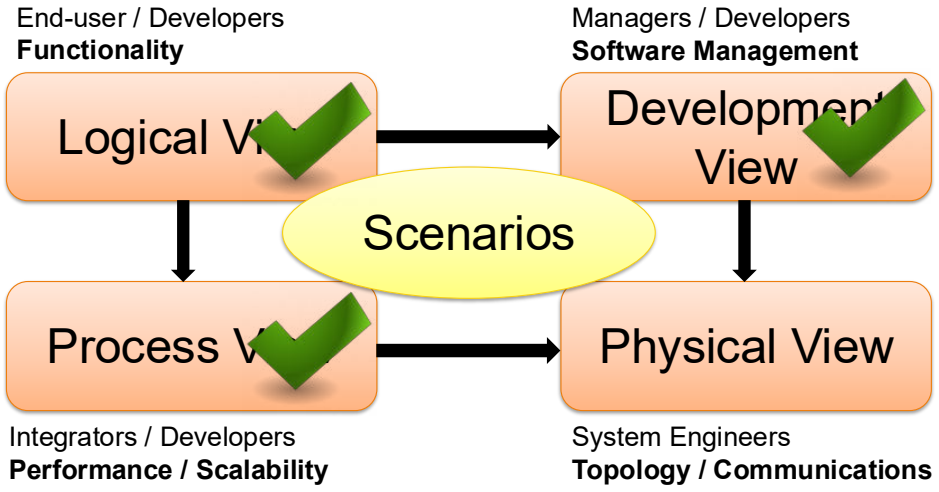
- Create a development architectural view of a web-based version of the LUDO game:
  - Focus on how the game can be development
  - 1-4 players (online)
  - Each player has four pieces of same color, One dice
  - High score list for players with fewest moves
  - Clients on PC, tablets and possibly smart phones



Solution: Development view (layered)

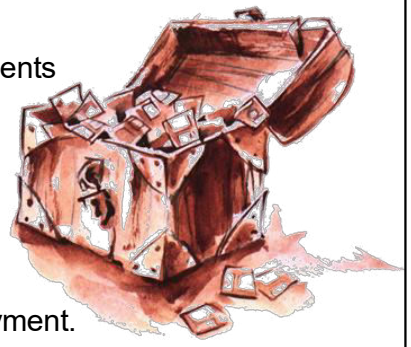


4+1 View model

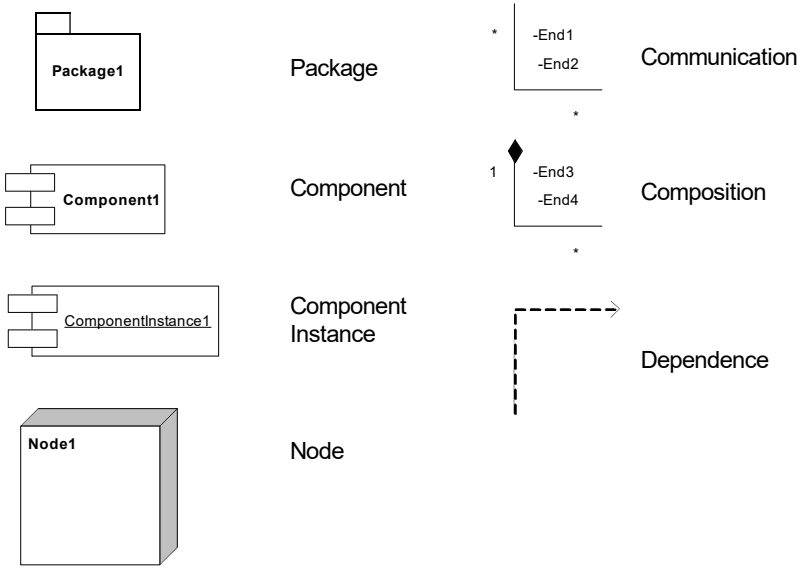


Physical view: System Engineers

- Focus on Deployability, Availability and Performance
- Software executes on a network of processing nodes (computers)
  - Networks, processes, tasks and components must be mapped onto hardware nodes.
- Common with different physical configurations:
  - Test, Development and Deployment phases, For various sites and users.
  - Architecture must support flexible deployment.

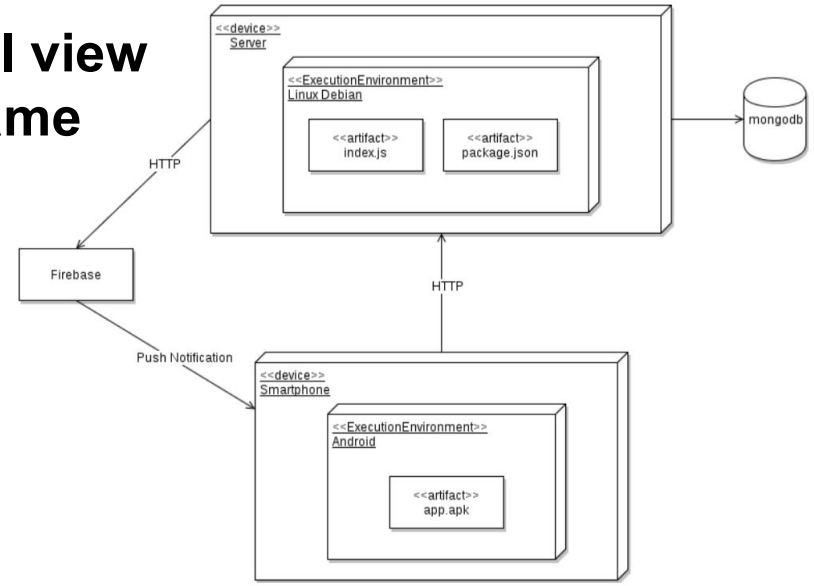


Notation for Physical View: Deployment diagram (UML)



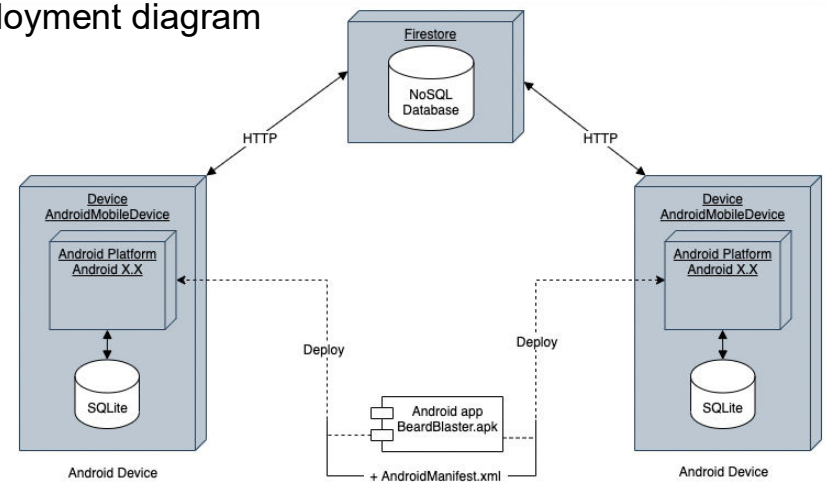
Physical view from Game project

Deployment diagram



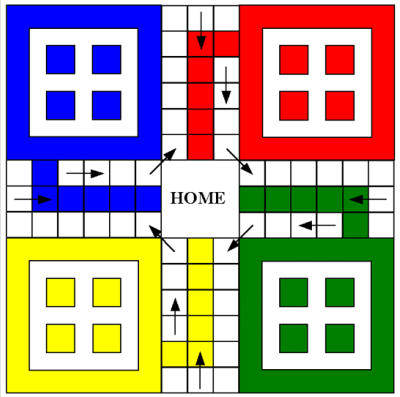
Physical view from Game Project

Deployment diagram

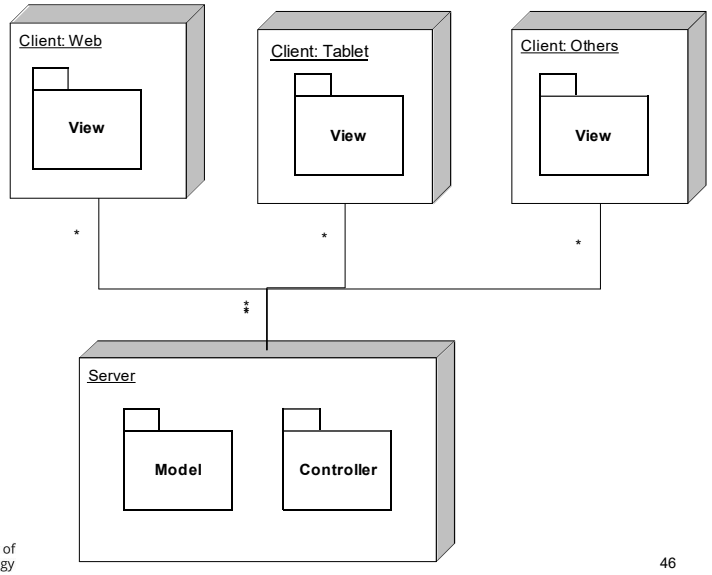


Exercise: Create Physical view [5 min]

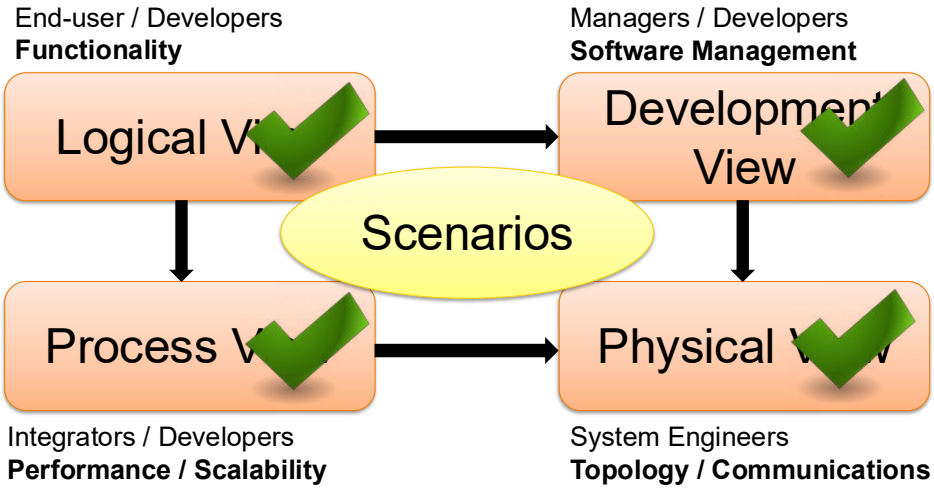
- Create a physical architectural view of a web-based version of the LUDO game:
  - Focus on mapping software onto hardware/network
  - 1-4 players (online)
  - Each player has four pieces of same color, One dice
  - High score list for players with fewest moves
  - Clients on PC, tablets and possibly smart phones



Solution: Physical view Ludo



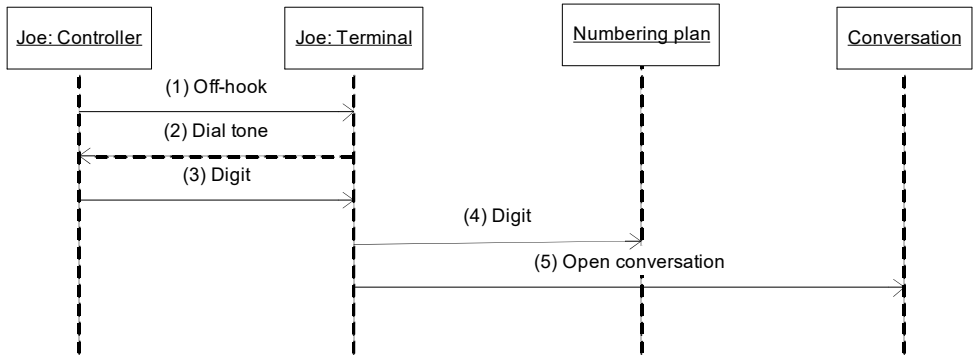
4+1 View model



Scenarios

- Four views work together using a small set of high-level scenarios.
- Scenarios abstract the most important requirements.
- Diagram: Use case or interaction.
- Scenario view is an add-on, with two main purposes:
  - Drives the discovery of architectural elements during design.
  - Validation when architecture is complete.

Example (UML): Scenario view of phone control system



Conclusion

- Quality attribute of non-trivial systems are *determined by its architecture*.
- Software architecture is a *vehicle for communication* among the stakeholders.
- Architecture should be described through *several views*.
- *Views represent the position* from where stakeholder see the architecture.