NTU SC2006 Notes

NTU SC2006 Software Engineering Notes

SC2006 Docs Team

Table of contents

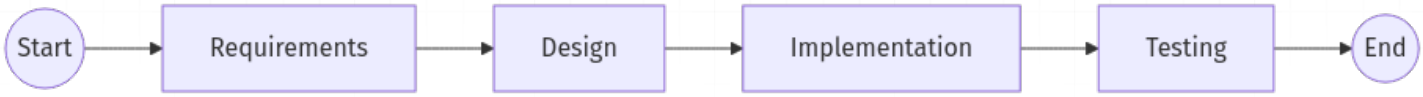
1. Software Engineering

Software Engineering is the:

> The production of **maintainable**, **fault-free** software that meets the user's **requirements** and is delivered **on time** and **within budget**.

not just coding.

2. Software Development Activities



Software Engineering Activities & their deliverables:

1. **Requirements** specify how the system should function
   - Requirements Ellicitation: Software Requirements Specification (SRS)
   - Requirements Analysis: Prototype System Design
2. **Design** System Design & review:
   - Software Design Document
   - Interface Design Document
   - Test Cases
   - Data Models
3. **Implementation**
   - Source Code
   - Software
   - Documentation: eg. User Manual
4. **Testing** checking that the software conforms to requirements
   - Test Report eg. User Acceptance test
5. **Maintenance** evolving software to changing customer needs.
   - Feature requests
   - Bug Fixes

3. Requirements

3.1 Requirements Elicitation

Correct requirements ellication is a the **foundation** of a successful Software Engineering project as it identifies the **purpose** of the software system.

3.2 Requirements Elicitation Process

1. **Identify Stakeholders**
    - Customers
    - Management
    - Developers
2. **Elicit Requirements**
    - from *problem domain*
    - from *customer day to day* activities.
3. **Validate Requirements** with stakeholders.
    - Customers: check that the requirements are **what they want**.
    - Development team: check that they understand what the requirements entail.

3.3 Software Requirements Specification (SRS)

Software Requirements Specification (SRS) typically contains:

1. Product Description
    - Purpose of the System: Mission Statement
    - Scope of the System
    - Users and Stakeholders
    - Assumptions and Constraints
2. Functional Requirements:
    - Use Case Model
    - Class Diagram
    - Sequence Diagram
    - Communication Diagram
    - Activity Diagram
3. Non-Functional Requirements
    - Availability
    - Security
    - Maintainability
    - Portability
4. Interface Requirements
    - User: UI Prototype
    - Hardware: hardware ports
    - Software: API compatibility
5. Data Dictionary

3.3.1 Project Mission Statement

Project Mission Statement defines the project in *2-3 sentences*:

- **Problem** scope of the project.
- **Stakeholders** Developers, Customers, Management.
- **Outcomes** benefits of the project.

3.3.2 Types of Requirements

- **Functional** what *features* must the system have? eg. must be interoperate with another system.
- **Non-functional** what *properties* must the system have? eg. Usability, Reliablility, Performance, Extensibility, Maintainability

3.3.3 Good Requirements

Good Requirements are:

- **Atomic** specify **only 1** requirement per requirement statement.
- **Verifiable** clear **testable** goalpost to satisfy requirement.
- **Unambiguious** interpretation of the requirement is not up to debate.
  - use words `Shall`, `Must`, `Must Not`, `Is required to`, `Are applicable`, `Responsible for`, `Will`.
- **Tracable** requirements can be cited by their requirement IDs back to the documents from which they where defined.

Example:

> REQ-002: The system **shall** require users to enter a valid email address during account registration.

3.3.4 UI Prototype

UI Prototype mock up to work out User Experience (UX) of the User Interface (UI)

3.3.5 Data Dictionary

Data Dictionary is a problem domain glossary that **unambiguiously** define terms so that they are not open for interpretation.

3.4 Unified Modelling Language (UML)

Set of Diagrams for designing Software, **not** a programming language.

- Class Diagram
- Activity Diagram
- Use Case Diagram
- State Machine Diagram
- Communication Diagram

3.5 Use Case Model

Use Case Model combines:

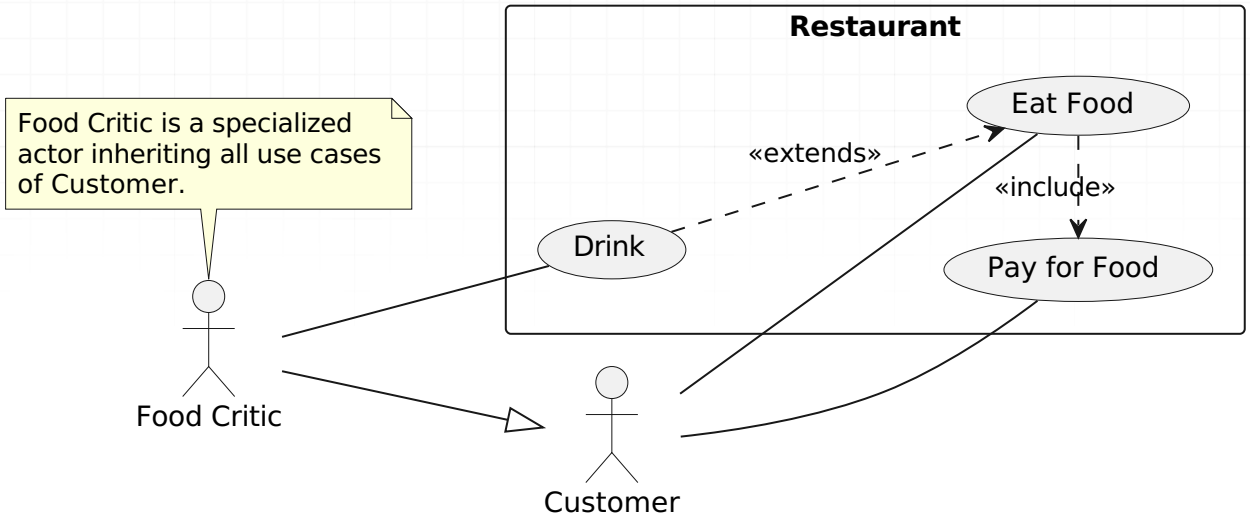
- Use Case Diagram
- Use Case Description

3.5.1 Use Case

Use Case:

- describes how a user **uses a system** to accomplish a particular **goal**.
- summary of ≥1 functional requirements **utilised together** by an **actor**.

3.5.2 Use Case Diagram



Use Case Diagram Associations

- `<<include>>` large use case includes functionality from smaller use case (arrow side).
- `<<extends>>` use case **optionally** extends functionality of another use case (arrow side).

3.5.3 Use Case Description

Use Case Description contains:

- **Participating Actors**
  - 1 **initiating actor** triggers the use case.
- **Entry Conditions** start state before the use case begins as a set of conditions.
- **Exit Conditions** end state after the use case ends as a set of conditions.
- **Flow of Events** steps performed in the **successful**/happy path:
  - Actor steps "The Actor ..."
  - System steps "The System ..."
- **Alternative Flows** steps performed on deviations from the successful path.
  - ID format: `AF-[0-9]+` **variations** from successful path.
  - ID format: `EX-[0-9]+` **exceptions** (errors) from successful path.

3.6 Requirements Analysis

Designing the software system based on requirements gathered in analysis:

3.6.1 Conceptual Model

Model the structure of the system via UML Class Diagram:

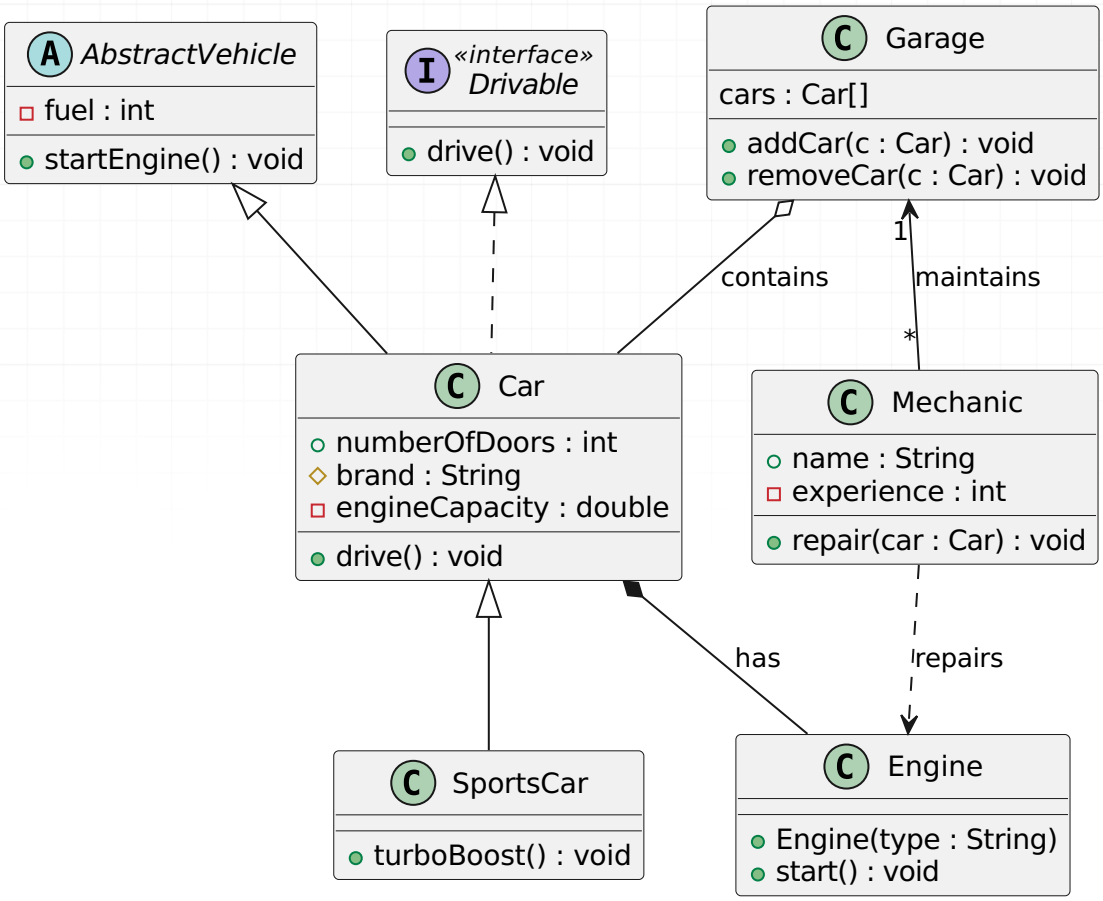
- Objects aka Classes
- Attributes aka Properties of objects.
- Operations aka Methods that can be performed on Objects.

3.6.2 Dynamic Model

Model the implementation of the system via UML Diagrams:

- **Single** Use Case
  - Sequence Diagram: focused on **timing & order of interactions** between objects.
  - Communication Diagram: Interactions between **objects**. Focused on objects.
- **Multiple** Use Cases
  - State Machine Diagram: visualise **single** system as a set of States.
  - Activity Diagram: visualise interactions **≥ 1** system(s) as a set of **workflow steps**.

3.7 Class Diagram



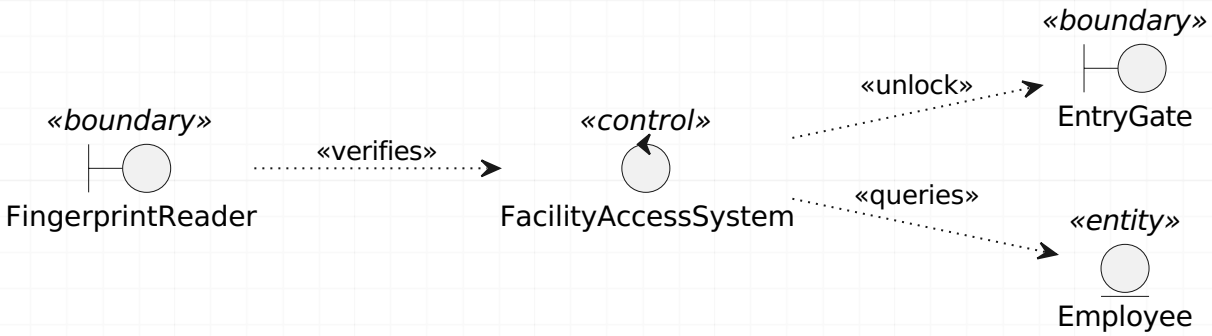
- **Abstract Class** class name is in *italics*.
- **Multiplicity** eg. many (*) `Mechanic` s to 1 (1) `Garage`.
- **Inheritance** subclass `SportsCar` inherits from class `Car`.
- **Implements** implementation `Car` implements `Drivable` interface.
- **Aggregation** `Car` is part of `Garage`, but **can exist independently**.
- **Composition** `Engine` is part of `Car`, but **cannot exist independently**.
- **Dependency** `Mechanic` uses `Engine` **temporarily**, but **does not** have an `Engine` attribute / property.

3.7.1 Access Modifiers

Access Modifier controls access to Attributes & Operations:

Access Modifier	Symbol	Description
Public	`+`	Members are accessible from anywhere.
Private	`-`	Members are accessible only within the class.
Protected	`#`	Members are accessible within the class and its subclasses.

3.8 Class Stereotype Diagram



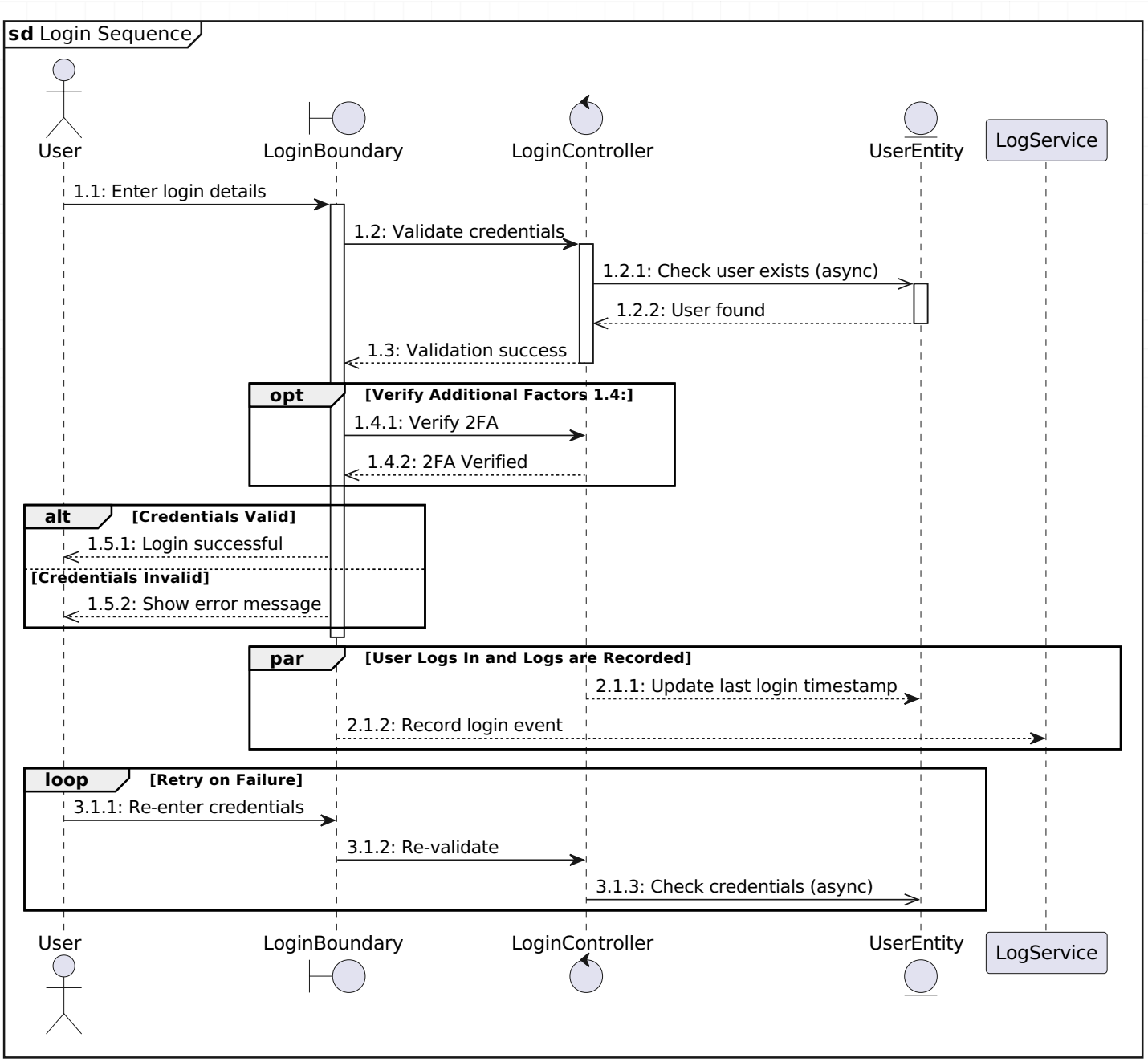
Class Diagram with **no details** (methods or attributes) visualising only:

- **Interactions** between classes via `<<usage>>` dependency.
- **Class Stereotype** of each class:
  - **Boundary** interface between actor and system.
  - **Control** app logic classes.
  - **Entity** data model classes.

3.9 Sequence Diagram

Visualises **timing and order** of interactions between objects:

- **1 Use Case** = 1 Sequence Diagram



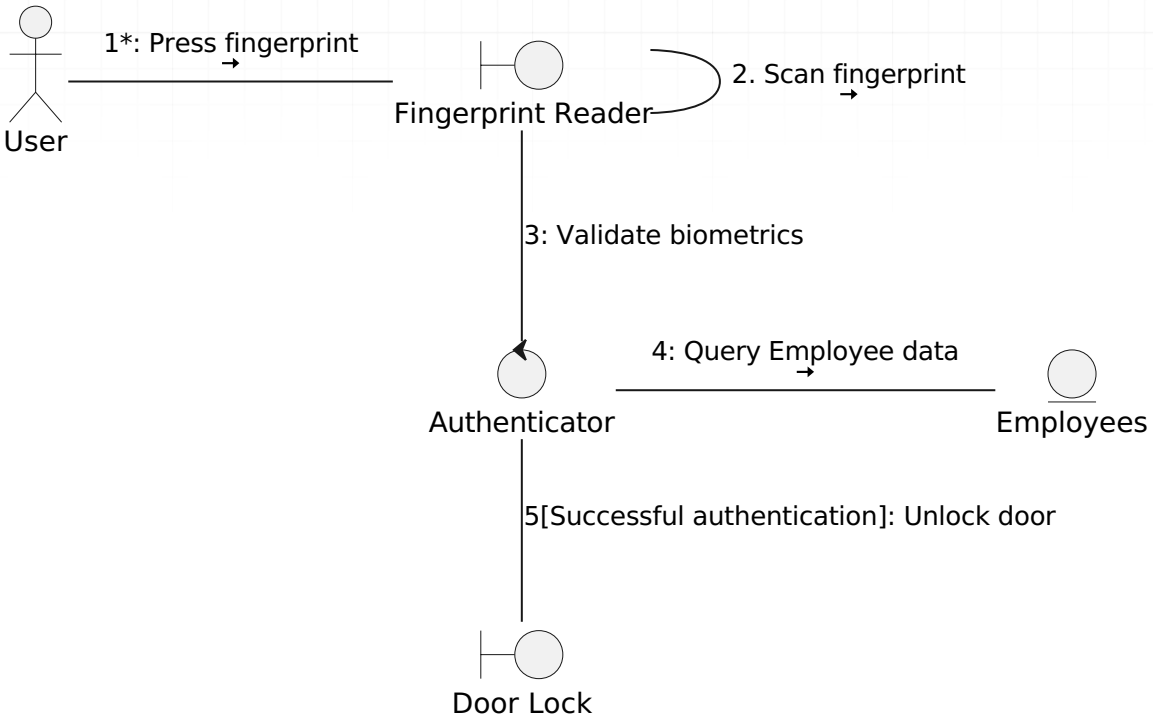
Visualises **timing and order** of interactions between objects:

- **1 Use Case** = 1 Sequence Diagram
- **Vertical Bar** on the object's lifeline indicates when the object is **active**.
- **Synchronous** blocking message is **solid arrow ->**
- **Asynchronous** non-blocking message is **thin arrow ->**
- **Return** non-blocking message is **thin dotted arrow** `<-`

Frames used in Sequence Diagram:

- **sd** frame wrapping entire sequence diagram
- **ref** reference another sequence diagram.
- **loop** repeating interactions.
- **alt** Alternative branch: if-else.
- **opt** Optional branch: if.
- **par** interactions run in parallel.

3.10 Communication Diagram

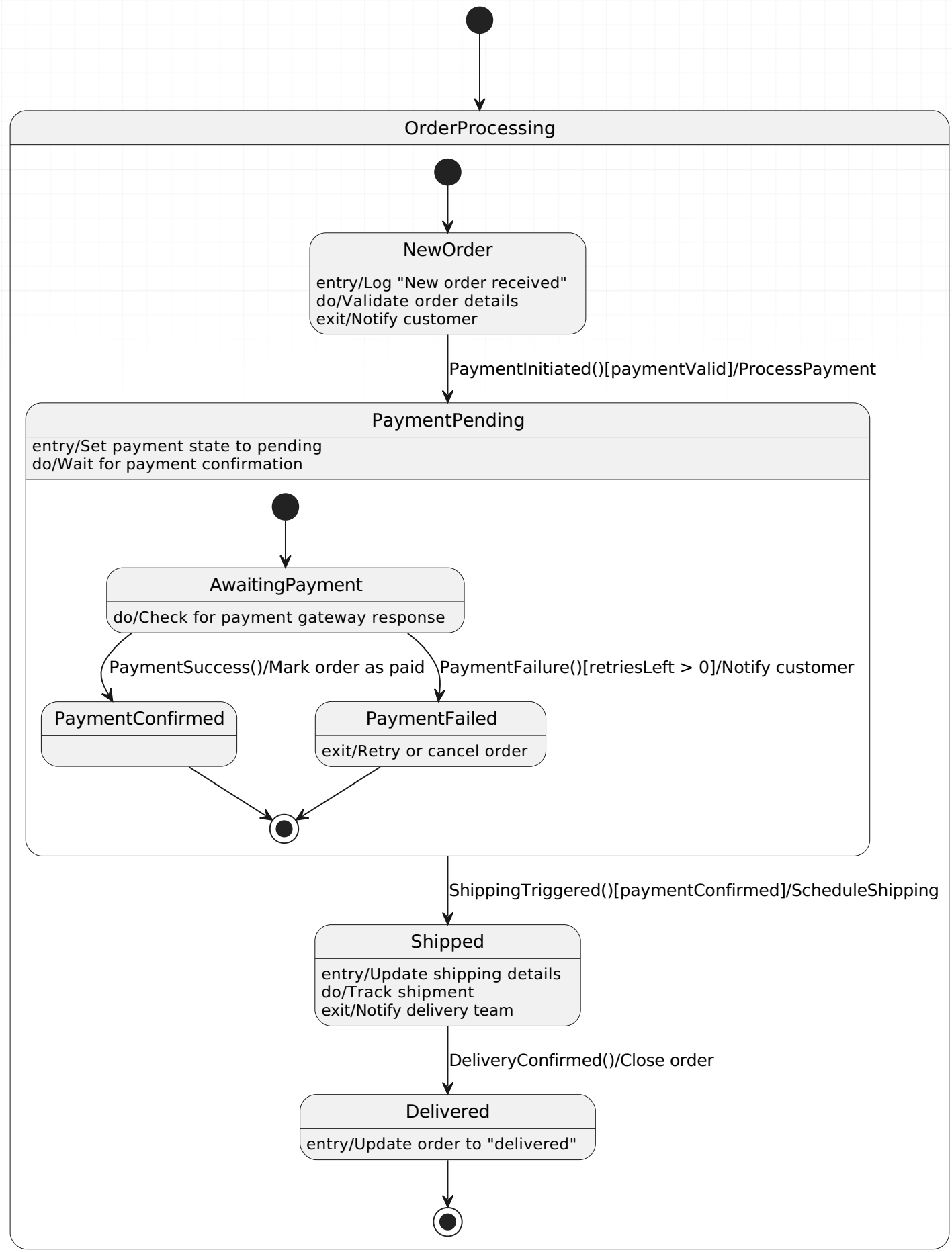


Messages are performed in the **order** of sequence no.:

- `*` **Iteration** indicates that the message may be performed **repeatedly**.
- `[CONDITION]` **Guard** only executes message if `CONDITION` is true.

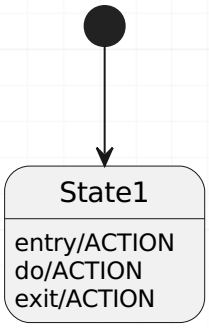
3.11 State Machine Diagram

State Machine Diagram aka Dialog Map Models system **States** and transition **Events** between states.



- **Start** black filled circle is the starting state.
- **End** outer line circle with inner black filled circle is end state.
- **Nesting** States can be nested. eg. `AwaitingPayment` is nested in `PaymentPending`.

3.11.1 State



Actions performed in the State Lifecycle are specified in the body of the state:

- `entry/ACTION` perform `ACTION` **once after** entering the state.
- `do/ACTION` perform `ACTION` **repeatedly** while the state is active.
- `exit/ACTION` perform `ACTION` **once before** exiting is active.

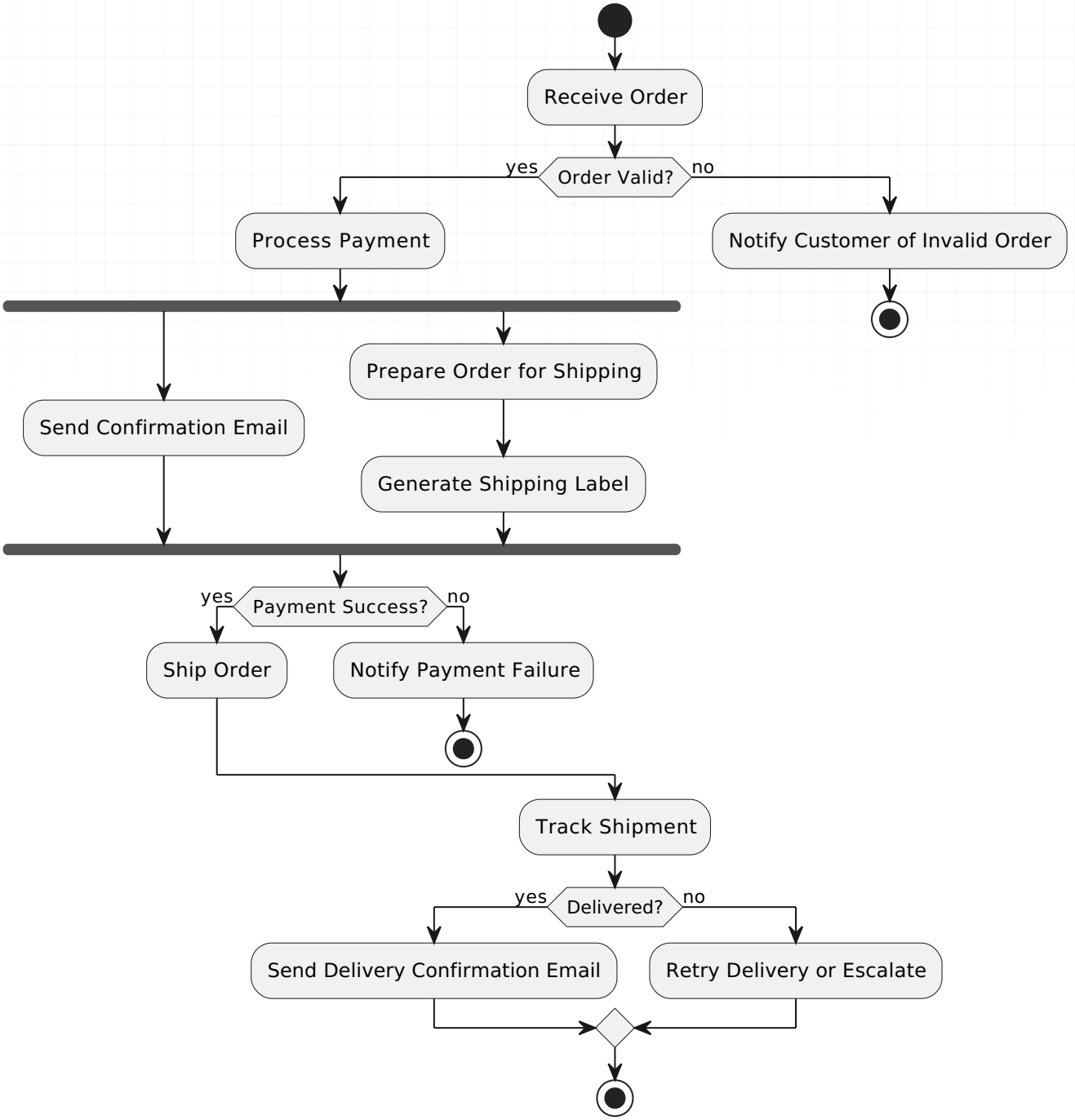
3.11.2 Event

```
EVENT(ARGS,....)[CONDITION]/ACTION
```

Event transitions format:

- `EVENT` the name of the event that caused the state transition
- `ARGS` arguments passed to the event handler. Can be **empty**.
- `CONDITION` **Optional**. Only performs the transition if `CONDITION` is **true**.
- `ACTION` **Optional**. Side effect action performed when transitioning.

3.12 Activity Diagram



- **Start** black filled circle is the starting step.
- **End** outer line circle with inner black filled circle is end step.
- **Decision** Diamond shape indicates a **conditional** decision.
- **Parallel** Solid liine indicates **parallel execution**.

4. Software Processes

4.1 Software Processes

Software Development LifeCycle (SDLC) Activities performed in Software Engineering **common** to all software processes:

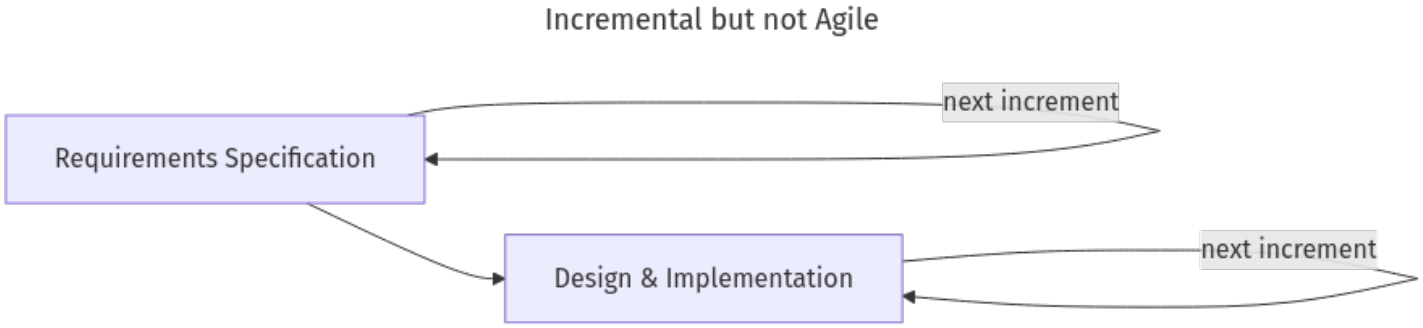
1. **Specification** Requirements Ellicitation, Requirements Analysis
2. **Design & Implementation** System design & implementation
3. **Validation** Testing
4. **Evaluation** Maintenance

4.1.1 Plan Drive vs Agile vs Incremental

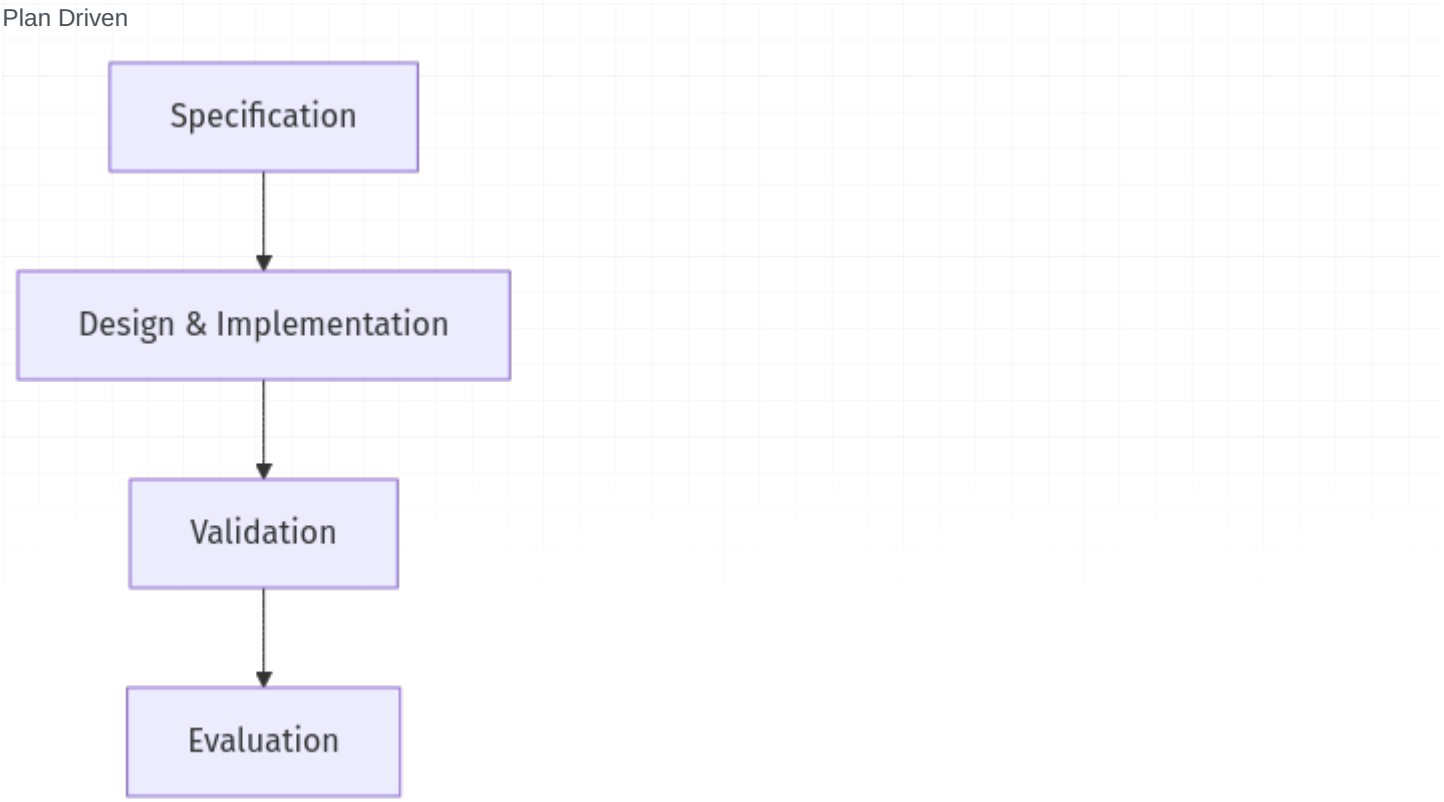
Software Processes can be compared by their characteristics:

- Plan Driven
- Agile: Iterative. Build software in a **cycle of repeating steps**.
- Incremental: build software in **small steps**

> **Agile ≠ Incremental** eg. Development in rigid stages with small step increments within each stage is **incremental but not agile**.

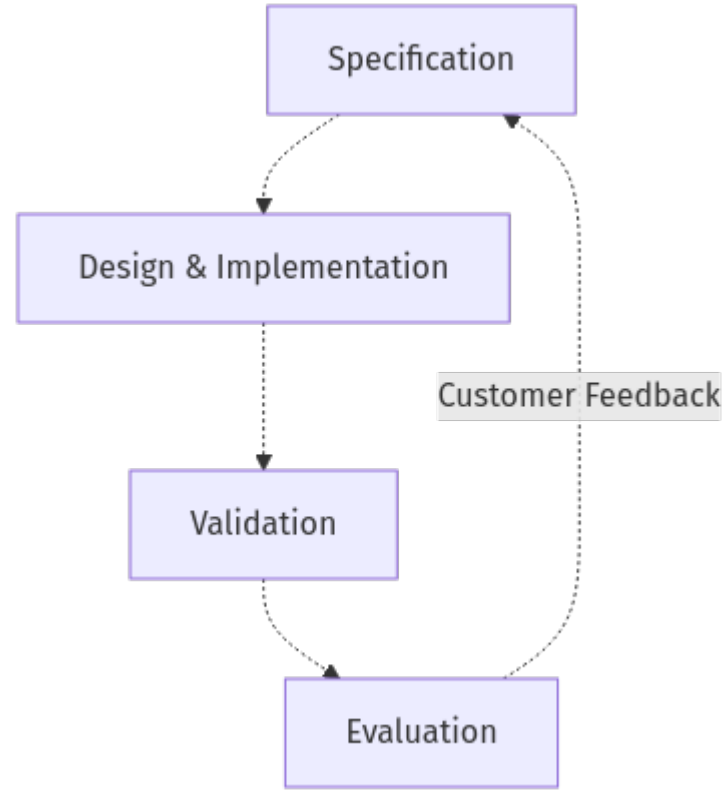


Plan Driven



Plan Driven aka Waterfall **never returns** to previous SDLC Activity.

Agile



Agile performs SDLC Activities **repeatedly** in iterative **sprint cycles**

4.2 Software Process Models

Model	Plan Driven	Agile	Incremental
Waterfall	Yes	No	No
Incremental (Masterplan)	Yes	No	Yes
Incremental (Agile)	No	Yes	Yes
Integration and Configuration	Yes	No	Yes

4.2.1 Waterfall

Waterfall performs SDLC Activities in a series of **rigid stages**:

- **Pros**
  - **Progress** clearly identifiable project progress.
  - **Documentation** up to date documentation.
  - **Large Systems**: suitable for building large systems with multiple components.
- **Cons**
  - **No Return** once a stage is completed.

4.2.2 Incremental

Incremental **interweaves** SDLC activities:

- **Pros**
  - **Flexible** to changing requirements.
  - **Rapid Delivery** of incremental versions.
  - **Custom Feedback** can be obtained for each incremental version.
- **Cons**
  - **Unclear Progress** No clearly defined project end.
  - **Poor System Design** Resulting from accommodating changing requirements over time.
    - Initial system design might **not be optimal** for **new requirements**.
    - Refactoring required to correct system design issues.

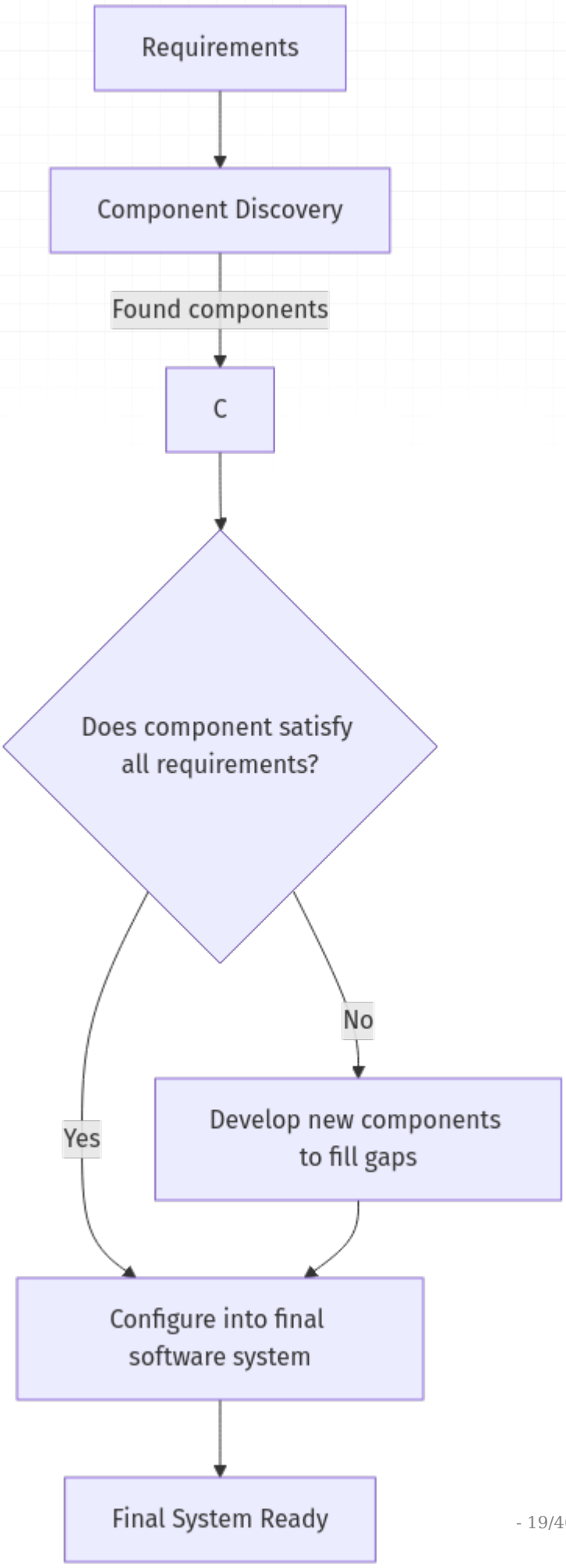
4.2.3 Integration & Configuration

Integration of **externally sourced reusable components** by **configuring** them to work together as a single software system:

- **Pros**
  - **Lower Development Cost** since we can reuse instead of paying developers to write our own.
  - **Faster Delivery** since we don't have to spend time to write our own.
- **Cons**
  - **Gaps in Requirements** components may not satisfy all requirements.
  - **Lack of Control** over reused components project direction.
  - **Limited Support** for reused components.

Reuse Oriented-Software Development

Integration & Configuration Software Process that prioritises the **reuse** of off-the-shelf components where possible:



4.3 Agile

Incremental Iterative Software Process for **rapid** software development:

- **Rapid** software has to **quickly** adapt rapidly changing requirements via frequent new version releases.
- **Code over Docs** Focus on writing code over creating extensive documentation.
  - Reduces overhead of keeping documentation in **sync** with **changing requirements**
- **Usage**
  - **Good** for **small-medium** projects, **experienced** developers.
  - **Bad** for **large projects**, inexperienced developers.

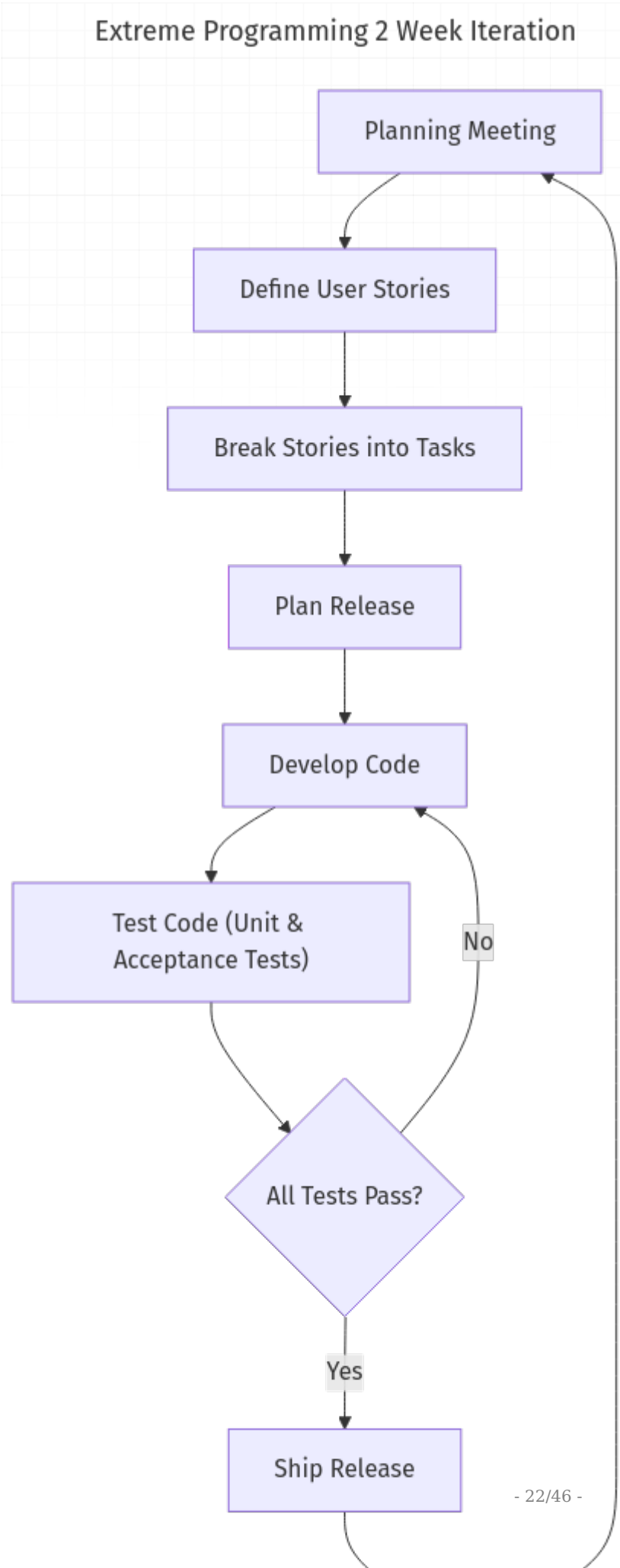
4.3.1 Agile Manifesto

Preferred	**Less Important**
Individuals & Interests	Processes & tools
Working Software	Comphensive Documentation
Customer Collaboration	Customer Negotiation
Responding to Change	Following a Plan

4.3.2 Agile Principles

- **Customer Involvement** throughout the development process to confirm requirements & give feedback.
- **Incremental Delivery** of software via releasing incremental versions.
- **People Not Process** development team should be allowed to follow their **own workflow** rather than a strict process.
- **Embrance Change** system design should be **extensive** since we expect changing requirements.
- **Maintain Simplicity** reduce complexity where possible.

4.4 Extreme Programming (XP)



Agile development method:

- **Continuous Integration** New software build & **tested** several times per day.
- **Rapid Release** 1 version released per 2 weeks.

4.4.1 User Stories

> **Format** As a ROLE, I want to perform ACTION, So that I gain BENEFIT.

Software Requirements as User Stories:

- **Epics** users stories are broken down from Epics (large feature).
- **Card** size limit constraints the **scope** of each User Story.
- **Conversation** include **background information** necessary to understand the User Story.
- **Confirmation** include both functional & non-functional **Acceptance Criteria**

Acceptance Criteria

Defines when the User Story is "done":

- **Intent not Solution** criteria should define "what" that needs to be, not the "how".
- **Implementation Independent** Developers should decide how to implement.
- **High Level** Includes only detail **necessary** to define requirements.

Release Planning

The Customer chooses which User Story to add to the next release.

User Story Tasks

User Stories are further decomposed by development team into implementation tasks:

- **Story Point** no. that gives a **workload** estimate for each task.

4.4.2 Refactoring

Making **code improvements** to "tidy" up the code even when not required:

- **Maintainability** code changes are easy to make due to estensible structure.
- **Understandablty** makes the code **understandable**, reducing need for documentation.

> eg. Reorganise classes, tidying up methods, extracting common code into functions etc

4.4.3 Test Driven Development (TDD)

TDD: write tests **before** code:

- **Clarifies Requirements** writing **Tests as Code** removes any **ambiguity** from requirements. Testing can then be performed by executing test code.
- **Automated** test harness facilitates automatic testing of software. Needed since we are testing frequently in Continuous Integration.
- **User Acceptance Test** customer can develop acceptance tests for requirements in User Stories.

4.4.4 Pair Programming

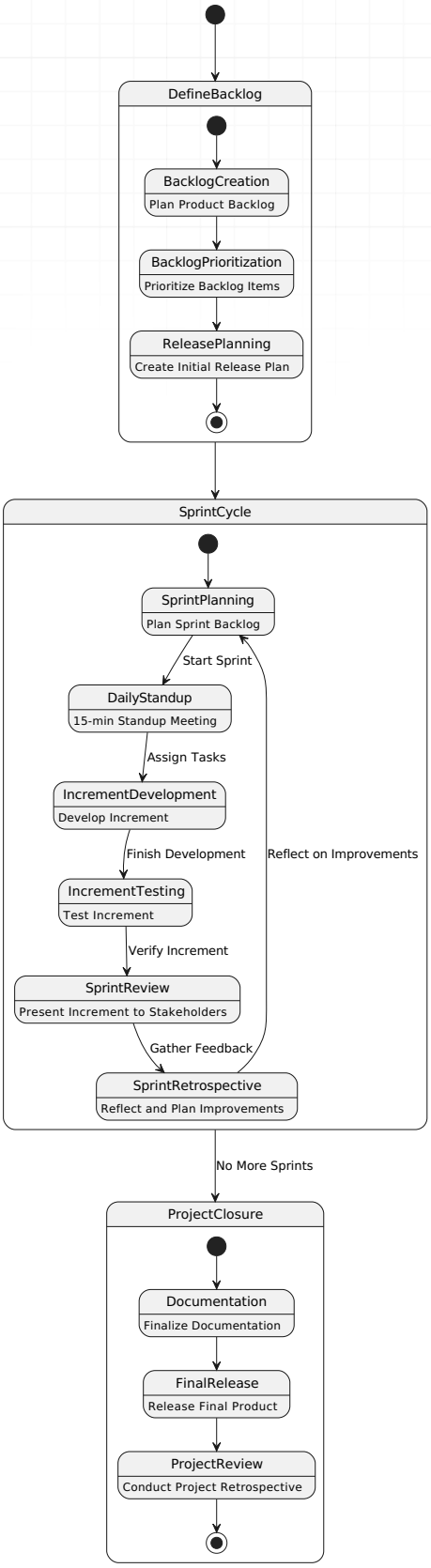
2 developers code together, **alternating** between roles:

- **Roles**:
  - **Developer** writes the code.
  - **Reviewer** checks the code for mistakes.
- **Knowledge Sharing** about the codebase will happen when the developers work together. Reduces the risk of employees leaving and now no one knows how the code works.

4.5 Project Management

Manage Software Engineering project to ensure **software delivery** happens **on time** & **on budget**.

4.6 Scrum



Project Management method for Iterative Software Processes (eg. Agile, XP) with phases:

1. **Initial Phase** Requirements, Design System Architecture.
2. **Sprint Cycle** Develop & release a **incremental software version** in **2-4 weeks**.
3. **Project Closure** Complete documentation, Retrospective.

4.6.1 Scrum Terminology

Term	Definition
Development team	A self-organizing group of up to 7 developers responsible for building software and essential project documents.
Potentially shippable product increment	A software increment delivered from a sprint, ideally in a finished **tested** state requiring no further work for final integration.
Product backlog	A **prioritized single source** of tasks, features, requirements, or supplementary items for the Scrum team to address, expressed as User Stories.
Sprint backlog (Sprint Goal)	A **fixed** (unchanged for entire sprint) set of Product Backlog Items (PBIs) selected to worked on the sprint cycle. **Unfinished** items are returned to the product backlog.
Product owner	A stakeholder responsible for defining and prioritizing features, owns / manages the Product Backlog, **maximising value of product** delivered.
Scrum	A daily **short 15 minute** team meeting to review progress and plan work for the day. **In depth discussion** should be done **outside** of Scrum.
Scrum Master	Ensures the Scrum process is followed, shields the team from distractions by **point of contact** for rest of organisation, removes **blockers** from progress.
Sprint	A **2-4 week** development iteration focused on delivering specific goals.
Sprint Review	A **1 hr × no. Sprint weeks** meeting at the end of a sprint between **Product Owner & External Stakeholders** on the state of the project to generate potential changes to product backlog.
Sprint Retrospective	A **45 min × no. Sprint weeks** meeting at the end of a sprint where the **Internal Stakeholders** reflect on what went well, what didn't, and how to improve next time.

4.6.2 Velocity

Velocity V is the **average** workload as User Story Points p_i a development over n no. of Sprints:

$$V = \frac{\sum_i^n p_i}{n}$$

Sprint Estimate Velocity gives an estimate of workload that **may be completed** per sprint:

- **Sprint Planning** useful for planning **workload allocation** of each sprint.
- **Performance Metric** benchmark for Scrum team performance.

4.6.3 Product Backlog

Good Product Backlogs should be:

- **Detailed** not ambiguous.
- **Emergent** up to date with latest requirements.
- **Estimated** product backlog items have workload estimates as story points.
- **Prioritised** items ranked by priority.

5. Software Testing

5.1 Software Bug

> A software bug is an **error, flaw, failure or fault** in a computer program or system that causes it to produce an **incorrect or unexpected result**, or to behave in **unintended ways**.

Bugs are **unexpected** behaviour in a Software System that **deviates** from requirements:

- **Debugging** identifying the **root cause** of the bug.

5.2 Software Testing

Testing: checking software system for **known bugs**:

- **User Acceptance** reduce risk of failing User Acceptance Test done by Customer.

> Software Testing can be used to show the **presence of bugs** but **never** to show their **absence**.

5.2.1 Black Box & White Box Testing

Types of Software Testing:

- **Black Box** testing done **without** knowledge of code implementation (requirements only).
  - Equivalence Class Testing
  - Boundary Value Testing
- **White Box** testing done **with** knowledge of code implementation (code + requirements).
  - Control Flow Testing

Testing	Test Complexity	Test Thoroughness	Test Coverage
Black Box	Lower	Lower	No
White Box	Higher	Higher	Yes

5.2.2 Unit, Integration, System, Acceptance Testing

- **Unit Test** Test a single **unit** of software (eg. function) in **isolation**.
- **Integration Test** Test **interoperability** of **multiple components**
- **System Test** Test functionality of system **as a whole**.
- **Acceptance Test** Testing done by customer to verify **quality** of software delivered.

5.2.3 Test Case

Components of a Test Case are derived from **verifiable** Requirements

Component	Description
Name	Name of the test case
Path	Location of the test case
Input	Test input
Oracle	**Expected** test output
Log	**Actual** test output

5.2.4 Order of Testing

Order of running Test Cases:

- **Cascading** test cases must be run **in order** as they **depend** on prior test cases.
- **Independent** test cases can be run in **any order**.

Order of Testing	Test Complexity	Parallel Execution
Cascading	Lower	No
Independent	Higher	Yes

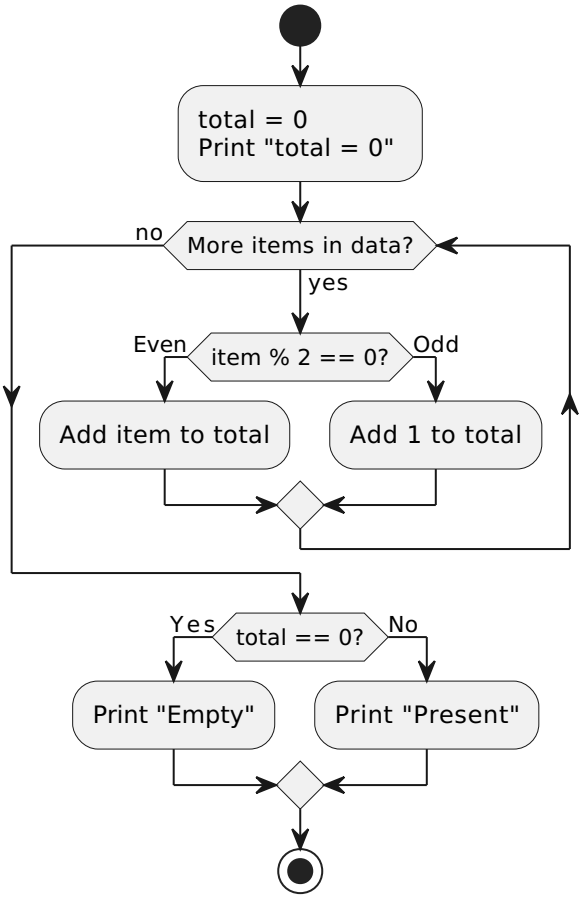
5.3 Control Flow Testing

White Box Testing method that focuses on testing **code paths** identified by its Control Flow Graph (CFG) by choosing inputs that exercise different code paths.

5.3.1 Control Flow Graph (CFG)

```
def process_data(data):
    total = 0
    print("total = 0")
    for item in data:
        if item % 2 == 0:
            total += item
        else:
            total += 1
    if total == 0:
        print("Empty")
    else:
        print("Present")
    return total
```

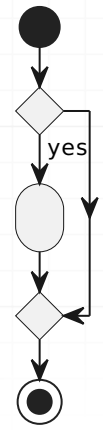
Directed Acyclic Graph that represents the Python code:



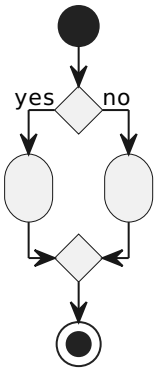
- **Process Block** contains a group of sequential statements.
- **Decision Point** represented by diamond. Can be **binary (2-case)** or **n-nary (n-case)**.

Common Programming Constructs CFG

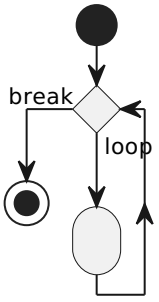
If statement:



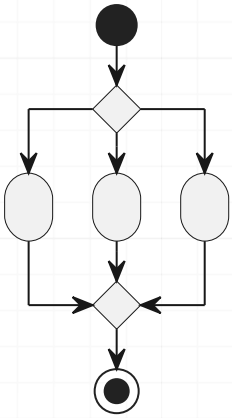
If/Else Statement:



While/For Loop:



Switch Statement:



5.3.2 Test Coverage

Test Coverage Levels:

Level	Coverage	Description
Level 1	100% **statement** coverage	Every **line of code** is tested
Level 2	100% **branch** coverage	Every **decision point branch** is taken
Level 3	100% **basis path** coverage	Every **linearly independent path** is tested
Level 4	100% **path** coverage	Every **path** is tested

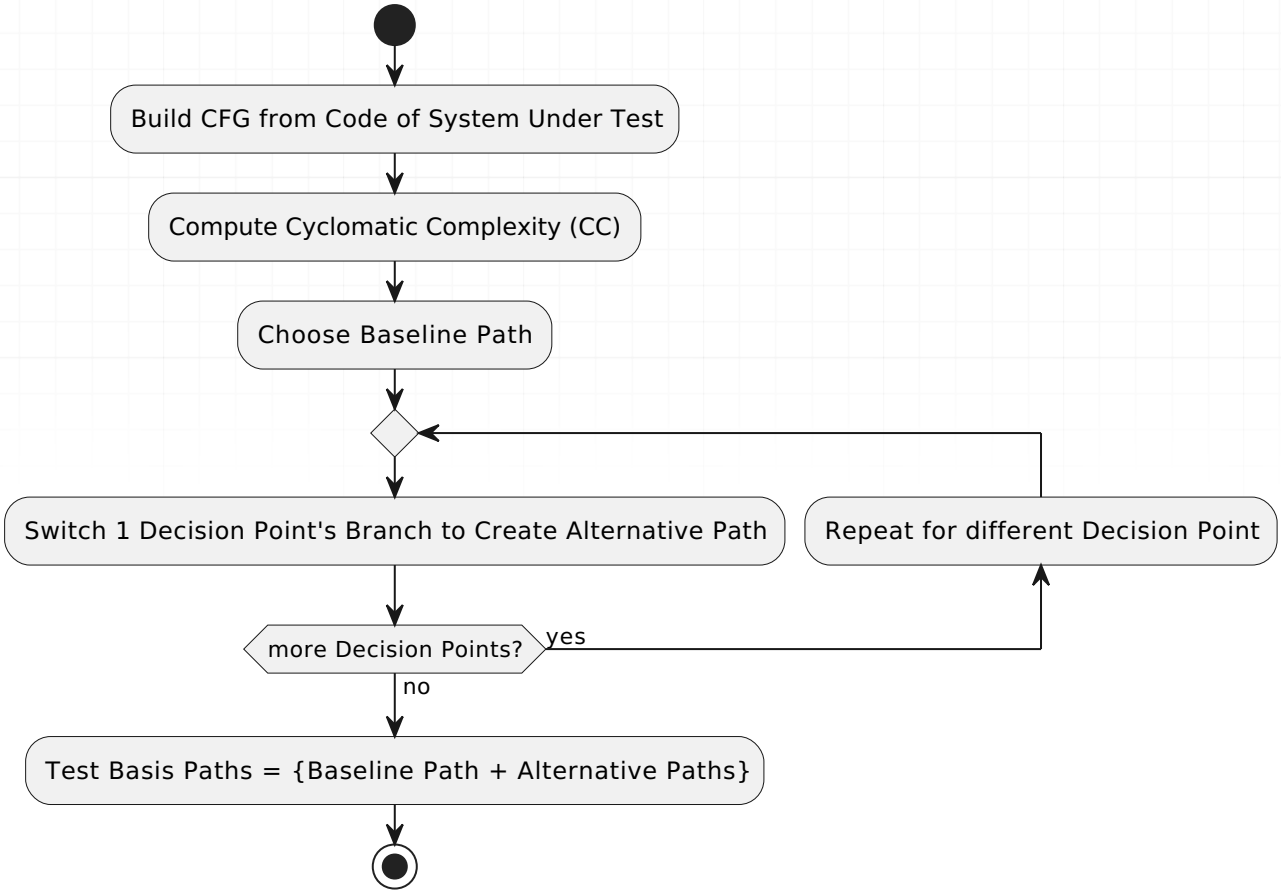
5.4 Total No. of Paths

Total No. of Paths P tested in **Level 4**: 100% **path** coverage testing:

- D is no. of **binary decision points**.
- L is no. of **loop iterations**

$$P = 2^D \times L$$

5.5 Basis Path Testing



Control Flow Testing that focuses on **Level 3** Test Coverage: Testing all **basis paths** aka linearly independent paths in the Control Flow Graph (CFG).

> **Non Unique** Set of basis paths is **not unique**, depends on initial baseline path chosen.

1. Build CFG from code of System Under Test.
2. Compute Cyclometric Complexity (CC) to determine no. of **basis paths** to test.

   > Not all basis paths identified by CC are **feasible** (reachable) in code. Such basis paths are **impossible** to test.

3. Choose a **baseline path** consisting of **false branches** choices for each decision point.
4. Switch **branch** of 1 decision point to create an **alternative path**.
5. Repeat step 4 for all other decision points to obtain **basis paths (Baseline + Alternative Paths)**.
6. Craft **test inputs** to test all identified basis paths.

> **Loops** Considerations when dealing with loops:
>
> - **No Iteration** skip the loop cycle entirely.
> - **1 Iteration** perform 1 iteration of loop cycle and then exit the loop cycle.

5.6 Cyclometric Complexity (CC)

CC computes the total no. of **basis paths** in a CFG:

- Method A: Edges E, Vertices V in CFG:

$$\mathrm{CC} = |E| + |V| + 2$$

- Method B: D No. of **binary decision points** in CFG:

$$\mathrm{CC} = D + 1$$

5.7 Equivalence Class Testing

Black Box testing method that partition possible **input domain** by **expected output** equivalence classes.

Test ≥ 1 set of inputs for **each equivalence class**:

- **Assumption** If the code works for the set of input(s), it should work for **all other inputs** in the same equivalence class.

5.7.1 Equivalence Classes

Equivalence Classes are sets of possible inputs with same **expected output**:

- **Valid (Testing by Contract)** Test successful / happy path for **valid** inputs eg. Login successful.
  - **Multiple Valid Values** test multiple valid input values for each test case.
  - **Exhaustive** Optionally, if the valid input domain is small **all valid inputs** can be tested.
- **Invalid** Test unsuccessful path for **invalid** inputs eg. Bad login credentials.
  - **Single Invalid Value** test only 1 single input value (rest are valid inputs) for each test case to check code correctly rejects even with only 1 invalid input.
- **Defensive Testing** testing both Valid + Invalid inputs.
- **Exception** error case. eg. Unable to connect to Database.

> **Numeric Equivalence Classes are Contiguous**\ Q: Suppose you have **Invalid** output for input range $-5 \leq x \leq -2$ and $3 \leq x \leq 10$.\ What are the Equivalence Classes?
>
> A: 2 Equivalence classes since the input ranges are **non overlapping**:
>
> - **Invalid** $-5 \leq x \leq -2$
> - **Invalid** $3 \leq x \leq 10$
>
> Does **not** apply to **discrete** test inputs since they have no notion of "ranges".

5.8 Boundary Value Testing

Black Box Testing **Heuristic** to select test input values for **numeric** input range $x \in [a, b]$: test **around** boundary values a & b:

- **Just Above** $x + \epsilon > a, \epsilon > 0$
- **At Boundary** $x = a$
- **Just Below** $x - \epsilon < a, \epsilon > 0$

> **Remove Duplicates** Suppose selected test inputs overlaps with the test inputs of another case. Remove the duplication, since it redundant to verify twice with exactly the inputs.

6. System Design

6.1 Software Architecture

High Level overview of how system **components** & **interactions** between them:

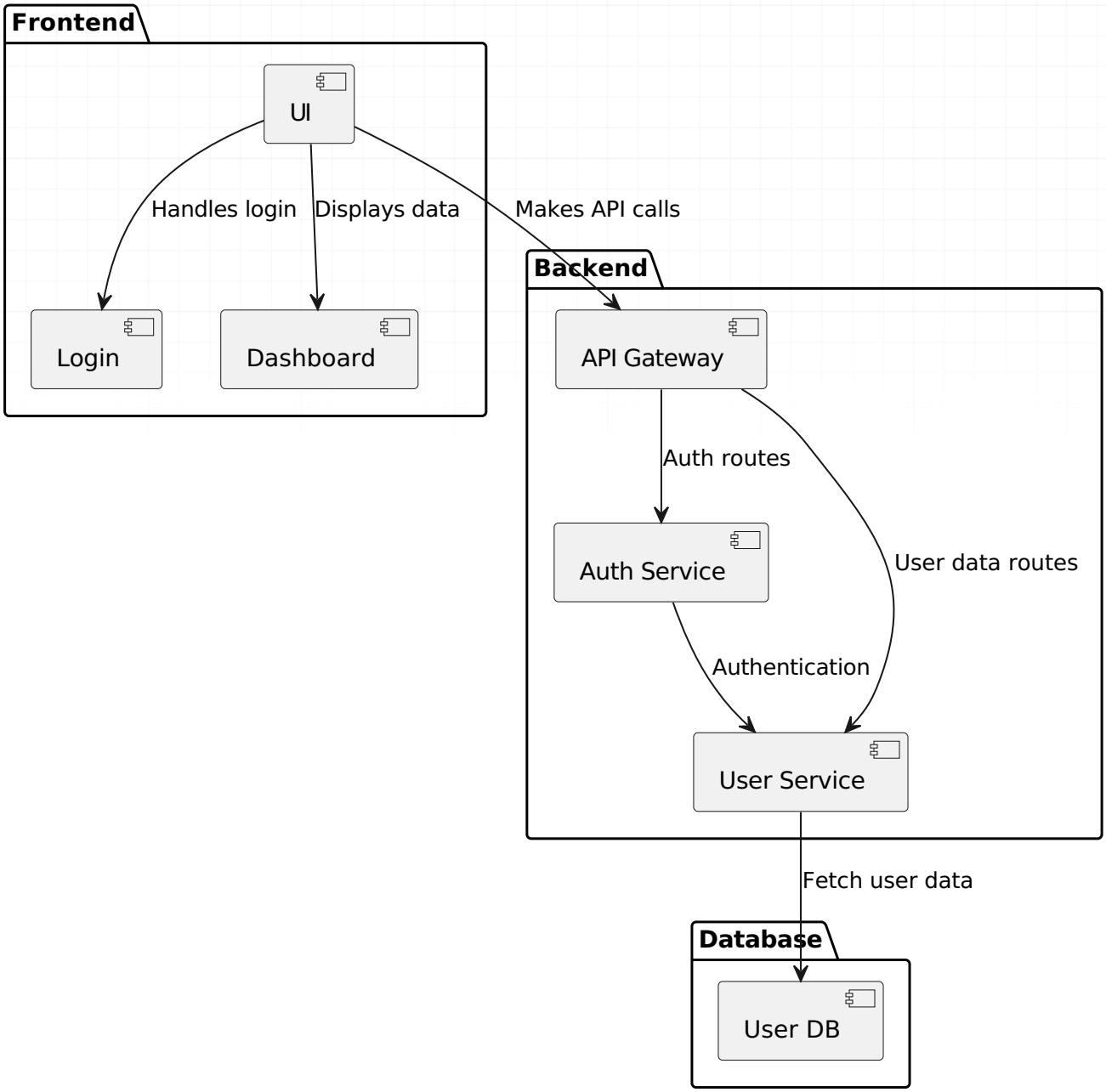
- **Components** units of software system eg. frontend, backend
- **Interactions** communication between components eg. API call.

6.1.1 Software Architecture Motivation

Software Architecture / System Design is needed for:

- **Non Functional Requirements** must be **implemented** in System Design.
- **Larger Software** Lowering **complexity** in larger software systems by organising components.
- **Costs & Schedule** Correcting **bad System Design** gets progressively **more costly** as development progresses and might **delay** timely software release.

6.2 Software Architecture Diagram

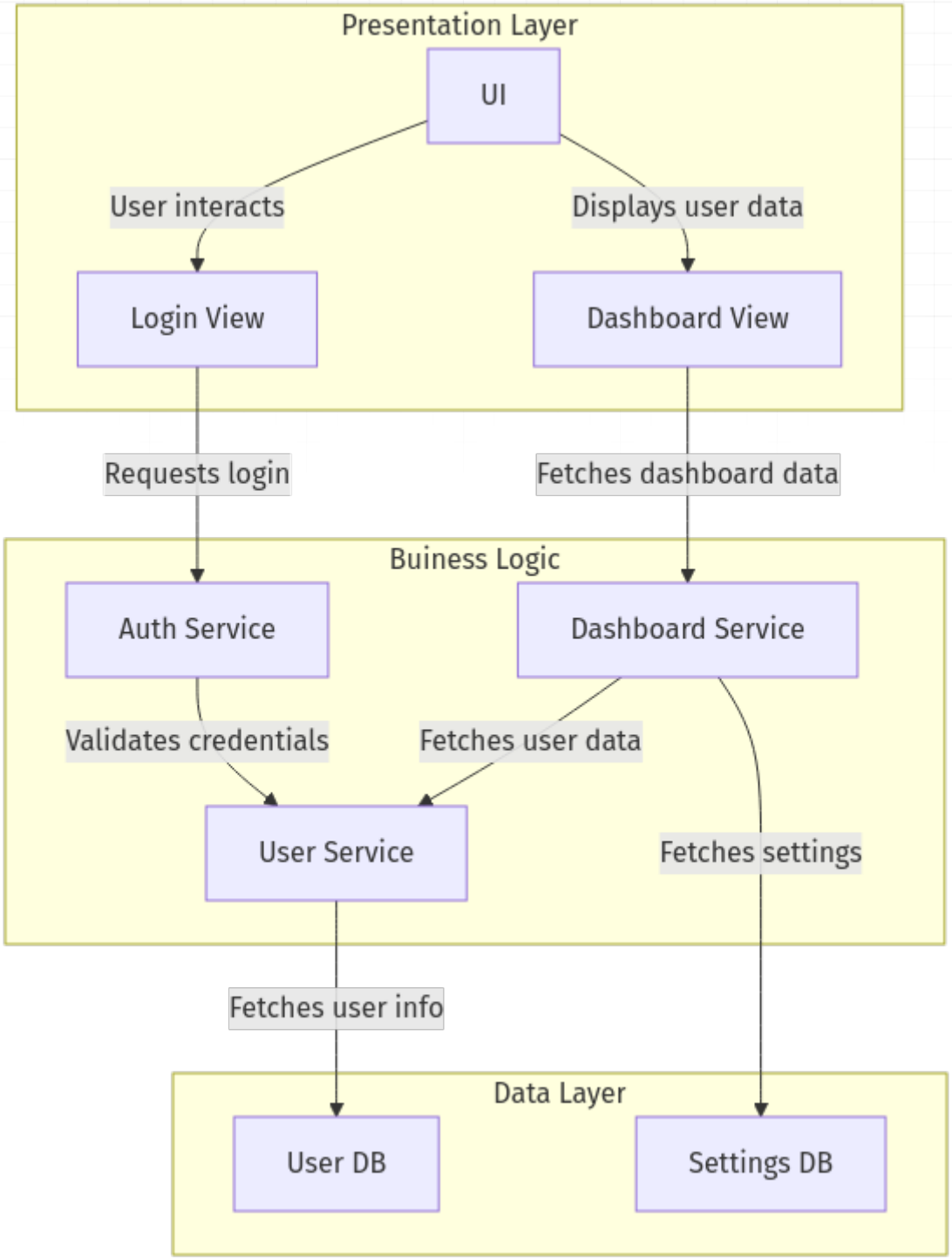


- **Hierarchical** child components (eg. `UI`) are **nested** with parent components (eg. `Frontend`)
- **Abstract** obmits **unnecessary** details.
- **Purposeful** focused on **structure & interactions** of components.

6.3 Software Architecture Style

Pattern (well known solution) of organising components in Software Architecture Designk

6.4 Layered Architecture



Software Architecture Style that groups components into **layers**

- **Upper -> Lower** Upper layers can call lower layers, but **not the other way around**.
- **Pros**
  - **Code Reuse** for components in **lower layers**.
  - **Extensibility** for components in **upper layers**.
- **Cons**
  - **Performance** overhead.
  - **Hard to Design** which layer a component should belong to could be **unclear**.

6.5 Object Design

Object design: how to design Objects / Classes in Class Diagram?

- **Interface Specification** defining **boundaries** between components eg. operations, arguments, properties.
- **Identifying Reuse** leveraging **existing** libraries & Design Patterns.
- **Restructuring** refactoring done to preserve code maintainability.
- **Optimisation** improve speed or memory performance.

6.6 Design Patterns

Existing **solution** to a **design problem**:

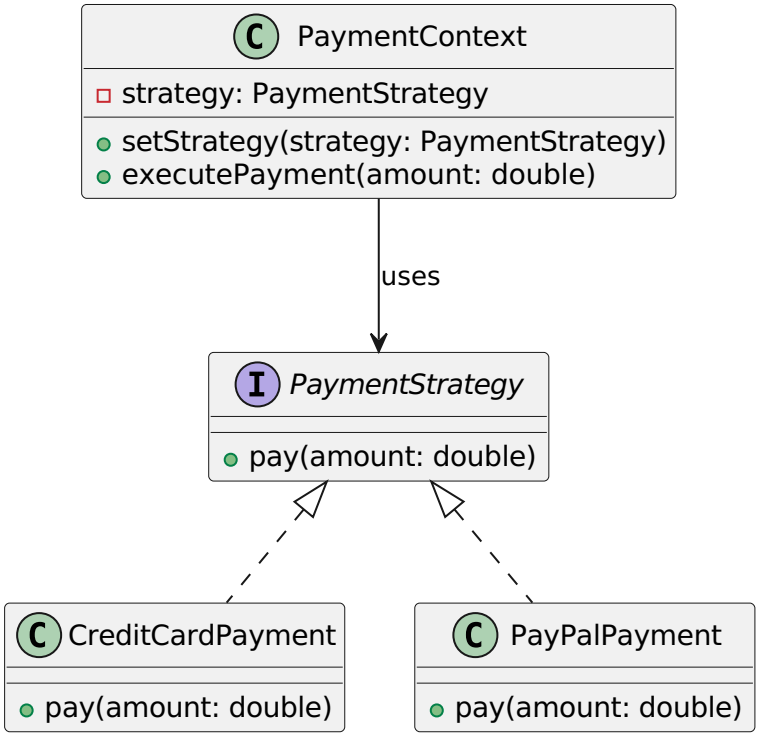
- **Name** terminology used to talk about the design pattern.
- **Problem** that design pattern attempts to resolve.
- **Solution** how to implement the design pattern.
- **Consequences** trade offs in implementing the design pattern.

6.6.1 Types of Design Patterns

Design Patterns classified by the problem they solve:

- **Creation Patterns**: how to **create** objects?
- **Structural Patterns**: how to **compose** (combine) objects?
- **Behavioural Patterns**: how to **implement** specific **behaviour** with objects?

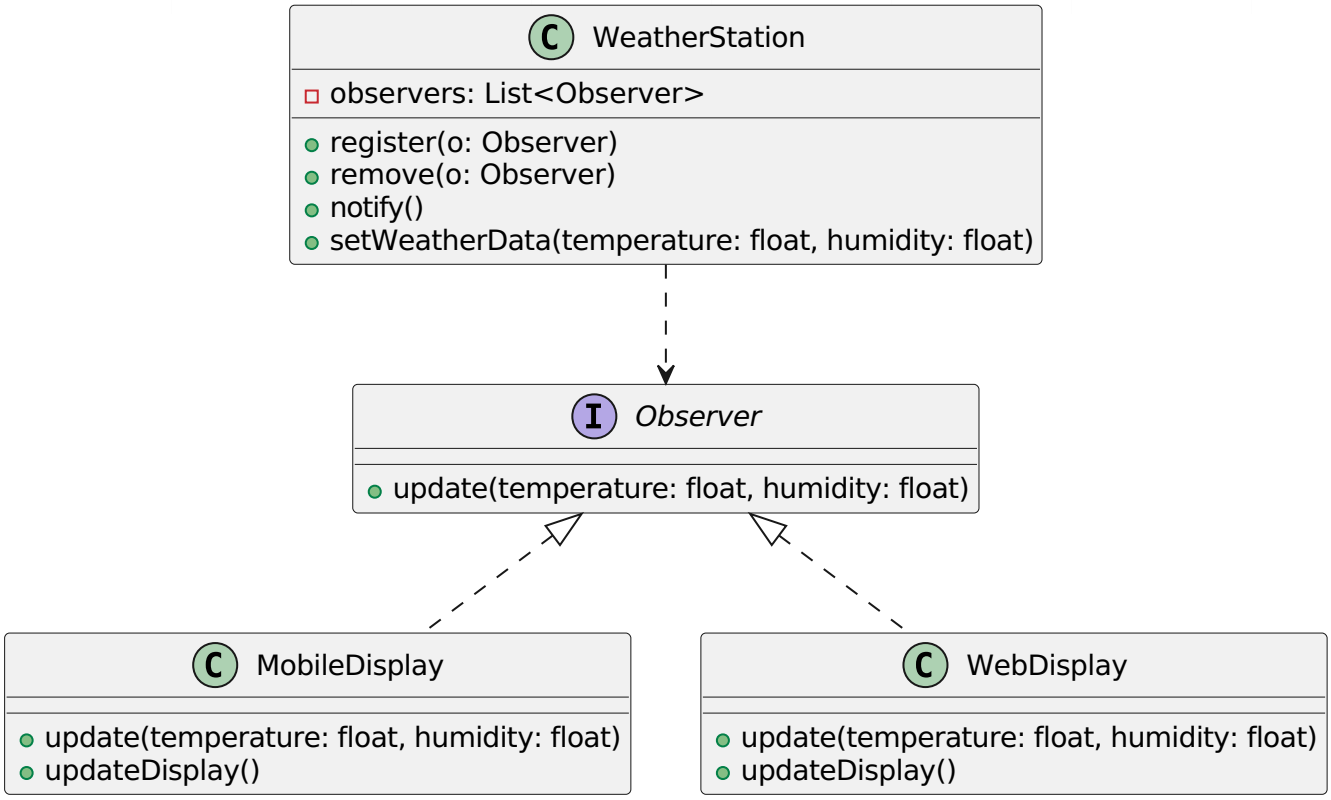
6.7 Strategy Pattern

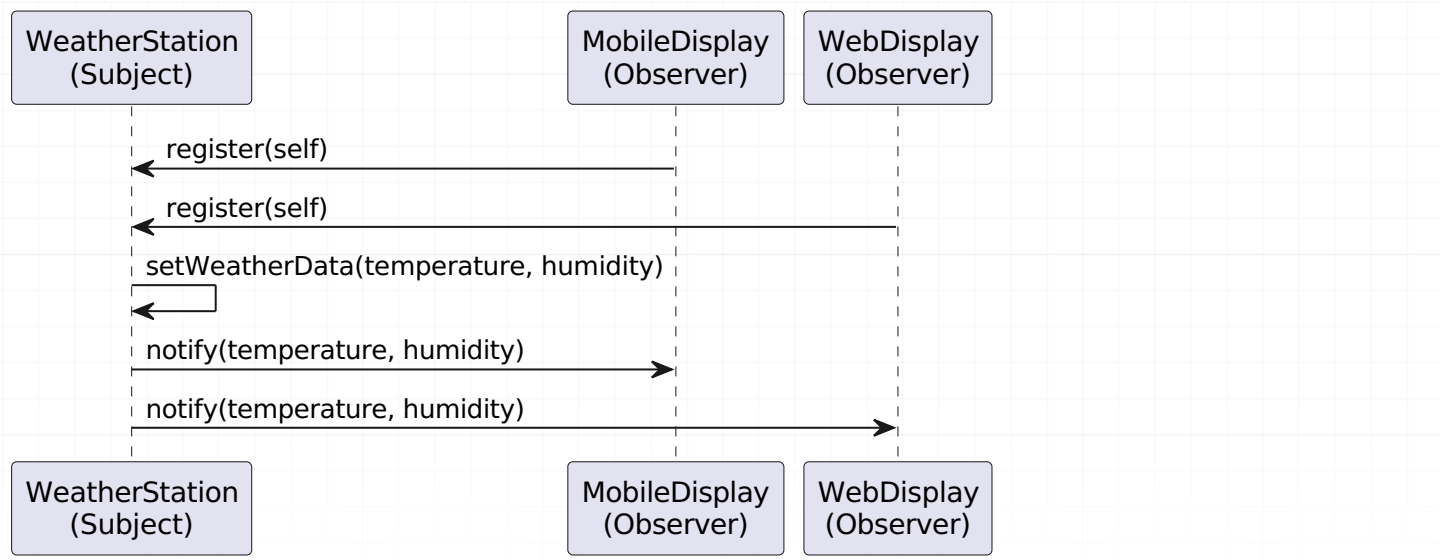


Strategy is **Behavioural Pattern**:

- **Problem** a set of algorithms (eg. `CreditCardPayment`, `PayPalPayment`) should be **interchangeable** (eg. `PaymentStrategy`)
- **Solution** implement algorithms behind a **common interface**.
- **Consequences**
  - **Pros**:
    - **Encapsulation** hides implementation details.
    - **Extensiblity** ie. code dependent on `PaymentStrategy` does **not need to change** to add a new `PaymentStrategy` implementation.
    - **Hot Swappable** Software behaviour change at runtime by swapping classes (eg. `CreditCardPayment` -> `PayPalPayment`).
  - **Cons**: Increases complexity.

6.8 Observer Pattern





Strategy is **Behavioural Pattern**:

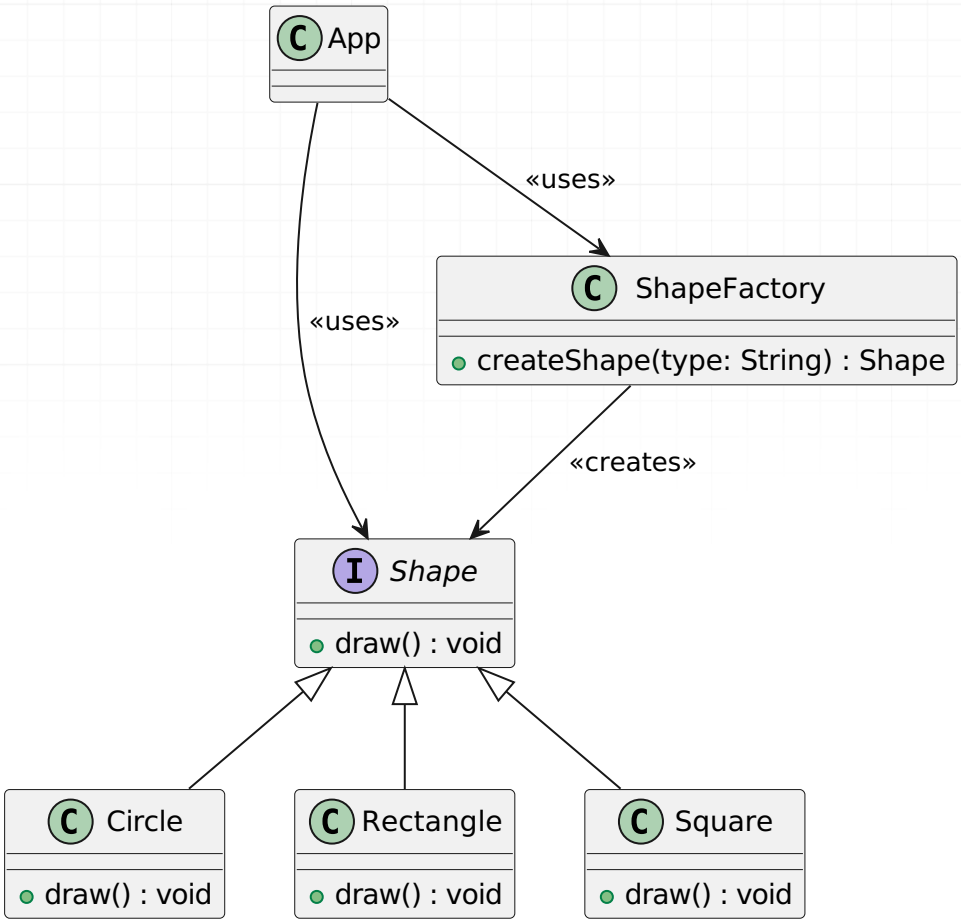
- **Problem** Broadcast: Update **≥1** observer objects when a subject object **changes** without polling.
- **Solution**
  1. Observers (eg. `MobileDisplay`, `WebDisplay`) `register()` themselves with the Subject (`WeatherStation`)
  2. Subject calls `notify()` on Observers to notify them of changes.
  3. Observer obtains changes from Subject and does what it needs to do.
- **Pros**
  - **Loose Coupling** subject is **not dependent** on Observer implementations. Observer does not have to **poll** Subject constantly for changes.
- **Cons**
  - **Performance** overhead.
  - **Complexity** increases code complexity.

6.8.1 Change Propagation

How changes are propagated from Subject to Observer:

- **Pull Approach** changes "pulled" by observer via calling methods on the Subject (call back).
  - **2-Way** communication (Subject -> Observer, Observer -> Subject) **increased** coupling.
  - **Selective Changes** each Observer can retrieve only the changes it needs by selective calling subject.
- **Push Approach** changes "pushed" by subject via `notify(changes)` parameter.
  - **1-Way** communication (Subject -> Observer) **reduced** coupling.
  - **All Changes** same set of changes are pushed to all observers.
- **Push + Pull** combines both approaches by having subject **push** minimal changes and the observer **pull** any additional changes that it requires.

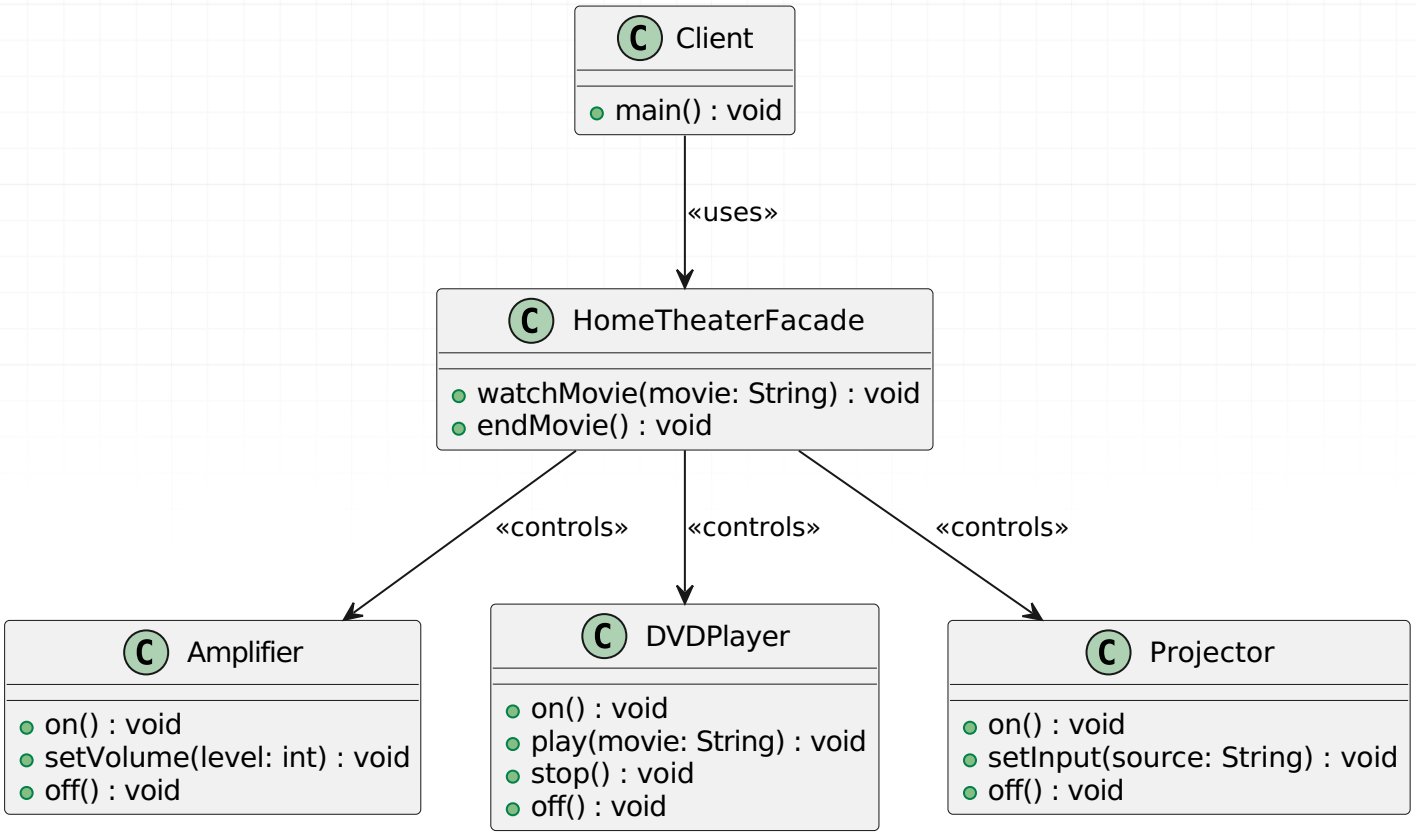
6.9 Factory Pattern



Factory is **Creational Pattern**

- **Problem** How to encapulate & defer creation of an implementation (eg. `Circle`, `Square`) of interface (eg. `Shape`)?
- **Solution** App uses Factory (eg. `ShapeFactory`) to create an implementation (eg. `Circle`) of the interface (eg. `Shape`).
- **Pros**
  - **Encapulation** hides creation logic.
  - **Extensiblity** in creation logic, adding new implementations to interface.

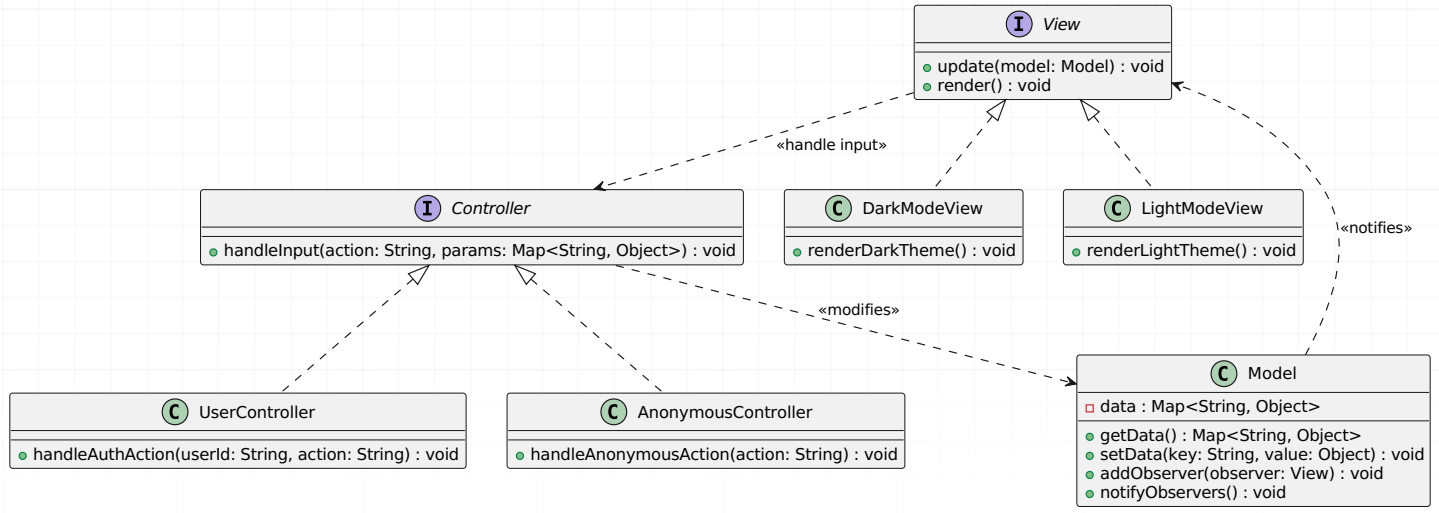
6.10 Facade Pattern



Facade is **Structural Pattern**

- **Problem** How can **reduce** dependencies on **multiple** objects (eg. `Amplifier`, `DVDPlayer`, `Projector`)?
- **Solution** Group interfaces into a single Facade (eg. `HomeTheaterFacade`) and depend on the Facade instead.
- **Pros**
  - **Ease of Use** code has to call 1 Facade instead of juggling multiple objects.
  - **Reduces Dependencies** on multiple objects to 1 Facade.
  - **Decoupling** code from multiple objects.
- **Cons**
  - **Extra Work** to replicate functionality behind Facade.
  - **Complexity** due to indirection.
  - **Performance** overhead.

6.11 Model View Controller (MVC)



Software Architecture Style for user facing **interactive** systems that **separates**:

- **Data (Model)** data structure & logic to manipulate data.
- **Presentation (View & Controller)**
  - **View** presents data to the user.
  - **Controller** handles user actions.

> **MVC ≠ Boundary-Control-Entity**
>
> - Model = Control + Entity
> - View + Controller = Boundary
>
> **Not Layered Architecture** Cyclic Dependency between Model, View, Controller makes it **impossible** to separate into **clear layers** required in layered architecture.

6.11.1 MVC Design Patterns

Design Patterns used in MVC:

- Strategy Pattern
  - **View** eg. Light Mode & Dark Mode presentation strategies.
  - **Controller** eg. Anonymous vs logged-in User functionality.
- Observer Pattern
  - **View** observes changes on the **Model**, which then notifies View to reflect changes.

6.11.2 MVC Tradeoffs

Pros

- **Loose Coupling** via indirection (ie. View makes changes to Model via Controller).
- **Simultaneous Development** of Model, View, Controller independently.
- **High Cohesion** related components are grouped together (eg. all Models are grouped).

Cons

- **Incompability** Model, View, Controller no longer interoperate together.
- **Complexity** due to additional indirection.

7. Software Maintenance

Software Maintenance is

> The process of **modifying** a software system after delivery to **correct faults, improve performance or other attributes**, or **adapt to a changed environment**.

7.1 Software Maintenance Problems

- **Unstructured Code** spaghetti code, bad naming, deep code block nesting etc.
- **Insufficient Knowledge** about the codebase, problem domain.
- **Insufficient Documentation** missing, out of date, insufficient documentation.

7.2 Software Mantainance Activities

- **Fault Repairs (24%)** fixing bugs, vulnerabilities.
- **Environmental Adaptation (19%)** changing software runtime environment eg. Windows Software to run on Linux OS.
- **Functionality Addition / Modification (58%)** modifying system to satisfy **new requirements**.

8. Software Refactoring

Making **improvements** to codebase **without changing functionality** to improve **structure**, reduce **complexity** and ease of code **understanding**.

8.1 Code Smells

Refactoring removes **Code Smells**:

- duplicate code: need to correct in **multiple places** if bugged.
- long methods / functions / classes: increased **complexity** of code.
- temporary variables: with **meaningless names** eg. `a`, `b`.
- switch statement: missing `default` case, missing `break`.
- lazy class: runtime initialisation
- data redundnancy / duplication
- tight coupling