Designing an Architecture (ADD Method)

Attribute Driven Design (ADD) - a systematic method that provides guidance in performing the complex design activity. Allows architecture to be designed systematically, repeatably and cost effectively, breaking down design decisions into manageable steps.

Architectural drivers: include the system's purpose, critical quality attribute scenarios, functional priorities, and technical constraints.

Steps in ADD:

1. **Review Inputs**
   Make sure architectural drivers are clearly identified: key functionality, quality attributes (like performance, availability, etc.), constraints (like tech or regulatory), and concerns. Requirements, constraints, collaboration with other systems...

2. **Establish Iteration Goal**
   Focus each iteration on a small set of drivers to prevent overloading the design process.

3. **Choose Elements to Refine**
   Can use top-down (starting from overall system), bottom-up (starting from reusable parts), or evolve previous iterations.

4. **Choose Design Concepts**
   In this step, the architect selects design concepts that can help meet the goals defined by the architectural drivers. This includes choosing architectural patterns, technologies, design tactics, and possibly reference architectures.

5. **Instantiate Architectural Elements**
   Once the design concepts are chosen, they are applied to define actual architectural elements. This means identifying concrete components, assigning them responsibilities, and defining their interfaces and how they interact. How will elements specifically be used (how many layers...), assign functionality to elements like presentation, business and data layers; specify relationships and information flow between components.

   Produce module structure, components and connector (runtime), and allocation structure

   Instantiating elements: reference architectures (modify predefined structures), patterns (tailor generic structures), tactics (adapt or reuse), add libraries or external executables

6. **Sketch Views and Record Decisions**
   Document diagrams and decisions (e.g., layers, dependencies, responsibilities). Can be informal (whiteboard) or formal (UML diagrams). Capture rationale and reasoning. It is intended more for communication, until now it was more about

architectural thinking and designing, this is about documenting and communicating rationale and visualization.

7. **Perform Analysis**
   Validate that the architecture satisfies the drivers. Use informal reviews or structured methods like ATAM. Iterate as needed. Evaluate design, review process, methods (atam) and tracking (architectural backlog)

The process supports clear reasoning, traceable design, and stakeholder communication, and helps avoid ad-hoc decisions.

Team Structure - Mirror the module decomposition

Create Skeletal System: build core system, implement core functions, add features gradually and test regularly

---

Example: Tippeliga-Ticket System (TTS)

This example shows the ADD method in practice. The TTS is a football ticket web system that must integrate with external servers, handle high loads before major matches, and offer secure payments with high availability.

1. **Review inputs**

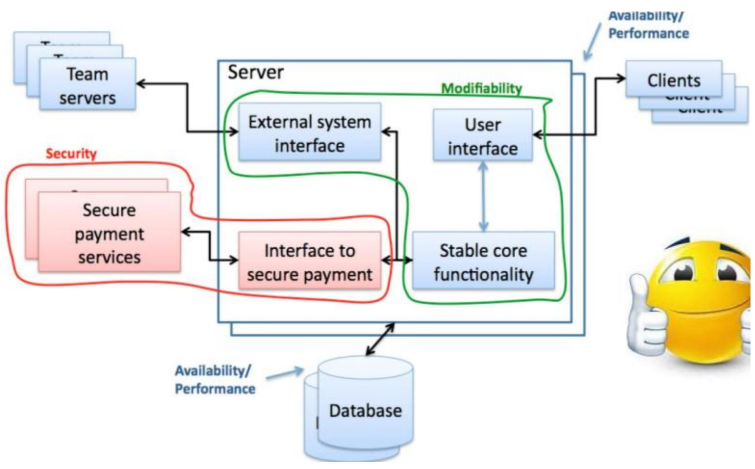
2. **Architectural Drivers Identified**:
   AD1: Availability (max 2 min downtime per week)
   AD2: Security (secure payment)
   AD3: Modifiability (teams change each year, varying server APIs)
   AD4: Performance (40,000 simultaneous users)

3. **Choose element to focus on**:
   Whole system

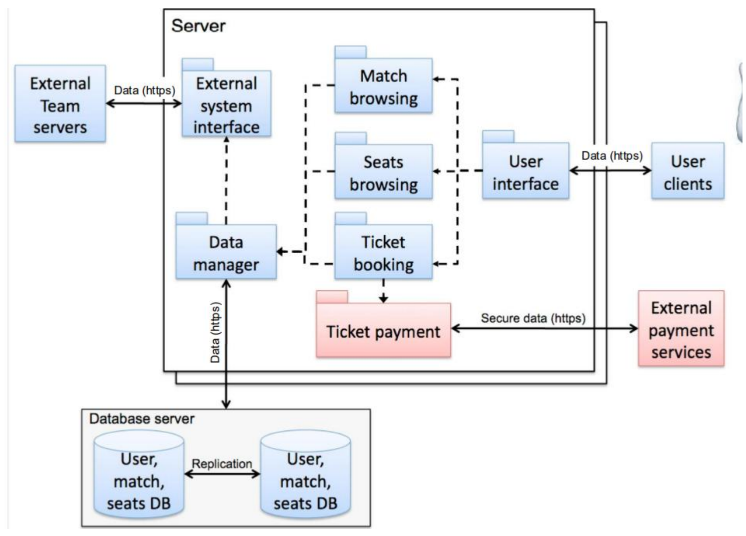
4. **Choose design concepts**
   - **Tactics Chosen**:
     - **Availability**: Server/database replication, heartbeat monitoring.
     - **Security**: HTTPS, authentication, external payment processors.
     - **Modifiability**: Modular structure with interfaces for external systems.
     - **Performance**: Server replication, caching.
   - **Patterns Chosen**:
     - Client-Server for the general structure.
     - MVC or n-tier for modularity and separation of UI/business/data.

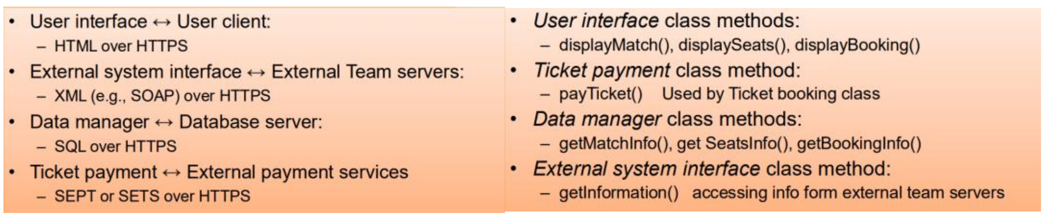
- Interface segregation for team servers.
- Partitioned secure zone for handling payments.

- **Sketch:**


Availability/ Performance
Team servers
Server
Modifiability
Clients
External system interface
User interface
Security
Secure payment services
Interface to secure payment
Stable core functionality
Availability/ Performance
Database

5. **Instantiate architectural elements and define interfaces**





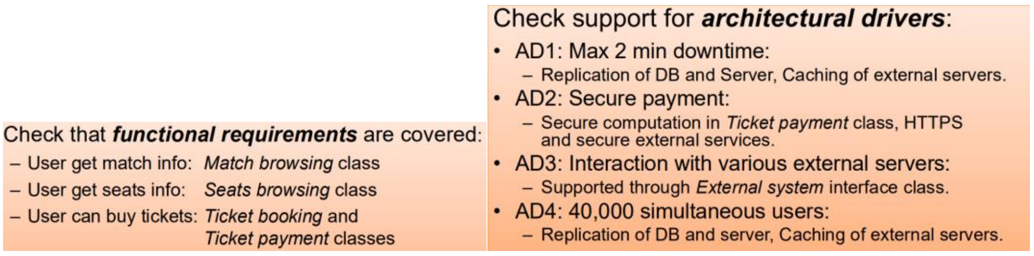
- User interface ↔ User client:
  - HTML over HTTPS
- External system interface ↔ External Team servers:
  - XML (e.g., SOAP) over HTTPS
- Data manager ↔ Database server:
  - SQL over HTTPS
- Ticket payment ↔ External payment services
  - SEPT or SETS over HTTPS

- *User interface* class methods:
  - displayMatch(), displaySeats(), displayBooking()
- *Ticket payment* class method:
  - payTicket() Used by Ticket booking class
- *Data manager* class methods:
  - getMatchInfo(), get SeatsInfo(), getBookingInfo()
- *External system interface* class method:
  - getInformation() accessing info form external team servers

6. Sketch views and record design decisions

   Record all structures from step 5.

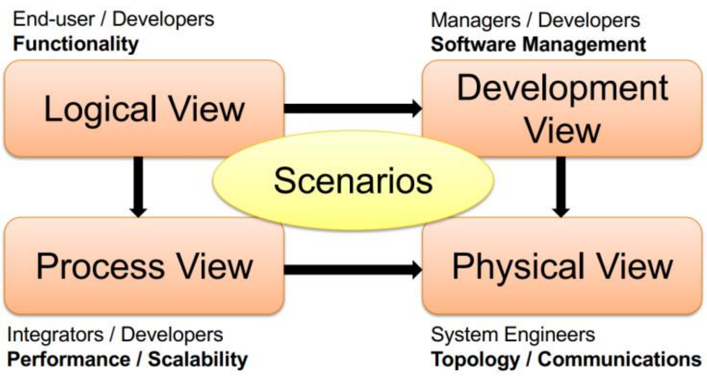
7. **Perform analysis**:
   Each architectural driver is clearly addressed in the structure. Views are used to trace responsibilities and tactics back to drivers.


Check that ***functional requirements*** are covered:
- User get match info: *Match browsing* class
- User get seats info: *Seats browsing* class
- User can buy tickets: *Ticket booking* and *Ticket payment* classes

Check support for ***architectural drivers***:
- AD1: Max 2 min downtime:
  - Replication of DB and Server, Caching of external servers.
- AD2: Secure payment:
  - Secure computation in *Ticket payment* class, HTTPS and secure external services.
- AD3: Interaction with various external servers:
  - Supported through *External system* interface class.
- AD4: 40,000 simultaneous users:
  - Replication of DB and server, Caching of external servers.

Documenting an Architecture – 4+1 View Model

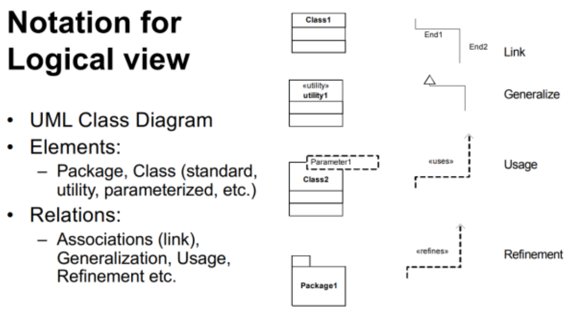
The **4+1 View Model** is a way to describe a system's architecture from multiple perspectives to satisfy different stakeholder concerns. It is especially useful in documentation and evaluation, including for your exam diagram tasks.



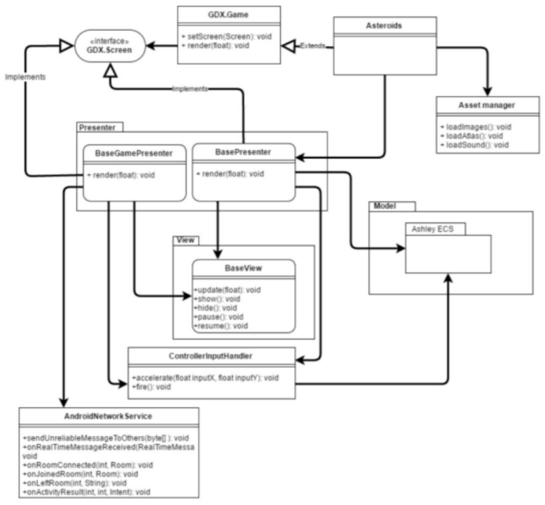
The 4+1 views:

1. Logical: assigns functionality to modules and defines their relationships

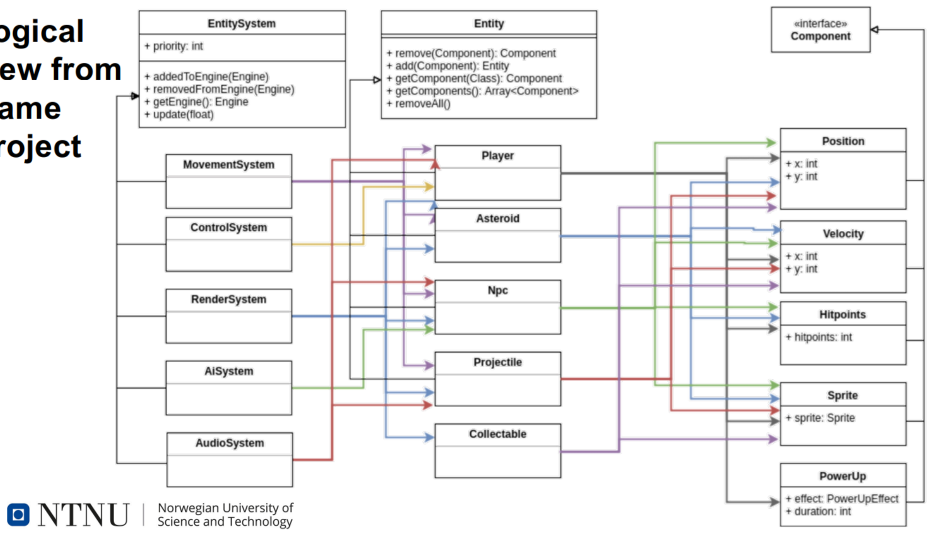
   End user/development. System decomposed into a set of key abstractions. Usually with an object oriented approach or data driven approach.


Notation for Logical view

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



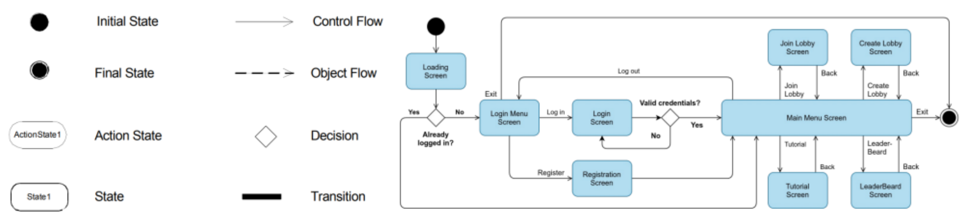
Logical view from Game Project

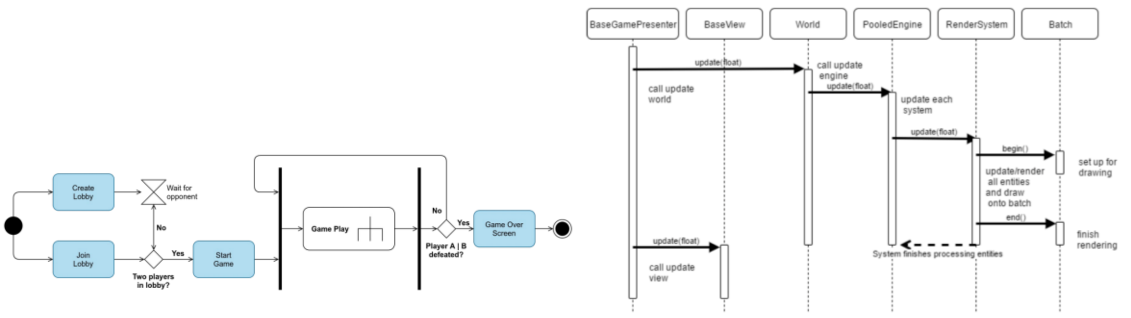


2. **Process View (Integrator, Developer)**
   Describes runtime behavior — processes, concurrency, communication. What does it look like when running the system. Represented using activity, state, or sequence diagrams.
   Addresses: performance, availability, fault tolerance, and system interaction.

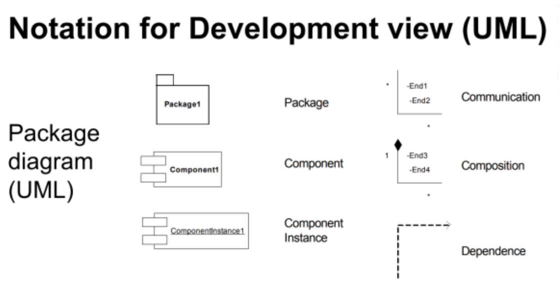
Notation for Processing View

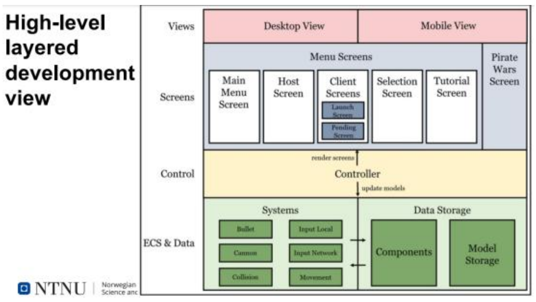
- Activity diagram (UML)
- Can also use following UML diagrams: State, sequence, collaboration, component

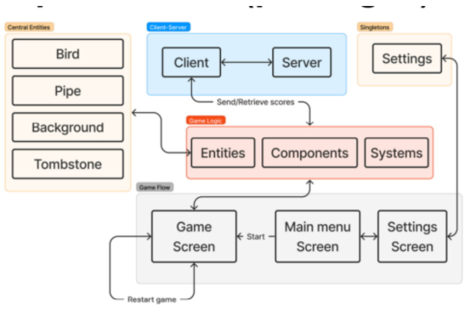




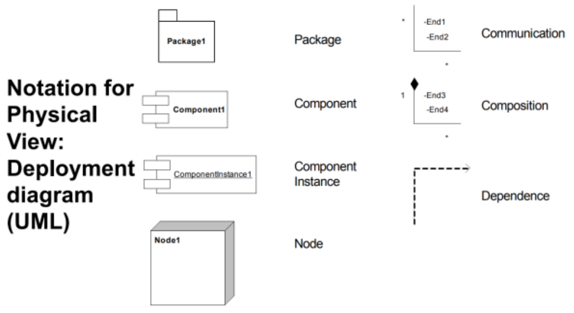
3. **Development View (Manager, Developer)**
   Shows the software's structure in the development environment.
   Focuses on modules, packages, and subsystems, often in layers.
   Addresses: code organization, team division, reuse, and build management.

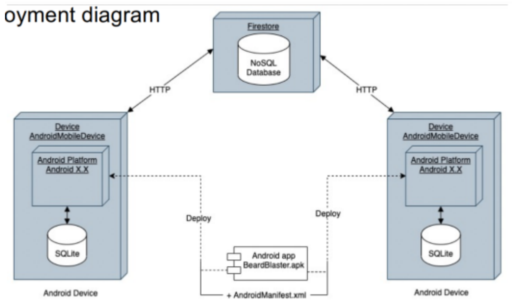




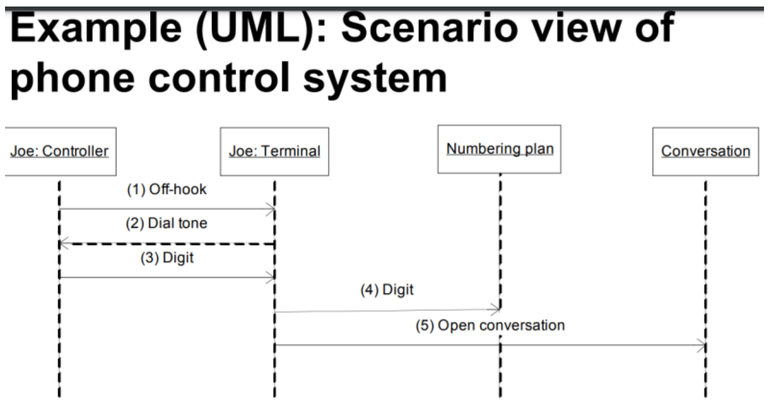


4. **Physical View (System Engineer)**
Describes deployment of software components to hardware.
Includes networks, nodes, execution units mapped onto hardware nodes
Important for: scalability, redundancy, and availability.





5. **+1: Scenarios**
High level overview of how system works, highlights key use cases for the system, describes key use cases to validate and connect the other views.
Helps ensure the views are coherent and the system supports required behavior.
Often captured in UML interaction diagrams (e.g., sequence or use case).


Example (UML): Scenario view of phone control system
Joe: Controller
Joe: Terminal
Numbering plan
Conversation
(1) Off-hook
(2) Dial tone
(3) Digit
(4) Digit
(5) Open conversation