COMP20050 - Software Engineering Project II

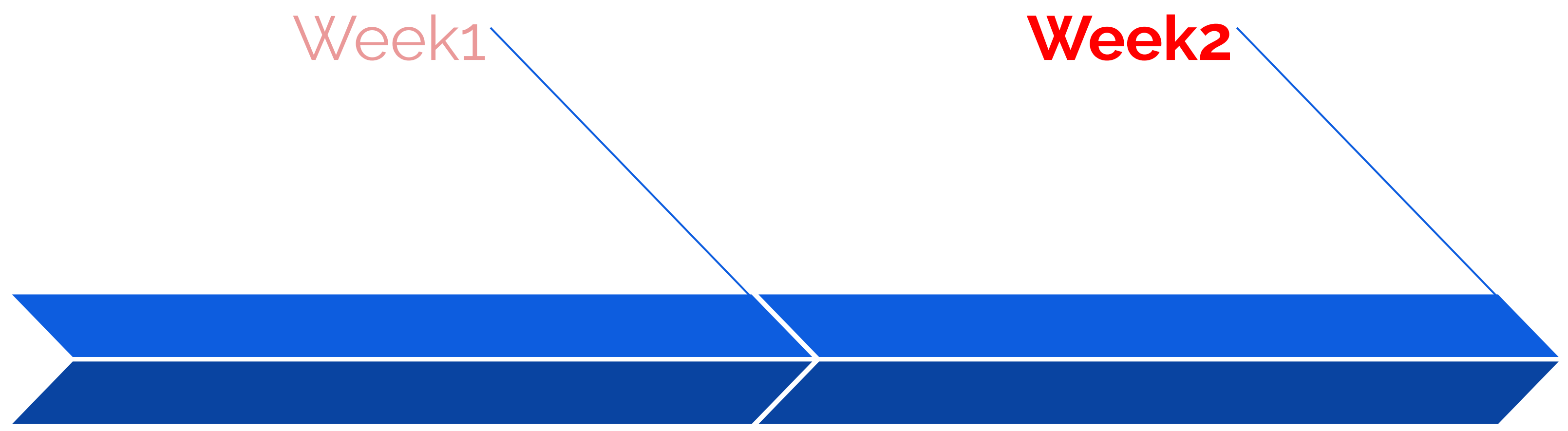
Software Architectural Design

Ravi Reddy Manumachu
ravi.manumachu@ucd.ie

UCD School of Computer Science.

Scoil na Ríomheolaíochta UCD.

COMP20050 - Weeks 1 & 2



Software Project Description

Module Introduction
Software Project Specification

Version Control
Software Design

Version Control
Software Architectural Design

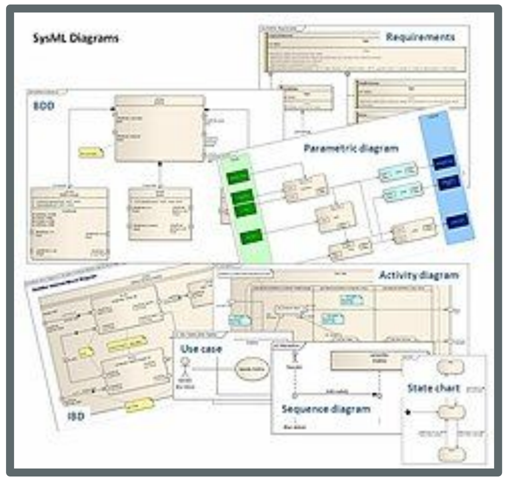
Outline (Learning Objectives)

- Understand how to **model** a software system through a contextual, interaction, structural, and behavioral perspectives.
- Become familiar with the basic diagram types in the **Unified Modelling Language**.
- Understand the **significance** of the **architectural design** of software.
- Become aware of the **architectural patterns**, which are highly reusable in system designs.

System Modelling

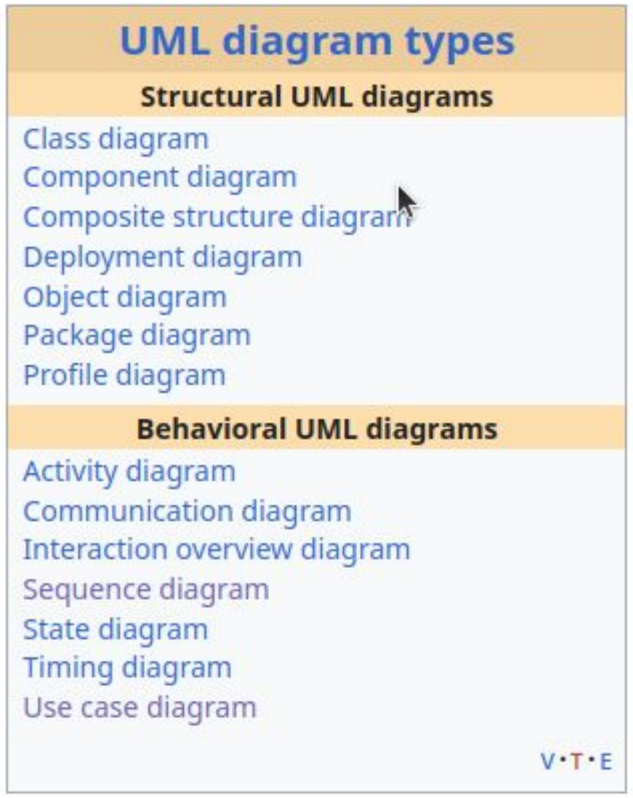
What is System Modelling?

- **System modeling** is the process of developing abstract models of a system using a graphical notation.
- Each **model** presents a different view or perspective of that system.
- **UML** is a standard modeling language.



UML (Unified Modified Language)

- Most UML users commonly employ five diagram types:
  - **Activity diagrams** that show the activities involved in a process.
  - **Use case diagrams** that show the interactions between a system and its environment.
  - **Sequence diagrams** that show interactions between actors and the system and between system components.
  - **Class diagrams** that show the object classes in the system and the associations between these classes.
  - **State diagrams** that show how the system reacts to internal and external events.



UML diagram types

Structural UML diagrams

Class diagram
Component diagram
Composite structure diagram
Deployment diagram
Object diagram
Package diagram
Profile diagram

Behavioral UML diagrams

Activity diagram
Communication diagram
Interaction overview diagram
Sequence diagram
State diagram
Timing diagram
Use case diagram

V·T·E

Types of Models

- Different models to represent the system from different perspective.
  - An external perspective, where you model the **context** or environment of the system.
  - An **interaction** perspective where you model the interactions between a system and its environment or between the components of a system.
  - A **structural** perspective, where you model the organization of a system or the structure of the data that is processed by the system.
  - A **behavioral** perspective, where you model the dynamic behavior of the system and how it responds to events.

NOTE: The system we are referring to is the HexOust software system that you will build.

Context Models

- Context models show how a system that is being modeled is positioned in an environment with other systems and processes.
- They help define the boundaries of the system to be developed.
- Context models normally show that the environment includes several other automated systems.
- *Since HexOust Software System (HOS) is a standalone system, there will not be any interesting context models.*

Interaction Models

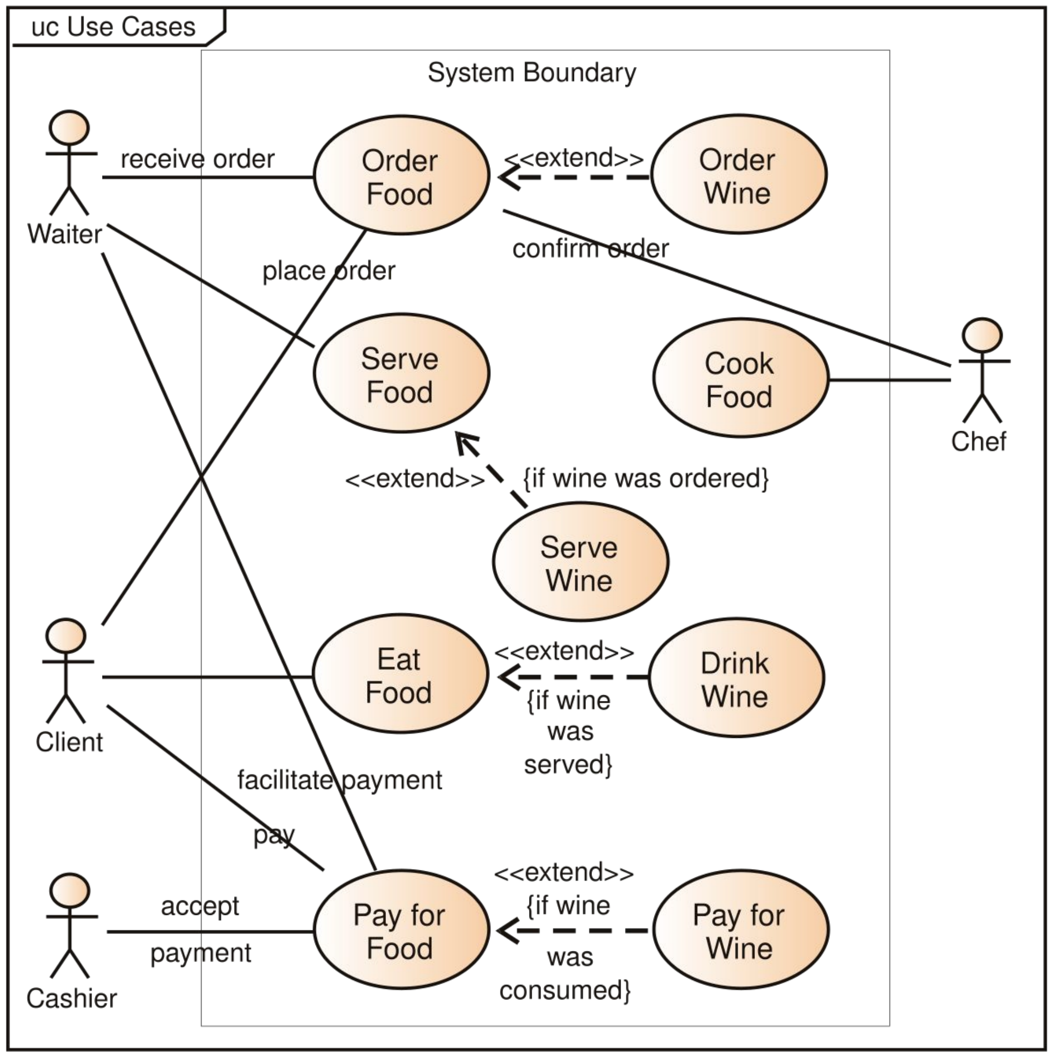
- User interaction with the system and interaction between system components can be modelled using two approaches:

- **Use Case Modelling:** Used to model interactions between a system and users.

- **Sequence diagrams:** Model interactions between system components.

Interaction Models: Use Case Modelling

- A **use case** can be taken as a simple scenario that describes what a user expects from a system.



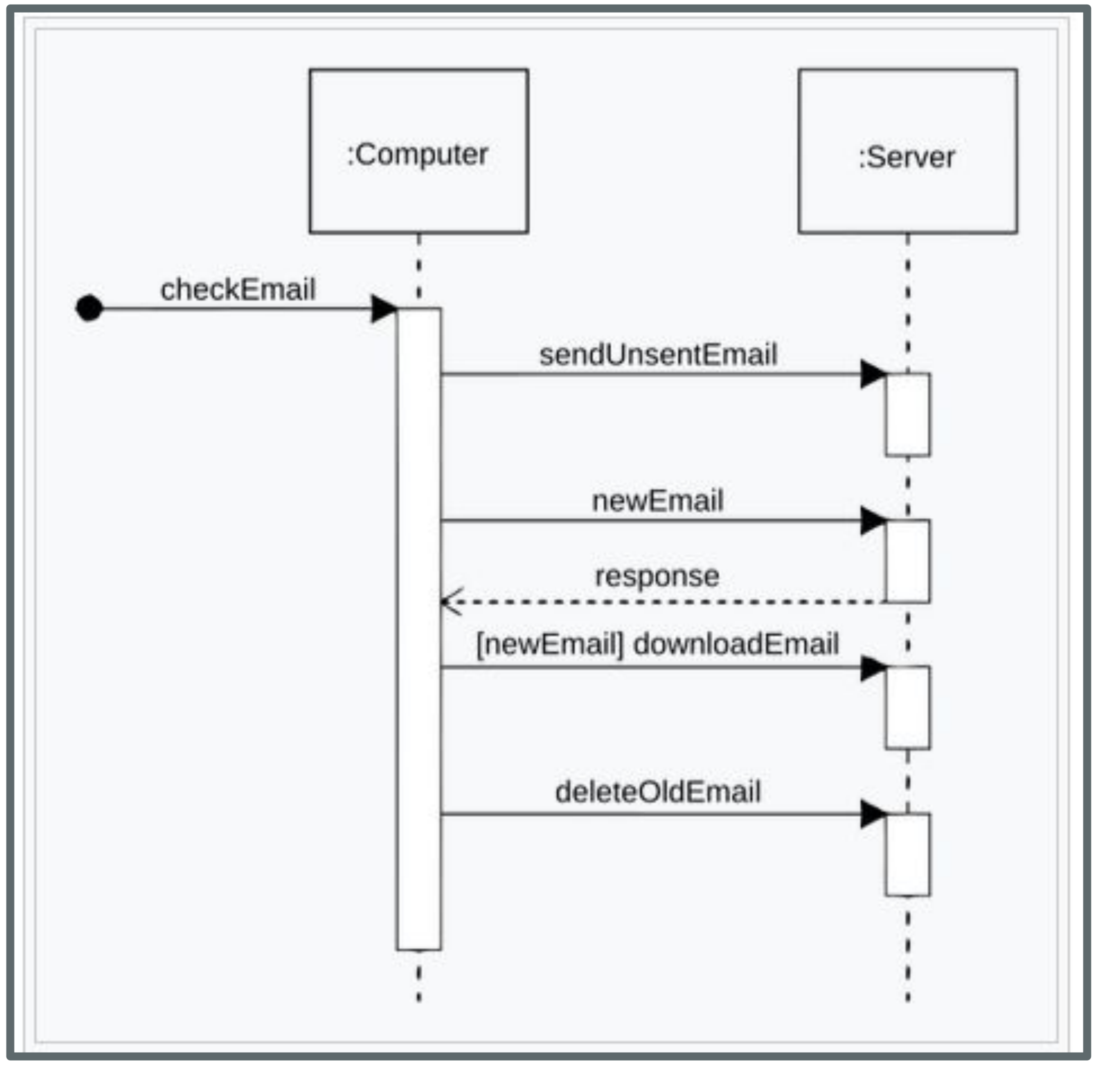
A UML use case diagram for the interaction of a client (the actor) within a restaurant (the system).

Interaction Models: Sequence Diagrams

- **Sequence diagrams** are primarily used to model:
    - The interactions between the actors and the objects in a system, and
    - The interactions between the objects.

- A **sequence diagram** shows the sequence of interactions that take place during a particular use case.

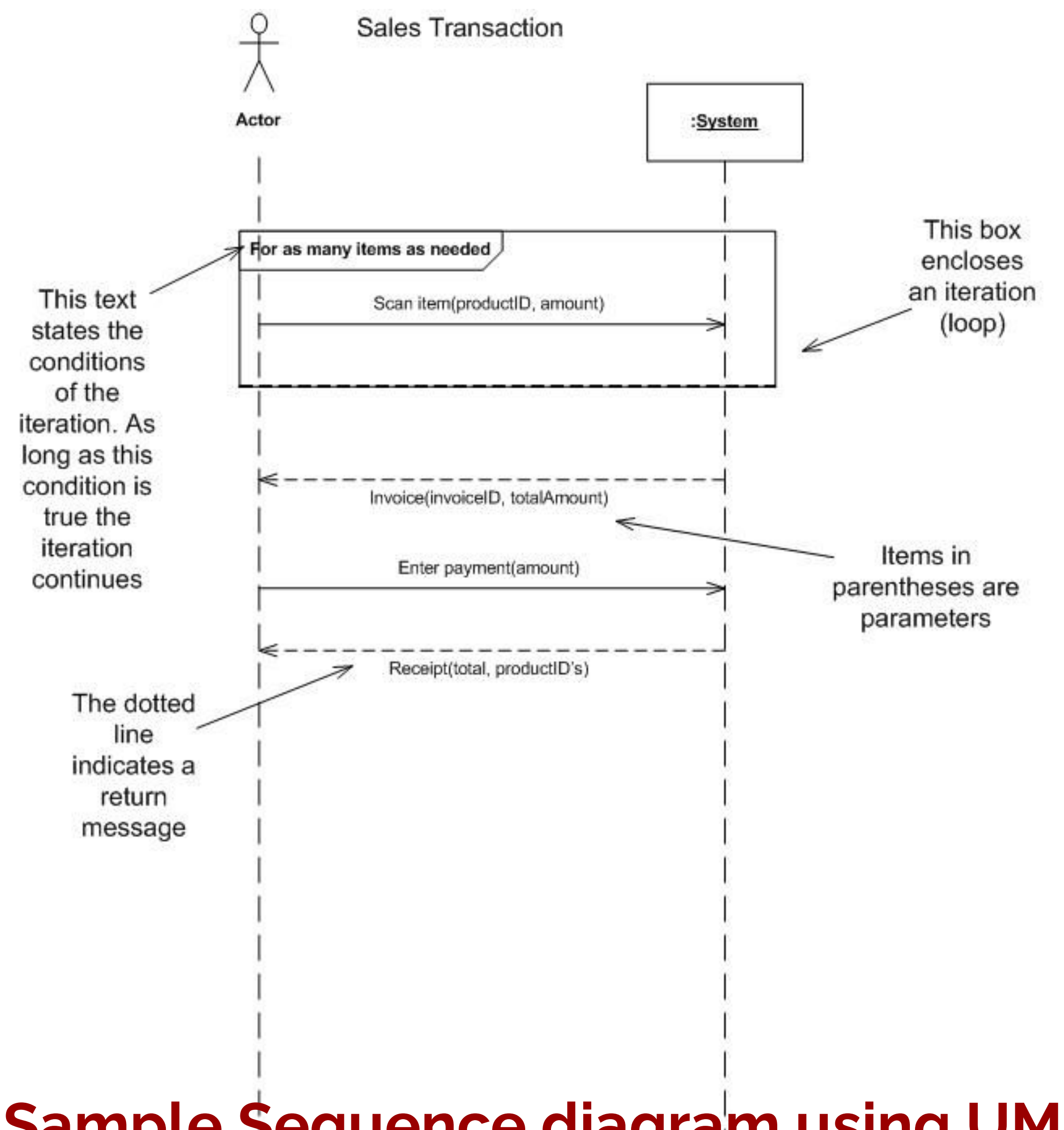
Sequence Diagrams in UML (1/2)



Sample Sequence diagram using UML

- The **objects** and **actors** involved are listed along the top of the diagram, with a dotted line drawn vertically from them.
- **Interactions** between objects are indicated by **annotated arrows**.
- The **rectangle** on the dotted lines indicates the **lifeline** of the object concerned.

Sequence Diagrams in UML (2/2)

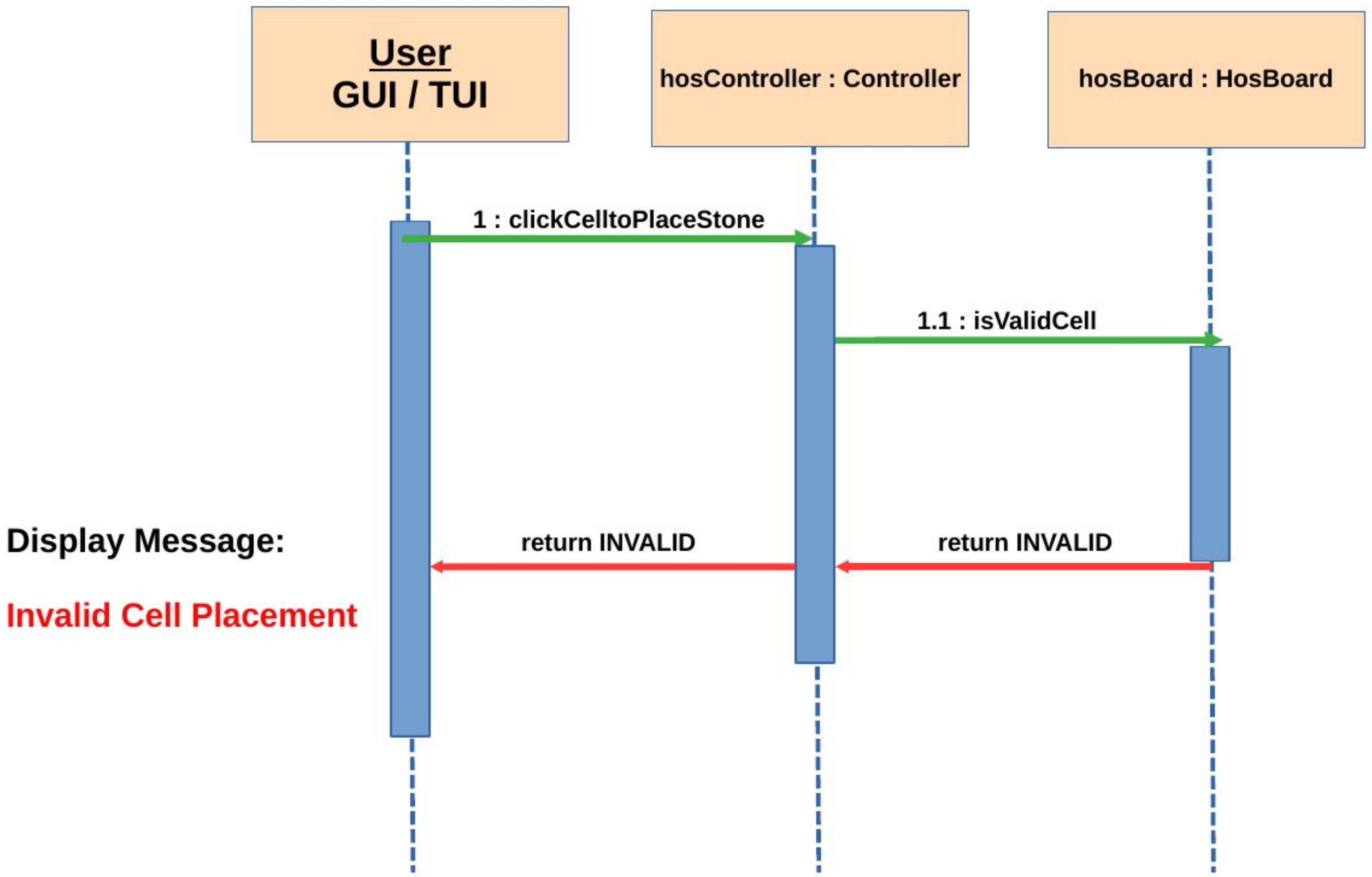


Sales Transaction
Actor
:System
For as many items as needed
Scan item(productID, amount)
Invoice(invoiceID, totalAmount)
Enter payment(amount)
Receipt(total, productID's)
This text states the conditions of the iteration. As long as this condition is true the iteration continues
This box encloses an iteration (loop)
Items in parentheses are parameters
The dotted line indicates a return message

Sample Sequence diagram using UML

- The **sequence** of interactions from is read from **top to bottom**.
- The **annotations** on the arrows indicate the calls to the objects, their parameters, and the return values.

A Sequence Diagram for HexOust SR4.1



A Sequence diagram for requirement SR4.1.

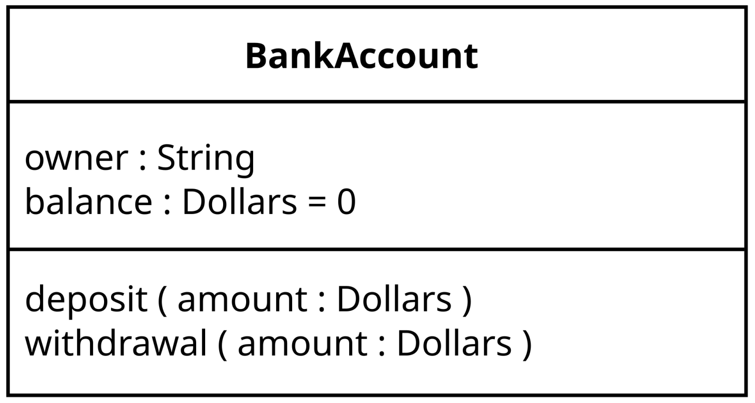
Structural Models

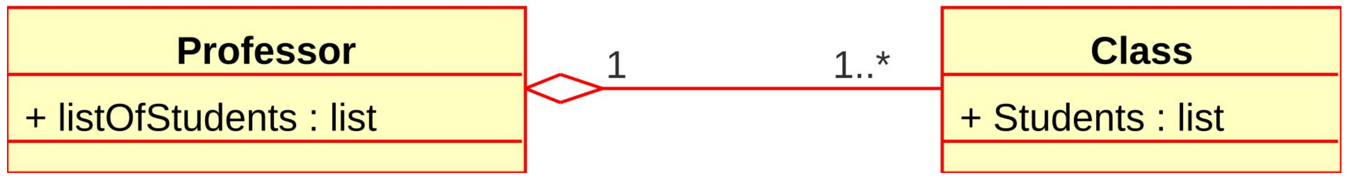
- **Structural models** of software display the organization of a system in terms of the components that make up that system and their relationships.

- **UML class diagrams** can be used for modeling the static structure of the object classes in a software system.

UML Class Diagrams

- **Class diagrams** are used to show the classes in a system and the associations between these classes.
- An **object class** can be thought of as a general definition of one kind of system object.
- An **association** is a link between classes that indicates that there is a relationship between these classes.

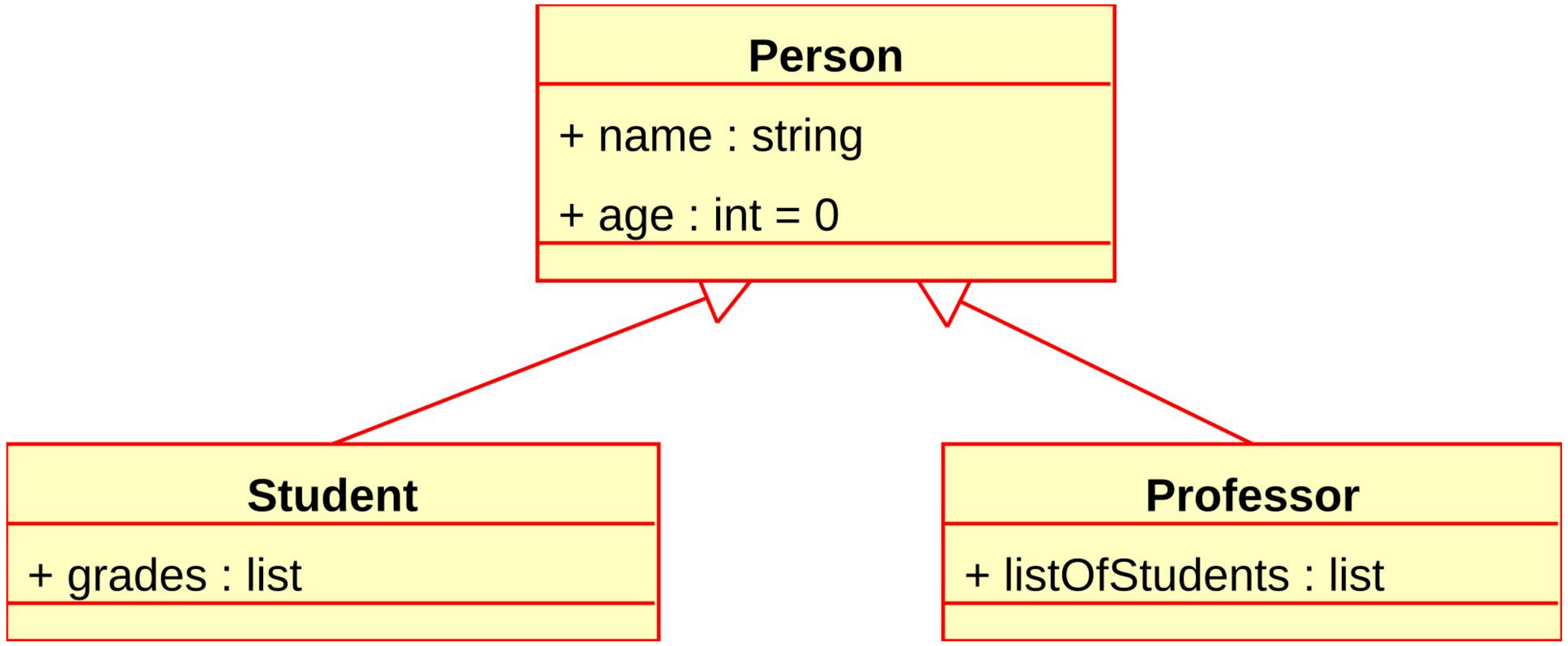




Class diagrams

Class Diagrams: Generalization (Inheritance)

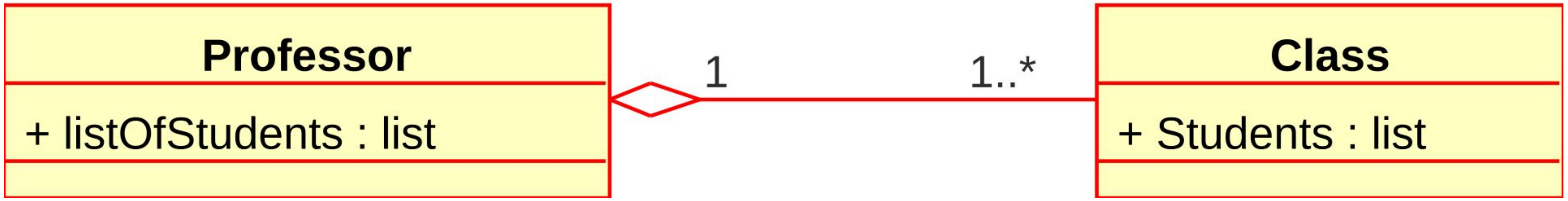
- In Java, **generalization** is implemented using the **class inheritance** mechanism.



Generalization between the superclass Person and the two subclasses, Student and Professor.

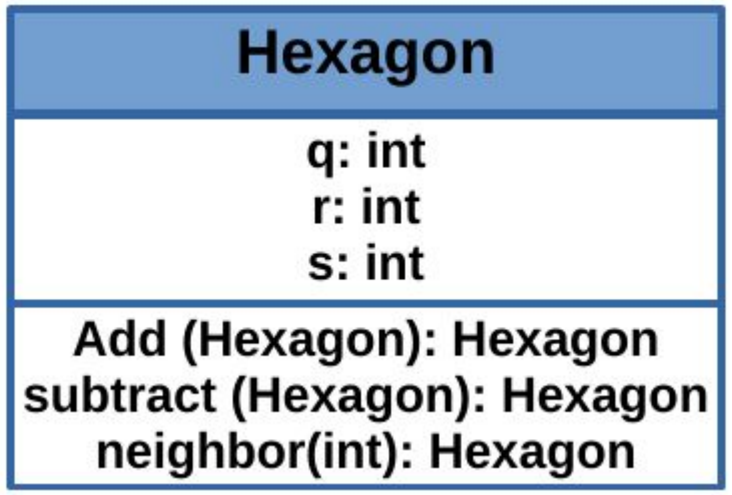
Class Diagrams: Aggregation

- **Aggregation** that means that one object (the whole) is composed of other objects (the parts).

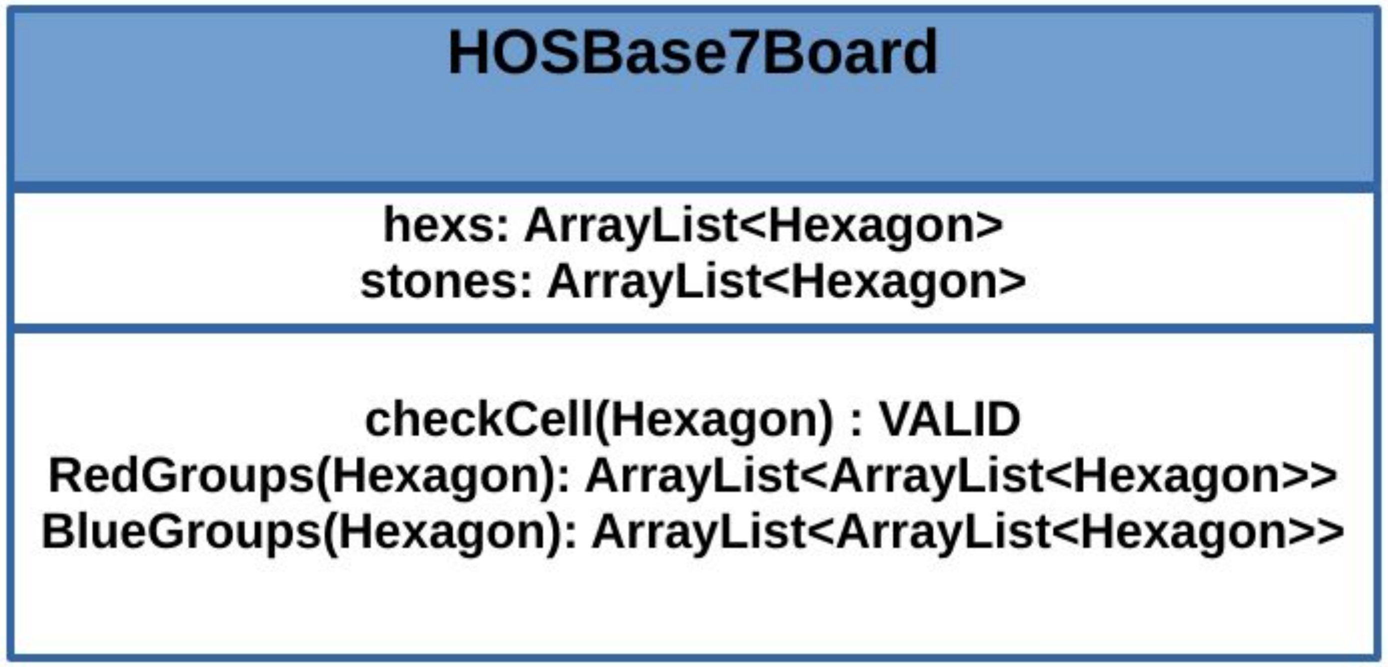


- Aggregation between two classes, Professor and Class.
- White diamond represents aggregation.
- Here, a Professor 'has a' class to teach.
- The Class is an aggregate.

Class Diagrams in HexOust



Using Cube coordinate system
(To be covered in next lecture)


HOSBase7Board
hexs: ArrayList<Hexagon>
stones: ArrayList<Hexagon>
checkCell(Hexagon) : VALID
RedGroups(Hexagon): ArrayList<ArrayList<Hexagon>>
BlueGroups(Hexagon): ArrayList<ArrayList<Hexagon>>

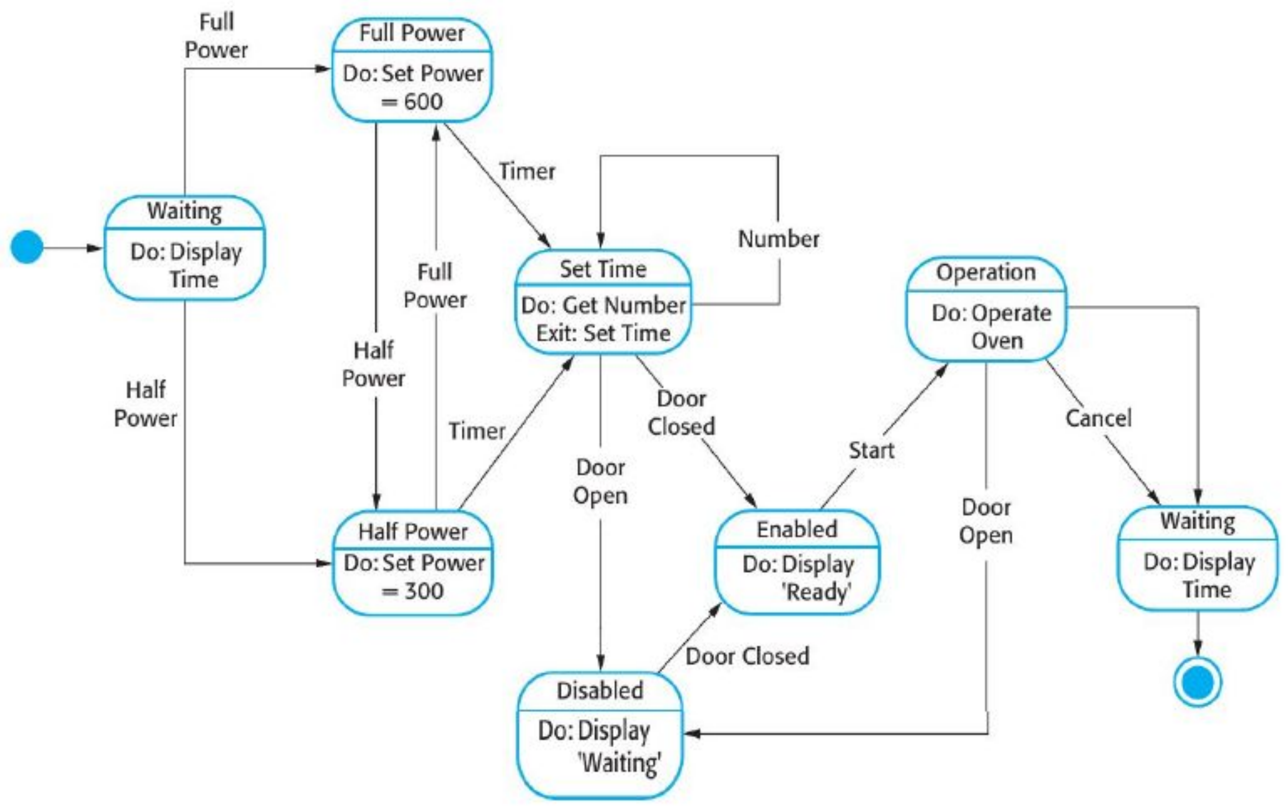
The variable **hexs** stores the list of hexagons.
The variable **stones** stores the list of hexagons with stones in them.
The method **RedGroups** returns the neighbouring red groups for a blue stone/hexagon.
The method **BlueGroups** returns the neighbouring red groups for a blue stone/hexagon.

Behavioral Models

- Behavioral models are models of the dynamic behavior of the system as it is executing.

- They show what happens when a system responds to a stimulus from its environment, which includes some data input to the system or some event that triggers system processing.

- UML 2.0 supports using behavior modeling using **activity**, **sequence** and **state** diagrams.



MicroWave Oven State Diagram

Software Architectural Design

What is Software Architectural Design?

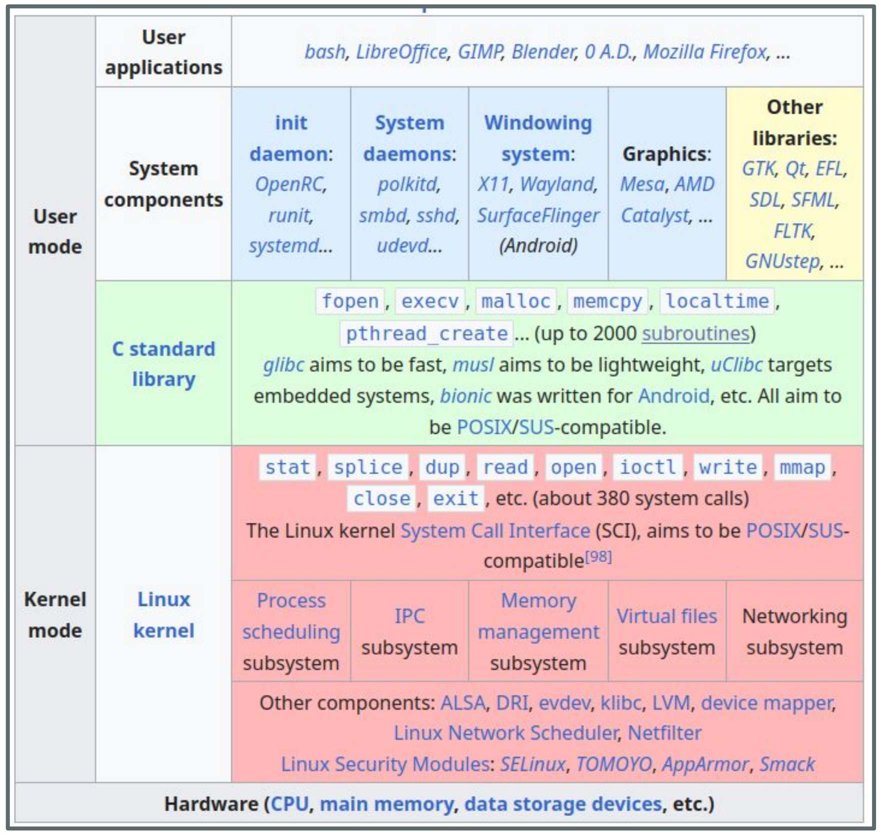
- **Architectural design** is concerned with understanding how a system should be organized and designing the overall structure of that system.

- **Architectural design** is a creative process where you design a system organization that will satisfy the **functional** and **non-functional** requirements of a system.

- The output of the **architectural design process** is an architectural model that describes how the system is organized as a set of communicating components.

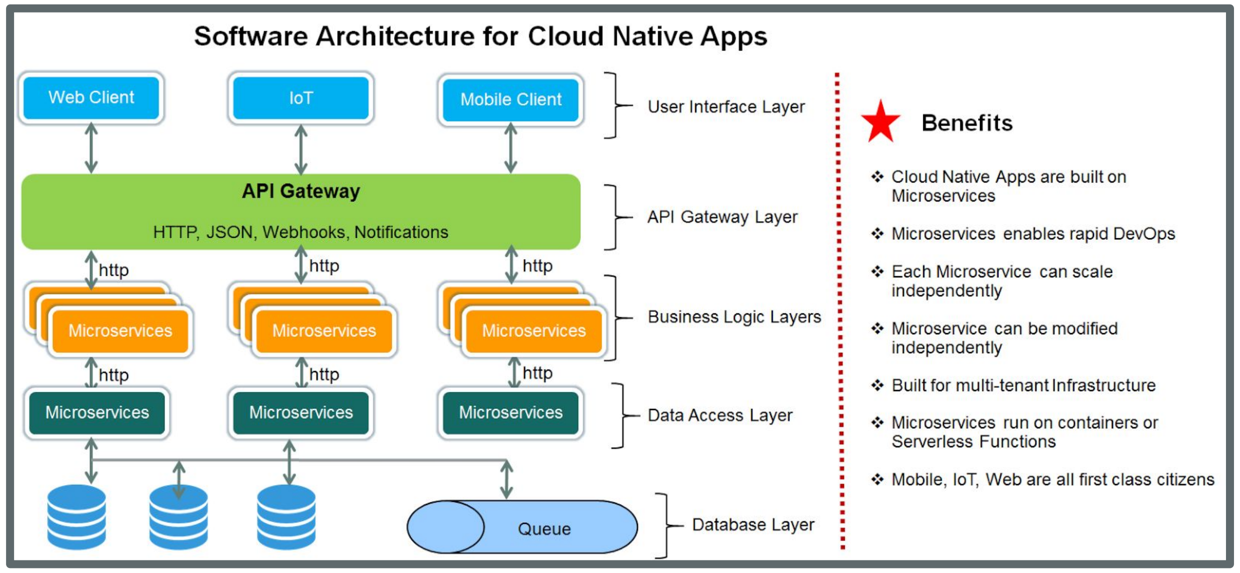
Why is Architectural Design Important?

- Software architecture is important because it affects the **performance**, **robustness**, **distributability**, and **maintainability** of a system.
- Individual system components implement the **functional system requirements**.
- The **non-functional requirements** depend on the system architecture, the way in which these components are organized and communicate.



User mode	User applications	bash, LibreOffice, GIMP, Blender, 0 A.D., Mozilla Firefox, ...				
	System components	init daemon: OpenRC, runit, systemd...	System daemons: polkitd, smbd, sshd, udevd...	Windowing system: X11, Wayland, SurfaceFlinger (Android)	Graphics: Mesa, AMD Catalyst, ...	Other libraries: GTK, Qt, EFL, SDL, SFML, FLTK, GNUstep, ...
	C standard library	fopen, execv, malloc, memcpy, localtime, pthread_create... (up to 2000 subroutines) glibc aims to be fast, musl aims to be lightweight, uClibc targets embedded systems, bionic was written for Android, etc. All aim to be POSIX/SUS-compatible.				
Kernel mode	Linux kernel	stat, splice, dup, read, open, ioctl, write, mmap, close, exit, etc. (about 380 system calls) The Linux kernel System Call Interface (SCI), aims to be POSIX/SUS-compatible[98]				
		Process scheduling subsystem	IPC subsystem	Memory management subsystem	Virtual files subsystem	Networking subsystem
		Other components: ALSA, DRI, evdev, klibc, LVM, device mapper, Linux Network Scheduler, Netfilter Linux Security Modules: SELinux, TOMOYO, AppArmor, Smack				
Hardware (CPU, main memory, data storage devices, etc.)						

Linux Software Architecture


Software Architecture for Cloud Native Apps
Web Client
IoT
Mobile Client
User Interface Layer
API Gateway
HTTP, JSON, Webhooks, Notifications
API Gateway Layer
http
Microservices
Business Logic Layers
Microservices
Data Access Layer
Queue
Database Layer
Benefits
Cloud Native Apps are built on Microservices
Microservices enables rapid DevOps
Each Microservice can scale independently
Microservice can be modified independently
Built for multi-tenant Infrastructure
Microservices run on containers or Serverless Functions
Mobile, IoT, Web are all first class citizens

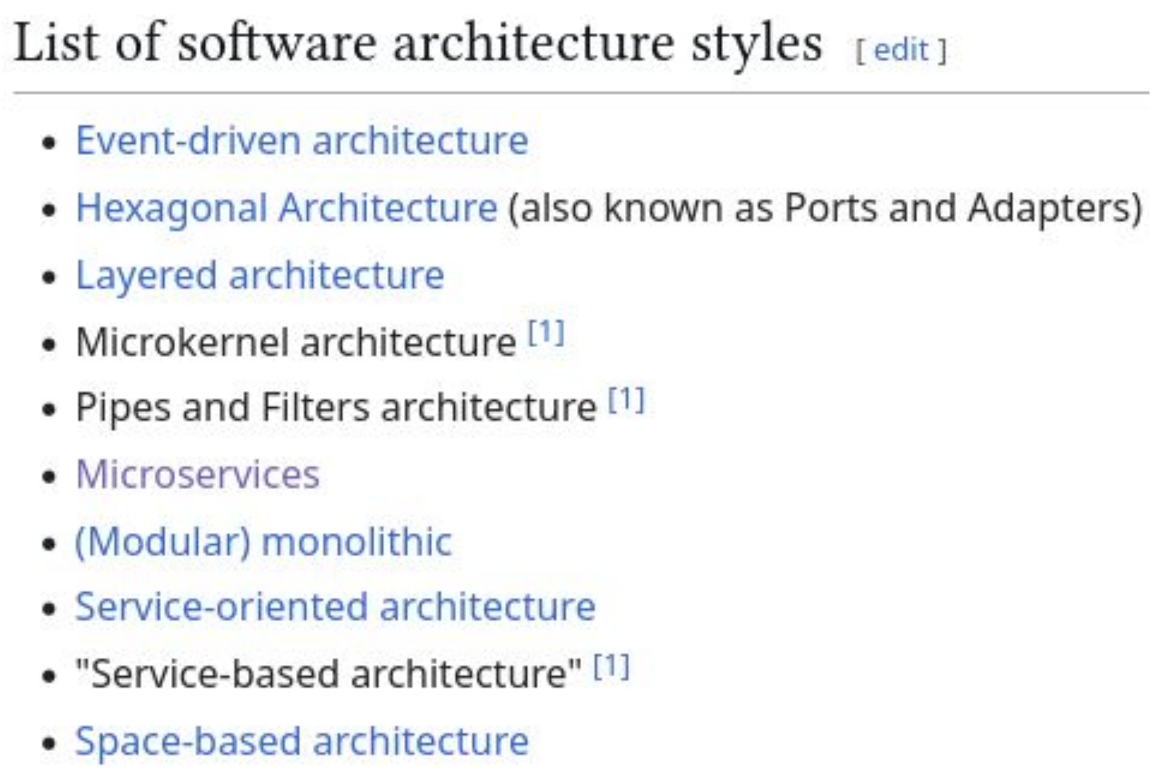
Architectural Design Decisions

- **Architectural design decisions** are the fundamental choices made during the software architecture process that shape the *structure*, *behavior*, and *quality* attributes of a system.
    - What **architectural styles and patterns** might be used?
    - What **technology stack** to use?
    - What is the **deployment strategy** (single machine, mobile, cloud)?
    - How should the **architecture** of the system be **documented**?
    - …

Architectural Styles and Patterns

What is an Architectural Style?

- An **architectural style** is a broad and general categorization of system organization principles.
- Here the focus is on structure and organizing principles rather than solving a specific problem.

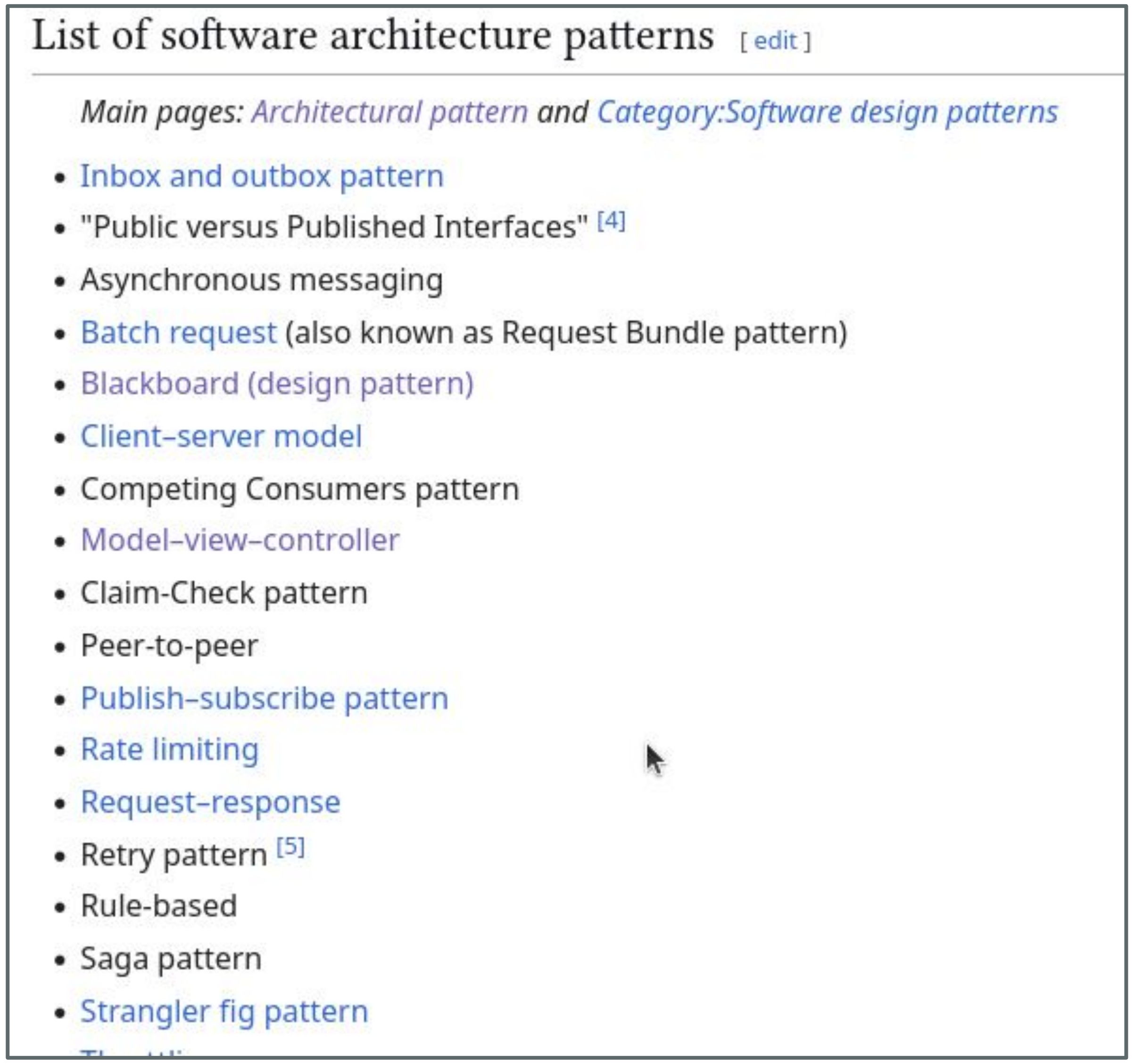


List of software architecture styles [edit]

- Event-driven architecture
- Hexagonal Architecture (also known as Ports and Adapters)
- Layered architecture
- Microkernel architecture [1]
- Pipes and Filters architecture [1]
- Microservices
- (Modular) monolithic
- Service-oriented architecture
- "Service-based architecture" [1]
- Space-based architecture

What is an Architectural Pattern?

- An **architectural pattern** is a reusable, proven solution to a specific, recurring problem, which can be applied within various architectural styles.
- Patterns are more specific, often detailing interactions, components, responsibilities, and relationships.

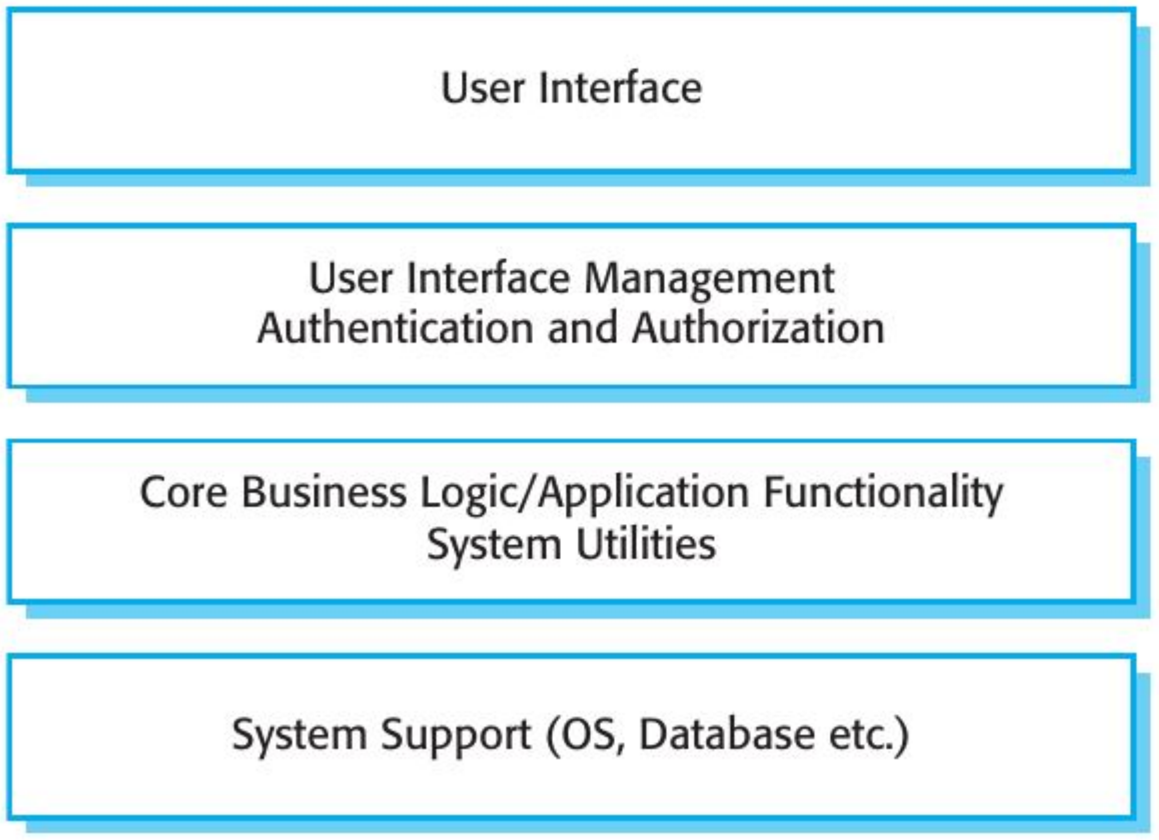


List of software architecture patterns [edit]

Main pages: Architectural pattern and Category:Software design patterns

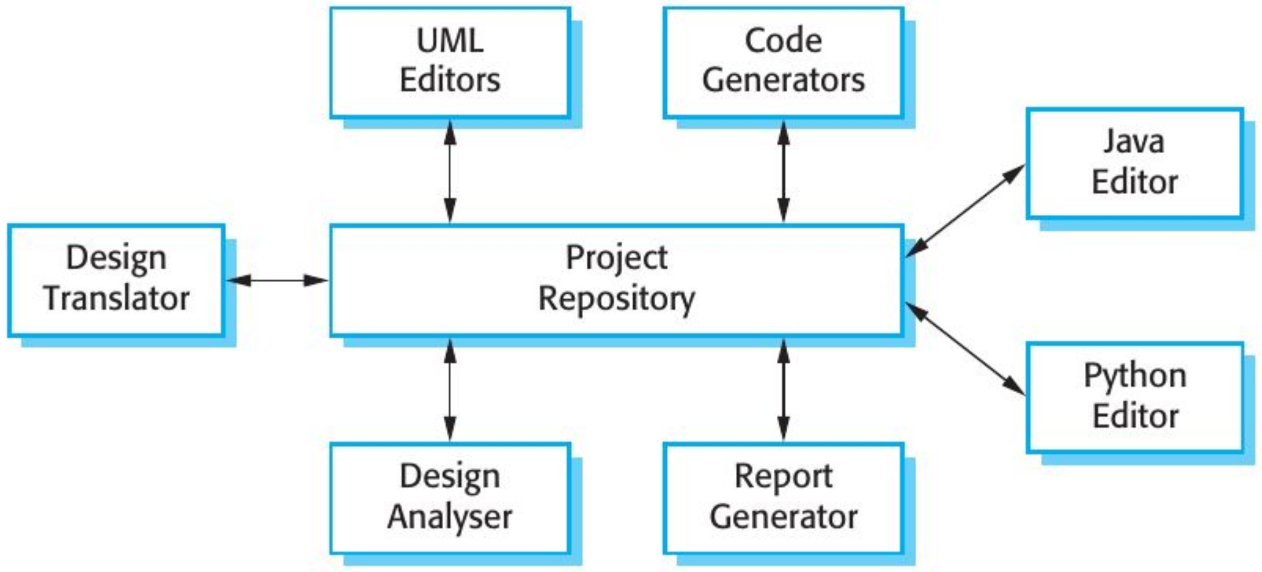
- Inbox and outbox pattern
- "Public versus Published Interfaces" [4]
- Asynchronous messaging
- Batch request (also known as Request Bundle pattern)
- Blackboard (design pattern)
- Client–server model
- Competing Consumers pattern
- Model–view–controller
- Claim-Check pattern
- Peer-to-peer
- Publish–subscribe pattern
- Rate limiting
- Request–response
- Retry pattern [5]
- Rule-based
- Saga pattern
- Strangler fig pattern

Layered Architecture


User Interface
User Interface Management
Authentication and Authorization
Core Business Logic/Application Functionality
System Utilities
System Support (OS, Database etc.)

- Organizes the system into **layers** with related functionality associated with each layer.
- A layer provides services to the layer above it so the lowest-level layers represent core services.
- Allows replacement of entire layers so long as the interface is maintained.

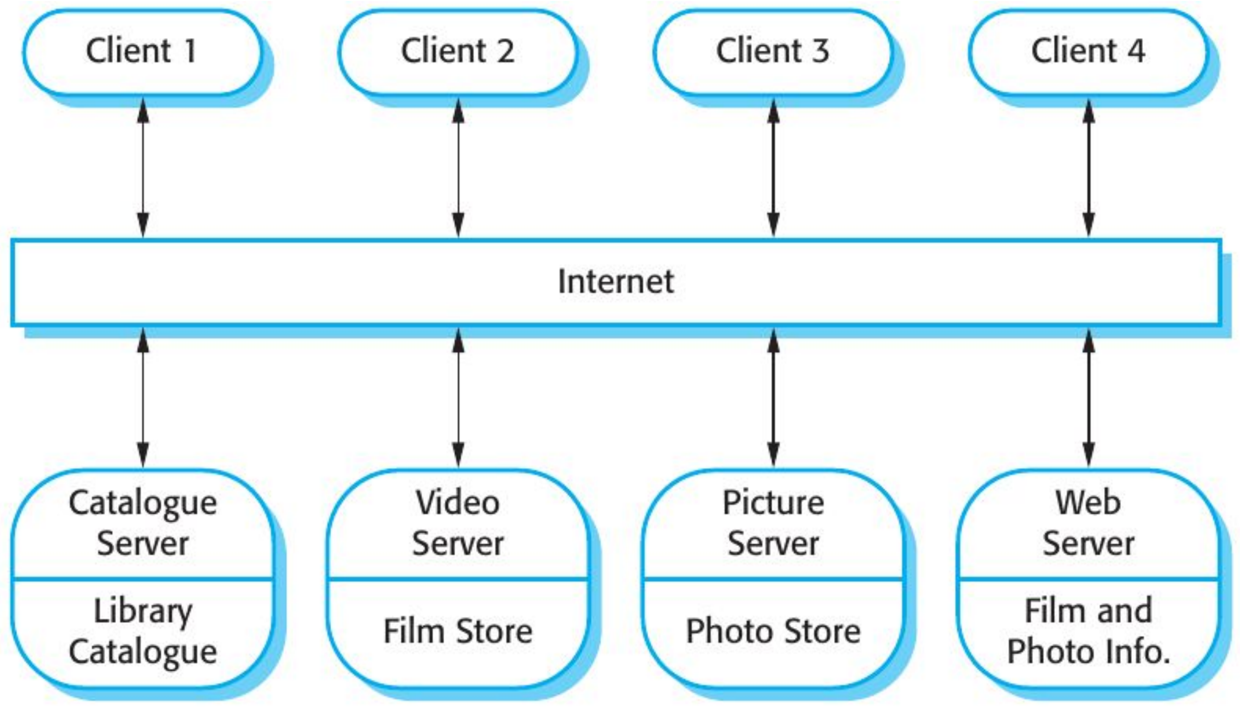
Repository Architecture



A Repository Architecture for an IDE.

- All data in a system is managed in a central repository that is accessible to all system components.
- Components interact only through the repository but not directly.
- Components can be independent. All data can be managed consistently as it is all in one place.

Client-Server Architecture

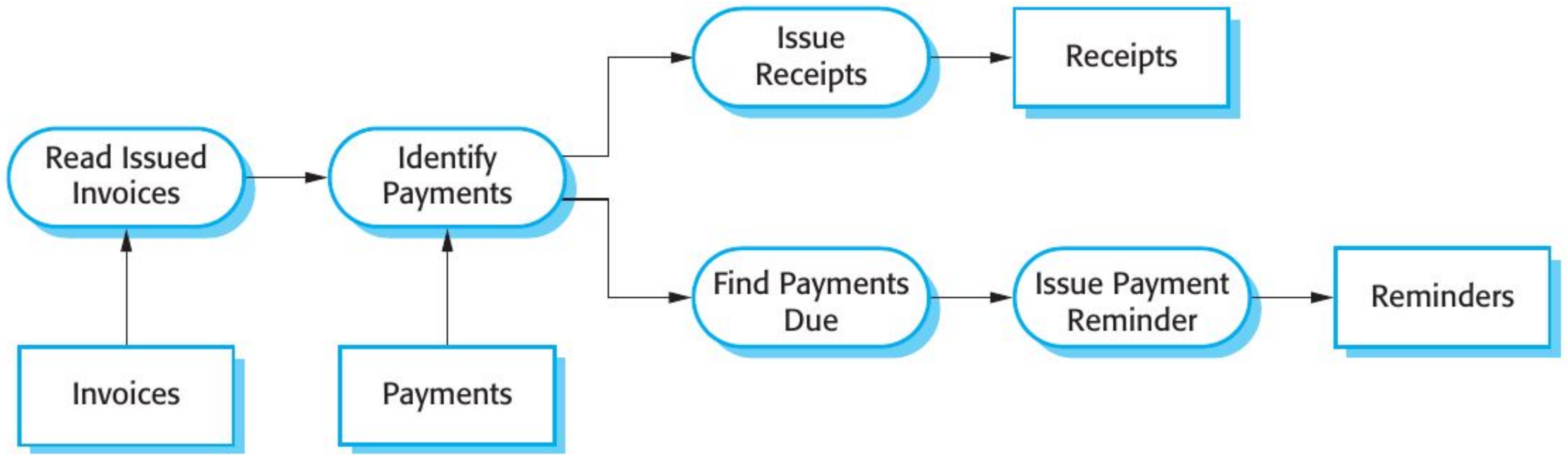


Film and Video/DVD Library.

- In a **client–server architecture**, the functionality of the system is organized into services.
- **Clients** are users of these services and access **servers** to make use of them.
- It is a **distributed architecture** where a new server can be easily integrated or servers upgraded transparently without affecting other parts of the system.

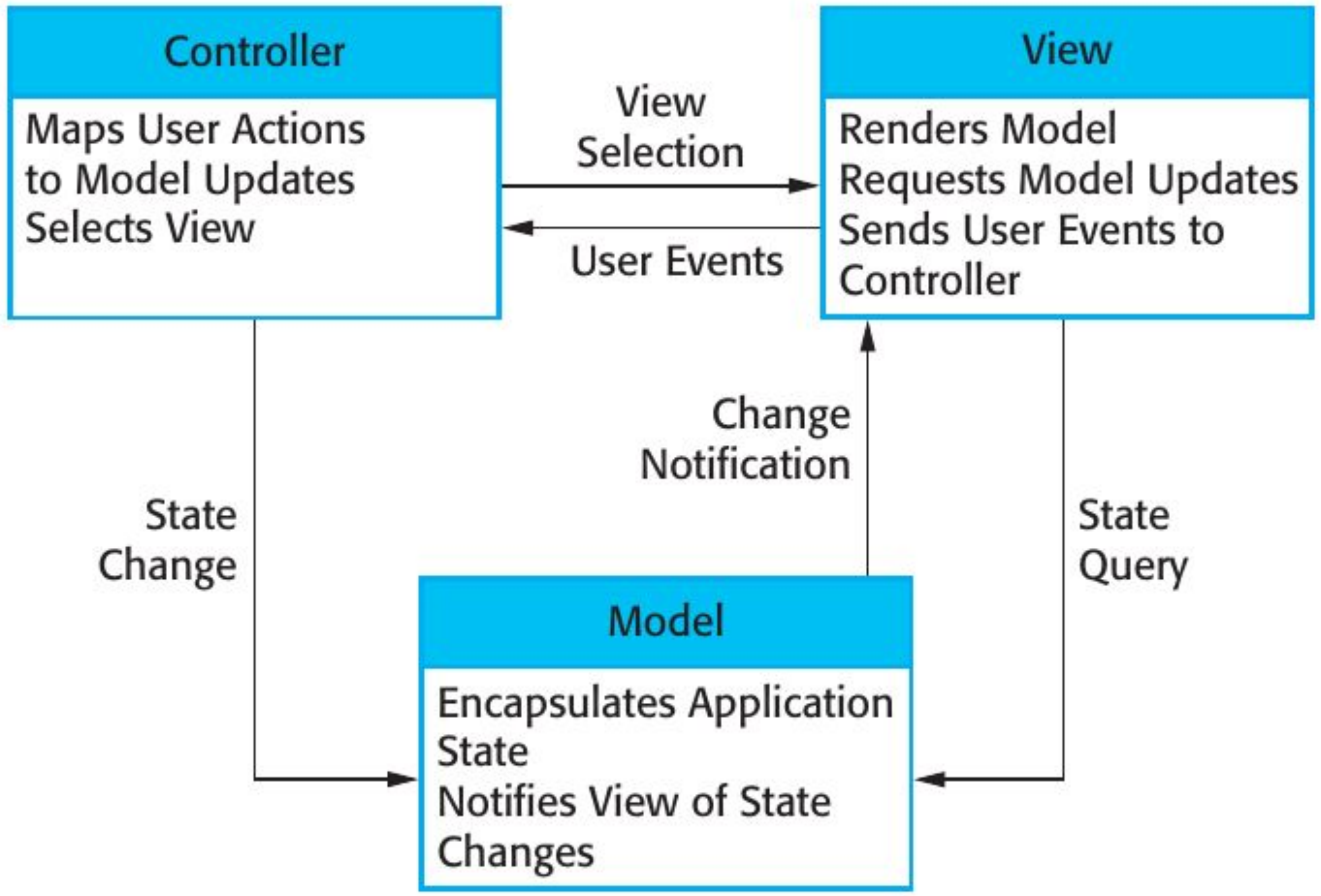
Pipe and Filter Architecture

- The processing of the data in a system is organized so that each processing component (**filter**) carries out one type of data transformation.
- The data flows (as in a **pipe**) from one component to another for processing.
- **UNIX Shell** is based on this architecture.



Example illustrating a pipe-and-filter architecture for processing invoices

Model-View-Controller Pattern

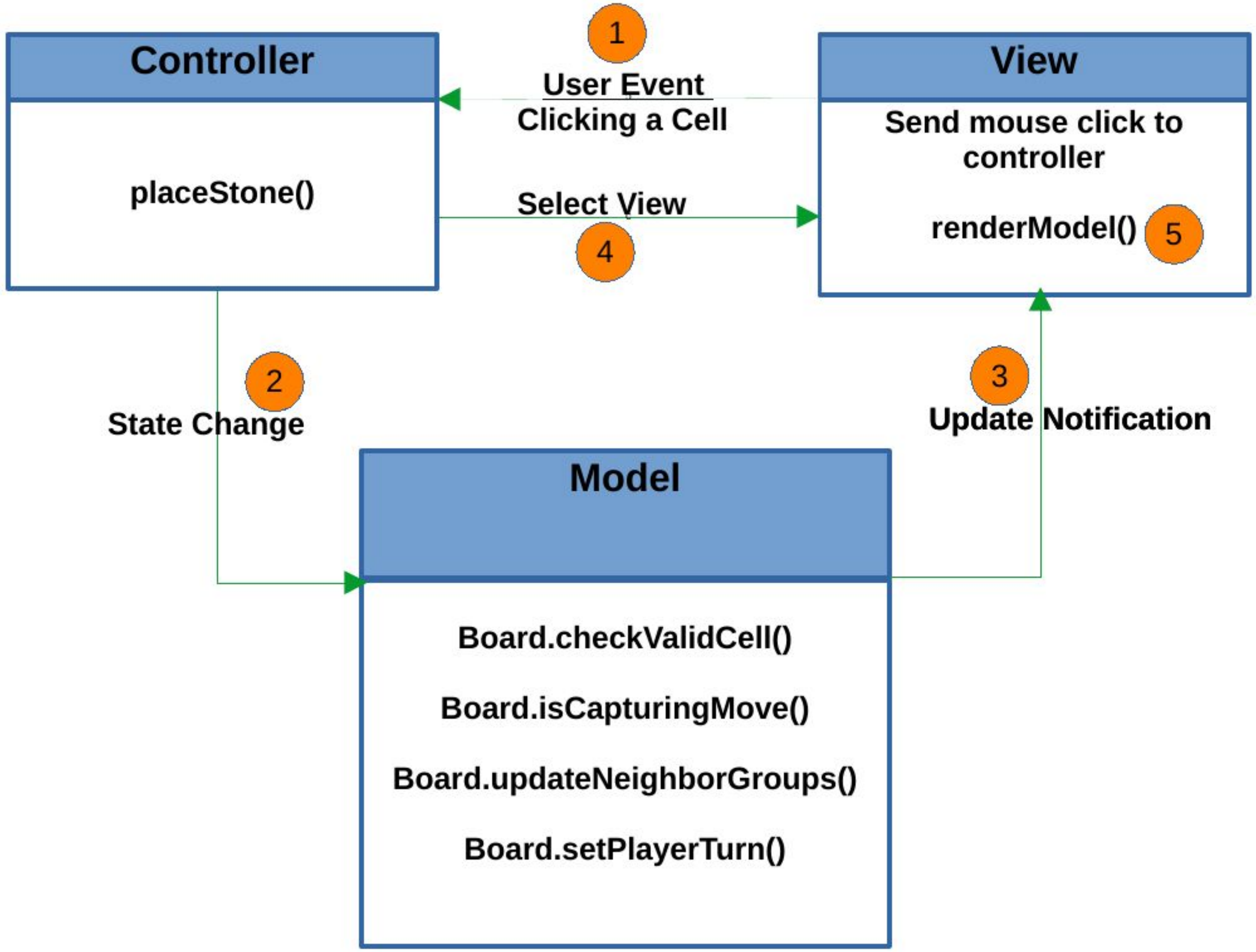


Controller
Maps User Actions to Model Updates Selects View
View Selection
User Events
View
Renders Model Requests Model Updates Sends User Events to Controller
Change Notification
State Change
State Query
Model
Encapsulates Application State Notifies View of State Changes

Commonly used for developing graphical user interfaces.

- The system is structured into three logical components that interact with each other.
- The **Model** component manages the system data and associated operations on that data.
- The **View** component defines and manages how the data is presented to the user.
- The **Controller** component:
  - Manages user interaction (key presses and mouse clicks), and
  - Passes these interactions to the *View* and the *Model*.

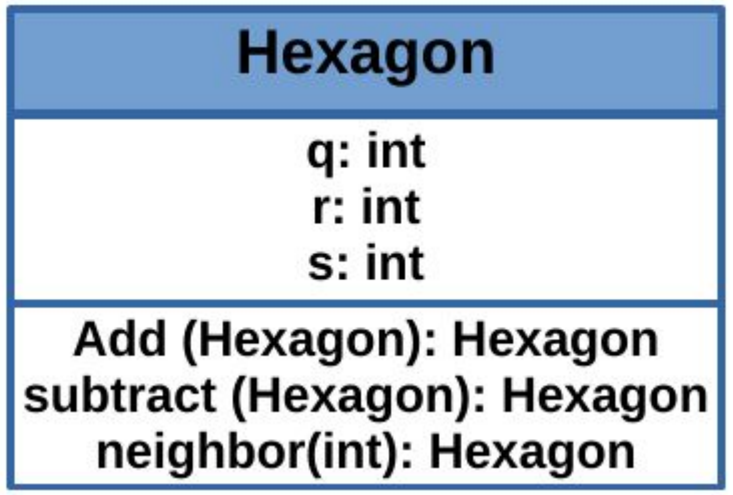
Model-View-Controller for HexOust

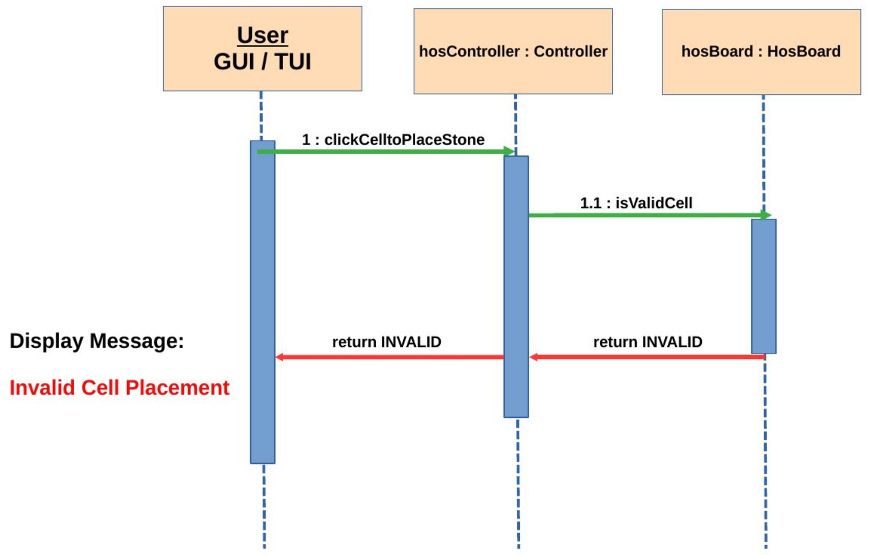


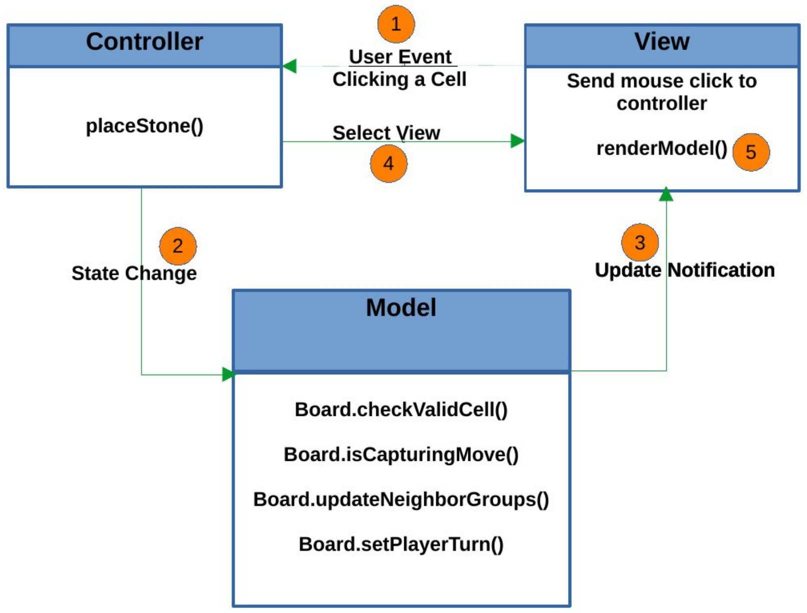
Architectural Design Submission Details

Architectural Design Submission Requirements

- **Three mandatory** components in your architectural design document:
  - One or more **class diagrams** of high-level components.
  - One or more **activity/sequence diagrams** describing use cases (requirement).
  - An **architectural pattern/style** that will be consistently adopted in your project.

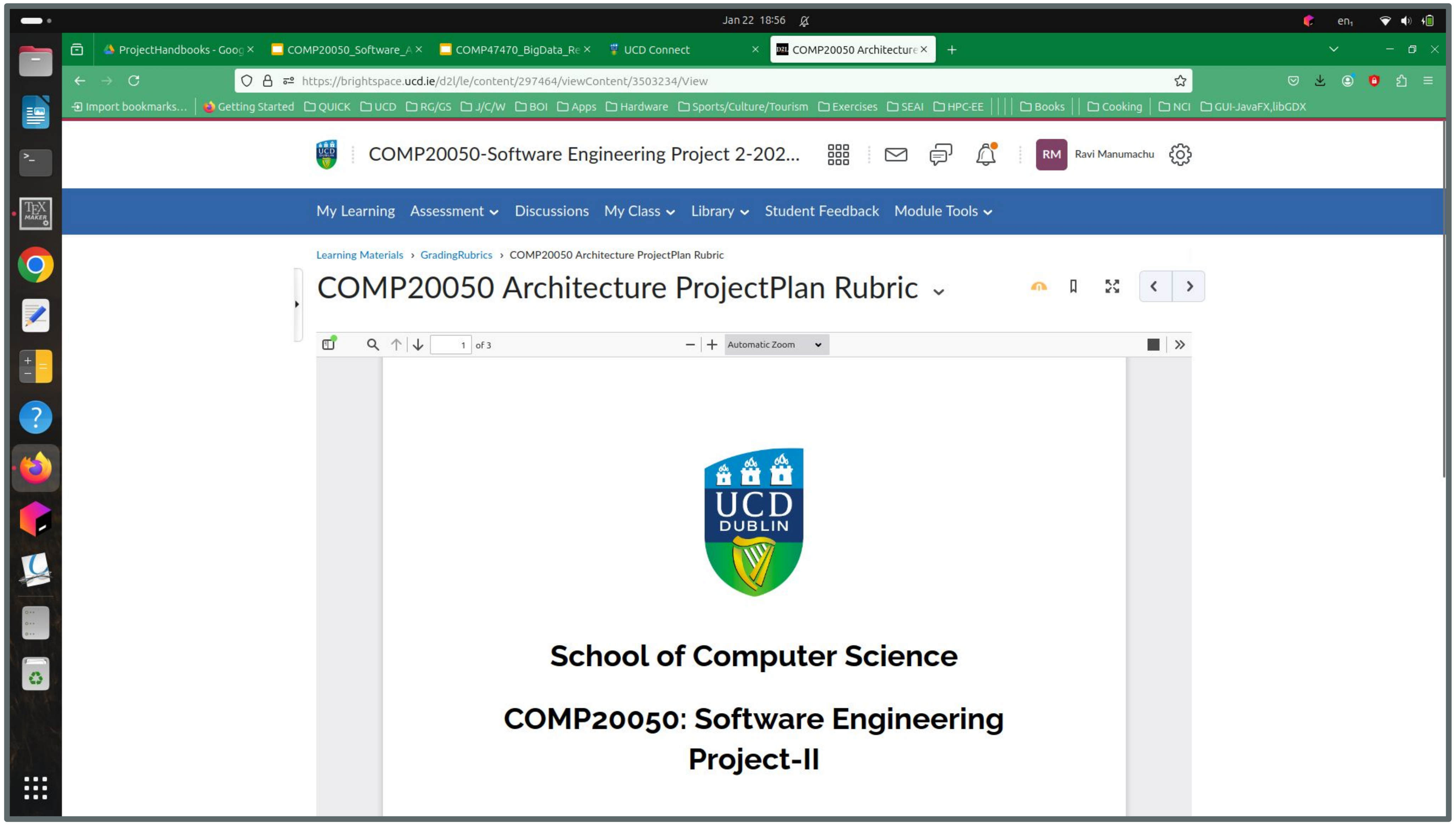






Architectural Sprint Rubric

- See rubric for evaluation criteria and submission requirements.
- **Deadline: 10 February**



COMP20050 Architecture ProjectPlan Rubric

School of Computer Science

COMP20050: Software Engineering Project-II

Q&A

To follow...

Hexagonal Grid: Design and Implementation