Software Engineering

Design
- Code is the ‘design’ (but at a lower level than the design that occurs during coding).
- Manufacturing is what is done by the compiler (fully automated)

Pros
- Making things useful to other people
- Always learning

Cons
- Need for perfection when developing software
- Requiring some amount of tedious, painstaking labor
- High dependence on others
- Seemingly never ending effort required for testing and debugging software
- Fast moving industry making our work obsolete quickly
- Other people set one's objectives, provide one's resources, and furnish one's information

---

Programming Paradigms

Object-Oriented Programming

Introduction

`[W1.1a] Paradigms → OOP → Introduction → What`

- OO guides us in how to structure the solution
- OO is modeled after how the objects in real world work
  - Groups operations and data into modular units called objects
  - Combine objects into structured networks to form a complete program
  - Objects and object interactions are the basic elements of design
- OO is a **higher level** mechanism than the procedural paradigm.
  - An **object** is an **abstraction** over data-related functions → OO works at higher level
  - Procedural languages work at the simple data structures (e.g., integers, arrays) and functions level

Some programming languages support multiple paradigms.	
Programming Paradigm	**Programming Languages**
Object-oriented programming	**Java**, JavaScript, Python, C++
Procedural Programming	**Java** **(not recommended)**, JavaScript, Python, C
Functional Programming	**Java** **(limited)**, JavaScript, Python, F#, Haskell, Scala
Logic Programming	Prolog

Objects

OOP concept	Example
OOP views the world as a *network* of *interacting objects*.	Objects usually match nouns in the description. *Tom, SE Textbook (Book for short), Text, (possibly) Highlighter*
An object in Object-Oriented Programming (OOP) has **state (data)** and **behavior (operations on data)**, similar to objects in the real world.	**Object** / **Examples of state** / **Examples of behavior** `Tom` / memory of the text read / read `Book` / title / show text `Text` / font size / get highlighted
Every object has: ***Interface*** - specifies how other objects interact with it, through which other objects can interact with it ***Implementation*** - internals of the object that facilitate those interactions, supports the interface but may not be accessible to the other object	**Object** / **Examples of interface** / **Examples of implementation** `Tom` / receive reading assignment / understand/memorize the text, read, remember the reading assignment `Book` / turn page / how pages are bound to the spine `Text` / read / how characters/words are connected together or fixed to the book
Objects interact by sending messages.	• `Tom` sends message `turn page` to the `Book`. • `Tom` sends message `show text` to the `Book`. When the `Book` shows the `Text`, `Tom` sends the message `read` to the `Text` which returns the text content to `Tom`. • `Tom` sends message `highlight` to the `Highlighter` while specifying which `Text` to highlight. Then the `Highlighter` sends the message `highlight` to the specified `Text`.

World	Sender	Receiver	Message	Response	State Change
Real	You	Adam	"What is your name?"	"Adam"	-
Real	as above	Beth	as above	"Beth"	-
Real	You	Pen	Put nib on paper and apply pressure	Makes a mark on your paper	Ink level goes down
Virtual	Main	Calculator (current total is 50)	add(int i): int i = 23	73	total = total + 23

OOP solutions try to create a similar object network inside the computer's memory so that a similar result can be achieved programmatically.

OOP does not demand that the virtual world object network follow the real world exactly.

A virtual world simulation of the above scenario can omit the `Highlighter` object. Instead, we can teach `Text` to highlight themselves when requested.

Object	Real World?	Virtual World?	Example of State (i.e. Data)	Examples of Behavior (i.e. Operations)
Adam	✓	✓	Name, Date of Birth	Calculate age based on birthday
Pen	✓	-	Ink color, Amount of ink remaining	Write
AgeList	-	✓	Recorded ages	Give the number of entries, Accept an entry to record
Calculator	✓	✓	Numbers already entered	Calculate the sum, divide
You/Main	✓	✓	Average age, Sum of ages	Use other objects to calculate

OBJECTS AS ABSTRACTIONS (higher level concepts)

OOP: Abstraction Mechanism	Example
Abstract away the lower level details (e.g. of data formats, method implementation) and work with **bigger granularity entities** (at the level of objects).	📦 You can deal with a `Person` object that represents the person Adam and query the object for Adam's age instead of dealing with details such as Adam's date of birth (DoB), in what format the DoB is stored, the algorithm used to calculate the age from the DoB, etc.

ENCAPSULATION OF OBJECTS

`[W1.1d] Paradigms → OOP → Objects → Encapsulation of objects`

An object is an **encapsulation** of some data and related behavior in terms of

PACKAGING	An object packages data and related behavior together into one self-contained unit
INFORMATION HIDING	The data in an object is hidden from the outside world and are only accessible using the object's interface.

Classes

A **class** contains instructions for creating a specific kind of **objects**.

CLASS-LEVEL MEMBERS

Class-level members (`static` members)	Accessed using the class name rather than an instance of the class
Class-level attributes	Variables whose value is shared by all instances of a class
Class-level methods	Called using the class instead of a specific instance

system: ToDo manager class: `Task` variable: `totalPendingTasks`	**[CLASS-LEVEL]** attribute `totalPendingTasks` should be maintained centrally and shared by all `Task` objects rather than copied at each `Task` object.
system: eLearning system class: `Course` variable: `totalStudents`	**[** variables managed at **INSTANCE LEVEL]** Value varies from instance to instance. e.g., `totalStudents` for one `Course` object will differ from `totalStudents` of another

ENUMERATIONS `[W1.1j] Paradigms → OOP → Classes → Enumerations`

- **ENUMERATION**: A fixed set of values that can be considered as a data type
- Enumeration reduces the risk of invalid values getting assigned to a variable
- Useful when using a regular data type such as int or String would allow invalid values to be assigned to a variable

Enumeration type: `Priority`	Three values: `HIGH`, `MEDIUM`, `LOW`
By defining the enumeration type, a variable of type Priority will never be assigned an invalid value because the compiler is able to catch such an error.	
Vs: You can declare the variable priority as of type int and use only values 2, 1, and 0 to indicate the three priority levels. However, this opens the possibility of an invalid value such as 9 being assigned to it.	

Inheritance

Define a new class (`subclass`) based on an existing class (`superclass`)

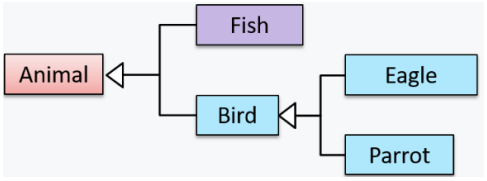
- `superclass` : more general aka base class, parent class
- `subclass` : more specialized aka derived class, child class, extended class

Apply inheritance on a group of similar classes → Common parts among classes extracted into more general classes.

`is a` relationship

!! NOTE: Inheritance does NOT necessarily mean a subtype relationship exists.

- derived class is a subtype of the base class
- base class is a super-type of the derived class

Inheritance hierarchies (inheritance trees) Parrot is a Bird and it is an Animal.	

Multiple Inheritance

A class inherits directly from multiple classes

Allowed: Python C++	Not allowed: Java C#

OVERLOADING `[W1.2b] Paradigms → OOP → Inheritance → Overloading`

Multiple methods with the same name but different `type signatures`

Type Signature

- **Type sequence** and order of **parameters**
- EXCLUDES return type, parameter names

OVERRIDING `[W1.2c] Paradigms → OOP → Inheritance → Overriding`

- A subclass changes the behavior inherited from the parent class
- Overridden methods have the
  - same method name
  - same type signature (parameter sequence)
  - same (or a subtype of the) return type

ABSTRACT CLASSES AND METHODS

`Abstract class`

- Cannot be instantiated
- Can be subclassed
- Represent commonalities among its subclasses

`Abstract method`

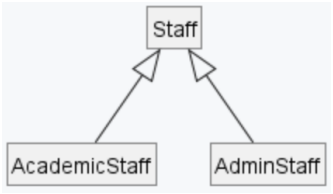
- Method signature without a method implementation
- When is is not possible to implement a method at the class level to fit all subclasses
- **A class that has an abstract method becomes an abstract class**
  - class definition is incomplete (due to missing method body)
  - not possible to create objects using an incomplete class definition

INTERFACES

- An interface is a **behavior specification** i.e. A collection of method specifications
  - Method specification: method signature without implementation
- A **class** `implementing` an **interface**
  - `is-a relationship` (like in class **inheritance**)
    - AcademicStaff `implements` SalariedStaff ⇒ AcademicStaff is a SalariedStaff.
      - An AcademicStaff object can be used anywhere a SalariedStaff object is expected e.g. SalariedStaff ss = new AcademicStaff().
  - **class** must implement all the methods specified by the **interface**

SUBSTITUTABILITY `[W1.3g] Paradigms → OOP → Inheritance → Substitutability`

- Every instance of a subclass is an instance of the superclass, but not vice-versa.
- Inheritance allows **substitutability**: the ability to substitute a child class object where a parent class object is expected

	`AcademicStaff` is an instance of a `Staff`. `AcademicStaff` is substitutable as a `Staff` object. `Staff staff = new AcademicStaff();`	staff is declared as a `Staff` type and therefore its value may or may not be of type `AcademicStaff`23. `Staff staff;` `AcademicStaff academicStaff = staff;` **// Not OK**

DYNAMIC AND STATIC BINDING `[W1.1a] Paradigms → OOP → Introduction → What`

Dynamic Binding

- Method calls in code are resolved at **runtime** (method call is executed), rather than at compile time
- **Overridden methods (subclass changes the behavior inherited from the parent class)** are resolved using **dynamic binding**
  - ⇒ resolves to the implementation in the actual type of the object

Consider the code below. The declared type of `s` is `Staff` and it appears as if the `adjustSalary(int)` operation of the `Staff` class is invoked.

```
void adjustSalary(int byPercent) {
    for (Staff s: staff) {
        s.adjustSalary(byPercent);
    }
}
```

However, at runtime `s` can receive an object of any subclass of `Staff`. That means the `adjustSalary(int)` operation of the actual subclass object will be called. If the subclass does not override that operation, the operation defined in the superclass (in this case, `Staff` class) will be called.

Static Binding

- When a method call is resolved at **compile time**
- **Overloaded methods (same name, different type signatures)** are resolved using **static binding**

Similarly, the `calculateGrade` method is overloaded in the code below and a method call `calculateGrade("A1213232")` is bound to the second implementation, at compile time.

```
void calculateGrade(int[] averages) { ... }
void calculateGrade(String matric) { ... }
```

Polymorphism

Write code targeting superclass objects, use that code on subclass objects, and achieve possibly different results based on the actual class of the object.

Assume classes `Cat` and `Dog` are both subclasses of the `Animal` class. You can write code targeting `Animal` objects and use that code on `Cat` and `Dog` objects, achieving possibly different results based on whether it is a `Cat` object or a `Dog` object. Some examples:	• Declare an array of type `Animal` and still be able to store `Dog` and `Cat` objects in it. • Define a method that takes an `Animal` object as a parameter and yet be able to pass `Dog` and `Cat` objects to it. • Call a method on a `Dog` or a `Cat` object as if it is an `Animal` object (i.e., without knowing whether it is a `Dog` object or a `Cat` object) and get a different response from it based on its actual class e.g., call the `Animal` class's method `speak()` on object `a` and get a `"Meow"` as the return value if `a` is a `Cat` object and `"Woof"` if it is a `Dog` object.

HOW `[W1.3i] Paradigms → OOP → Polymorphism → How` Java - Polymorphism

To achieve polymorphism: **substitutability**, **operation overriding**, and **dynamic binding**

To achieve polymorphism		**Polymorphism**
Substitutability	Write code that expects objects of a parent class and yet use that code with objects of child classes	Polymorphism is able to **treat objects of different types as one type**

Operation Overriding (not operation overloading)	Operation in the superclass needs to be overridden in each of the subclasses	Overriding allows objects of different subclasses to **display different behaviors in response to the same method call**
Dynamic binding	Calls to overridden methods are bound to the implementation of the actual object's class dynamically during the runtime	Polymorphic code can call the method of the parent class and yet execute the implementation of the child class

---

Associations

- Connections between objects
- Can change over time
- Associations among objects can be generalized as associations between the corresponding classes too.
- Use **instance level** variables to implement associations
    - `Course` class can have a `students` variable to keeps track of students associated with a particular course

NAVIGABILITY UML - Navigability

- Which object in the association knows about (has a reference to) the other object

Unidirectional	Bidirectional
Navigability is from Box to Rope → b will have a reference to r but r will not have a reference to b → Can navigate from b to r using the b's object reference of r	b will have a reference to r and r will have a reference to b (point to each other for the same single instance of the association)

- Two unidirectional associations in opposite directions do not add up to a single bidirectional association

In the code below, there is a bidirectional association between the `Person` class and the `Cat` class i.e., if `Person p` is the owner of the `Cat c`, `p` it will result in `p` and `c` having references to each other.

```
class Person {
    Cat pet;
    //...
}

class Cat{
    Person owner;
    //...
}
```

The code below has two unidirectional associations between the `Person` class and the `Cat` class (in opposite directions) because the breeder is not necessarily the same person keeping the cat as a pet i.e., there are two separate associations here, which rules out it being a bidirectional association.

```
class Person {
    Cat pet;
    //...
}

class Cat{
    Person breeder;
    //...
}
```

MULTIPLICITY UML Class Diagrams - Multiplicity

Multiplicity is the aspect of an OOP solution that dictates how many objects take part in each association.

Normal instance-level variable → `0..1 multiplicity` (optional associations)

- A variable can hold a reference to a single object or `null`

A variable can be used to implement a `1` multiplicity too (compulsory associations)

- Eg The `Logic` class will always have a `ConfigGenerator` object, provided the variable is not set to `null` at some point.

```
class Logic {
    ConfigGenerator cg = new ConfigGenerator();
    ...
}
```

To implement other multiplicities, choose a suitable data structure eg Arrays, ArrayLists, HashMaps, Sets.

- Eg Two-dimensional array to implement a `1-to-many` association from Minefield to Cell

```
class Minefield {
    Cell[][] cell;
    //...
}
```

COMPOSITION UML Class Diagrams - Composition

An association that represents a strong **whole-part** relationship.

- When the ***whole*** is destroyed, ***parts*** are destroyed too
  - The part cannot exist without being attached to a whole.
  - Consider the case of `Folder` object `subF` is a sub-folder of `Folder` object `F`.
    - If `F` is deleted, `subF` will be deleted with it.
- There cannot be cyclical links.
  - `F` cannot be a sub-folder of `subF`
    - i.e., no cyclical 'sub-folder' association between the two objects)
- Whether a relationship is a composition can depend on the context.
  - Application that sends emails: Composition where the email subject is part of an email
  - Application that gather analytics about email traffic: Might not be a composition relationship
- A common use of composition is when parts of a big class are carved out as smaller classes
  - The classes extracted out still act as parts of the bigger class
- Cascading deletion alone is not sufficient for composition.
  - Eg Person object get deleted whenever Task object is deleted → Does not mean composition relationship
    - For it to be composition, a Person must be an integral part of a Task in the context of that association, at the concept level (not simply at implementation level).
- Identifying and keeping track of composition relationships in the design has benefits
  - Maintain the data integrity of the system
- Composition is implemented using a normal variable.

```
class Email {
    private Subject subject;
  ...
}
```

AGGREGATION UML Class Diagrams - Aggregation

- **Aggregation** represents a **container-contained** relationship
  - `SportsClub` can act as a container for `Person` objects who are members of the club.

    - `Person` objects can survive without a `SportsClub` object.
- **Containee object can exist even after the container object is deleted**

📦 In the code below, there is an aggregation association between the `Team` class and the `Person` class in that a `Team` contains a `Person` object who is the leader of the team.

```
class Team {
    Person leader;
    ...
    void setLeader(Person p) {
        leader = p;
    }
}
```

DEPENDENCIES UML Class Diagrams - Dependencies

- A **dependency** is a need for one class to depend on another without having a direct association in the same direction.
- Example: `Foo` has a dependency on `Bar` but it is not an association because it is only a **transient interaction** and there is **no long term relationship** between a `Foo` object and a `Bar` object. i.e. the Foo object does not keep the Bar object it receives as a parameter.

```
class Foo {

    int calculate(Bar bar) {
        return bar.getValue();
    }
}
```

```
class Bar {
    int value;

    int getValue() {
        return value;
    }
}
```

ASSOCIATION CLASSES UML Class Diagram - Association Class

- An association class represents additional information about an association.

`Transaction` class is an **association class** that represents a transaction between a `Person` who is the seller and another `Person` who is the buyer.	`class Transaction {`
	`    //all fields are compulsory`
	`    Person seller;`
	`    Person buyer;`
	`    Date date;`
	`    String receiptNumber;`
	
	`    Transaction(Person seller, Person buyer, Date date, String receiptNumber) {`
	`        //set fields`
	`    }`
	`}`

Requirements

A software project may be
- a **brownfield** project i.e., develop a product to replace/update an existing software product
- a **greenfield** project i.e., develop a totally new system from scratch

A software requirement specifies a need to be fulfilled by the software product.
- Requirements come from stakeholders.
- Identifying requirements is often not easy

Non-functional requirements

Functional requirements	Non-functional requirements
Specify what the system should do	Specify the constraints under which the system is developed and operated

- NFRs are easier to miss e.g., stakeholders tend to think of functional requirements first
- Sometimes NFRs are critical to the success of the software.

Non-functional requirement categories:
- Data requirements e.g. size, volatility, persistency etc.,
- Environment requirements e.g. technical environment in which the system would operate in or needs to be compatible with.
- Accessibility, Capacity, Compliance with regulations, Documentation, Disaster recovery, Efficiency, Extensibility, Fault tolerance, Interoperability, Maintainability, Privacy, Portability, Quality, Reliability, Response time, Robustness, Scalability, Security, Stability, Testability,

Some concrete examples of NFRs
- Business/domain rules: e.g. the size of the minefield cannot be smaller than five.
- Constraints: e.g. the system should be backward compatible with data produced by earlier versions of the system; system testers are available only during the last month of the project; the total project cost should not exceed $1.5 million.
- Technical requirements: e.g. the system should work on both 32-bit and 64-bit environments.
- Performance requirements: e.g. the system should respond within two seconds.
- Quality requirements: e.g. the system should be usable by a novice who has never carried out an online purchase.
- Process requirements: e.g. the project is expected to adhere to a schedule that delivers a feature set every one month.
- Notes about project scope: e.g. the product is not required to handle the printing of reports

---

Quality of requirements

Well-defined requirements:
- Unambiguous
- Testable (verifiable)

- Clear (concise, terse, simple, precise)
- Correct
- Understandable
- Feasible (realistic, possible)
- Independent
- Atomic
- Necessary
- Implementation-free (i.e. abstract)

Requirements as a whole: Consistent, Non-redundant, Complete

Prioritising requirements

- Based on importance and urgency
- Example schemes
    - Essential, Typical, Novel
    - High, Medium, Low
    - Must-have, Nice-to-have, Unlikely-to-have
- Some requirements can be discarded if they are considered ‘out of scope’

Gathering requirements

- In a **brainstorming** session there are no "bad" ideas. Aim is to generate ideas; not to validate them
- **Product Surveys** - Studying existing products can unearth shortcomings of existing solutions that can be addressed by a new product.
- **Observing** users in their natural work environment can uncover product requirements.
- **Surveys** can be used to **solicit responses and opinions** from a large number of stakeholders
- **Interviewing** stakeholders and domain experts can produce useful information about project requirements
- **Focus groups** are a kind of informal interview within an interactive group setting.
- **Prototyping**: A prototype is a mock up, a scaled down version, or a partial system constructed
    - to get users’ feedback.
    - to validate a technical concept (a "proof-of-concept" prototype).
    - to give a preview of what is to come, or to compare multiple alternatives on a small scale before committing fully to one alternative.
    - for early field-testing under controlled conditions.

  ⇒ Prototyping can uncover requirements related to **how users interact with the system**
  ⇒ Prototyping can be used for **discovering** as well as **specifying** requirements e.g. a UI prototype can serve as a specification of what to build.

Specifying Requirements

Prose `[W5.3a] Requirements → Specifying Requirements → Prose → What`

A textual description (i.e. **prose**) can be used to describe requirements.

Feature Lists

A list of features of a product grouped according to some criteria e.g. aspect, priority, order of delivery, etc.

User Stories

- Short, simple descriptions of a feature told from the perspective of the person who desires the new capability
  - Not detailed enough to tell us exact details of the product.
- User story format: `As a {user type/role} I can {function} so that {benefit}`
  - **As an** employee, **I can** view my leave balance, **so that** I know how many leave days I have left.

Details `[W5.3d] Requirements → Specifying Requirements → User Stories → Details`

- `{benefit}` can be omitted if it is obvious. (confirm there is a concrete benefit first)
- Add more characteristics to the `{user role}` eg expert user, forgetful user
- Write user stories at various levels
  - **Epics** (or themes): High-level user stories that cover bigger functionality
  - Example: [Epic] As a user, I can cancel a reservation
    - As a premium site member, I can cancel a reservation up to the last minute
    - As a non-premium member, I can cancel up to 24 hours in advance
    - As a member, I am emailed a confirmation of any cancelled reservation
- Add conditions of satisfaction to a user story
  - Tthings that need to be true for the user story implementation to be accepted as ‘done’
- Priority: how important the user story is
- Size: the estimated effort to implement the user story
- Urgency: how soon the feature is needed

Usage `[W5.3e] Requirements → Specifying Requirements → User Stories → Usage`

- User stories capture user requirements in a way that is convenient for scoping, estimation, and scheduling.
- User stories differ from traditional requirements specifications (prose) mainly in the level of detail
- User stories can capture non-functional requirements too
- User stories are quite handy for recording requirements during early stages of requirements gathering

- Don't be too hasty to discard 'unusual' user stories
- Don't go into too much detail
- Don't be biased by preconceived product ideas
- Don't discuss implementation details or whether you are actually going to implement it

---

Glossary `[W5.3f] Requirements → Specifying Requirements → Glossary → What`

All stakeholders have a common understanding of the noteworthy terms, abbreviations, acronyms etc.

Supplementary Requirements

`[W5.3g] Requirements → Specifying Requirements → Supplementary Requirements → What`

Requirements that do not fit elsewhere

Use Cases `[W7.1a] Requirements → Specifying Requirements → Use Cases → Introd…`

- Describes an **interaction** between the **user** and **system** for a **specific functionality** of the system
- Capture the **functional requirements** of a system

System: Online Banking System (OBS) **Use case:** UC23 - Transfer Money **Actor:** User **Preconditions**: User is logged in **Guarantees:** • Money will be deducted from the source account only if the transfer to the destination account is successful. • The transfer will not result in the account balance going below the minimum balance required. **MSS:** 1. User chooses to transfer money. 2. OBS requests for details of the transfer. 3. User enters the requested details. 4. OBS requests for confirmation. 5. User confirms. 6. OBS transfers the money and displays the new account balance. Use case ends.	**Extensions:** 3a. OBS detects an error in the entered data. 3a1. OBS requests for the correct data. 3a2. User enters new data. Steps 3a1-3a2 are repeated until the data entered are correct. Use case resumes from step 4. 3b. User requests to effect the transfer in a future date. 3b1. OBS requests for confirmation. 3b2. User confirms future transfer. Use case ends. *a. At any time, User chooses to cancel the transfer. *a1. OBS requests to confirm the cancellation. *a2. User confirms the cancellation. Use case ends. *b. At any time, 120 seconds lapse without any input from the User. *b1. OBS cancels the transfer. *b2. OBS informs the User of the cancellation. Use case ends.

ACTORS in Use Cases

- A **ROLE** played by a user
- Can be a human or another system
- Not part of the system, reside outside the system

- A use case can involve **multiple actors**.
- An actor can be involved in **many use cases**.
- A single person / system can play **many roles (actors)**
- Many persons / systems can play a single role (actor)
- Use cases can be specified at various levels of detail (eg conduct vs take vs answer survey)

DETAILS `[W7.1c] Requirements → Specifying Requirements → Use Cases → Details`

- Main body: A sequence of steps that describes the interaction between the system and actors
    - Each step: ***Who does what***
- Use case describes only the **EXTERNALLY visible behavior** (eg LMS uploads the file), not internal details (eg save files into cache), of a system
- A step gives the intention of the actor (not the mechanics) → Usually omit UI details
    - `User clears input` instead of `User right clicks the box and chooses clear`

LOOPS

Software System: SquareGame
Use case: UC02 - Play a Game
Actors: Player (multiple players)

MSS:
1. A Player starts the game.
2. SquareGame asks for player names.
3. Each Player enters his own name.
4. SquareGame shows the order of play.
5. SquareGame prompts for the current Player to throw a die.
6. Current Player adjusts the throw speed.
7. Current Player triggers the die throw.
8. SquareGame shows the face value of the die.
9. SquareGame moves the Player's piece accordingly.
   Steps 5-9 are repeated for each Player, and for as many rounds as required until a Player reaches the 100th square.
10. SquareGame shows the Winner.

Use case ends.

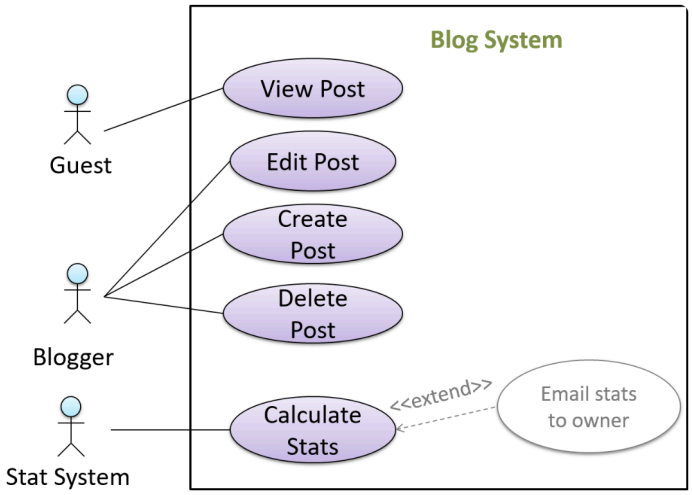
MAIN SUCCESS SCENARIO (MSS)

- The most straightforward interaction for a given use case, which assumes that nothing goes wrong
- Should be **self-contained** - give a complete usage scenario

EXTENSIONS

- “add-ons” to the MSS that describe *exceptional / alternative flow of events*

- Not useful to mention events such as power failures or system crashes (system cannot function)



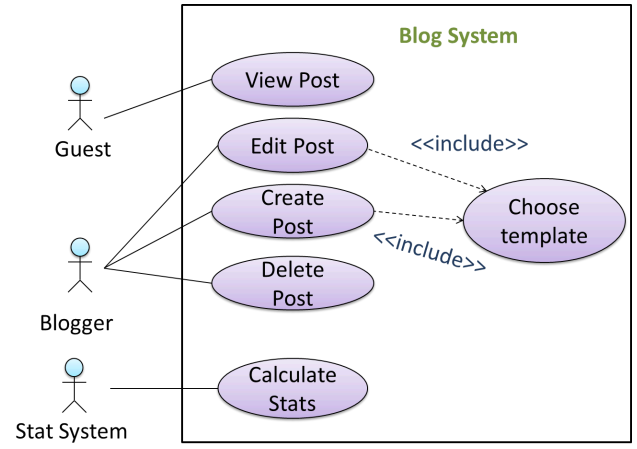
  - `<<extend>>` arrow (dashed line) **from extension** to the use case it extends

INCLUDE ANOTHER USE CASE

- Underlined text to show inclusion of a use case

  - MSS:
    1. Staff creates the survey (UC44).
    2. Student completes the survey (UC50).
    3. Staff views the survey results.

    Use case ends.



  - `<<include>>` arrow from main use case **to included use case**

PRECONDITIONS

- Specify the specific state you expect the system to be in before the use case starts

GUARANTEES

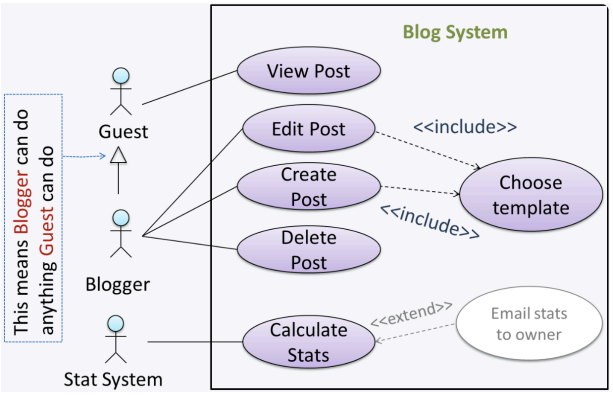
- Specify what the use case promises to give us at the end of its operation

USE CASE DIAGRAMS

- UML includes **use case diagrams** - can illustrate use cases of a system visually

Usage `[W7.1d] Requirements → Specifying Requirements → Use Cases → Usage`

- Can use actor generalization in use case diagrams



- Some include 'System' as an actor to indicate that something is done by the system itself without being initiated by a user or an external system.
  - DON'T use the system as a user.
    - `view daily report` should be the use case and `generate daily report` (done by the system) is not shown as it supports `view daily report`
- UML is not very specific about the text contents of a use case.
- Advantages of documenting system requirements as use cases
  - Simple notation, plain English descriptions → Easy for users to understand, give feedback
  - Decouple user intention from mechanism (exclude UI-specific details) → System designers have more freedom to optimize how a functionality is provided to a user
  - Identify all possible extensions → Can consider all situations that a software product might face during its operation
  - Separate typical scenarios from special cases → Optimize the typical scenarios
- Disadvantage
  - Not good for capturing requirements that do not involve a user interacting with the system

Design

- **Product / external design**: Designing the external behavior of the product to meet the users' requirements.
- **Implementation / internal design**: Designing how the product will be implemented to meet the required external behavior.

Design Fundamentals

Abstraction `[W7.3a] Design → Design Fundamentals → Abstraction → What`

- **Only details that are relevant** to the current perspective or the task at hand need to be considered
- Not limited to just data or control abstractions

Data abstraction	Control abstraction
Abstracting away the lower level data items and thinking in terms of bigger entities • E.g. User data type, ignore details in the user data item such as name	Abstracting away details of the actual control flow to focus on tasks at a higher level • E.g. print("Hello") is an abstraction of the actual output mechanism within the PC

  - OOP class = An abstraction over related data and behaviors
  - Architecture = Higher-level abstraction of the design of a software
  - Models (e.g., UML models) = Abstractions of some aspect of reality.
- Abstraction can be applied repeatedly to obtain progressively *higher levels of abstraction*.

Coupling `[W7.3b] Design → Design Fundamentals → Coupling → What`

- A measure of the **degree of *dependence*** between components, classes, methods, etc.
- High coupling (tight coupling / strong coupling)
  - More components are required to be integrated at once, or
  - More drivers and stubs are required when integrating incrementally
  - ⇒ Value of automated regression testing increases
- Disadvantages
  - Maintenance is harder
    - A change in one module could cause changes in other modules coupled to it → **Higher risk of regressions**
  - Integration is harder
    - Multiple components coupled with each other have to be integrated at the same time
  - Testing and reuse of the module is harder due to its dependence on other modules.
  - **Decreases testability**
    - SUT is coupled to many other components → Difficult to test the SUT in isolation of its dependencies

`[W7.3c] Design → Design Fundamentals → Coupling → How`

X is <u>COUPLED</u> to Y if a change to Y can *<u>POTENTIALLY</u>* require a change in X.

- If the `Foo` class calls the method `Bar#read()`, `Foo` is coupled to `Bar`
  - If the signature of `Bar#read()` is changed, `Foo` needs to change
- `A` is coupled to `B` if,
  - `A` has access to the internal structure of `B` (→ very high level of coupling)
  - `A` and `B` depend on the same global variable
  - `A` calls `B`
  - `A` receives an object of `B` as a parameter or a return value
  - `A` inherits from `B`
  - `A` and `B` are required to follow the same data format or communication protocol

---

Cohesion `[W7.3e] Design → Design Fundamentals → Cohesion → What`

- A measure of **how strongly-related** and **focused the various responsibilities** of a component are
- <u>Disadvantages of low (weak) cohesion:</u>
  - Lowers understandability of modules (difficult to express module functionalities at higher level)
  - Lowers maintainability
    - A module can be modified due to unrelated causes (reason: the module contains code unrelated to each other)
    - Many modules may need to be modified to achieve a small change in behavior (reason: the code related to that change is not localized to a single module)
  - Lowers reusability of modules (as they do not represent logical units of functionality)

`[W7.3f] Design → Design Fundamentals → Cohesion → How`

Cohesion can be present in many forms

- Code related to a single concept / invoked close together / manipulates the same data structure is kept together

---

Modeling

Model	Class Diagram
Representation of something else	Model that represents a software design
[**Abstraction**] Provides a simpler view of a complex entity, captures only a selected aspect	Captures the structure of the software design but not the behavior.
Multiple models of the same entity may be needed to capture it fully.	Other diagrams may be needed to capture various interesting aspects of the software.

WHY `[W4.1b] Design → Modelling → Introduction → How`

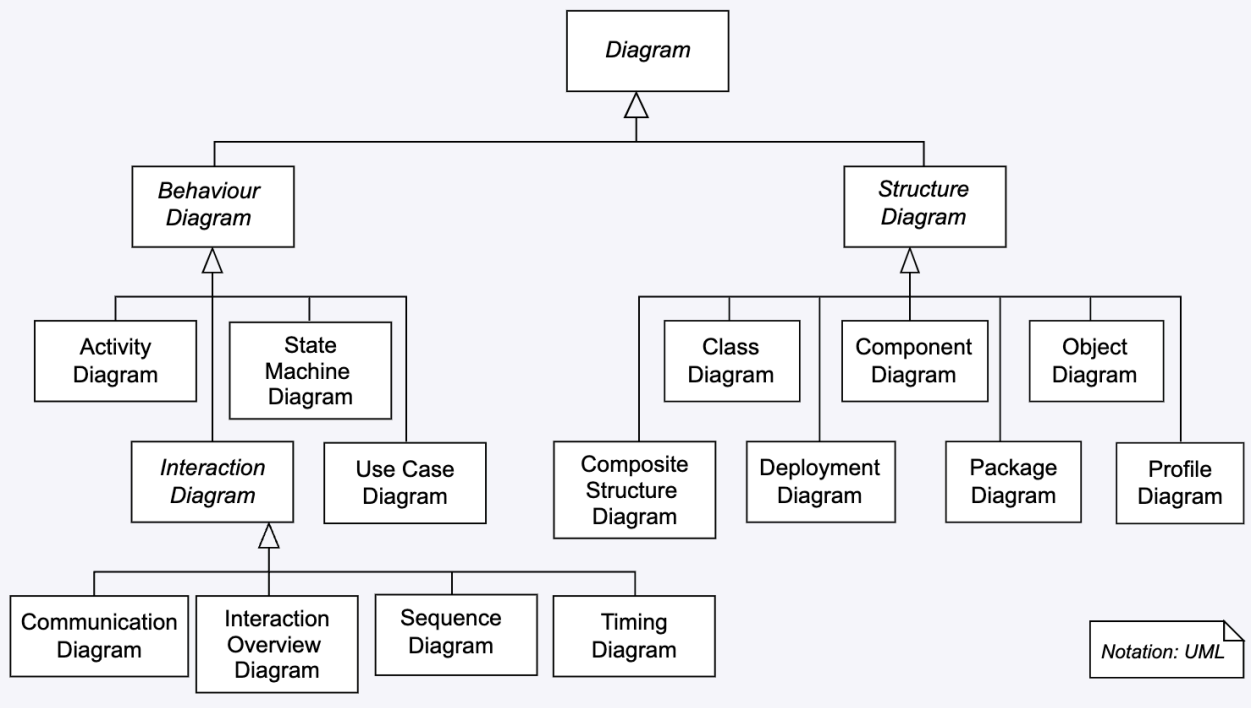
Models are useful in software development

- Analyze a complex entity related to software development.
- Communicate information among stakeholders.
- As a blueprint for creating software.

UML MODELS `[W4.1c] Design → Modelling → Introduction → UML models`

- Unified Modeling Language (UML)

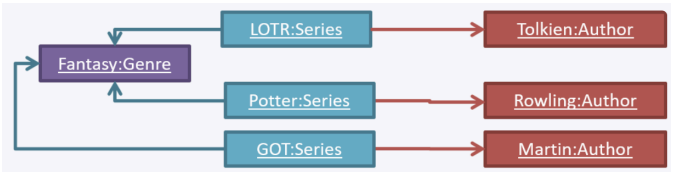
The following uses class diagram notation to show the different types of UML diagrams.



Modelling STRUCTURE

OO STRUCTURES `[W4.2a] Design → Modelling → Modelling Structure → OO structure…`

- Useful to be able to model how the relevant objects are 'networked' (connected) together
- Object structures within the same software can change over time based on a set of rules
    - Rules that object structures need to follow can be illustrated as a ***class structure***

UML Object Diagrams model **object** structures.	UML **Class** Diagrams model **class** structures.
	

CLASS DIAGRAMS `[W4.2b] Design → Modelling → Modelling Structure → Class diagr…`

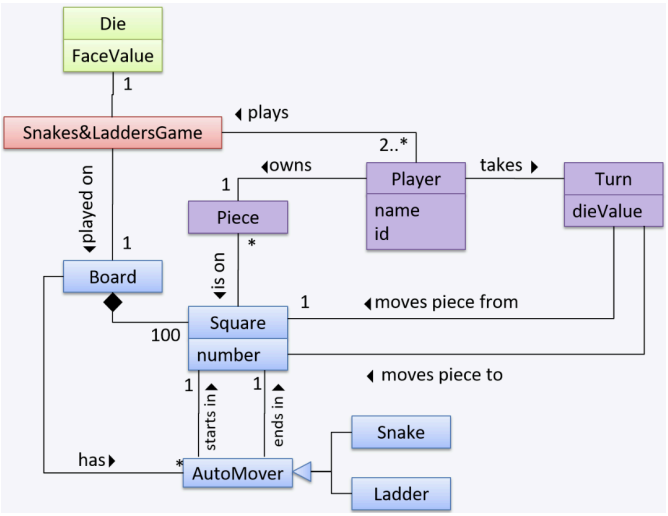
- Classes form the basis of class diagrams.
- UML class diagrams describe the **STRUCTURE (but not the behavior)** of an OOP solution.
- A class diagram can also show different types of relationships between classes: inheritance, compositions, aggregations, dependencies.

OBJECT DIAGRAMS `[W4.2c] Design → Modelling → Modelling Structure → Object di…`

- Shows an object **STRUCTURE** at a given point of time. Can complement class diagrams

Conceptual Class Diagrams (OO domain models, OODMs)

| **UML model that captures CLASS structures in the PROBLEM domain**
• Domain modeling = Modeling the problem domain (relevant information that needs to be examined to understand a problem)
  ○ Domain-specific modeling notation
  ○ General purpose modeling notation (Eg UML)
  ○ Other general purpose notations

CCDs **do not contain solution-specific classes** (i.e.used in solution domain but do not exist in problem domain) |  |

Conceptual Class Diagrams	Class Diagrams
Represents **CLASS STRUCTURE** of the problem / solution domain (not behavior)	

Describes the problem domain	Describes the solution domain
Uses a **subset of the class diagram notation (OMITS METHODS, NAVIGABILITY)**	

Modelling BEHAVIORS

SEQUENCE DIAGRAMS

Sequence diagrams model the interactions between various entities in a system, in a specific scenario.

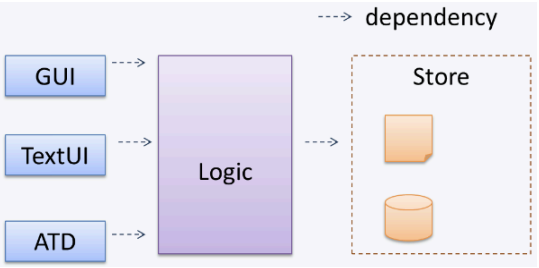
ACTIVITY DIAGRAMS - model workflows

Modelling a SOLUTION

You can use models to analyze and design software before you start coding.

Software Architecture

- Shows overall organization of the system
- A very high-level design
- Typically designed by the software architect

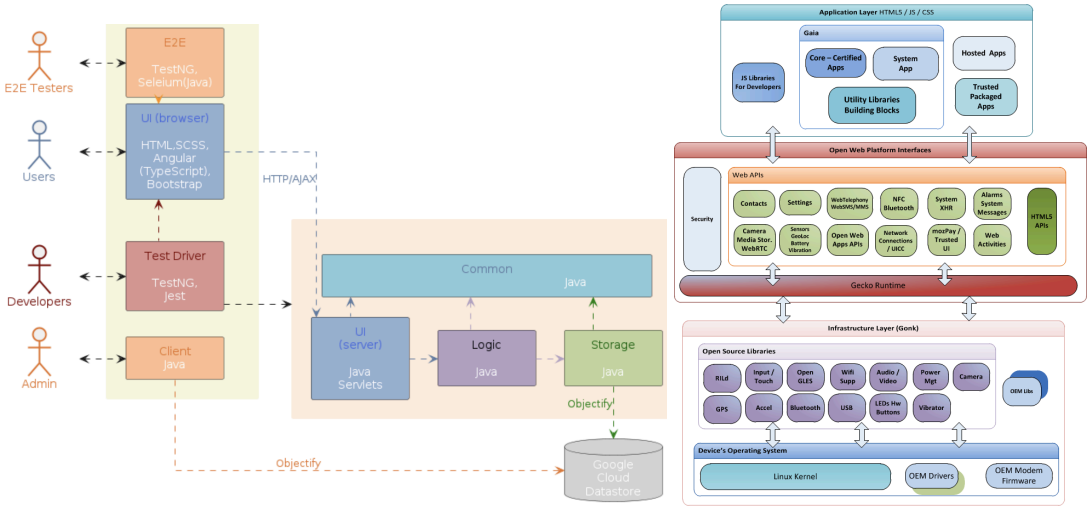


Main components:

- `GUI` : Graphical user interface
- `TextUi` : Textual user interface
- `ATD` : An automated test driver used for testing the game logic
- `Logic` : Computation and logic of the game
- `Store` : Storage and retrieval of game data (high scores etc.)

Architecture Diagrams

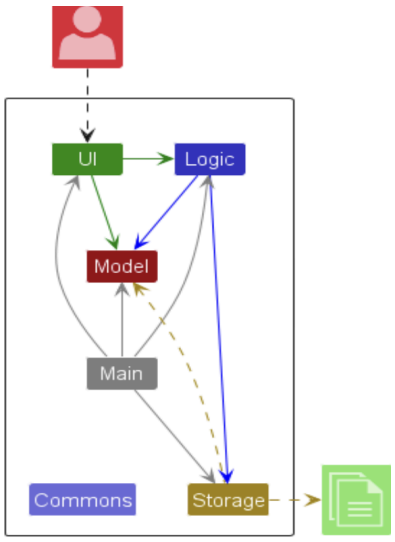
- Free-form diagrams
  - Minimize variety of symbols. If the symbols do not have widely-understood meanings (drum symbol is widely-understood as representing a database), explain their meaning.
  - Avoid indiscriminate use of double-headed arrows to show interactions between components



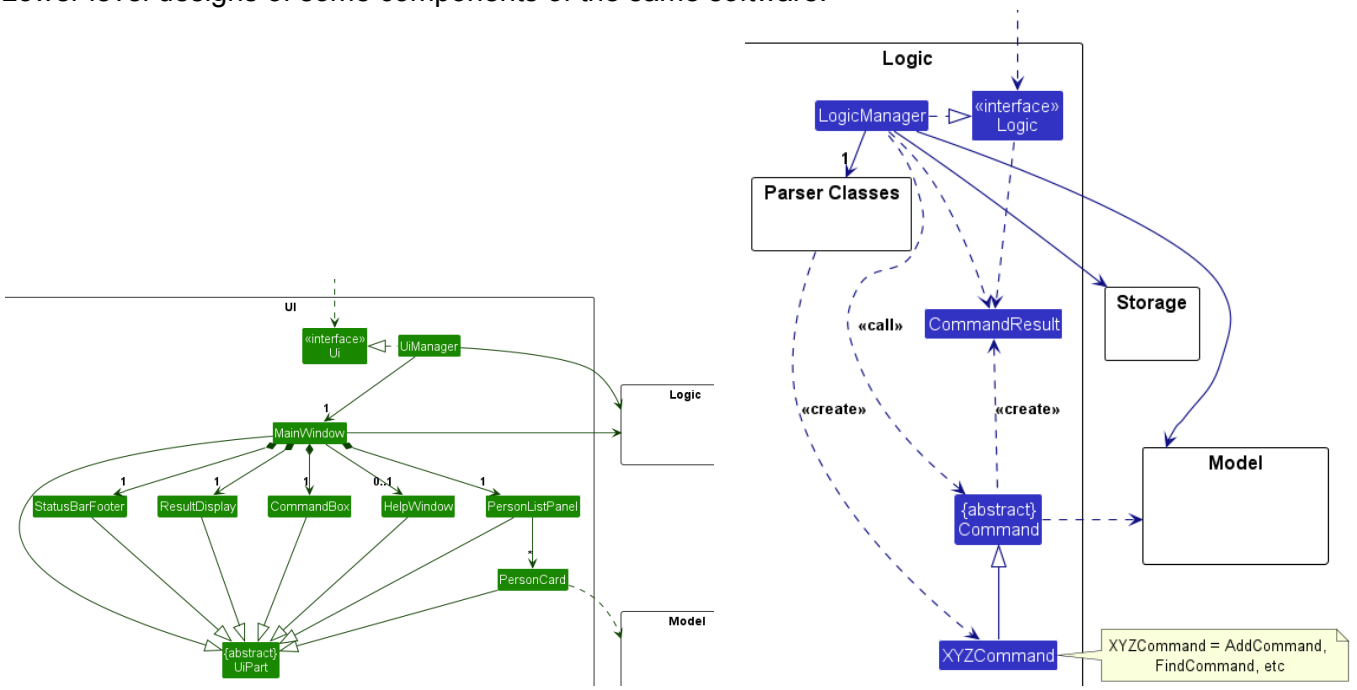
Multi-level Design

- Design of bigger systems needs to be done/shown at multiple levels.

Architecture diagram depicts the **high-level** design of the software



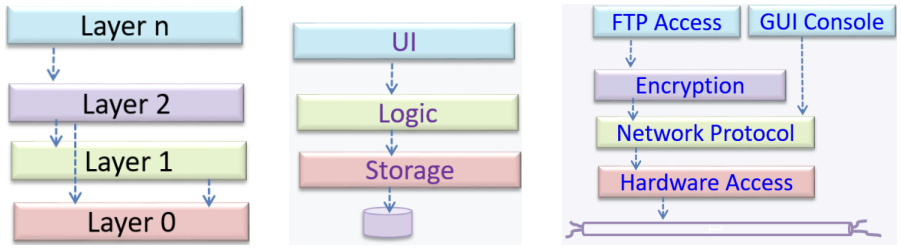
Lower level designs of some components of the same software:



Architectural Styles

- Software architectures follow various high-level styles (architectural patterns)
- Most applications use a mix of the architectural styles. Eg Minesweeper game:
    - Client-server – Clients can be the game UI running on player PCs. The server can be the game logic running on one machine.
    - Transaction-processing – Each player action can be packaged as transactions (by the client component running on the player PC) and sent to the server. Server processes them in the order they are received.
    - SOA – The game can access a remote web service for things such as getting new puzzles, validating puzzles, charging players subscription fees, etc.
    - Multi-layer – The server component can have two layers: the logic layer and the storage layer.

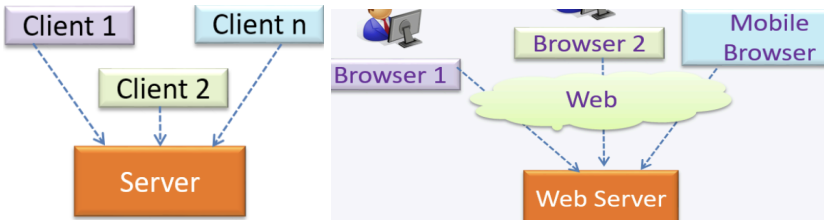
N-TIER (multi-layered, layered) architectural style



- Higher layers make use of services provided by lower layers
- Lower layers are independent of higher layers

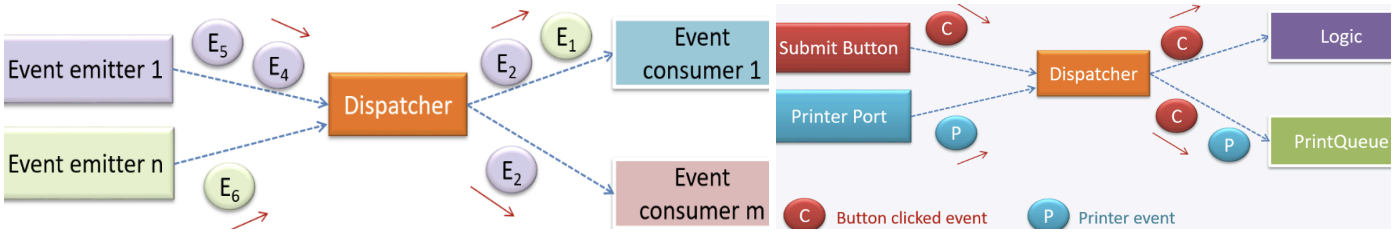
CLIENT-SERVER architectural style

- At least one component playing the role of a **server**
- At least one **client** component accessing services of **server**
- Often in distributed application



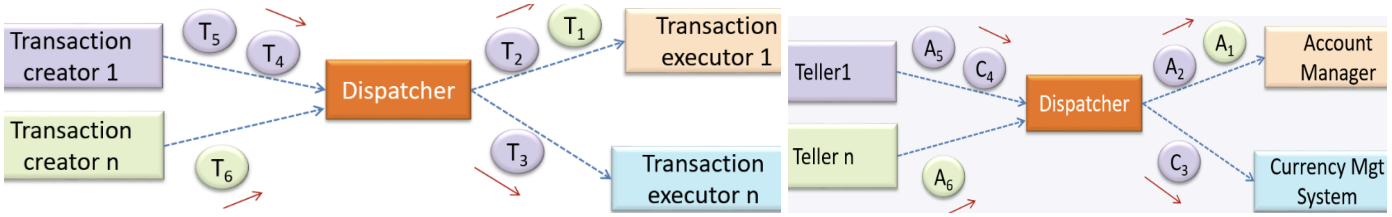
EVENT-DRIVEN architectural style

Controls the flow of the application by detecting events from event (eg button click, timer running out) ***emitters*** and communicating those events to interested event ***consumers***. Often used in **GUIs**.



TRANSACTION PROCESSING architectural style

Divides the workload of the system down to a number of **transactions** which are then given to a **dispatcher** that controls the execution of each transaction



SERVICE-ORIENTED architectural (SOA) style

Builds applications by combining functionalities packaged as **programmatically accessible services**

- Aims to achieve interoperability between distributed services
- Because both Amazon and HSBC services follow the **SOA architecture**, their web services can be reused by the web application, even if all three systems use different programming platforms.

Software Design Patterns

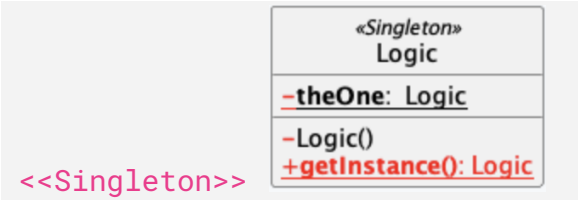
An **elegant reusable solution** to a **commonly recurring problem** in a given context in software design.

Format to describe a pattern: Context, Problem, Solution

- Optional: Anti-patterns, Consequences, Other useful information

SINGLETON (single instances) design pattern

- **Context**: Certain classes should have no more than just one instance (e.g. main controller class)
- **Problem**: A normal class can be instantiated multiple times by invoking the constructor.
- **Solution**:
  - **Constructor** of the singleton class is `private` (public → allow others to instantiate)
  - **Single instance** of the singleton class is maintained by a `private` **class-level variable**
  - `public` **class-level method** to access the **single instance**
    - `Logic m = Logic.getInstance();` instead of `Logic m = new Logic();`

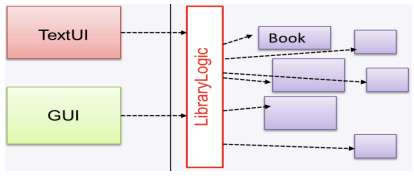


General format: `<<role/purpose>>`

```
class Logic {
    private static Logic theOne = null;
    private Logic() {...}
    public static Logic getInstance() {
        if (theOne == null) {
            theOne = new Logic();
        }
        return theOne; ...
```

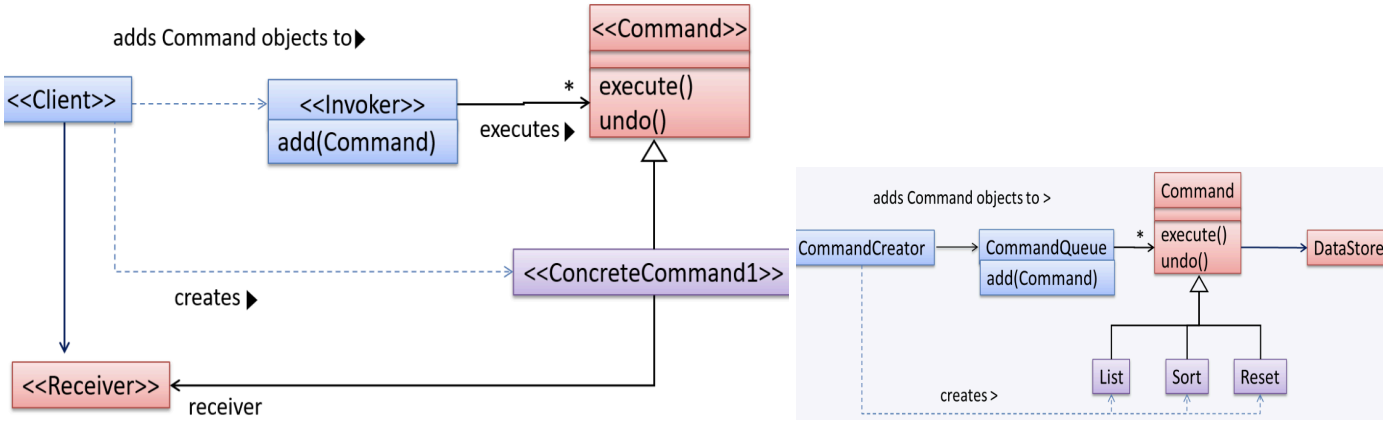
Pros of Singleton design pattern	Cons of Singleton design pattern
• Easy to apply • Effective in achieving its goal with minimal extra work • Easy way to access the singleton object from anywhere in the codebase	• A singleton object acts like a global variable that increases coupling across codebase. • In testing, it is difficult to replace Singleton objects with stubs (static methods cannot be overridden). • In testing, singleton objects carry data from one test to another even when you want each test to be independent.

FACADE design pattern



Sits between the component internals and users of the component such that all access to the component happens through the `Facade` class, without exposing internal details of the component

COMMAND design pattern



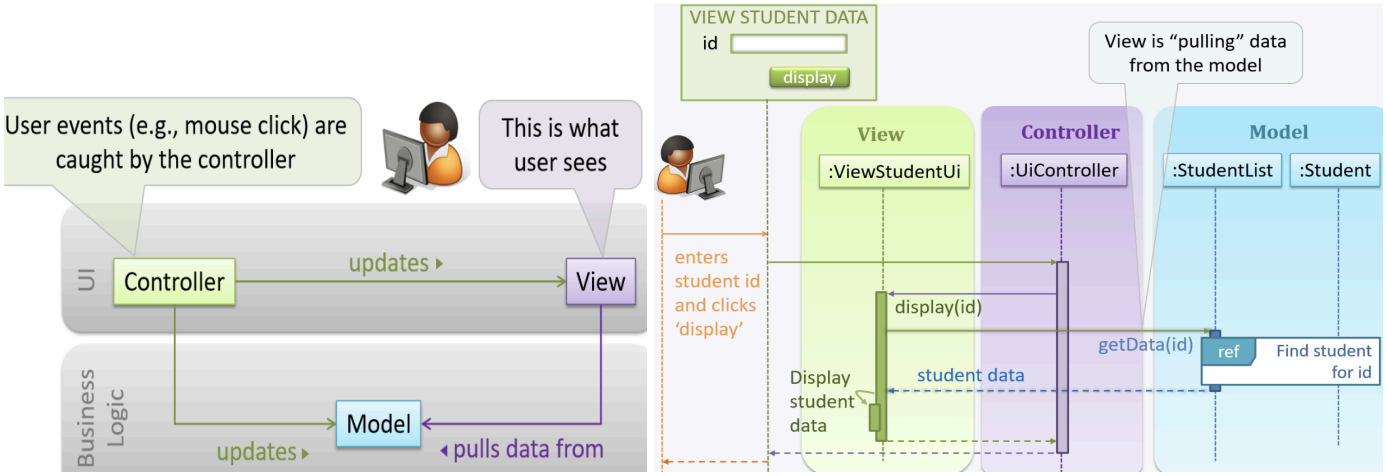
- The `<<Client>>` creates a `<<ConcreteCommand>>` object, and passes it to the `<<Invoker>>`.
- The `<<Invoker>>` object treats all commands as a **general** `<<Command>>` type, without knowledge of each command type and what each command does.
    - `<<Command>>` object can be passed around, stored, executed, etc without knowing the type of command (i.e. via polymorphism).
    - **`<<Command>>` class can also be an abstract class or interface.**
- `<<Invoker>>` issues a request by calling `execute()` on the command.
- If a command is undoable, `<<ConcreteCommand>>` will store the state for undoing the command prior to invoking `execute()`.
- In addition, the `<<ConcreteCommand>>` object may have to be linked to any `<<Receiver>>` of the command (the object the command will operate on, in case different commands operate on different objects) before it is passed to the `<<Invoker>>`.

Note that an application of the command pattern does not have to follow the structure given above.

MODEL VIEW CONTROLLER (MVC) design pattern

Decouple data, presentation, and control logic by separating them into different components:

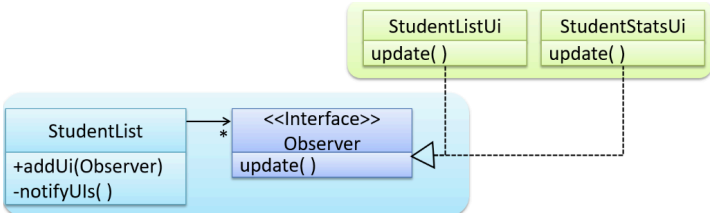
- **MODEL**: Stores and maintains data. Updates the **view** if necessary.
- **VIEW**: Displays data, interacts with the user, and pulls data from the **model** if necessary.
- **CONTROLLER**: Detects UI events such as mouse clicks and button pushes, and takes follow up action. Updates or changes the **mode** or **view** when necessary.



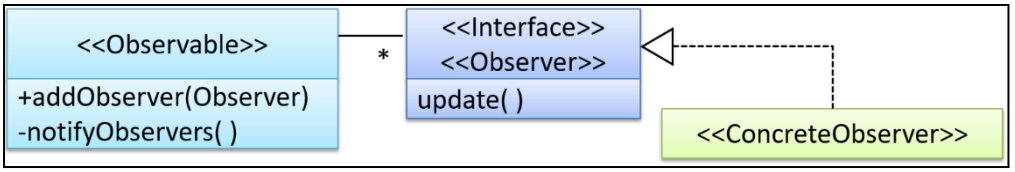
OBSERVER design pattern

Problem: The 'observed' object does not want to be coupled to objects that are 'observing' it.

Solution:



- Whenever the data in `StudentList` changes (e.g. new student is added to StudentList),
  - All interested observers are updated by calling the `notifyUIs` operation.
  - UIs can then pull data from the `StudentList` whenever the `update` operation is called.
  - ** `StudentList` is unaware of the exact nature of the two UIs but still manages to communicate with them via an intermediary.
- When applying the **Observer** pattern to an **MVC** structure, `Views` can get **notified** and **update** themselves about a change to the `Model` without the `Model` having to depend on the `Views`.



- `<<Observer>>` is an interface
  - Any class that implements it can observe an `<<Observable>>`.
  - Any number of `<<Observer>>` objects can observe (i.e. listen to changes of) the `<<Observable>>` object.
- The `<<Observable>>` maintains a list of `<<Observer>>` objects.
  - `addObserver(Observer)` operation adds a new `<<Observer>>` to the list of `<<Observer>>`s.
- Whenever there is a change in the `<<Observable>>`, the `notifyObservers()` operation is called that will call the `update()` operation of all `<<Observer>>`s in the list.
  - **Polymorphic behavior of `update()`**
    - When the `Observable` object invokes the `notifyObservers()` method, it is treating all `ConcreteObserver` objects as a general type called `Observer` and calling the `update()` method of each of them.
    - `update()` method of each `ConcreteObserver` could potentially show different behavior based on its actual type

Design Approaches

Top-down and bottom-up design

Multi-level design can be done in a top-down manner, bottom-up manner, or as a **mix**.

- **Top-down**: Useful when designing big and novel systems where the high-level design needs to be stable before lower levels can be designed.
- **Bottom-up**: Not usually scalable for bigger systems. Examples where this might work: Designing a variation of an existing system or Repurposing existing components to build a new system.

Agile design

- Agile designs are **emergent (change over time)**, they're not defined up front.
- Some initial architectural modeling at the very beginning

Implementation

Integrated Development Environments (IDEs)

Support most development-related work within the same tool

- Source code editor
  - syntax coloring
  - auto-completion
  - easy code navigation (e.g., to navigate from a method call to the method implementation)
  - error highlighting
  - code-snippet generation
- Compiler and/or an Interpreter
  - Compilation / linking / running / deployment
- Debugger
  - Execute the program one step at a time to observe the run-time behavior and locate bugs
- Support for automated testing
- Visual programming
  - e.g. write programs using 'drag and drop' actions instead of typing code
  - Drag-and-drop construction of UI components
- Version management support
- Simulation of the target runtime platform
  - e.g., run a mobile app in a simulator
- Modeling support
- AI-assisted coding help
- Collaborative coding with others
- Code analysis
  - e.g. to find unreachable code
- Reverse engineering design/documentation
  - e.g. generate diagrams from code
- Syntax assistance
  - e.g., show hints as you type
- Code generation
  - e.g., to generate the code required by simply specifying which component/structure you want to implement
- Extension
  - i.e., ability to add more functionality to the IDE using plugins

---

Debugging `[W6.3a] Implementation → IDEs → Debugging → What`

- Debugging is the process of discovering defects in the program.
  - BAD: Inserting temporary print statements
  - BAD: Manually tracing through the code
  - GOOD: Use a debugger (recommended approach for debugging)

Code Quality

- Production code needs to be of high quality

Style `[W3.5b] Implementation → Code Quality → Style → Introduction`

- Follow a consistent style → Improve code quality
    - Follow a strict coding standard (style guide)
        - Make the entire codebase look like it was written by one person

Naming `[W4.6a] Implementation → Code Quality → Naming → Introduction`

- Proper naming improves the readability of code. Reduces bugs caused by ambiguities regarding the intent of a variable or a method.

Use nouns for classes/variables and verbs for methods/functions.

Name for a	👎 Bad	👍 Good
Class	`CheckLimit`	`LimitChecker`
Method	`result()`	`calculate()`

- Distinguish between single-valued and multi-valued variables.

```
Person student;
ArrayList<Person> students;
```

Use standard words. Avoid foreign language words, slang, and names that are only meaningful within specific contexts/times

Use name to explain.

- A name is not just for differentiation
    - Should explain the named entity to the reader accurately and at a sufficient level of detail.

👎 Bad	👍 Good
`processInput()` (what 'process'?)	`removeWhiteSpaceFromInput()`
`flag`	`isValidInput`
`temp`	

- If a name has multiple words, they should be in a sensible order.

👎 Bad	👍 Good
`bySizeOrder()`	`orderBySize()`

- Don't use numbers or case to distinguish names.

👎 Bad	👎 Bad	👍 Good
`value1`, `value2`	`value`, `Value`	`originalValue`, `finalValue`

Not too long or short.

- While it is preferable not to have lengthy names, names that are 'too short' are even worse.
- If you must abbreviate or use acronyms, do it consistently.
- Explain their full meaning at an obvious location.

Avoid misleading names.

- Related things should be named similarly, while unrelated things should NOT.
- Avoid misleading or ambiguous names (e.g. those with multiple meanings)

Bad	Good	Reason
`phase0`	`phaseZero`	Is that zero or letter O?
`rwrLgtDirn`	`rowerLegitDirection`	Hard to pronounce
`right` `left` `wrong`	`rightDirection` `leftDirection` `wrongResponse`	`right` is for 'correct' or 'opposite of 'left'?
`redBooks` `readBooks`	`redColorBooks` `booksRead`	`red` and `read` (past tense) sounds the same
`FiletMignon`	`egg`	If the requirement is just a name of a food, `egg` is a much easier to type/say choice than `FiletMignon`

Readability [W5.4a] Implementation → Code Quality → Readability → Introduction

Among various dimensions of code quality, such as run-time efficiency, security, and robustness, one of the most important is readability

- **Avoid long methods**: Consider if shortening is possible when a method goes beyond 30 LoC
- **Avoid deep nesting:** Not more than 3 levels of indentation. No arrowhead style code.
- **Avoid complicated expressions**, especially those having many negations and nested parentheses

Bad:
```
return ((length < MAX_LENGTH) || (previousSize != length))
         && (typeCode == URGENT);
```

Good:
```
boolean isWithinSizeLimit = length < MAX_LENGTH;
boolean isSameSize = previousSize != length;
boolean isValidCode = isWithinSizeLimit || isSameSize;

boolean isUrgent = typeCode == URGENT;

return isValidCode && isUrgent;
```

- **Avoid any magic literals.** Avoid magic numbers (meaning of the number is unexplained).
- Make the code as explicit as possible, even if the language syntax allows them to be implicit.
  - Java Use explicit type conversion instead of implicit type conversion.
  - Java Python Use parentheses/braces to show groupings even when they can be skipped.

    - Java Python Use enumerations when a certain variable can take only a small number of finite values
- **Structure code logically**.
    - E.g. order; can use blank lines to separate groups of related statements
- **Do not 'Trip Up' the reader.** Avoid things that would make the reader go 'huh?'
- **Practice KISSING.** Do not try to write 'clever' code. "Keep it simple, stupid" (KISS)
- **Avoid premature optimisations.**
    - Optimizing code prematurely has several drawbacks:
        - May not know which parts are the real performance bottlenecks.
        - Complicate the code
        - Hand-optimized code can be harder for the compiler to optimize
- Make it work, make it right, make it fast
    - Getting the code to perform correctly should take priority over optimizing it
    - There are cases in which optimizing takes priority over other things e.g. when writing code for resource-constrained environments

SLAP hard.

- **Avoid having multiple levels of abstraction within a code fragment.**
- **Single Level of Abstraction Principle (SLAP)**: One Level of Abstraction per Function

👎 **Bad** (`readData();` and `salary = basic * rise + 1000;` are at different levels of abstraction)

```
1 readData();
2 salary = basic * rise + 1000;
3 tax = (taxable ? salary * 0.07 : 0);
4 displayResult();
```

Good (all statements are at the same level of abstraction)

```
1 readData();
2 processData();
3 displayResult();
```

- Sometimes possible to pack two levels of abstraction into the code
    - Each step in the higher-level logic is clearly marked using comments and separated eg:

```
//high-level step A
low-level statement A1
low-level statement A2
low-level statement A3

//high-level step B
low-level statement B1
low-level statement B2
```

- **The happy path should be clear and prominent.** The execution path taken when everything goes well should be less-nested as much as possible

`if (!isUnusualCase) {  //detecting an unusual condition`
`    if (!isErrorCase) {`
`        start();    //main path`
`        process();`
`        cleanup();`
`        exit();`
`    } else {`
`        handleError();`
`    }`
`} else {`
`    handleUnusualCase(); //handling that unusual`
	BAD
	
	Unusual condition detections are separated from their handling.
	
	The main path is nested deeply.

condition
}
 	
if (isUnusualCase) { //Guard Clause
  handleUnusualCase();
  return;
}
 	
if (isErrorCase) { //Guard Clause
  handleError();
  return;
}
 	
start();
process();
cleanup();
exit();
	GOOD
	
	Deals with unusual conditions as soon as they are detected so that the reader doesn't have to remember them for long.
	
	Keeps the main path un-indented.

Reduce the nesting of the happy path inside a loop using `continue`

BAD

```
for (condition1)
   if (condition2)
       statement A
       statement B
       statement C
       statement D
statement E
```

GOOD

```
for (condition1)
   if (not condition2)
       continue
   statement A
   statement B
   statement C
   statement D
statement E
```

Error-Prone Practices

- Safer to use language constructs in the way they are meant to be used

- **Always include a `default` branch in case statements**
  - Use the `default` branch / final `else` of an `if-else` construct for the intended `default` action and not just to execute the last option.
    - Can use the `default` branch or final `else` to detect errors

BAD

```
if (red) print "red";
else print "blue";
```

GOOD

```
if (red) print "red";
else if (blue) print "blue";
else error("incorrect input");
```

- **Don't recycle variables or parameters.**
  - Use one variable for one purpose
  - Do not reuse formal parameters as local variables
- Avoid empty `catch` statements
- **Delete dead code.** Get rid of unused code the moment it becomes redundant.

- **Minimize global variables**. Define variables in the least possible scope
- **Minimise code duplication.** Code duplication, especially when you copy-paste-modify code, often indicates a poor quality implementation

---

Code Comments `[W5.4t] Implementation → Code Quality → Comments → Introductio…`

- Comment **minimally but sufficiently**
  - Avoid writing comments to explain bad code. Improve the code to make it self-explanatory.
- **Do not repea**t in comments information that is already **obvious** from the code
- **Write to the reader.** Write comments targeting other programmers reading the code.
  - Always useful: Header comment that you write for a class or an operation to explain its purpose
- Comments should explain
  - **WHAT**
    - The specification of what the code is *supposed* to do
  - **WHY**
    - Rationale for current implementation
  - NOT the HOW (explanation for how the code works)

Refactoring

- Process of **improving a program's internal structure** in small steps without modifying its external behavior
  - MUST do regression testing after each change
- NOT:
  - Rewriting Refactoring needs to be done in small steps.
  - Bug fixing Refactoring alters the **internal** (not external, eg adding a feature) behaviour
- Secondary benefits
  - Hidden bugs become easier to spot
  - Improve performance, understandability
- Example
  - Move duplicate conditional fragments to outside the expression.
  - Group the code fragment into a method, with the method name explained.

When `[W5.5d] Implementation → Refactoring → When`

- One way to identify refactoring opportunities is by **code smells**
  - A surface indication that usually corresponds to a deeper problem in the system
- Periodic refactoring is a good way to pay off the **technical debt** a codebase has accumulated
- Too much refactoring when the benefits no longer justify the cost
  - ⇒ Some refactorings are ‘opposites’ of each other (e.g. extract method vs inline method)

Documentation

- Documentation for **developer-as-user** - for other developers to reuse software components
- Documentation for **developer-as-maintainer** - for other developers to maintain and evolve the code
- Good software documentation: Tutorials, How-to guides, Explanation, Technical reference
- Software documentation (both user-facing and developer-facing) is best kept in a text format for ease of version tracking.
- A writer-friendly source format is also desirable.

- Not enough for documentation to be accurate and comprehensive; should also be **comprehensible**
- **Describe Top-Down**
  - Top-down breadth-first explanation is easier to understand than a bottom-up one
  - Reader can travel down a path she is interested in until she reaches the component she is interested to learn in-depth
- **Minimal yet Sufficient**: Aim for 'just enough' developer documentation.
  - Provide higher level information that is not readily visible in the code or comments
  - Avoid duplicating. Describe the similarities in one place and emphasize only the differences in other places.

JAVADOC

`[W3.4a] Implementation → Documentation → Tools → JavaDoc → What`

- JavaDoc: Generates API documentation in HTML format from comments in the source code

An example method header comment in JavaDoc format:

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url An absolute URL giving the base location of the image.
 * @param name The location of the image, relative to the url argument
 * @return The Image at the specified URL.
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

Generated HTML documentation:

getImage
`public Image getImage(URL url, String name)`

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:
`url` - an absolute URL giving the base location of the image.
`name` - the location of the image, relative to the `url` argument.

Returns:
the image at the specified URL.

See Also:
`Image`

A minimal JavaDoc comment example for methods:

```
/**
 * Returns lateral location of the specified position.
 * If the position is unset, NaN is returned.
 *
 * @param x X coordinate of position.
 * @param y Y coordinate of position.
 * @param zone Zone of position.
 * @return Lateral location.
 * @throws IllegalArgumentException If zone is <= 0.
 */
public double computeLocation(double x, double y, int zone)
    throws IllegalArgumentException {
    // ...
}
```

A minimal JavaDoc comment example for classes:

```
package ...

import ...

/**
 * Represents a location in a 2D space. A <code>Point</code> ob
 * a coordinate represented by two integers e.g., <code>3,6</co
 */
public class Point {
    // ...
}
```

Error Handling

- Allows applications to recover gracefully from unexpected errors
    - Request user intervention
    - Application recovers on its own
    - Extreme cases: Application log the user off or shut down the system

---

Logging `[W6.4a] Implementation → Error Handling → Logging → What`

- The deliberate recording of certain information during a program execution for future reference
- Useful for troubleshooting problems
- A log file is like the black box of an airplane
    - Don't prevent problems but they can be helpful in understanding what went wrong

`[W6.4b] Implementation → Error Handling → Logging → How`

- Most programming environments come with logging systems that allow sophisticated forms of logging

```
import java.util.logging.*;
private static Logger logger = Logger.getLogger("Foo");

// log a message at INFO level
logger.log(Level.INFO, "going to start processing");
// ...
processInput();
if (error) {
    // log a message at WARNING level
    logger.log(Level.WARNING, "processing error", ex);
}
// ...
logger.log(Level.INFO, "end of processing");
```

Exceptions

Exceptions are used to deal with 'unusual' but not entirely unexpected situations that the program might encounter at runtime. E.g.:

- A network connection encounters a timeout due to a slow server.
- The code tries to read a file from the hard disk but the file is corrupted and cannot be read.

HOW `[W1.5d] Implementation → Error Handling → Exceptions → How`

1. Error occurs
2. Code being executed creates an **exception object** and hands it off to the runtime system
    - Method **throws the exception**

3. Runtime system attempts to find the **exception handler** in the call stack to handle it
    - Call stack: Ordered list of methods that had been called to get to the method where the error occurred
4. Exception handler **catches** the exception

Advantages of exception handling in this way:
- Ability to propagate error information through the call stack
- Separation of code that deals with 'unusual' situations from the code that does the 'usual' work

WHEN [W1.5f] Implementation → Error Handling → Exceptions → When

- Use exceptions only for 'unusual' conditions
- Use normal return statements to pass control to the caller for conditions that are 'normal'.

Assertions [W5.6a] Implementation → Error Handling → Assertions → What

- **Assertions** are used to define assumptions about the program state so that the runtime can verify them.
- If the runtime detects an **assertion failure**, it typically takes some drastic action

[W5.6b] Implementation → Error Handling → Assertions → How

- `assert`

```
x = getX();
assert x == 0 : "x should be 0";
```

- Assertions can be disabled without modifying the code.
    - java `-enableassertions` HelloWorld: run HelloWorld with assertions enabled
    - java `-disableassertions` HelloWorld: run it without verifying assertions
        - **Java disables assertions by default.**
            - Could create a situation where you think all assertions are being verified as true while in fact they are not being verified at all.
            - Remember to enable assertions when you run the program if you want them to be in effect

Java `assert`	JUnit assertions
Both check for a given condition	
In functional code	More powerful and customized for testing

[W5.6c] Implementation → Error Handling → Assertions → When

- Assertions be used liberally in the code
- Do not use assertions to do **work**.
    - Eg `assert` writeFile() : "File writing is supposed to return true";
    - Because assertions can be disabled.
    - If not, program will stop working when assertions are not enabled.
- Assertions are suitable for verifying assumptions about *Internal Invariants, Control-Flow Invariants, Preconditions, Postconditions, and Class Invariants*

Assertion	Exception
Handle errors in software	
Unusual condition created by the **user** (e.g. user inputs an unacceptable input) or the environment (e.g., a file needed for the program is missing).	Indicates the **programmer** made a mistake in the code (e.g., a null value is returned from a method that is not supposed to return null under any circumstances).

Defensive Programming

A defensive programmer codes under the assumption "if you leave room for things to go wrong, they will go wrong". ⇒ Try to eliminate any room for things to go wrong

- E.g. Config getConfig() { return config`.copy()`; // return a **defensive copy** }
- public Man getHusband() throws Exception { `if (h == null)` { `throw new Exception`(...); …
- **Not necessary to be 100% defensive all the time**
    - May be less prone to be misused or abused, but can also be more complicated and slower

Integration

- **Integration**: Combining parts of a software product to form a whole

---

Build Automation

- Build automation tools automate the steps of the build process, usually by means of build scripts.
  - Java Gradle, Maven, Apache Ant, GNU Make
  - JavaScript Grunt
  - Ruby Rake
- Some build tools also serve as **dependency management** tools
  - Download the correct version of the required libraries and update them regularly
  - Gradle, Maven

CONTINUOUS INTEGRATION AND CONTINUOUS DEPLOYMENT

An extreme application of build automation is called **Continuous Integration (CI)**

- integration, building, and testing happens automatically after each code change

A natural extension of CI is **Continuous Deployment (CD)**

- the changes are not only integrated continuously, but also deployed to end-users at the same time

CI/CD tools: Travis, Jenkins, Appveyor, CircleCI, GitHub Actions

---

Integration approaches

Late and one time VS Early and frequent

Late and one-time (not recommended)	**Early and frequent**
Wait till all components completed and integrate all finished components near the end of the project ● Late integration → Incompatibilities found → major rework required → cannot meet the delivery date	Integrate early and evolve each part in parallel, in small steps, re-integrating frequently ● Walking skeleton (high-level components needed for the first version in minimal form, compiles, and runs but may not produce any useful output yet) can be written first

Big-bang VS Incremental integration

Big-bang integration (not recommended)	**Early and frequent**
Integrate all (or too many) components (changes) at the same time ● Will uncover too many problems at the same time → Could make debugging and bug-fixing more complex	Integrate a few components at a time ● Surfaces integration problems in a more manageable way

Reuse

By reusing tried-and-tested components, the **robustness** of a new software system can be enhanced while **reducing the manpower and time requirement**.

Associated costs:

- Reused code may be an **overkill** (eg increase size, degrade performance of software)
- Reused software **may not be mature/stable enoug**h
    - Non-mature → Risk of dying off
- Might have **bugs, missing features, or security vulnerabilities**
- **Malicious cod**e can sneak into your product via compromised dependencies

---

Application Programming Interface (API)

Application Programming Interface (API) specifies the interface through which other programs can interact with a software component

- Contract between the component and its clients / component developer and the component user
- **Defining component APIs early is useful for developing components in parallel**
    - As the future behavior of the other components are now more predictable
- A software component can have an API.
- A **PUBLIC** method **(not private method)** of a class is part of its API.
- Sequence diagrams can be used to show how components interact with each other via APIs.

Examples

- A class has an API (e.g., API of the Java String class, API of the Python str class) which is a collection of public methods that you can invoke to make use of the class
- GitHub API is a collection of web request formats that the GitHub server accepts and their corresponding responses

---

Frameworks

The overall structure and execution flow of a specific category of software systems can be very similar. The similarity is an opportunity to reuse at a high scale.

- **Software framework** = Reusable implementation of a software (or part thereof) providing **generic** functionality that can be selectively customized to produce a **specific** application
    - Eclipse is an IDE framework that can be used to create IDEs for different programming languages.
- Some frameworks provide a complete implementation of a **default** behavior which makes them immediately usable
    - Eclipse is a fully functional Java IDE out-of-the-box
- A framework facilitates the adaptation and customization of some desired functionality
    - Eclipse plugin system can be used to create an IDE for different programming languages while reusing most of the existing IDE features of Eclipse

---

Frameworks vs Libraries

- A **library** is a collection of modular code that is general and can be used by other programs.

Libraries	Frameworks
Reuse mechanisms; more concrete than principles and patterns	
Meant to be used **'as is'**	Meant to be **customized / extended**
Your **code calls the library code**	**Framework code calls your code** (instead of code calling framework) ● **INVERSION OF CONTROL** technique ("Hollywood principle") ● E.g. Write test methods (code) that will be called by JUnit (framework)

Platform

- Provides a **runtime environment** for applications
- Examples:
    - JavaEE, .NET : Platform for writing enterprise applications

Quality Assurance

Static Analysis

Static Analysis	Dynamic Analysis
● Analysis of code without actually executing the code. ● Can find useful information such as unused variables, unhandled exceptions, style errors, and statistics ● Higher-end static analysis tools (static analyzers) can perform more complex analysis such as locating potential bugs ● **Linters** are a subset of static analyzers	● Requires the code to be executed to gather additional information about the code

---

Code Reviews

- The systematic examination of code with the intention of finding where the code can be improved
  - Pull Request reviews
  - In pair programming
  - Formal inspections
- Advantages of code review over testing:
  - Detect functionality defects and other problems such as coding standard violations.
  - Can verify non-code artifacts and incomplete code.
  - Does not require test drivers or stubs.
- Disadvantages:
  - Manual process → Error prone

Test Case Design

- Except for trivial SUTs, exhaustive testing is not practical
- Every test case adds to the cost of testing
- Test cases need to be designed to make the best use of testing resources → **E&E**
  - **EFFECTIVE testing** i.e., it finds a high percentage of existing bugs
  - **EFFICIENCY testing** i.e., it has a high rate of success (bugs found/test cases)

⇒ Each new test added should target a potential fault that not already targeted by existing test case

Positive and Negative Test Cases

- **Positive test case**: Designed to produce an expected/valid behavior
- **Negative test case**: Designed to produce a behavior that indicates an invalid / unexpected situation, eg error message, i == null; …

Black box vs Glass box

Based on how much of the SUT's internal details are considered when designing test cases:

- **Black-box** (specification-based / responsibility-based)
  - Test cases are designed exclusively based on the SUT's specified external behavior.
- **White-box** (**glass-box** / structured / implementation-based)
  - Test cases designed based on what is known about the SUT's implementation, i.e. the code

Testing based on Use Cases

Use cases can be used for **system testing** and **acceptance testing**

- MSS can be 1 test case. Each variation (due to extensions) can be another test case
- Tester has to choose data by considering equivalence partitions and boundary values
- High-priority use cases are given more attention, eg tested using a scripted approach

Equivalence Partitioning (EP)

- Most SUTs do not treat each input in a unique way. Instead, they process all possible inputs in a small number of distinct ways.
  - **Equivalence partitioning (EP)** is a test case design technique that uses the above observation to improve the effectiveness and efficiency of testing
- **Equivalence partition** (equivalence class)
  - A group of test inputs that are likely to be processed by the SUT in the same way

Specification of SUT ⇒	⇒ Equivalence Partitions
returns true if `m` (an int) is in the range [1..12]	[MIN_INT ... 0] [1 … 12] [13 … MAX_INT]
`isValidFlag(String s)`: boolean Returns `true` if `s` is one of ["F", "T", "D"]. The comparison is case-sensitive	["F"] ["T"] ["D"] ["f", "t", "d"] [any other string][null]
`squareRoot(String s): int` `s` is a `String` representing a positive integer Returns the square root of `s` if the square root is an integer; returns `0` otherwise.	[`s` does not represent a valid number] [`s` is a negative integer] [`s` has an integer square root] [`s` does not have an integer square root]

- By **dividing possible inputs into equivalence partitions**, you can
  - avoid testing too many inputs from one partition
  - ensure all partitions are tested

Identify the EPs of all data participants that can potentially influence the behaviour of the method, such as,

- the target object of the method call
- input parameters of the method call
- other data/objects accessed by the method such as global variables

Consider this method in the `DataStack` class: `push(Object o): boolean`

- Adds `o` to the top of the stack if the stack is not full.
- Returns `true` if the push operation was a success.
- Throws
  - `MutabilityException` if the global flag `FREEZE==true` .
  - `InvalidValueException` if `o` is null.

EPs:

- `DataStack` object: [full] [not full]
- `o` : [null] [not null]
- `FREEZE` : [true][false]

Consider the `Logic` component of the Minesweeper application. What are the EPs for the `markCellAt(int x, int y)` method? The partitions in **bold** represent valid inputs.

- `Logic` : PRE_GAME, **READY**, **IN_PLAY**, WON, LOST
- `x` : [MIN_INT..-1] **[0..(W-1)]** [W..MAX_INT] (assuming a minefield size of WxH)
- `y` : [MIN_INT..-1] **[0..(H-1)]** [H..MAX_INT]
- `Cell` at `(x,y)` : **HIDDEN**, MARKED, CLEARED

Consider a simple Minesweeper app. What are the EPs for the `newGame()` method of the `Logic` component?

As `newGame()` does not have any parameters, the only obvious participant is the `Logic` object itself.

Note that if the glass-box or the grey-box approach is used, other associated objects that are involved in the method might also be included as participants. For example, the `Minefield` object can be considered as another participant of the `newGame()` method. Here, the black-box approach is assumed.

Next, let us identify equivalence partitions for each participant. Will the `newGame()` method behave differently for different `Logic` objects? If yes, how will it differ? In this case, yes, it might behave differently based on the game state. Therefore, the equivalence partitions are:

- `PRE_GAME` : before the game starts, minefield does not exist yet
- `READY` : a new minefield has been created and the app is waiting for the player's first move
- `IN_PLAY` : the current minefield is already in use
- `WON` , `LOST` : let us assume that `newGame()` behaves the same way for these two values

Boundary Value Analysis (BVA) - Test case design heuristic

- Based on the observation that bugs often result from incorrect handling of boundaries of equivalence partitions
- When picking test inputs from an equivalence partition, **values near boundaries** (i.e. boundary values / corner cases) are more likely to find bugs
- Choose three values around the boundary to test: one value **from the boundary**, one value **just below the boundary**, and one value **just above the boundary**

Equivalence partition	Some possible test values (boundaries are in bold)
[1-12]	0,**1**,2, 11,**12**,13
[MIN_INT, 0] (MIN_INT is the minimum possible integer value allowed by the environment)	**MIN_INT**, MIN_INT+1, -1, **0** , 1
[any non-null String] (assuming string length is the aspect of interest)	**Empty String**, a String of **maximum possible length**
[prime numbers] ["F"] ["A", "D", "X"]	No specific boundary No specific boundary No specific boundary
[non-empty Stack] (assuming a fixed size stack)	Stack with: no elements, **one element**, two elements, **no empty spaces**, only one empty space

Combining Multiple Test Inputs

- An SUT can take multiple inputs
- Testing all possible combinations is effective but not efficient
- Need smarter ways to combine test inputs that are both **effective** and **efficient**

- **All combinations strategy**: Generates test cases for each unique combination of test inputs.
  - 3 inputs, each with possible number of values to be 3, 3, and 2 ⇒ 3 x 3 x 2 test cases
- **At least once strategy**: Includes each test input at least once. E.g. 3 inputs ⇒ 3 test cases
- **All pairs strategy**: For any given pair of inputs, all combinations between them are tested

    - Number of test cases is lower than all combinations strategy, but higher than at least once approach
    - Inputs p1 and p2 can generate 3 x 3 = 9 combinations. p1 and p3 can generate 3 x 2 = 6 combinations. p2 and p3 can generate 3 x 2 = 6 combinations. Total = 9 + 6 + 6
- **Random strategy**: Generates test cases using one of the other strategies and then picks a subset randomly

Heuristics for Combining Test inputs

- **Each VALID input AT LEAST ONCE in a POSITIVE test case**
    - Ensure that the valid input produces a correct result
    - If not, if you combine the valid input with an invalid one, you might not know if the valid input causes the bug.
    - Each valid test input should appear at least once in a test case that doesn't have any invalid inputs
- It is okay to combine valid values for different inputs.
- No more than one invalid test input should be in a given test case.
- **NOT to say NEVER have more than one invalid input in a test case**
    - SUT might work correctly when only one invalid input is given but not when a certain combination of multiple invalid inputs is given
    - Useful to have test cases with multiple invalid inputs, after confirming that the SUT works when only one invalid input is given
- **Test INVALID inputs INDIVIDUALLY *before combining* them**
    - To verify the SUT is handling a certain invalid input correctly, it is better to test that invalid input without combining it with other invalid inputs
    - A test case with multiple invalid inputs by itself does not confirm that the SUT works for each of those invalid inputs

Example

- SUT: `calculateGrade(participation, projectGrade, isAbsent, examScore)`
- Values to test: invalid values are underlined
    - participation: 0, 1, 19, 20, 21, 22
    - projectGrade: A, B, C, D, F
    - isAbsent: true, false
    - examScore: 0, 1, 69, 70, 71, 72

At least once strategy:

Case No.	participation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	VV/IV	69	...
4	20	D	VV/IV	70	...
5	21	F	VV/IV	71	Err Msg
6	22	VV/IV	VV/IV	72	Err Msg

- VV/IV = Any Valid or Invalid Value, Err Msg = Error Message

Apply **each valid input at least once in a positive test case**.

- Test case 5 has a valid value for projectGrade=F that doesn't appear in any other positive test case.
  ⇒ Replace test cases 5 with 5.1 and 5.2 to rectify that.

Case No.	participation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	VV	69	...
4	20	D	VV	70	...
5.1	VV	F	VV	VV	...
5.2	21	VV/IV	VV/IV	71	Err Msg
6	22	VV/IV	VV/IV	72	Err Msg

Apply **no more than one invalid input in a test case**.

- Test cases 5.2 and 6 don't follow this heuristic.

5.2	21	VV	VV	VV	Err Msg
5.3	22	VV	VV	VV	Err Msg
6.1	VV	VV	VV	71	Err Msg
6.2	VV	VV	VV	72	Err Msg

Assume there is a dependency between the inputs `examScore` and `isAbsent` such that an absent student can only have `examScore=0`. To cater for the hidden invalid case arising from this, add a new test case where `isAbsent=true` and `examScore!=0`. In addition, test cases 3-6.2 should have `isAbsent=false` so that the input remains valid.

Case No.	participation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	false	69	...
4	20	D	false	70	...
5.1	VV	F	false	VV	...
5.2	21	VV	false	VV	Err Msg
5.3	22	VV	false	VV	Err Msg
6.1	VV	VV	false	71	Err Msg
6.2	VV	VV	false	72	Err Msg
7	VV	VV	true	!=0	Err Msg

-

Quality Assurance

Software Quality Assurance (QA) is the process of ensuring that the software being built has the required levels of quality.

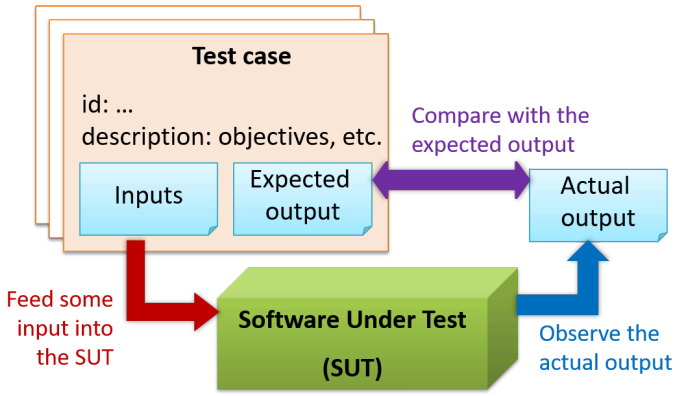
Quality Assurance = Validation + Verification

- **Validation**: are you *building the right system* i.e., are the requirements correct?
- **Verification**: are you *building the system right* i.e., are the requirements implemented correctly?

Formal Verification

- Uses mathematical techniques to prove the correctness of a program
- Advantages:
  - Can **prove the absence of errors** (vs **TESTING** - only prove the **presence** of errors)
- Disadvantages:
  - Only proves the compliance with the specification, but not the actual utility of the software.
  - Requires highly specialized notations and knowledge → Expensive → **More commonly used in safety-critical software such as flight control systems**

Testing



- When testing, you execute a set of test cases

Example: A minimal test case for testing a browser:

- **Input** – Start the browser using a blank page (vertical scrollbar disabled). Then, load longfile.html located in the test data folder.
- **Expected behavior** – The scrollbar should be automatically enabled upon loading longfile.html.

Test cases can be determined based on the specification, reviewing similar existing systems, or comparing to the past behavior of the SUT.

For each test case:

1. Feed the input to the SUT
2. Observe the actual output
3. Compare actual output with the expected output

Test case failure

- Mismatch between the expected behavior and the actual behavior
- A failure indicates a potential **defect** (or a bug)
    - 'potential' as error could be in the test case itself

Regression Testing

WHAT `[W2.6b] Quality Assurance → Testing → Regression Testing → What`

- Modification may result in **some unintended and undesirable effects (REGRESSION)**
- **Regression testing**
    - Re-testing of the software to detect regressions
    - Need not be automated
- Regression testing is more practical when automated.

Test Automation

An automated test case can be run programmatically and the result of the test case (pass or fail) is determined programmatically.

AUTOMATED TESTING OF CLI APPLICATIONS

Semi-automate testing of a CLI (Command Line Interface) app

- Use input/output re-direction
  - Technique only suitable when testing CLI apps, and
  - only if the exact output can be predetermined
    - (Vs output varies from one run to the other (e.g. it contains a time stamp))

1. Feed the app with a sequence of test inputs that is stored in a file while redirecting the output to another file.
2. Compare the actual output file with another file containing the expected output.

Example: Testing a CLI app called `AddressBook`

1. Store the test input in the text file `input.txt`
   ```
   add Valid Name p/12345 valid@email.butNoPrefix
   add Valid Name 12345 e/valid@email.butPhonePrefixMissing
   ```
2. Store the output you expect from the SUT in another text file `expected.txt`
   ```
   Command: || [add Valid Name p/12345 valid@email.butNoPrefix]
   Invalid command format: add

   Command: || [add Valid Name 12345 e/valid@email.butPhonePrefixMissing]
   Invalid command format: add
   ```
3. Run the program `java AddressBook < input.txt > output.txt`
   - redirect the text in `input.txt` as the input to `AddressBook`
   - redirect the output of `AddressBook` to a text file `output.txt`

   - Based on language eg for AddressBook.**py**
     - **python** `AddressBook < input.txt > output.txt`
   - Windows: use a normal MS-DOS terminal (i.e., cmd.exe) to run the app, not a PowerShell window
4. `FC output.txt expected.txt`
   - Compare output.txt with the expected.txt
   - Utility
     - Windows' `FC` (i.e. File Compare) command
     - Unix's `diff` command
     - GUI tool such as WinMerge

TEST AUTOMATION USING TEST DRIVERS

- **Test driver**: The code that ‘drives’ the SUT for the purpose of testing

- Invoking the SUT with test inputs and verifying if the behavior is as expected

`PayrollTest` 'drives' the `Payroll` class by sending it test inputs and verifies if the output is as expected.

```
public class PayrollTest {
    public static void main(String[] args) throws Exception {

        // test setup
        Payroll p = new Payroll();

        // test case 1
        p.setEmployees(new String[]{"E001", "E002"});
        // automatically verify the response
        if (p.totalSalary() != 6400) {
            throw new Error("case 1 failed ");
        }
        System.out.println("All tests passed");
    }
}
```

TEST AUTOMATION TOOLS

- JUnit is a tool for automated testing of Java programs

```
@Test
    public void testTotalSalary() {
        Payroll p = new Payroll();

        // test case 1
        p.setEmployees(new String[]{"E001", "E002"});
        assertEquals(6400, p.totalSalary());
}
```

AUTOMATED TESTING OF GUIs

- Testing the GUI is much harder than testing the CLI (Command Line Interface) or API
- Moving as much logic as possible out of the GUI can make GUI testing easier
- There are testing tools that can automate GUI testing: TestFX, Visual Studio, Selenium

---

Developer Testing

Developer testing = Testing done by the developers themselves

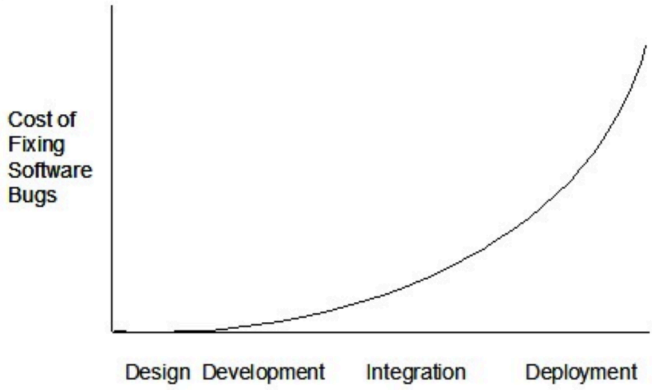
WHY `[W3.6b] Quality Assurance → Testing → Developer Testing → Why`

Delaying testing until the full product is complete has disadvantages:

- Locating the cause of a test case failure is difficult due to the larger search space
- Fixing a bug found during such testing could result in major rework

- One bug might 'hide' other bugs
- The delivery may have to be delayed if too many bugs are found during testing.

Better to do early testing



Developers testing their own code	
Pros	Cons
• Can be done early (the earlier we find a bug, the cheaper it is to fix). • Can be done at lower levels, for example, at the operation and class levels (testers usually test the system at the UI level). • Possible to do more thorough testing as developers know the expected external behavior and the internal structure of the component. • Forces developers to take responsibility for their own work (cannot claim that "testing is the job of the testers").	• Subconsciously only test situations that he knows to work (i.e. test it too 'gently'). • Blind to his own mistakes (if he did not consider a certain combination of input while writing the code, it is possible for him to miss it again during testing). • May have misunderstood what the SUT is supposed to do in the first place. • May lack the testing expertise.

Unit Testing `[W3.7c] Quality Assurance → Testing → Unit Testing → What`

- Testing individual units (methods, classes, subsystems, ...) to ensure each piece works correctly.

```
class FooTest {
    @Test
    void read() {
        // a unit test for Foo#read() method
    }
}
```

STUBS `[W3.7e] Quality Assurance → Testing → Unit Testing → Stubs`

A proper unit test requires the **unit** to be tested **in isolation**

- Bugs in the dependencies cannot influence the test
- If a Logic class depends on a Storage class, unit testing the Logic class requires isolating the Logic class from the Storage class

Stubs can isolate the SUT (Software Under Test e.g. the unit being tested) from its dependencies.

- A stub has the same interface as the component it replaces, but its implementation is so simple that it is unlikely to have any bugs

Not testing Logic in isolation from its dependencies	Proper unit test
`class DatabaseStorage implements Storage { ... }`	`class StorageStub implements Storage { ... }`
`@Test void getName() { Logic logic = new Logic(new DatabaseStorage()); assertEquals("Name: John", logic.getName(5)); }`	`@Test void getName() { Logic logic = new Logic(new StorageStub()); assertEquals("Name: Adam", logic.getName(5)); }`

```
class DatabaseStorage implements Storage
{

    @Override
    public String getName(int index) {
        return
readValueFromDatabase(index);
    }
}
```

```
class StorageStub implements Storage {

    @Override
    public String getName(int index) {
        if (index == 5) {
            return "Adam";
        } else {
            throw new
UnsupportedOperationException();
        }
    }
}
```

```
@Test
void getName() {
    Logic logic = new Logic(new
DatabaseStorage());
    assertEquals("Name: John",
logic.getName(5));
}
```

```
@Test
void getName() {
    Logic logic = new Logic(new
StorageStub());
    assertEquals("Name: Adam",
logic.getName(5));
}
```

Integration Testing

- Testing whether different parts of the software work together (i.e. integrates) as expected
- Not simply repeating the unit test cases using the actual dependencies

Pure integration test:

Suppose a class `Car` uses classes `Engine` and `Wheel`.

1. **Unit** test `Engine` and `Wheel`.
2. **Unit** test `Car` **in isolation of** `Engine` and `Wheel`, using **STUBS** for `Engine` and `Wheel`.
3. **Integration test** for `Car` by using it together with the `Engine` and `Wheel` classes.

Developers often use a **hybrid** of unit and integration tests to minimize the need for stubs:

Suppose a class `Car` uses classes `Engine` and `Wheel`.

1. **Unit** test `Engine` and `Wheel`.
2. **Integration test** for `Car` by using it together with the `Engine` and `Wheel` classes. This step should include test cases that are meant to **unit** test `Car` (used in 2. above) and test cases meant to test the integration of Car with Wheel and Engine (i.e. pure integration test used in 3. above)

* `Car` is never tested in isolation of its dependencies. Given that its dependencies are already **unit** tested, the risk of bugs in `Engine` and `Wheel` affecting the testing of `Car` can be considered minimal.

System Testing

- Take the **whole system** and test it against the **system specification**
- System test cases are based on the specified external behavior of the system
- Includes testing against **non-functional requirements**
    - Performance testing – to ensure the system responds quickly.
    - Load testing (stress testing / scalability testing) – ensure system can work under heavy load.
    - Security testing – to test how secure the system is.
    - Compatibility testing, interoperability testing – check if system can work with other systems.
    - Usability testing – to test how easy it is to use the system.
    - Portability testing – to test whether the system works on different platforms.

Acceptance Testing / User Acceptance Testing (UAT)

- Test the system to ensure it meets the **user requirements**
- Passing system tests does not necessarily mean passing acceptance testing

SYSTEM Testing	ACCEPTANCE Testing
Done against the **SYSTEM specification**	Done against the **REQUIREMENTS specification**
Done by testers of the project team	Done by a team that represents the customer
Done on the development environment or a test bed	Done on the deployment site or on a close simulation of the deployment site
Both negative and positive test cases	More focus on positive test cases

In many cases one document serves as both a requirement specification and a system specification

SYSTEM Specification	REQUIREMENTS Specification
Can also include details on how it will **fail gracefully when pushed beyond limits**, how to recover, etc. specification	Limited to how the system behaves in **normal working conditions**
Written in terms of how the system solves those problems (e.g. explain the email search feature)	Written in terms of problems that need to be solved (e.g. provide a method to locate an email quickly)
Could contain additional APIs not available for end-users (for the use of developers/testers)	Specifies the interface available for intended end-users

Alpha / Beta Testing

ALPHA Testing	BETA Testing
Performed by the users, under **controlled conditions** set by the software development team	Performed by a selected **subset** of target users of the system in their **natural work setting**

Exploratory vs Scripted Testing

Exploratory testing = Reactive testing, error guessing technique, attack-based testing, and bug hunting

Better to use a MIX of both:

SCRIPTED Testing (predetermined test cases)	**EXPLORATORY Testing (usually manual)**
First write a set of test cases based on expected behavior of SUT. Then perform testing based on that.	Devise test cases on-the-fly (eg by playing a game in various ways), creating new test cases based on the results of the past test cases
More systematic → Likely to discover more bugs given sufficient time	• Success depends on tester's prior experience, intuition • May detect some problems in a relatively short time, but not prudent as the sole means of testing a critical system

Dependency injection (with stubs)

- Process of 'injecting' objects to **replace current dependencies with a different object**
- E.g. Inject stubs to isolate the SUT from its dependencies so that it can be tested in isolation.

Testability

- Indication of how easy it is to test an SUT

Test Coverage

- Metric used to measure the extent to which testing exercises the code
  - Function / method coverage: Based on functions executed
  - Statement coverage : LOC executed
  - Decision / branch coverage: Eg an `if` statement evaluated to both true and false with separate test cases during testing is considered 'covered'.
  - Condition coverage: Based on the boolean sub-expressions, each evaluated to both true and false with different test cases.
  - Path coverage: Possible paths through a given part of the code executed.
  - Entry / exit coverage: Possible calls to and exits from the operations in the SUT
- Measuring coverage is often done using **coverage analysis tools**
- Coverage analysis can be useful in improving the quality of testing

Test-Driven Development (TDD)

- Advocates writing the tests before writing the SUT, while evolving functionality and tests in small increments
  - First define the precise behavior of the SUT using test code
  - Then update the SUT to match the specified behavior

For TDD	**Against TDD**

● Testing will not be neglected due to time pressure (as it is done first). ● Forces developers to think about how exactly the component should behave, before jumping into implementing it. ● Avoids wasting programmer effort (i.e. the code ends up doing exactly what's needed; no less, no more). ● Forces programmer to automate all tests	● Since tests can be seen as ‘executable specifications’, programmers tend to neglect other forms of documentation. ● Promotes ‘trial-and-error’ coding instead of making programmers think through their algorithms (i.e. ‘just keep hacking until all tests pass’). ● Gives a false sense of security. (What if you forgot to test certain scenarios?) ● Not intuitive. Some programmers might resist adopting TDD.

Project Management

Revision Control

Revision control: The process of managing multiple versions of a piece of information

- Track the history and evolution of your project
- Easier to collaborate
  - Identify and resolve conflicts in incompatible changes made simultaneously
- Recover from mistakes
  - Revert to an earlier version
- Work simultaneously on, and manage the drift between, multiple versions of your project

Revision / Version

- State of a piece of information at a specific time that is a result of some changes

Revision control software (RCS) / Version Control Software (VCS)

- Software tools that automate the process of Revision Control
- **Git**, Mercurial, Subversion (SVN), Perforce, CVS (Concurrent Versions System), Bazaar, TFS (Team Foundation Server), and Clearcase

Github

- Web-based project hosting platform for projects using **Git** for revision control
- Similar services: GitLab, BitBucket, SourceForge

Repositories `[W2.3b] Project Management → Revision Control → Repositories`

Repository: Database that stores revision history of a directory being tracked by RCS software (e.g. Git)
Working directory: Root directory revision-controlled by Git (eg directory in which the repo was initialized)

Can have multiple **repos** in your computer, each **repo** revision-controlling files of a different **working directory**, e.g. files of different projects.

Saving history `[W2.3d] Project Management → Revision Control → Saving histor…`

Tracking and ignoring

- In a repo, you can specify which files to track and which files to **ignore** (eg temporary files created during the build / test process)

Staging and committing

- **Committing** saves a snapshot (commit) of the current state of the tracked files in the revision control history
  - Commit: a change (revision) saved in the Git revision history
- When ready to commit, first add the specific changes you want to commit to a **staging area** (index)
  - Stage: Instructing Git to prepare a file for committing

Using history `[W2.3g] Project Management → Revision Control → Using history`

RCS tools store the history of the working directory as a series of commits.

Each commit in a repo is a recorded point in the history of the project that is uniquely identified by an auto-generated **hash** e.g. a16043703f28e5b3dab95915f5c5e5bf4fdc5fc1.

You can **tag** a specific commit with a more easily identifiable name e.g. v1.0.2.

To see what changed between two points of the history, you can ask the RCS tool to **diff** the two commits in concern.

To restore the state of the working directory at a point in the past, you can **checkout** the commit in concern.

Remote repositories

`[W2.4a] Project Management → Revision Control → Remote repositories`

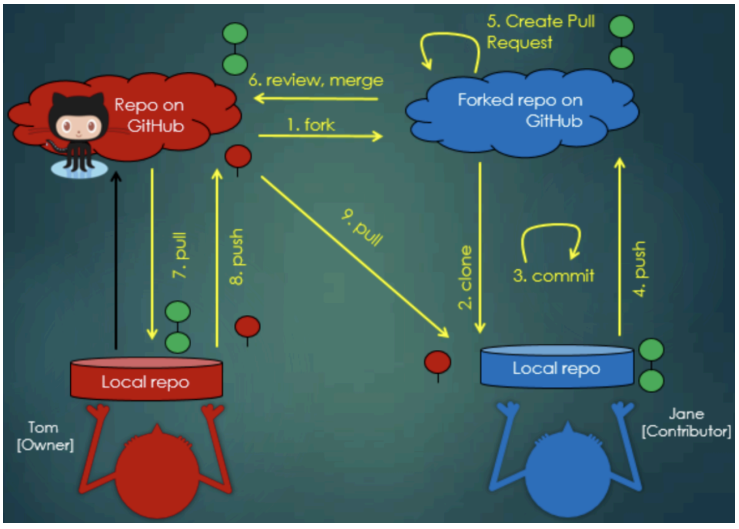
- Repos that are hosted on remote computers
- Possible to set up your own remote repo on a server

A repo can work with any number of other repositories as long as they have a shared history

- **Clone** a repo
  - Original repo is the **upstream** repo
    - A repo can have multiple upstream repos
- Can **pull** (**fetch**) from one repo to another, to receive new commits in the second repo
  - Allows you to **sync** your clone with the upstream repo
- Can **push** new commits in one repo to another repo

Fork: A remote copy of a remote repo

Pull request (PR): A mechanism for contributing code to a remote repo



Branching `[W3.1a] Project Management → Revision Control → Branching`

- Process of evolving multiple versions of the software in parallel

- Merge conflicts Git - Dealing with Merge Conflicts
    - Merge two branches that had changed the same part of the code

---

Forking flow `[W7.7a] Project Management → Revision Control → Forking flow`

In the **forking workflow**, the 'official' version of the software is kept in a remote repo designated as the 'main repo'. All team members fork the main repo and create pull requests from their fork to the main repo.

- Benefit: Does not require most contributors to have write permissions to the main repository. Only those who are merging PRs need write permissions.
- Drawback: Extra overhead of sending everything through forks.

Centralized / Distributed RCS (CRCS / DRCS)

Centralized RCS (CRCS)	Distributed / Decentralized RCS (DRCS)
Central remote repo that is shared by the team	Allows **multiple remote / local** repos working together

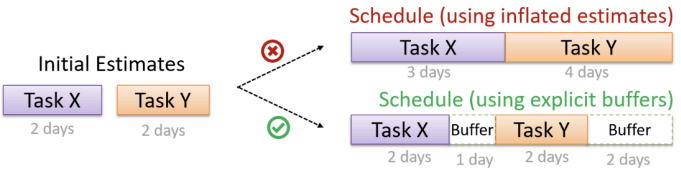
Project Planning

MILESTONES [W7.6a] Project Management → Project Planning → Milestones

- A milestone is the end of a stage which indicates significant progress.

BUFFERS [W7.6b] Project Management → Project Planning → Buffers

- A buffer is time set aside to absorb any unforeseen delays

• Do not inflate task estimates to create hidden buffers • Explicit buffers → Easier to detect incorrect effort estimates which can serve as feedback to improve future effort estimates	

ISSUE (BUG) TRACKERS

- Track task assignment and progress

WORK BREAKDOWN STRUCTURE (WBS)

- Depicts information about tasks and their details in terms of subtasks
- Effort is traditionally measured in man hour/day/month
- All tasks should be well-defined (should be clear as to when the task will be considered done)

Bad	Better
more coding	implement component X
do research on UI testing	find a suitable tool for testing the UI

GANTT CHARTS [W7.6e] Project Management → Project Planning → Gantt charts

- A 2-D bar-chart, drawn as time vs tasks (represented by horizontal bars)

TEAM STRUCTURES

Egoless team (democratic)	Chief programmer team	Strict hierarchy team
Every team member is equal in terms of responsibility and accountability	A single authoritative figure, the chief programmer (directs and coordinates the effort)	Strictly defined organization among the team members
Higher risk of falling apart due to the absence of an authority figure to manage the team.	Under a suitably qualified leader, such a team structure is known to produce successful work.	Reduce communication overhead in a large, resource-intensive, complex project

SDLC Process Models

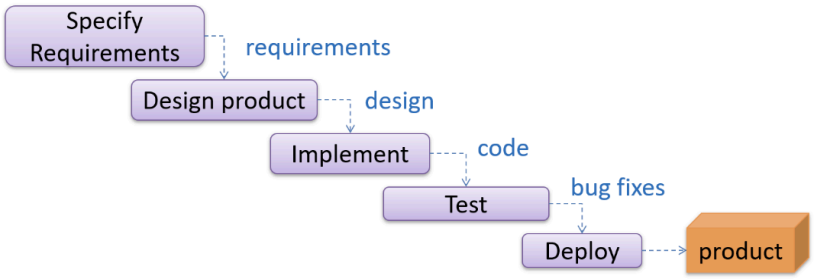
Software development lifecycle (SDLC)

- Stages: **REQUIREMENTS, ANALYSIS, DESIGN, IMPLEMENTATION, TESTING**
- Approach: Software development lifecycle models (software process models)

Sequential Approach	Iterative Approach
Final quality depends on the quality of each phase. Any quality problems in any phase could result in a low quality product.	Frequent reworking can deteriorate the design. Frequent refactoring should be used to prevent this. Frequent customer feedback can help to improve the quality (i.e. quality as seen by the customer).
High risk of overshooting the deadline. Any delay in any phase can result in overshooting the deadline with nothing to deliver	**Less risk of overshooting the deadline**. If the last iteration got delayed, we can always deliver the previous version. However, this does not guarantee that all features promised at the beginning will be delivered by the deadline.
Carry on even if the budget is exceeded because there is no intermediate version to fall back on	**Can always stop before the project budget is exceeded.** However, this does not guarantee that all features promised at the beginning will be delivered under the estimated cost.
Customer satisfaction is guaranteed if the product was delivered as promised and initial requirements proved to be accurate. Customer does not need to give frequent feedback during the project's development.	The customer gets many opportunities to guide the product in the direction he wants. The customer gets to change requirements even in the middle of the product's development. Both of these can **increase customer satisfaction**
Easier to measure progress against the plan, although this does not ensure eventual success	Hard to measure progress against a plan, as the plan itself keeps changing
Can save time because we minimize rework.	Requirements are not fixed. Overshooting the deadline is not an option. Gives a chance to learn lessons from one iteration and apply them in the next.

SEQUENTIAL (waterfall) models

Organizes the project based on activities. Views software development as a **linear process**

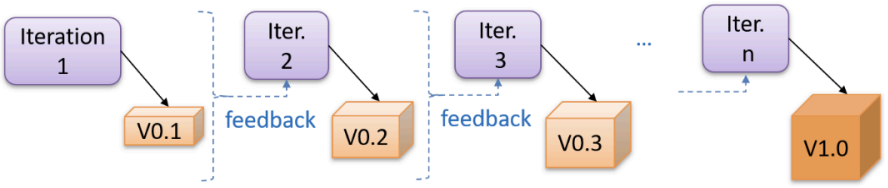


- When one stage (eg requirements phase) of the process is completed, it produces some artifacts (eg list of requirements) to be used in the next stage (eg design phase)
- A strict sequential model project moves only in the **forward direction**
    - Complete the stage before starting the next
- For a project that produces software to solve a **well-understood problem**

    - Requirements can remain stable
    - Effort can be estimated accurately
- However, real-world projects often tackle problems that are not well-understood at the beginning

ITERATIVE models - Organizes project based on functionality

Produce the software by going through **several iterations**, build upon version produced previous iteration



- Each iteration produces a new version of the product; can potentially go through all stages of SDLC.

A project can be done as a mixture of breadth-first and depth-first iterations

Breadth-first approach	Depth-first approach
An iteration evolves all major components and all functionality areas in parallel	An iteration focuses on fleshing out only some components or some functionality area
Working product at the end of each iteration	Might not produce a working product

Agile models (eXtreme Programming (XP) and Scrum)

Better ways of developing software by doing it and helping others do it. Value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

Scrum

A process skeleton that contains sets of practices and predefined roles.

- Scrum Master - maintains the processes (typically in lieu of a project manager)
- Product Owner - represents the stakeholders and the business
- Team - cross-functional group who do the actual analysis, design, implementation, testing, etc.

- A Scrum project is divided into iterations called **Sprints**
- Each sprint is preceded by a planning meeting
- During each sprint, the team creates a potentially deliverable product increment
- Scrum enables creation of self-organizing teams by encouraging co-location of all team members
- **Recognises that during a project, customers can change their minds about what they want and need** (requirements churn)
    - Problem cannot be fully understood or defined → Focus on maximizing the team's ability to deliver quickly and respond to emerging requirements

Daily Scrum

- In Scrum, on each day of a sprint, the team holds a daily scrum meeting called the "daily scrum.
- During the daily scrum, each team member answers 3 questions

    - What did you do yesterday? What will you do today? Any impediments in your way?
- Issues that are raised are taken offline and usually dealt with by the relevant subgroup immediately after the meeting.

Extreme Programming (XP)

- Stresses **customer satisfaction**; delivers the software you need as you need it
- Empower developers to confidently respond to changing customer requirements
- Emphasizes teamwork
- Improve a project in 5 essential ways: communication, simplicity, feedback, respect, courage
- A set of simple rules
- Related: Pair programming, CRC cards, project velocity, and standup meetings

Principles

SOLID Principles

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP): No client should be forced to depend on methods it don't use.
- Dependency Inversion Principle (DIP)
    - High-level modules should not depend on low-level modules. Both should depend on abstractions.
    - Abstractions should not depend on details. Details should depend on abstractions.

Separation Of Concerns principle (SoC)

- To achieve better modularity, separate the code into distinct sections, such that each section addresses a separate **concern** (a set of information that affects the code)
- Examples for concerns:
    - A specific feature, such as the code related to the add employee feature
    - A specific aspect, such as the code related to persistence or security
    - A specific entity, such as the code related to the Employee entity
- **Reduces functional overlaps among code sections** and also **limits the ripple effect when changes are introduced** to a specific part of the system
- Can be applied at the class level and s at higher levels
    - n-tier architecture: Each layer has a well-defined functionality that has no functional overlap with each other
- Should lead to **HIGHER COHESION** and **LOWER COUPLING**
    - **Cohesion**: How strongly-related and focused the various responsibilities of a component are
    - **Coupling**: Degree of dependence between components, classes, methods, etc
- Eg By making 'user interaction' the GUI class's sole responsibility, we increase its **cohesion**. This is also in line with the **separation of concerns** (i.e., we separated the concern of user interaction) and the **single responsibility principle** (the GUI class has only one responsibility).

Single Responsibility Principle (SRP)

- A class should have one, and only one, reason to change.

Liskov Substitution Principle (LSP)

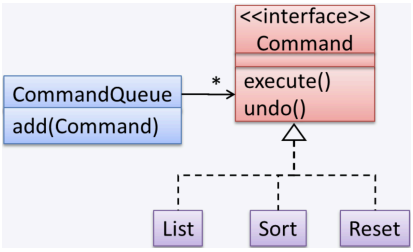
- Derived classes must be substitutable for their base classes
- **SUBCLASS should NOT be more RESTRICTIVE than the behavior specified by SUPERCLASS**
- `Rectangle#resize()` method can take any integers for height and width. This is **violated by the subclass** `Square#resize()` because it does not accept a height that is different from the width.

`class Rectangle {`	`class Square extends Rectangle {`

`void resize(int height, int width) {`
`  ...`
`}`
	`@Override`
	`void resize(int height, int width) {`
	`  if (height != width) { ...`

Open-Closed Principle (OCP)

- Make a code entity easy to adapt and reuse **without needing to modify** the code entity itself
- A module should be open for extension (extend the behaviour) but closed for modification
- Often requires separating the **specification (i.e. interface)** of a module from its **implementation**

	Behavior of `CommandQueue` class can be altered by adding more concrete `Command` subclasses. For example, by including a `Delete` class alongside `List`, `Sort`, and `Reset`, the `CommandQueue` can now perform delete commands without modifying its code at all. ⇒ **Behavior was extended without having to modify its code** ⇒ Open to extensions, but closed to modification
Alter behavior of a Java generic class by passing a different class as a parameter	ArrayList students = new `ArrayList`<`Student`>(); ArrayList admins = new `ArrayList`<`Admin`>();

Law of Demeter (LoD)

- **Prevent objects from navigating the internal structures of other objects; REDUCE COUPLING**
- An object should have limited knowledge of another object.
- An object should only interact with objects that are closely related to it.
- Don't talk to strangers. / Principle of least knowledge

A method `m` of an object `O` should invoke only the methods of the following kinds of objects:

- The object `O` itself
- Objects passed as parameters of `m`
- Objects created / instantiated in `m` (directly or indirectly) ; **Can invoke methods of created objects**
- Objects from the direct association of `O`

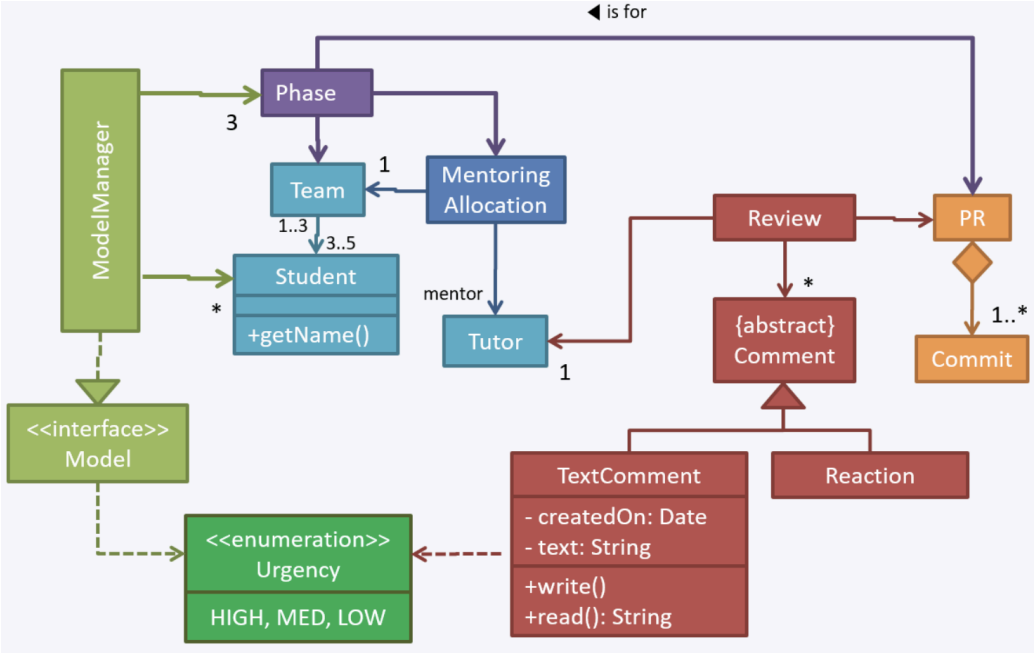
`void foo(Bar b) {`
`  Goo g = b.getGoo();`
`  g.doSomething();`
`}`
	Violates LoD
	While `b` is a 'friend' of `foo` (because it receives it as a parameter), `g` is a 'friend of a friend' (which should be considered a 'stranger'), and `g.doSomething()` is analogous to 'talking to a stranger'.

Tools

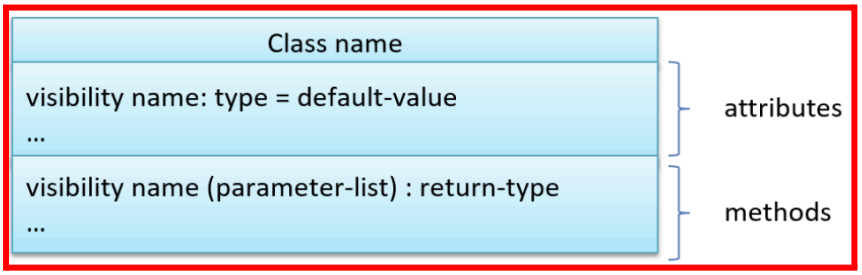
UML

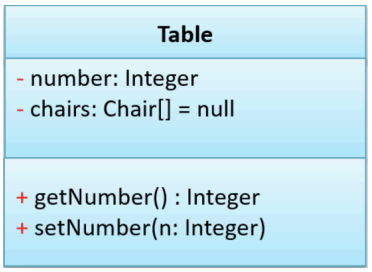
Class Diagrams (model STRUCTURE)

- UML class diagrams describe the **STRUCTURE** (NOT BEHAVIOR) of an OOP solution

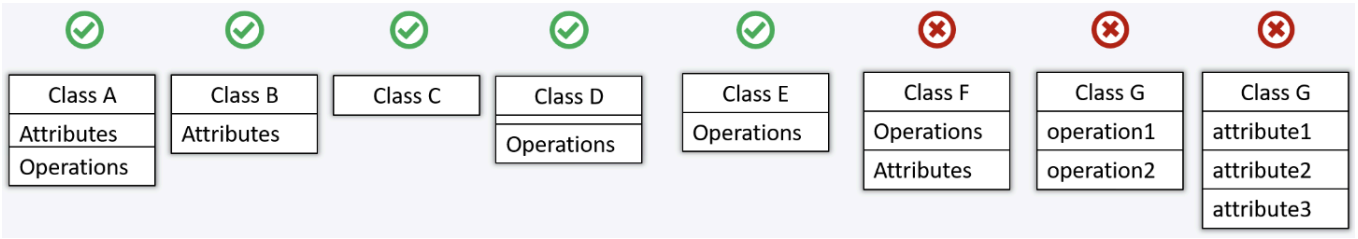


CLASSES





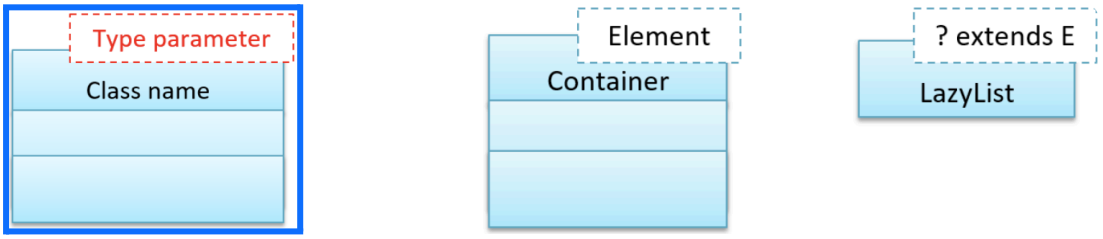
- The 'Operations' and / or the 'Attributes' compartment **may be omitted / empty**
- 'Attributes' always appear above the 'Operations'
- All operations / attributes in one compartment.
    - NOT each operation / attribute in a separate compartment.



Visibility of attributes and operations: Level of access allowed for each member

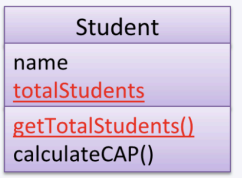
- `+` : public `-` : private `#` : protected `~` : package private
- No *default* visibility (will be unknown if the visibility is not specified)

Generic classes can be shown as given below:



CLASS-LEVEL MEMBERS

Underlines denote class-level attributes and methods.

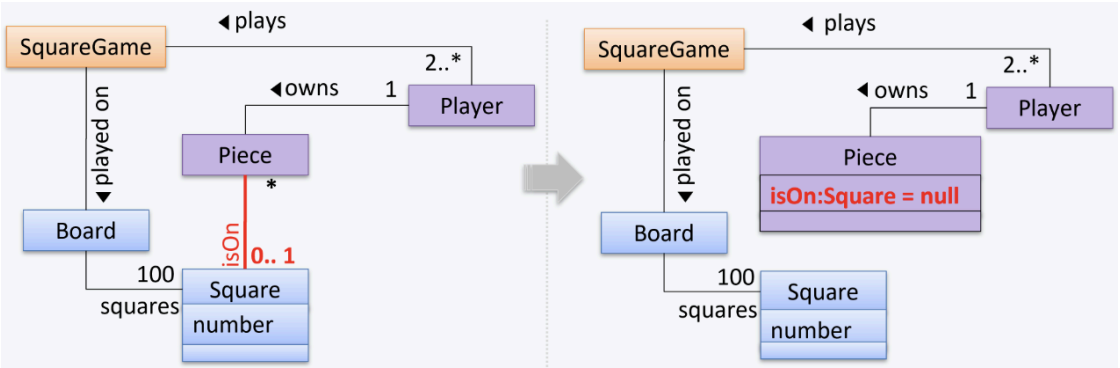


ASSOCIATIONS (main connections among classes)

Solid line to show an association between two classes.

Associations can be shown as Attributes (instead of a line)

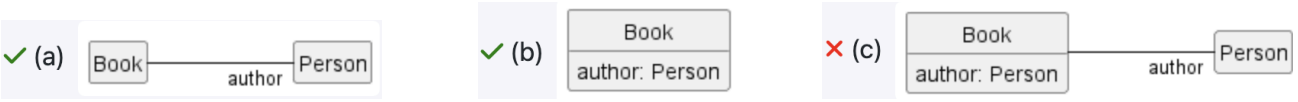
- Association multiplicities and default value can be shown as part of attribute (both are optional)
  - `name: type [multiplicity] = default value`



- `isOn` attribute of the `Piece` class
  - A `Piece` may or may not be on a `Square`
- `isOn` attribute can be `null` or refer to a `Square` object
  - Replaces the 0..1 multiplicity of the association

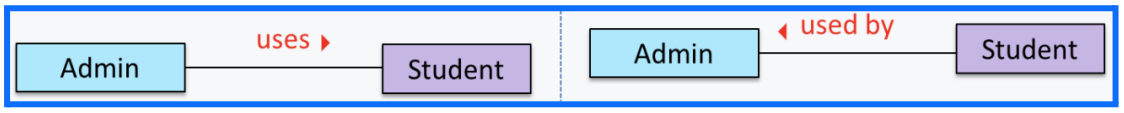
The association that a `Board` has 100 `Square`s can be shown in either of these two ways:

Show each association as EITHER an attribute or a line but NOT BOTH.



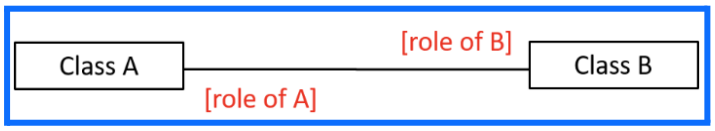
ASSOCIATION LABELS in Associations

- Describe the meaning of the association



ASSOCIATION ROLE in Associations

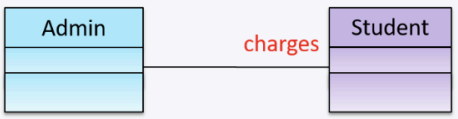
- Indicate the role played by the classes in the association.




```
class Man {
    Woman wife;
}
```

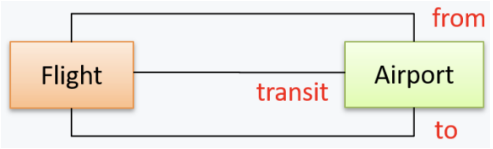
```
class Woman {
    Man husband;
}
```

The role of `Student` objects in this association is `charges`
Admin is in charge of students)




```
class Admin {
    List<Student> charges;
}
```

- Useful for differentiating among multiple associations between the same two classes.



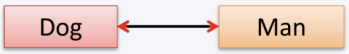
NAVIGABILITY in Associations (in class diagrams and object diagrams)

- Tells us which objects hold references to which other objects
  - If a Category object is related to another Category object, each of them will have a reference to the other. (bidirectional, not self-association)
- Arrow**head** (not the entire arrow) to indicate the **navigability** of an association

In this example, the navigability is unidirectional, and is from the `Logic` class to the `Minefield` class. That means if a `Logic` object `L` is associated with a `Minefield` object `M`, `L` has a reference to `M` but `M` doesn't have a reference to `L`.


```
class Logic {
    Minefield minefield;
    // ...
}

class Minefield {
    //...
}
```

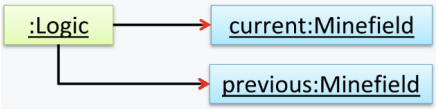
Here is an example of a bidirectional navigability; i.e., if a `Dog` object `d` is associated with a `Man` object `m`, `d` has a reference to `m` and `m` has a reference to `d`.




```
class Dog {
    Man man;
    // ...
}

class Man {
    Dog dog;
```

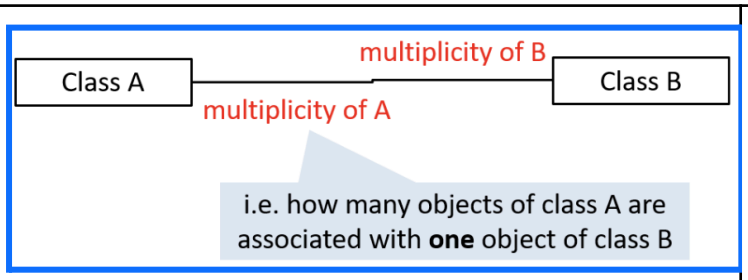
- Navigability can be shown in **class** diagrams and **object** diagrams.

According to this object diagram, the given `Logic` object is associated with and aware of two `MineField` objects.

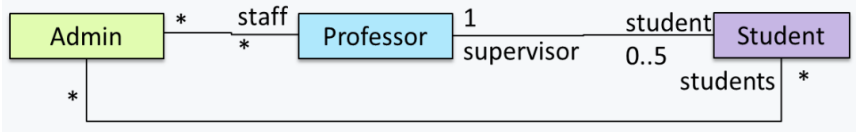


MULTIPLICITY in Associations

No default multiplicity (will be unknown if the multiplicity is not specified)



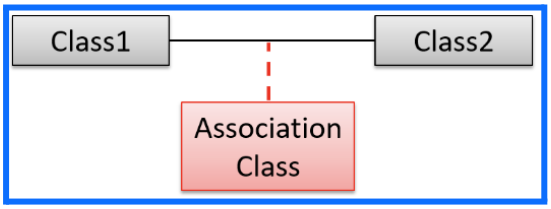
- `0..1` : optional association (0 or 1); for normal instance-level variables
- `1` : compulsory, must be linked to one object at all times.
- `*` : can be linked to 0 or more objects.
- `n..m` : the number of linked objects must be within n to m inclusive.

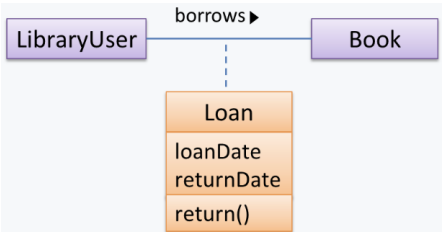


- Each student must be supervised by exactly one professor. i.e. There cannot be a student who doesn't have a supervisor or has multiple supervisors.
- A professor cannot supervise more than 5 students but can have no students to supervise.
- An admin can handle any number of professors and any number of students, including none.
- A professor/student can be handled by any number of admins, including none.

ASSOCIATION CLASSES

- Dashed line to show connection to an association link



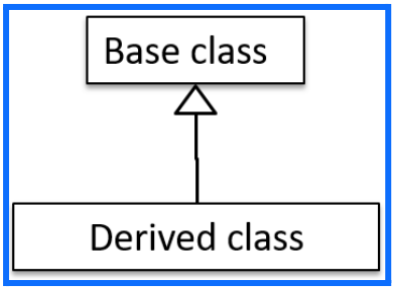


`Loan` is an association class

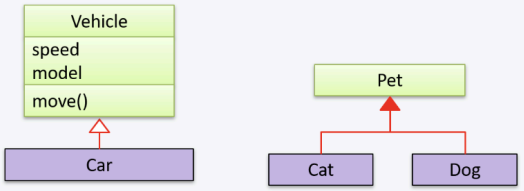
- Stores information about the `borrows` association between the `User` and the `Book`.

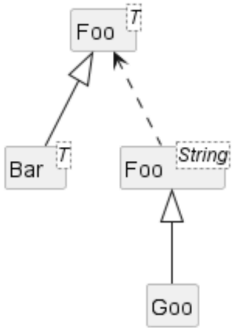
INHERITANCE

- Triangle (filled or empty) and a solid line (not to be confused with an arrow)



Examples: The `Car` class *inherits* from the `Vehicle` class. The `Cat` and `Dog` classes *inherit* from the `Pet` class.






```
class Foo<T> {
}

class Bar<T> extends Foo<T> {
}

class Goo extends Foo<String> {
}
```

Inheritance and Generics

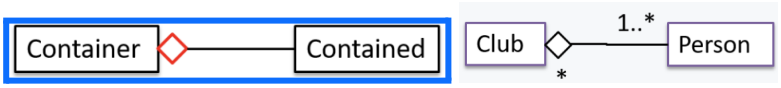
COMPOSITION

Solid diamond symbol

- A Book is composed of Chapter objects.
  - When a Book object is destroyed, its Chapter objects are destroyed too.

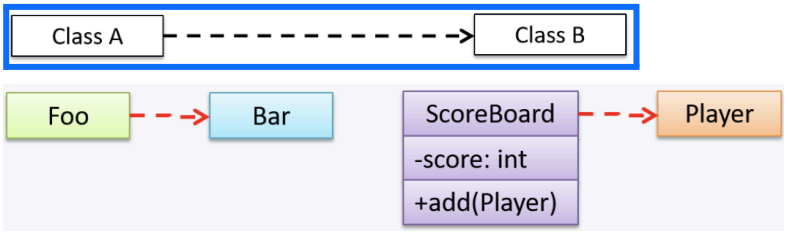
AGGREGATION

- Hollow diamond



DEPENDENCIES

- Dashed arrow

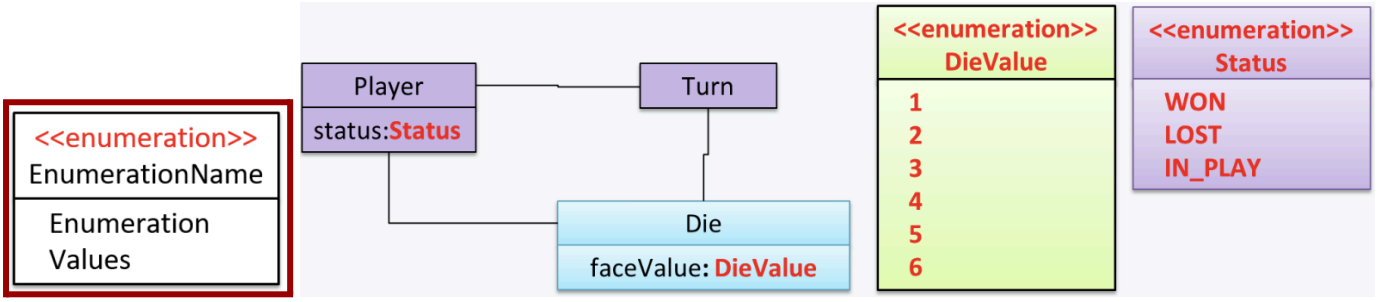


- **Dependencies** vs **associations**
  - Association = A relationship resulting from one object keeping a reference to another object
    - i.e., storing an object in an instance variable
    - Need not show dependency if association already indicated
  - **Use a dependency arrow to indicate a dependency only if that dependency is not already captured by the diagram in another way** (eg as an association or an inheritance)

- E.g., class Foo accessing a constant in Bar (Foo has a dependency on Bar) but there is no association / inheritance from Foo to Bar.

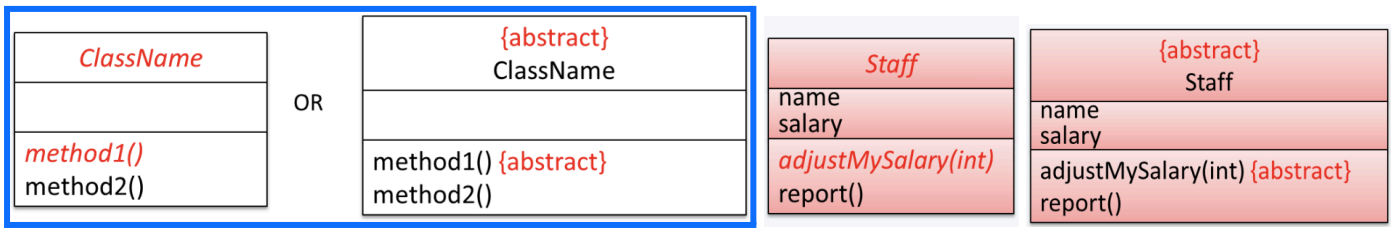
ENUMERATION

A class diagram can also show different types of class-like entities: Enumeration



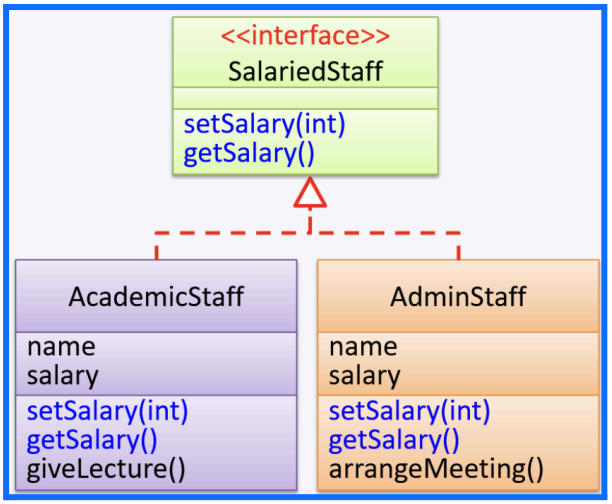
ABSTRACT CLASSES

Use *italics* or `{abstract}` (preferred) keyword to denote abstract classes / methods.
(NOTE: Italics do **not** mean anything in **sequence** diagrams)



INTERFACES (behavior specification) <<interface>>

- Similar to class inheritance except a **DASHED line** is used instead of a solid line.



- AcademicStaff and the AdminStaff classes implement the SalariedStaff interface.

Object Diagrams

OBJECTS

Notation 

- The class name and object name are **underlined** e.g. `car1:Car`.
- `objectName:ClassName` = An instance of ClassName identified as objectName
- Unlike classes, there is no compartment for methods.
- *Attributes* compartment can be omitted if it is not relevant to the purpose of the diagram.
- Object name can be omitted too e.g. `:Car` = An **unnamed instance** of a Car object

ASSOCIATION

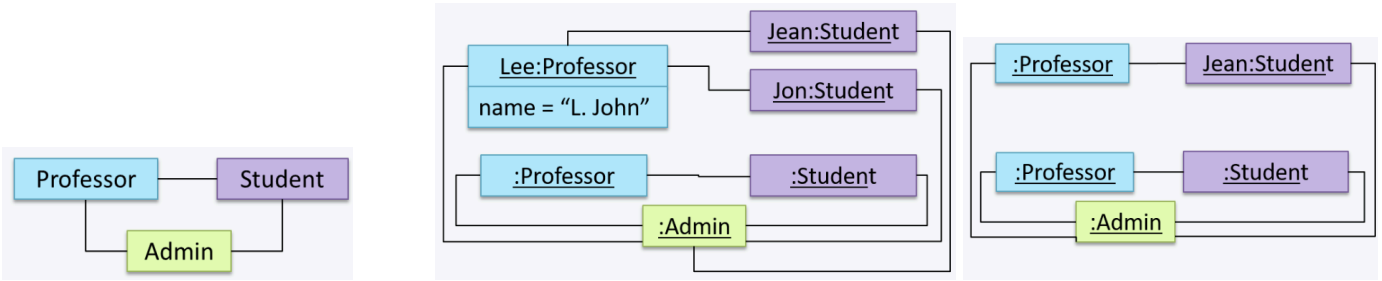
Solid line indicates an association between two objects 

Object vs Class Diagrams

Object diagrams differ from class diagrams in the following ways:

- Show objects instead of classes:
  - Instance name may be shown
  - There is a `:` before the class name
  - Instance and class names are underlined
- **Methods are omitted**
- **Multiplicities are omitted**. An **association line** in an **object** diagram represents a connection to exactly one object (i.e., the **multiplicity is always 1**).

Multiple object diagrams can correspond to a **single class** diagram.



When the class diagram has an **INHERITANCE** relationship, the **OBJECT diagram should show EITHER an object of the *parent class or the child class*, but not both**.

Suppose `Employee` is a child class of the `Person` class. The class diagram will be as follows:

Now, how do you show an `Employee` object named `jake`?

- ✗ 

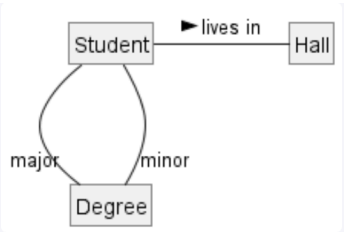
This is not correct, as there should be only one object.
- ✓ 

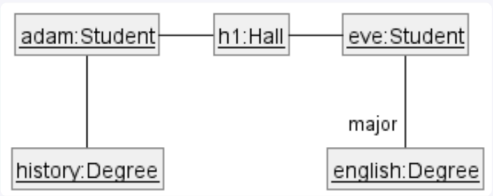
This is OK.
- ✓ 

This is OK, as `jake` *is* a `Person` too. That is, we can show the parent class instead of the child class if the child class doesn't matter to the purpose of the diagram (i.e., the reader of this diagram will not need to know that `jake` is in fact an `Employee`).

Association labels / roles can be omitted unless they add value.

- Useful if there are multiple associations between the two classes

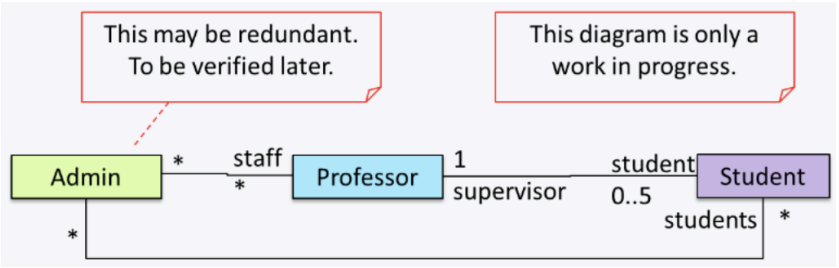




We can clearly see that both Adam and Eve lives in hall h1 (i.e., OK to omit the association label `lives in`) but we can't see if History is Adam's major or his minor (i.e., the diagram should have included either an association label or a role there). In contrast, we can see Eve is an English major.

UML Notes `[W4.3a] Tools → UML → Notes`

- UML notes can augment UML diagrams with additional information

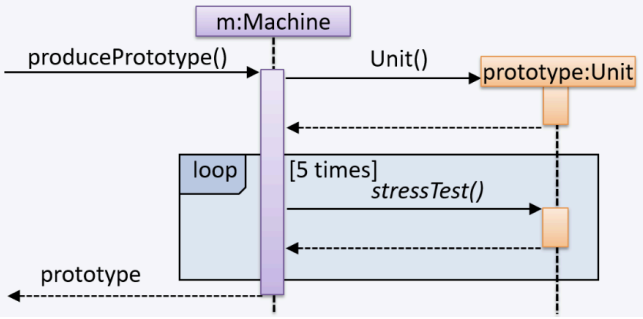


Sequence Diagrams - Model BEHAVIOUR

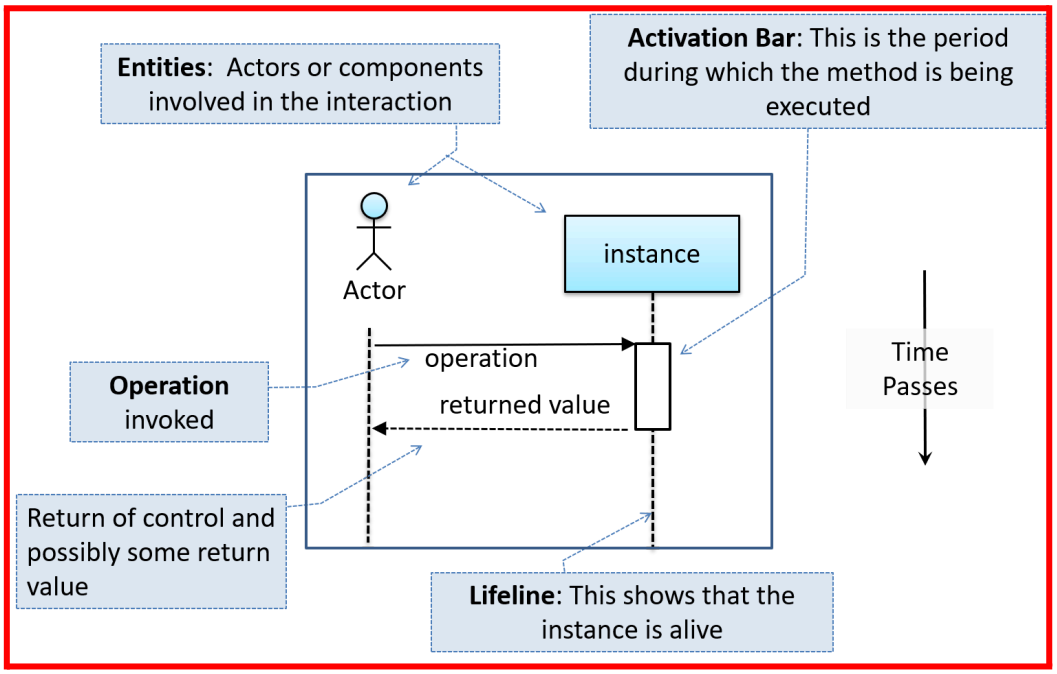
A UML sequence diagram captures the **interactions** between **multiple entities** for a given scenario.

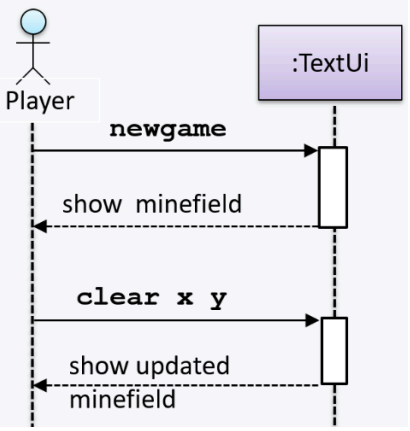
```
class Machine {

    Unit producePrototype() {
        Unit prototype = new Unit();
        for (int i = 0; i < 5; i++) {
            prototype.stressTest();
        }
        return prototype;
    }
}

class Unit {

    public void stressTest() {

    }
}
```

Here is the sequence diagram to model the interactions for the method call `produceProtoype()` on a `Machine` object.



Basic notation - Sequence Diagram

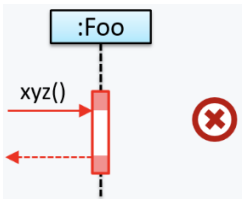
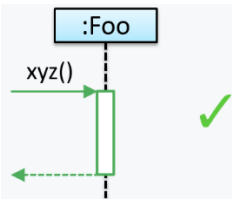
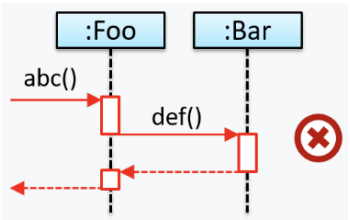
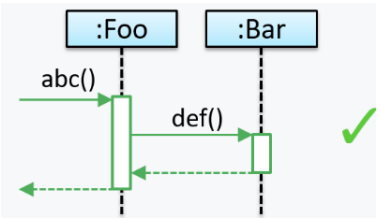


This sequence diagram shows some interactions between a human user and the Text UI of a CLI Minesweeper game. The player runs the `newgame` action on the `TextUi` object which results in the `TextUi` showing the minefield to the player. Then, the player runs the `clear x y` command; in response, the `TextUi` object shows the updated minefield. `:TextUi` denotes an ***UNNAMED INSTANCE* of the class** `TextUi`. If there were two instances of `TextUi`, they can be distinguished by naming them e.g. `TextUi1:TextUi` and `TextUi2:TextUi`.	

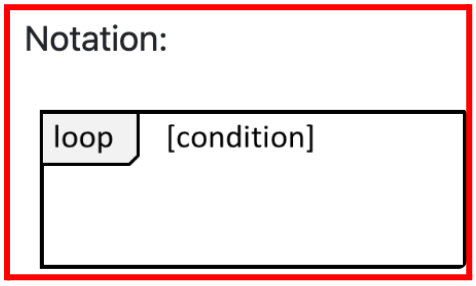
- Arrows representing method **calls** → **solid** arrows
- Arrows representing method **returns** → **dashed** arrows
- Unlike in object diagrams, the **class / object name is NOT underlined** in sequence diagrams
- Arrowhead style depends on the **type of method call**
  - **filled** arrowheads (→) (USE THIS)
    - Synchronous (i.e., the caller method is blocked from doing anything else until the called method returns)
  - lined arrowheads (→)
- 'Unroll' chained/compound method calls before drawing a sequence diagram.
  - Consider the Java statement `new Book().add(new Chapter());` equivalent to:

```
Book b = new Book();
Chapter c = new Chapter();
b.add(c);
```

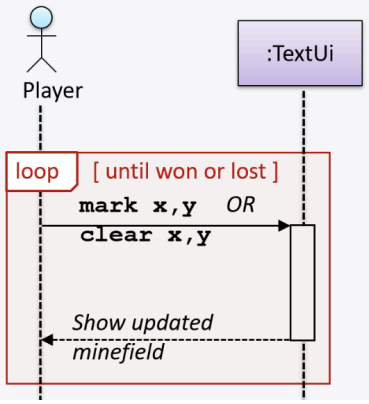
Errors

Activation bar too long The activation bar of a method **cannot start before** the method call arrives and a method **cannot remain active after** the method has returned.		
Broken activation bar The activation bar (eg for the method `Foo#abc()`) should remain **unbroken** while the method is being executed, from the point the method is called until the method returns		

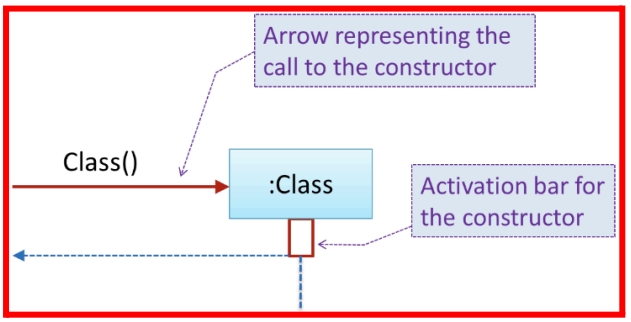
Loops - Sequence Diagram



The `Player` calls the `mark x,y` command or `clear x y` repeatedly until the game is won or lost.



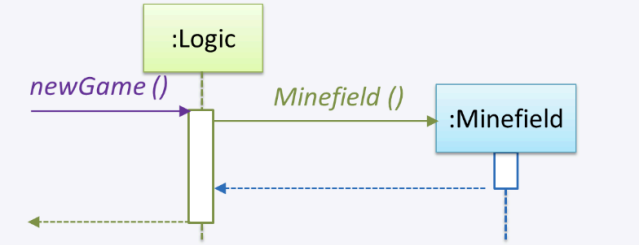
Object creation - Sequence Diagram



The **activation bar** represents the period the constructor is active

The **arrow that represents the constructor** arrives at the side of the box representing the instance.

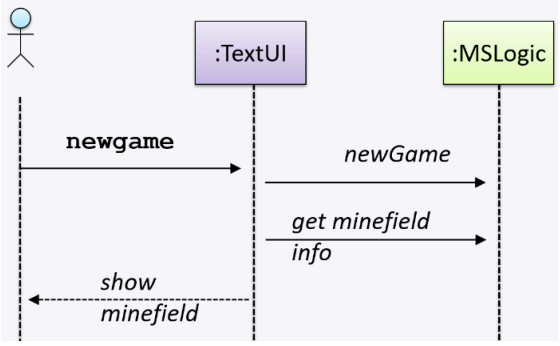
The `Logic` object creates a `Minefield` object.



Minimal notation - Sequence Diagram

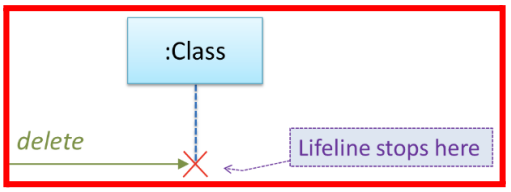
Optional elements (e.g, activation bars, return arrows) may be omitted if the omission does not result in ambiguities or loss of relevant information.

If method parameters don't matter to the purpose of the sequence diagram, omit them using `...` e.g., use `foo(...)` instead of foo(int size, double weight)



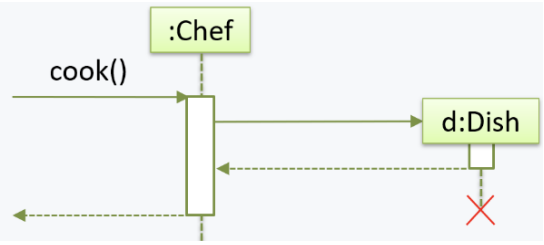
Object deletion - Sequence diagrams

- UML uses an `X` at the end of the lifeline of an object to show its deletion.
- Indicate the point at which the object becomes ready to be garbage-collected (i.e., the point at which it ceases to be referenced.

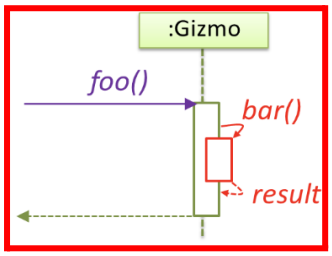


Note how `d` lifeline ends with an `X` to show that it is 'deleted' (ready to be garbage collected) after the `cook()` method returns.

```
class Chef {
    void cook() {
        Dish d = new Dish();
    }
}
```

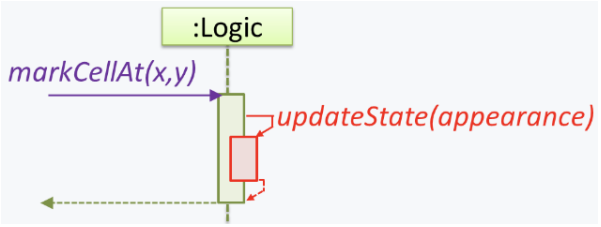


Self-Invocation - Sequence diagrams



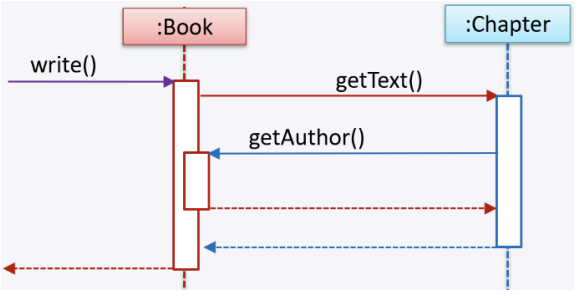
UML can show a method of an object **calling another of its own methods**.

`markCellAt(...)` method of a `Logic` object is calling its own `updateState(...)` method



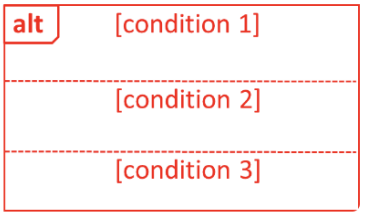
`Book#write()` method is calling the `Chapter#getText()` method which in turn does a **call back** by calling the `getAuthor()` method of the calling object.

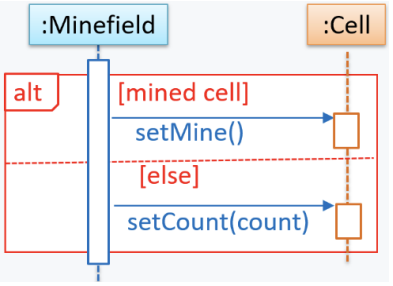
Call back - Object A calls the method of Object B, and Object B calls the method of Object A back



Alternative paths - Sequence Diagram

`alt` frames to indicate alternative paths

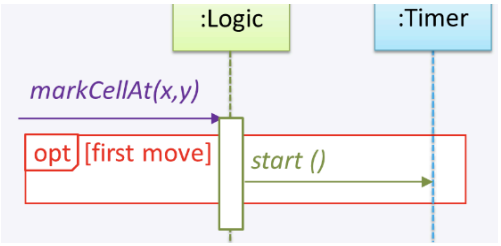


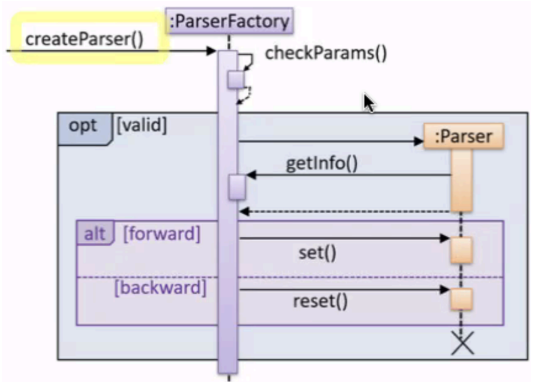
`Minefield` calls the `Cell#setMine` method if the cell is supposed to be a mined cell, and calls the `Cell:setMineCount(...)` method otherwise **No more than one** alternative partitions be executed in an `alt` frame	

Optional paths - Sequence Diagram

`opt` frames to indicate alternative paths

`Logic#markCellAt(...)` calls `Timer#start()` only if it is the first move of the player.

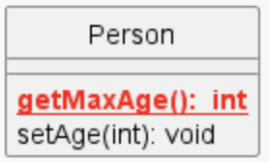


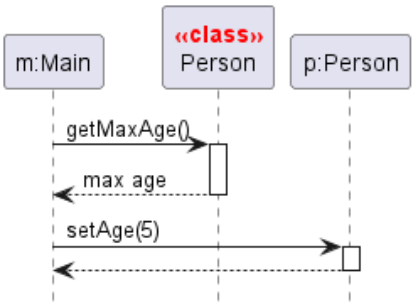
	Assuming forward == ! backward, an execution of the createParser () method can result in • either the set() or the reset() method but not both • **None** of the methods being called ○ Due to the opt box `ParserFactory` class has the `getInfo()` method

Calls to static methods - Sequence Diagram

- Use `<<class>>` to show that a participant is the class itself

`m` calls the static method `Person.getMaxAge()` and also the `setAge()` method of a `Person` object `p`



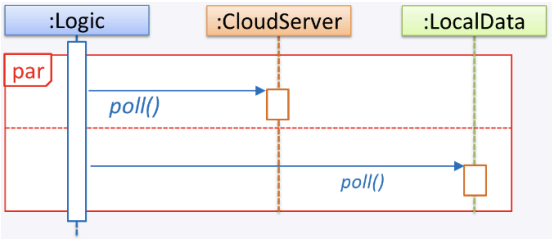


Parallel paths - Sequence diagrams

`par` frames to indicate alternative paths

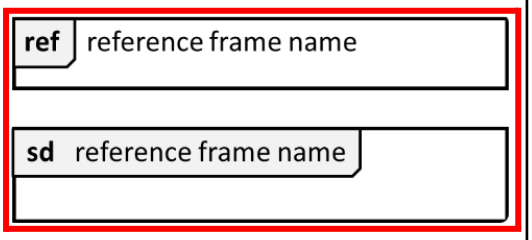
If you show parallel paths in a sequence diagram, the corresponding Java implementation is likely to be multi-threaded because a normal Java program cannot do multiple things at the same time.

`Logic` is calling methods `CloudServer#poll()` and `LocalData#poll()` in parallel.

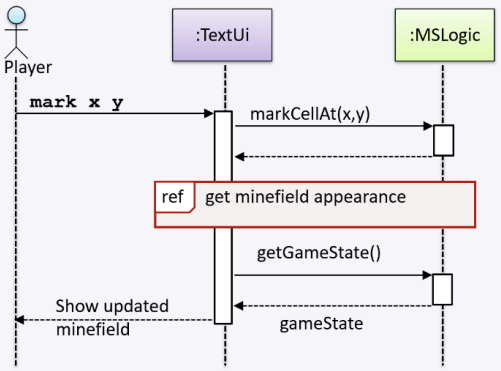


Reference paths - Sequence diagrams

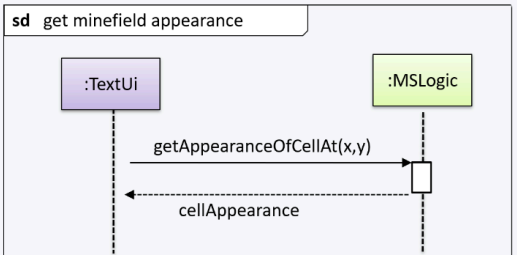
`ref` frame to allow a segment of the interaction to be omitted and shown as a separate sequence diagram



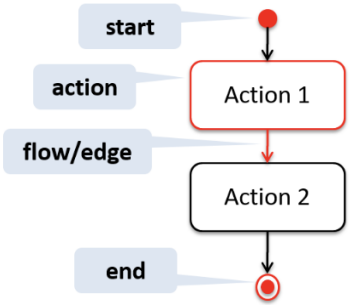
The details of the `get minefield appearance` interactions have been omitted from the diagram.



Those details are shown in a separate sequence diagram given below.



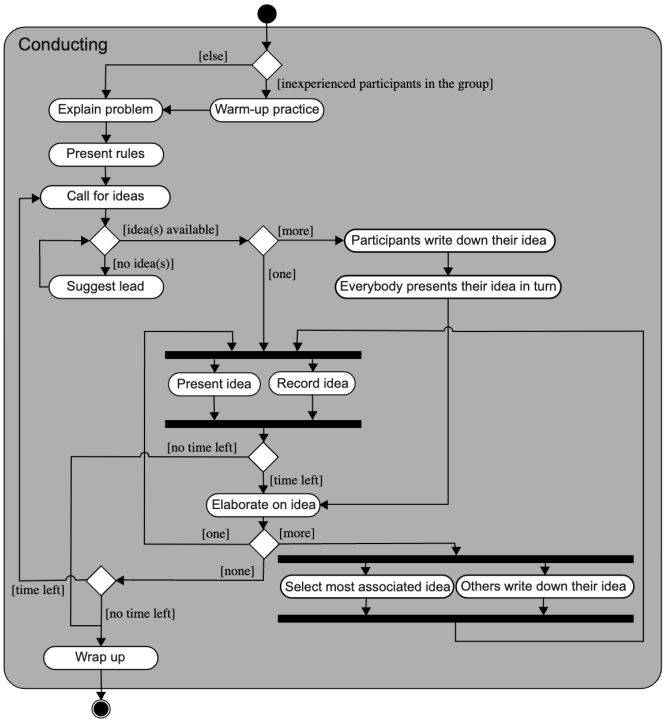
Activity Diagrams (model BEHAVIOURS - workflows)



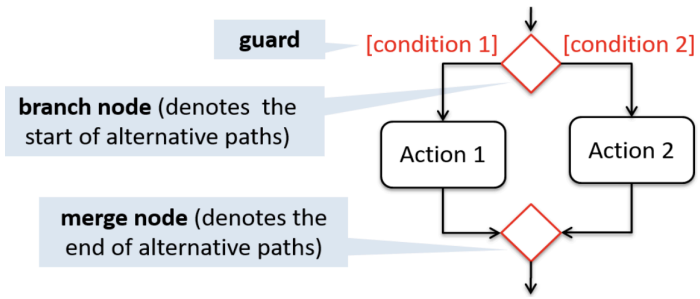
Activity diagram (AD) captures an **activity** through

- **Actions**
  - Single step in an activity
  - Rectangle with **ROUNDED corners**
- **Control flows**
  - Flow of control from one action to the next
  - An arrow-head to show the direction of the flow

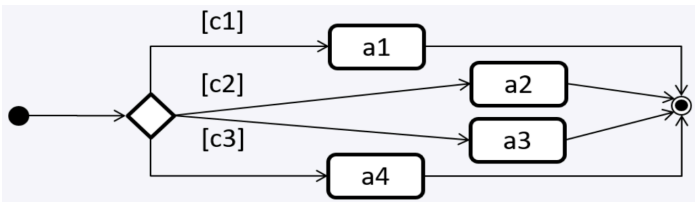
Note START NODE and END NODE



Alternate paths (Branch node, Merge node) - Activity Diagram



- Branch / Merge node
  - Start / End of alternate paths
  - **DIAMOND shapes**
  - *Exactly ONE of the guard conditions should be true at any given branch node*
- Guard condition - boolean condition should be true for execution in that path
  - **SQUARE BRACKETS !!!**



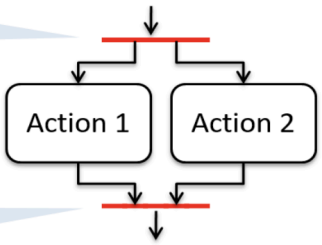
Acceptable

- Omit the merge node if it doesn't cause any ambiguities.
- Multiple arrows can start from the same corner of a branch node.
- Omit the [Else] condition.

Parallel paths (Fork node, Join node) - Activity Diagram

Fork (denotes the start of parallel paths - many outgoing edges)

Join (denotes the end of parallel paths – many incoming edges)

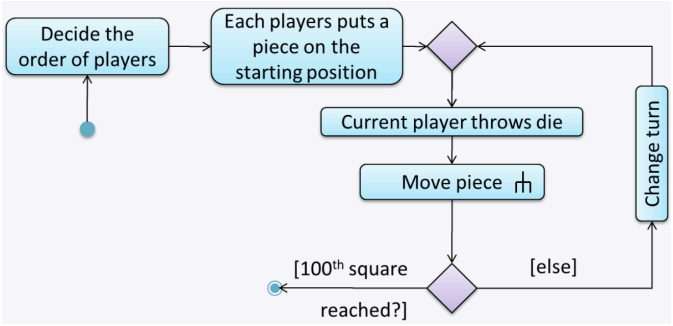


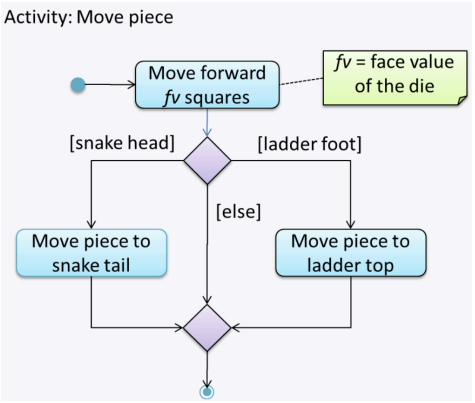
All parallel paths should **complete before** execution can start on the **outgoing control flow of the join** i.e. end in the **same join node**

Actions in parallel paths can interleave if all paths complete

Rake notation - Activity Diagram

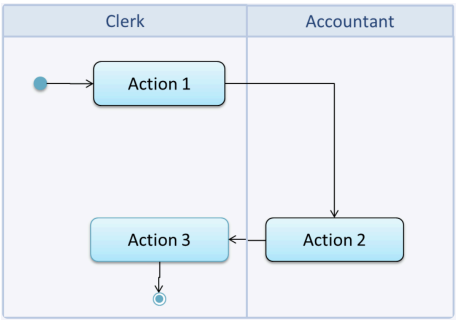
Rake notation is used to indicate that a part of the activity is given as a separate diagram





Swim Lanes - Activity Diagram

Partition an activity diagram to show who is doing which action



Git and Github

init: Getting started

1. Install Sourcetree (Git + GUI for Git); or Git (use Git via command line without GUI)
2. Create a directory (e.g. named `things`) for the repo
3. Initialize a repository in the directory

Sourcetree	Command Line Interface (CLI)

Windows:
File → Clone/New…. → Create
 	
Mac:
New... → Create New Repository
 	
hidden folder .git created in `things` folder
	Open a Git Bash Terminal (or `Terminal` button in Sourcetree).
	Navigate to `things` directory.
	
	`$ cd /c/repos/things`
	`$ git init  # initialize the repo`
	`Initialized empty Git repository in c:/repos/things/.git/`
	
	`$ ls -a`
	`.  ..  .git`
	
	`$ git status`
	`# On branch master`
	`#`
	`# No commits yet`
	`#`
	`nothing to commit (create/copy files and use "git add" to track)`

---

commit: Saving changes to history

`[W2.3e] Tools → Git and GitHub → commit: Saving changes to history`

After initializing a repository, Git can help with revision controlling files inside the working directory. However, it is not automatic.

1. Do some changes to the content inside the working directory
2. File is detected by Git → File shown as 'unstaged'
3. **Stage** the changes to commit
    a. Change is moved to the staging area (index)

Sourcetree	Command Line Interface (CLI)

Select the file → "Stage Selected" button
 	
Selected file appears in the "Staged files" panel
	`$ git add fruits.txt`
	`$ git status`
	`# On branch master`
	`#`
	`# No commits yet`

`\ No newline at the end of the file` message → Move the cursor to end of the last line in that file and hit enter → tell Git that last line is complete	`#`
	`# Changes to be committed:`
	`#   (use "git rm --cached <file>..." to unstage)`
	`#`
	`#       new file:   fruits.txt`

4. **Commit** the stage version of the file.

Sourcetree	Command Line Interface (CLI)

Commit button, enter a commit message	`$ git commit -m "Add fruits.txt"`
	
	`$ git log  # see commit history`
	`commit 8fd30a6910efb28bb258cd01be93e481caeab846`
	`Author: ... < ... @... >`
	`Date:   Wed Jul 5 16:06:28 2017 +0800`
	
	`  Add fruits.txt`

5. You can decide what to stage and what to leave unstaged.

6. See the revision graph

Sourcetree	Command Line Interface (CLI)
WORKSPACE → History	`$ gitk`

Undo / Delete a commit

Undo the last commit, but keep the changes in the staging area	`$ git reset --soft HEAD~1`
Undo the last commit, and remove the changes from the staging area	`$ git reset --mixed HEAD~1`
Delete the last commit entirely	`$ git reset --hard HEAD~1`
To undo/delete last n commits	`HEAD~n`

Omitting files from revision control

`[W2.3f] Tools → Git and GitHub → Omitting files from revision control`

`.gitignore` file

Files to be omitted
- Binary files generated when building project
  - e.g., `*.class`, `*.jar`, `*.exe`
- Temporary files
  - Eg log files generating when testing

- Local files
- Sensitive content

`tag`: Naming commits

`[W2.3h] Tools → Git and GitHub → tag: Naming commits`

Tag: Entity that points to the commit

Add a tag to the current commit as v1.0	`$ git tag v1.0`
View tags	`$ git tag`

`diff`: Comparing revisions

`[W2.3i] Tools → Git and GitHub → diff: Comparing revisions`

	Sourcetree	Command Line Interface (CLI)

See which files changes and what changed in the file	Click on the
	- Commit
	- File name
		`$ git show < part-of-commit-hash >`
		
		`$ git show 5bc0e306`
		
		`commit 5bc0e30635a754908dbdd3d2d833756cc4b52ef3`
		`Author: ... < ... >`
		`Date: Sat Jul 8 16:50:27 2017 +0800`
		
		`fruits.txt: replace banana with berries`
		
		`diff --git a/fruits.txt b/fruits.txt`
		`index 15b57f7..17f4528 100644`
		`--- a/fruits.txt`
		`+++ b/fruits.txt`
		`@@ -1,3 +1,3 @@`
		`apples`
		`-bananas`
		`+berries`
		`cherries`

Difference between 2 points in history	Ctrl + Click to select the 2 points to be compared	`$ git diff # shows the changes (uncommitted) since the last commit`
		
		`$ git diff 0023cdd..fcd6199 # shows changes between the points indicated by commit hashes (first 7-10 chars)`

		`$ git diff v1.0..HEAD  # shows changes that happened from the commit tagged as v1.0 to most recent commit`

`checkout`: Retrieving a specific revision

`[W2.3j] Tools → Git and GitHub → checkout: Retrieving a specific revision`

Git can load a specific version of the history to the working directory.

Sourcetree	Command Line Interface (CLI)

Double-click the commit or Right-click then `Checkout…`
 	
`HEAD` is a reference to the currently checked out commit
 	
If you `checkout` a commit that comes before the commit in which you added the `.gitignore` file, Git will now show ignored files as 'unstaged modifications'
	`$ git checkout <commit-identifier>  # change the working directory to the state at a specific past commit the repo`
	
	`$ git checkout v1.0  # loads the state as at commit tagged v1.0`
	
	`$ git checkout 0023cdd`
	
	`$ git checkout HEAD~2  # loads the state that is 2 commits behind the most recent commit`

`clone`: Copying a repo

`[W2.4b] Tools → Git and GitHub → clone: Copying a repo`

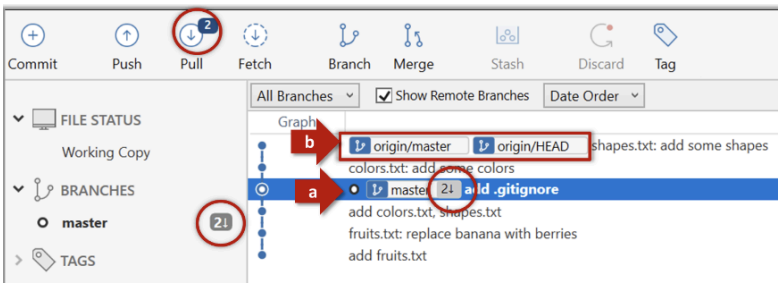
GitHub project URL: https://github.com/… Git repo URL: https://github.com/….**git**

`pull, fetch`: Downloading data from other repos

`[W2.4c] Tools → Git and GitHub → pull, fetch: Downloading data from other repos`

1. Clone a repo.
2. Delete the last few commits to simulate cloning the repo a few commits ago.

Sourcetree	Command Line Interface (CLI)
Right click current commit → Reset current branch to this commit → Hard - discard all working copy changes	`$ git reset --hard HEAD~2`

 `a`: The local repo is now at this commit, marked by `master` `b`: `origin/master` label shows the latest commit in the `master` branch in the remote repo	

3. Pull from remote repo

Sourcetree	Command Line Interface (CLI)
`Pull` Pull from remote: `origin` Remote branch to pull: `master` `master` and `origin master` both point to the same commit	`$ git pull origin`

Working with multiple remotes

- When you clone a repo, Git automatically adds a remote repo named `origin`
- To communicate with another remote repo, first add it as a **remote** of your repo

Sourcetree	Command Line Interface (CLI)

1. Open local clone of the repo
2. `Repository` → `Repository Settings`
3. Add a new remote
- Remote name: name to assign to the remote repo e.g., `upstream1`
- URL/path: URL of repo (ending in `.git`)
4. Choose the remote name (instead of `origin`) of the repo to `pull` from
5. There will be one more commit into the local repo
	1. Navigate to the folder containing the local repo.
	2. Set the new remote repo as a remote of the local repo.
	`git remote add {remote_name} {remote_repo_url}`
	3. **Pulling** will **fetch** the branch and **merge** the new code to the current branch) from the new remote
	`git fetch upstream1 master` followed by `git merge upstream1/master`, OR, `git pull upstream1 master`

---

fork: Creating a remote copy

`[W2.4d] Tools → Git and GitHub → Fork: Creating a remote copy`

Forking is not a Git feature, but provided by remote Git hosting services such as Github

`push`: Uploading data to other repos

`[W2.4e] Tools → Git and GitHub → push: Uploading data to other repos`

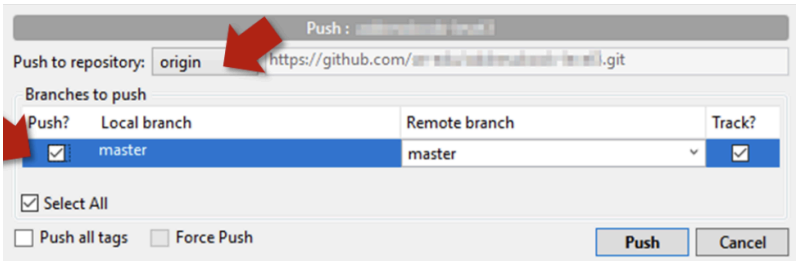
	Command Line Interface (CLI)
Push new commits clone of fork on local to fork on Github	`$ git push origin master`

Push to repos (with shared history) other than the cloned one
• Select the name of target remote instead of `origin`. Select the `Track` checkbox.
	`$ git push (name of remote) master`
	
	`$ git push upstream1 master`

Push a specific tag	`$ git push origin v1.0b`
Push all tags	`$ git push origin --tags`

Push an entire local repository to GitHub

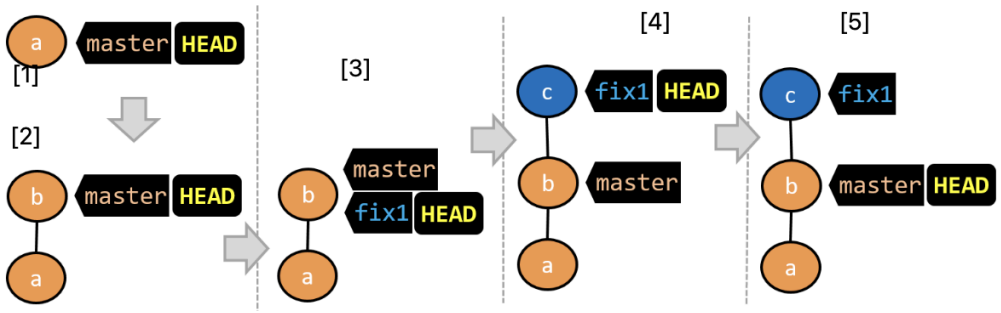
1. Create an **empty** remote repo on GitHub.
2. Keep the "Initialize this repository with a README" tick box unchecked
3. Add the GitHub repo URL as a remote of the local repo
4. Push the repo to the remote

Sourcetree	Command Line Interface (CLI)
	`$ git push -u origin master` `-u` to track (remember which branch in the remote repo corresponds to which branch in the local repo) the branch

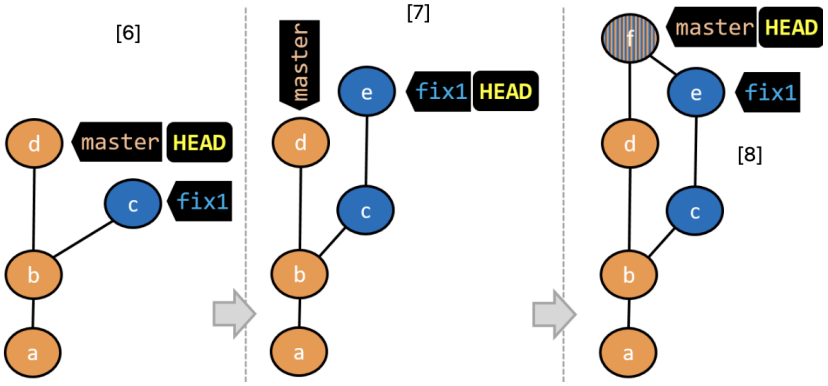
`branch`: Doing multiple parallel changes

`[W3.1b] Tools → Git and GitHub → branch: Doing multiple parallel changes`

- Git branch: A *named label* pointing to a commit
- `HEAD` label indicates which branch you are on
- Always remember to switch back to the master branch before creating a new branch.
    - If not, your new branch will be created on top of the current branch.



1. Only one branch (`master`) and one commit on it.
2. New commit (`b`) added (with `master` and `HEAD` labels).
3. New branch (`fix1`) and repo switched to this branch (with `HEAD` label).
4. New commit (`c`) with branch (`fix1`) and `HEAD` labels.
5. Repo switched back to `master` branch (with `HEAD` label). Working directory reflects code at commit `b` (not `c`).



6. New commit (`d`) added (with `master` and `HEAD` labels).
7. Repo switched back to `fix1` branch, with a new commit `e` to it.
8. Repo switched back to `master` branch. `fix1` branch merged into `master` branch, creating a merge commit `f`.

	Command Line Interface (CLI)
Create a new branch	`$ git branch feature1`
Switch to a specific branch	`$ git checkout (or switch) feature1`
Create a branch and switch to it at the same time	`$ git checkout -b feature1` `$ git switch -c feature1`
Merge the `master` branch to `feature1` branch	`$ git merge master`
Merge the `feature1` branch to `master` branch	`$ git merge feature1`
Force Git to create a merge commit even if fast forwarding is possible (→ no fast forward)	`$ git merge --no-ff add-countries`

Dealing with merge conflicts

[W3.1c] Tools → Git and GitHub → Dealing with merge conflicts

- Conflicted part is marked
  - between a line starting with `<<<<<<` and a line starting with `>>>>>>`
  - separated by another line starting with `=======`

Conflicting part	Resolve the conflict by editing the file

`blue`
`<<<<<< HEAD`
`black (from master branch)`
`=======`
`green (from fix1 branch)`
`>>>>>> fix1`
`red`
	`blue`
	`black`
	`green`
	`red`
	`white`

Working with remote branches

[W3.1d] Tools → Git and GitHub → Working with remote branches

- Git branches in a local repo can be linked to a branch in a remote repo

Pushing a new branch to a remote repo

`-u` (or `--set-upstream`) flag tells Git that you wish the local branch to 'track' the remote branch that will be created as a result of this push	`$ git push -u origin add-intro` `(remote) (branch)`

Pulling a remote branch for the first time

1. Fetch details from the remote. e.g., if the remote is named `myfork`	`$ git fetch myfork`
2. List the branches to see the name of the branch you want to pull	`$ git branch -a` `(-a : list both local and remote branches)`

3. Create a matching local branch and switch to it
 	
`-c` flag tells Git to create a new local branch
	`$ git switch -c branch1 myfork/branch1`
	
	`Switched to a new branch 'branch1'`
	`branch 'branch1' set up to track 'myfork/branch1'.`

Syncing branches

Push new changes in local branch to the corresponding remote branch (no `-u` flag)	`$ git push origin add-intro` `(remote) (branch)`

Pull new changes from a remote branch to the corresponding local branch	
1. Switch to the branch to be updated	`$ git checkout branch`
2. Pull the updated in the remote branch to the local branch	`$ git pull <remote> <branch>` `$ git pull origin branch1`

Creating PRs `[W3.2a] Tools → Git and GitHub → Creating PRs`

- Set the appropriate target repo and the branch that should receive your PR
  - base repository
  - base
- Indicate which repo:branch contains your proposed code
  - head repository
  - compare

Problem: merge conflicts in ongoing PRs

- Upstream repo's `master` branch has been updated in a way that the PR code conflicts with that `master` branch

1. Pull `master` branch from upstream repo to local repo	`$ git checkout master` `$ git pull upstream master`

2. In the local repo, attempt to merge the `master` branch (that you updated in the previous step) onto the PR branch	`$ git checkout pr-branch`
	`  # pr-branch is the name of branch in the PR`
	
	`$ git merge master`

3. Resolve the merge conflicts manually. 4. Push PR branch to the fork.	

Supplementary

C++ to Java

Classes

DEFINING CLASSES `[W1.1f] C++ to Java → Classes → Defining classes`

- Defining a class introduces a new object type.
- Every object belongs to some object type i.e. it is an instance of some class.
- `PascalCase` format for class names

Constructors

- Special methods that construct the object and initialize the instance variables
- Keyword `static` is omitted
- Takes in no arguments
- Returns a reference to the new object
- Can be overloaded
    - multiple constructors with different parameters

`this` keyword

- A reference variable in Java that refers to the current object (i.e., the enclosing object, or myself)
- Can read and write the instance variables of `this`
- Can pass `this` as an argument to other methods
- Do not declare `this`
- Cannot make an assignment to `this`

Example: The parameters shadow (hide) the instance variables, so `this` is necessary to tell them apart

```
public Time(int hour, int minute, int second) {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
}
```

`this` can be used to refer to a **constructor** of a class within the same class too.

```
public Time() {
    this(0, 0, 0); // call the overloaded constructor Time(int, int, int)
}

public Time(int hour, int minute, int second) {
```

```
    // ...
}
```

Instance methods

- Methods to a class which can be used from the objects of that class
- ***Instance*** methods do not have the `static` keyword in their method signature

GETTERS AND SETTERS [W1.1g] C++ to Java → Classes → Getters and setters

`getAttribute`: Access attributes	`public double getRadius() {`
	`  return radius;`
	`}`

`setAttribute`: Modify attributes	`public void setRadius(double radius) {`
	`  this.radius = Math.max(radius, 0);`
	`}`

CLASS-LEVEL MEMBERS [W1.1i] C++ to Java → Classes → Class-level members

`static` fields / class variables

- Fields with the `static` modifier in their declaration
- Associated with the class, rather than with any object
- Every instance of the class shares a class variable, which is in one fixed location in memory
- Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class
- Referenced by the class name itself
    - `Bicycle.numberOfBicycles`

`static` methods

- Invoked with the class name, without the need for creating an instance of the class
    - `ClassName.methodName(args)`

Example:

```
public class Bicycle {
    private int gear;
    private int speed;

    // an instance variable for the object ID
    private int id;

    // a class variable for the number of Bicycle objects instantiated
    private static int numberOfBicycles = 0;

    public static int getNumberOfBicycles() {
       return numberOfBicycles;
   }
```

```
}
```

`static final` modifier to define CONSTANTS

- `final` : value of this field cannot change
- E.g. `static final` double PI = 3.141592653589793;

`System.out.println(...)`

- `out` is a **class-level** public attribute of the `System` class.
- `println` is an **instance** level method of the `out` object.

Inheritance

INHERITANCE [`extends`] (Basics)

A **subclass inherits all** the members (**fields, methods, nested classes**) from its superclass

- Constructors
  - Not members → not inherited
  - Can be invoked from subclass

Every class has one and only one direct superclass (single inheritance)

- Exception: `Object` has no superclass
- Every class is implicitly a subclass of `Object`

Java does not support multiple inheritance among classes

`java.lang.Object` class

- Defines and implements behavior common to all classes

`super`

- Invoke the overridden superclass's method `super.method()`
- Refer to a hidden field (superclass variable has the same name as subclass variable)

`Subclass constructor`

- Can invoke the superclass constructor
- `super()` invokes the no-argument constructor of the superclass
- `super(parameters)` invoke the superclass constructor with a matching parameter list

NOTE

- If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass
  - Superclass does not have a no-argument constructor → compile-time error
- Object does have such a constructor, so if Object is the only superclass → no problem

```
public class MountainBike extends Bicycle {
```

```
    // Bicycle has fields int gear, int speed
    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startSpeed, int startGear) {
        super(startSpeed, startGear);      // calls the superclass constructor
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

Access Modifiers (simplified)

- Access level modifiers determine whether other classes can use a particular field or invoke a particular method.

Assuming you have **not yet started placing your classes in different packages** i.e., all classes are placed in the root level. Two levels of access control:

- **Class level**
  - `public` the class is visible to all other classes
  - no modifier same as `public`
- **Member level**
  - `public` the member is visible to all other classes
  - `protected` same as `public`
  - no modifier same as `public`
  - `private` the member is not visible to other classes (but can be accessed in own class)

POLYMORPHISM [W1.3b] C++ to Java → Inheritance → Polymorphism

Java is a `strongly-typed` language

- Code works with only the object types that it targets, eg only works with Cat, not Dog objects:

```
public class PetShelter {
    private static Cat[] cats = new Cat[]{
            new Cat("Mittens"),
            new Cat("Snowball")};

    public static void main(String[] args) {
        for (Cat c: cats){
            System.out.println(c.speak());
        }
    }
}
```

```
public class Cat {
    private String name;

    public Cat(String name) {
        this.name = name;
    }

    public String speak() {
        return name + ": Meow";
    }
}
```

- **Can lead to unnecessary verbosity**
    - Repetitive similar code that do similar things with different object types
    - If the PetShelter is to keep both cats and dogs, you'll need two arrays and two loops:

```
public class PetShelter {
    private static Cat[] cats = new Cat[]{
            new Cat("Mittens"),
            new Cat("Snowball")};
    private static Dog[] dogs = new Dog[]{
            new Dog("Spot")};

    public static void main(String[] args) {
        for (Cat c: cats){
            System.out.println(c.speak());
        }
        for(Dog d: dogs){
            System.out.println(d.speak());
        }
    }
}
```

SOLUTION: `POLYMORPHISM` - target a superclass so that it works with any subclass objects

- The `PetShelter2` uses one data structure to keep both types of animals and one loop to make them speak. The code targets the `Animal` superclass (assuming `Cat` and `Dog` inherits from the Animal class) instead of repeating the code for each animal type.

```
public class PetShelter2 {
    private static Animal[] animals = new Animal[]{
            new Cat("Mittens"),
            new Cat("Snowball"),
            new Dog("Spot")};

    public static void main(String[] args) {
        for (Animal a: animals){
            System.out.println(a.speak());
        }
```

```
    }
}
```

Polymorphic code is better

- Shorter
- Simpler
- More flexible (above example: main method will work even if we add more animal types)

ABSTRACT CLASSES AND METHODS

[W1.3d] C++ to Java → Inheritance → Abstract classes and methods

Abstract method (hence abstract class)

- declared with `abstract`
- no implementation (no method body)

Abstract class

- declared with `abstract`
- can be used as reference type `Account a;` OK
- cannot be instantiated `a = new Account();` **COMPILE ERROR**
- A class that does not have any abstract methods can be declared as an abstract class
- When an abstract class is subclassed
    - **Subclass should provide implementations for all abstract methods** in its superclass
    - or else the subclass must also be declared abstract

The `Feline` class below inherits from the abstract class `Animal` but it does not provide an implementation for the abstract method speak. As a result, the `Feline` class needs to be `abstract` too.	The `DomesticCat` class inherits the abstract `Feline` class and **provides the implementation** for the abstract method speak. As a result, it **need not be (but can be) declared as** `abstract`**.**
`Animal a = new Feline("Mittens");` **COMPILE ERROR. Feline is abstract.**	`Animal a = new DomesticCat("Mittens");` DomesticCat **can be instantiated** and **assigned** to a variable of Animal type (allowed by **polymorphism**)

`public abstract class Feline extends Animal {`
`    public Feline(String name) {`
`        super(name);`
`    }`
 	
`}`
	`public class DomesticCat extends Feline {`
	`    public DomesticCat(String name) {`
	`        super(name);`
	`    }`
	
	`    @Override`
	`    public String speak() {`
	`        return "Meow";`
	`    }`
	`}`

INTERFACES [W1.3f] C++ to Java → Inheritance → Interfaces

An `interface` is a **reference type** (similar to a `class`)

- Mainly containing method signatures which
  - have no braces (`{ }`)
  - are terminated with a semicolon
- Can contain **constants** and `static` **methods**
- **Cannot** be instantiated
- Can only be `implemented` by classes
  - When an instantiable **class** `implements` an **interface**, it provides a method body for each of the methods declared in the **interface**

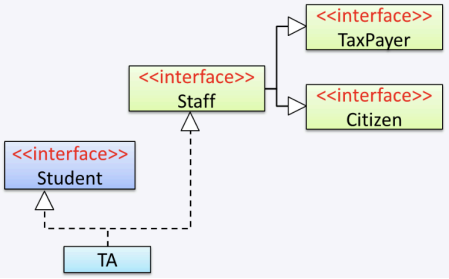
```
public interface DrivableVehicle {
    int MAX_SPEED = 150;

    static boolean isSpeedAllowed(int speed){
       return speed <= MAX_SPEED;
    }

    void turn(Direction direction);
    void changeLanes(Direction direction);
    void signalTurn(Direction direction, boolean signalOn);
}

public class CarModelX implements DrivableVehicle {
    @Override
    public void turn(Direction direction) {
       // implementation
    }
    // implementation of other methods
}
```

An interface can be used as a type e.g., `DrivableVehicle dv = new CarModelX();`

Interfaces can inherit from other interfaces

- `extends`
- **Multiple inheritance** among interfaces
  - Java interface can inherit multiple other interfaces

A Java **class** can `implement` **multiple interfaces** (and **inherit** from **one class**)

UML notation used: solid lines indicate normal inheritance; dashed lines indicate interface inheritance; the triangle points to the parent.  	1. `Staff` interface inherits (note the solid lines) the interfaces `TaxPayer` and `Citizen`. 2. `TA` class implements both `Student` interface and the `Staff` interface. 3. Because of point 1 above, `TA` class has to implement all methods in the interfaces `TaxPayer` and `Citizen`. 4. Because of points 1,2,3, a `TA` *is a* `Staff`, *is a* `TaxPayer` and *is a* `Citizen`.

Exceptions

WHAT ARE EXCEPTIONS [W1.5c] C++ to Java → Exceptions → What are Exceptions?

Checked Exceptions

- Exceptional conditions that a well-written application should **anticipate** and recover from
- All exceptions, **except** for `Error`, `RuntimeException`, and their subclasses
- Example

User provides the name of an existing, readable file → construction of the `FileReader` object succeeds	User supplies the name of nonexistent file → Constructor throws `java.io.FileNotFoundException` → Exception caught, prompt user for a corrected file name

[Unchecked Exception] Errors

- Exceptional conditions that are **external** to the application, and that the application **usually cannot anticipate or recover** from
- Exceptions indicated by `Error` and its subclasses
- E.g. Unsuccessful read will throw `java.io.IOError`
  - Application catches this exception to notify the user
  - Or program to print a stack trace and exit

[Unchecked Exception] Runtime Exceptions

- Conditions that are **internal** to the application, and that the application **usually cannot anticipate or recover** from
- Exceptions indicated by `RunTimeException` and its subclasses
- Usually indicate programming bugs, eg logic errors or improper use of an API
- E.g. If a logic error causes a null to be passed to the constructor, the constructor will throw `NullPointerException`.
  - The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

HOW TO USE EXCEPTIONS [W1.5e] C++ to Java → Exceptions → How to use Exceptions

`try` block

- identifies a block of code in which an exception can occur

`catch` block

- identifies an exception handler that can handle a particular type of exception

`finally` block

- code that is guaranteed to execute with or without the exception

Example:

```
public void writeList() {
    print("starting method");
    try {
        print("starting process");
        process();
        print("finishing process");

    } catch (IndexOutOfBoundsException e) {
        print("caught IOOBE");

    } catch (IOException e) {
        print("caught IOE");

    } finally {
        // clean up
        print("cleaning up");
    }
    print("finishing method");
}
```

Some possible outputs:

No exceptions ↓	`IOException` ↓	`IndexOutOfBoundsException` ↓
starting method	starting method	starting method
starting process	starting process	starting process
finishing process	~~finishing process~~	~~finishing process~~
cleaning up	caught IOE	caught IOOBE
finishing method	cleaning up	cleaning up
	finishing method	finishing method

`throw` an exception

- `throw` statement
  - Requires a throwable object as the argument
    - instances of any subclass of the `Throwable` class

```
private static int calculateArea(String descriptor) throws IllegalShapeException {
        String[] dimensions = descriptor.split("x");

        if (dimensions.length > 2) {
            throw new IllegalShapeException("Too many dimensions: " + descriptor);
        }
        if (dimensions.length < 2) {
            throw new IllegalShapeException("WIDTH or HEIGHT is missing: " +
descriptor);
        }

        return Integer.parseInt(dimensions[0]) * Integer.parseInt(dimensions[1]);
    }
```

In Java, **CHECKED exceptions are subject to the `Catch` or Specify Requirement**

- Code that might throw checked exceptions must be enclosed by either
  - `try` statement that catches and provides a handler for the exception
  - method must provide a `throws` clause that lists the exception
- Example example of a method specifying that it throws certain checked exceptions

```
public void writeList() throws IOException, IndexOutOfBoundsException {
    print("starting method");
    process();
    print("finishing method");
}
```

No exceptions ↓	`IOException` ↓	`IndexOutOfBoundsException` ↓
starting method finishing method	starting method ~~finishing method~~	starting method ~~finishing method~~

Collections

COLLECTIONS FRAMEWORK [W1.4a] C++ to Java → Collections → The collections framework

Collection (container)

- An object that groups multiple elements into a single unit

The collections framework

- A unified architecture for representing and manipulating collections
- Contains
  - **Interfaces**
    - Abstract data types that represent collections

        - Allow collections to be manipulated independently of the details of their representation
        - `List<E>` `interface` can be used to manipulate list-like collections which may be implemented in different ways such as `ArrayList<E>` or `LinkedList<E>`
    - **Implementations**
        - Concrete implementations of the collection interfaces
        - Reusable data structures
        - `ArrayList<E>` class `implements` the `List<E>` interface
        - `HashMap<K, V>` class `implements` the `Map<K, V>` interface
    - **Algorithms**
        - Methods that perform useful computations, (eg searching, sorting), on objects that implement collection interfaces
        - Polymorphic
            - Same method can be used on many different implementations of the appropriate collection interface
        - `sort(List<E>)` method can sort a collection that implements the `List<E>` interface
- The Collections Framework ≠ C++ Standard Template Library (STL)

Core collection interfaces

- `Collection`
    - Root of collection hierarchy
    - Represents its elements (a group of objects)
    - Least common denominator that all collections implement
    - Used to pass collections around and manipulate them when maximum generality is desired
    - May allow duplicate elements
    - May be ordered or unordered
    - The Java platform doesn't provide any direct implementations of this interface, but provides implementations of more specific subinterfaces, such as `Set` and `List`
- `Set`
    - Collections that cannot contain duplicate elements
    - Represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine
- `List`
    - Ordered collection (sequence)
    - Can contain duplicate elements
    - Control over where in the list each element is inserted
    - Can access elements by their integer index (position)
- `Queue`
    - Collection to hold multiple elements prior to processing
    - Additional insertion, extraction, and inspection operations
- `Map`
    - An object that maps keys to values
    - Cannot contain duplicate keys
    - Each key can map to at most one value
- `Dequeue`
- `SortedSet`
- `SortedMap`

THE `ArrayList` CLASS [W1.4b] C++ to Java → Collections → The ArrayList class

`ArrayList` class

- Resizable-array implementation of the `List` interface
- Unlike a normal `array`, an `ArrayList` can grow in size as you add more items

Example: `ArrayList` of `String` objects

```
import java.util.ArrayList;

public class ArrayListDemo {

    public static void main(String args[]) {
        ArrayList<String> items = new ArrayList<>();

        System.out.println("Before adding any items:" + items);

        items.add("Apple");
        items.add("Box");
        items.add("Cup");
        items.add("Dart");
        print("After adding four items: " + items);

        items.remove("Box"); // remove item "Box"
        print("After removing Box: " + items);

        items.add(1, "Banana"); // add "Banana" at index 1
        print("After adding Banana: " + items);

        items.add("Egg"); // add "Egg", will be added to the end
        items.add("Cup"); // add another "Cup"
        print("After adding Egg: " + items);

        print("Number of items: " + items.size());

        print("Index of Cup: " + items.indexOf("Cup"));
        print("Index of Zebra: " + items.indexOf("Zebra"));

        print("Item at index 3 is: " + items.get(2));

        print("Do we have a Box?: " + items.contains("Box"));
        print("Do we have an Apple?: " + items.contains("Apple"));

        items.clear();
        print("After clearing: " + items);
    }

    private static void print(String text) {
        System.out.println(text);
```

```
    }
}

//output
Before adding any items:[]
After adding four items: [Apple, Box, Cup, Dart]
After removing Box: [Apple, Cup, Dart]
After adding Banana: [Apple, Banana, Cup, Dart]
After adding Egg: [Apple, Banana, Cup, Dart, Egg, Cup]
Number of items: 6
Index of Cup: 2
Index of Zebra: -1
Item at index 3 is: Cup
Do we have a Box?: false
Do we have an Apple?: true
After clearing: []
```

Example: Addition of numbers in the `ArrayList` of `Integer` objects

```
    private static ArrayList<Integer> numbers = new ArrayList<>();

    private static void addNumber(int i) {
        numbers.add(Integer.valueOf(i));
        System.out.println(numbers);
    }
```

`HashMap` CLASS [W1.4b] C++ to Java → Collections → The ArrayList class

`HashMap`

- Implementation of the `Map` interface
- Store a collection of **key-value** pairs

Example: `HashMap<String, Point>` to maintain a list of coordinates and their identifiers

- e.g., the identifier `x1` (key) is used to identify the point `0,0` (value)

```
import java.awt.Point;
import java.util.HashMap;
import java.util.Map;

public class HashMapDemo {
    public static void main(String[] args) {
        HashMap<String, Point> points = new HashMap<>();

        // put the key-value pairs in the HashMap
        points.put("x1", new Point(0, 0));
        points.put("x2", new Point(0, 5));
        points.put("x3", new Point(5, 5));
```

```
        points.put("x4", new Point(5, 0));

        // retrieve a value for a key using the get method
        print("Coordinates of x1: " + pointAsString(points.get("x1")));

        // check if a key or a value exists
        print("Key x1 exists? " + points.containsKey("x1"));
        print("Key y1 exists? " + points.containsKey("y1"));
        print("Value (0,0) exists? " + points.containsValue(new Point(0, 0)));
        print("Value (1,2) exists? " + points.containsValue(new Point(1, 2)));

        // update the value of a key to a new value
        points.put("x1", new Point(-1,-1));

        // iterate over the entries
        for (Map.Entry<String, Point> entry : points.entrySet()) {
            print(entry.getKey() + " = " + pointAsString(entry.getValue()));
        }

        print("Number of keys: " + points.size());
        points.clear();
        print("Number of keys after clearing: " + points.size());

    }

    public static String pointAsString(Point p) {
        return "[" + p.x + "," + p.y + "]";
    }

    public static void print(String s) {
        System.out.println(s);
    }
}

// output
Coordinates of x1: [0,0]
Key x1 exists? true
Key y1 exists? false
Value (0,0) exists? true
Value (1,2) exists? false
x1 = [-1,-1]
x2 = [0,5]
x3 = [5,5]
x4 = [5,0]
Number of keys: 4
Number of keys after clearing: 0
```

Example: `HashMap` to store the number of entries for each day

```java
import java.util.HashMap;
import java.util.Map;

public class Main {
    private static HashMap<String, Integer> roster = new HashMap<>();

    private static void addToRoster(String day) {
        if (roster.containsKey(day)){
            Integer newValue = Integer.valueOf(roster.get(day).intValue() + 1);
            roster.put(day, newValue);
        } else {
            roster.put(day, Integer.valueOf(1));
        }
    }

    public static void main(String[] args) {
        addToRoster("Monday"); // i.e., one person signed up for Monday
        addToRoster("Wednesday"); // i.e., one person signed up for Wednesday
        addToRoster("Wednesday"); // i.e., another person signed up for Wednesday
        addToRoster("Friday");
        addToRoster("Monday");
        printRoster();
    }
}
```

JUnit

JUnit: Basic [W3.7d] C++ to Java → JUnit → JUnit: Basic

```java
public class IntPair {
    int first;
    int second;

    public IntPair(int first, int second) {
        this.first = first;
        this.second = second;
    }

    /**
     * Returns The result of applying integer division first/second.
     * @throws Exception if second is 0.
     */
    public int intDivision() throws Exception {
        if (second == 0){
            throw new Exception("Divisor is zero");
        }
```

```
        return first/second;
    }
}
```

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.fail;

public class IntPairTest {

    @Test
    public void intDivision_nonZeroDivisor_success() throws Exception {
        // normal division results in an integer answer 2
        assertEquals(2, new IntPair(4, 2).intDivision());

        // normal division results in a decimal answer 1.9
        assertEquals(1, new IntPair(19, 10).intDivision());

        // dividend is zero but divisor is not
        assertEquals(0, new IntPair(0, 5).intDivision());
    }

    @Test
    public void intDivision_zeroDivisor_exceptionThrown() {
        try {
            assertEquals(0, new IntPair(1, 0).intDivision());
            fail(); // the test should not reach this line
        } catch (Exception e) {
            assertEquals("Divisor is zero", e.getMessage());
        }
    }
}
```

- Each test method is marked with a `@Test` annotation
- `assertEquals(expected, actual)`, `assertNull`, `assertNotNull`, `assertTrue`, `assertFalse` methods
- camelCase for method names but for test methods, sometimes another convention is used:
  - `unitBeingTested_descriptionOfTestInputs_expectedOutcome`
    - e.g., intDivision_zeroDivisor_exceptionThrown
- Verify the code throws the correct exception.
  - If the exception is thrown, it will be caught and further verified inside the catch block.
  - But if it is not thrown as expected, the test will reach fail() line and will fail.

Miscellaneous Topics

ENUMERATIONS [W1.1k] C++ to Java → Miscellaneous Topics → Enumerations

- `enum` keyword
- names of an enum type's fields are in UPPERCASE letters e.g. `FLAG_SUCCESS`
- While enumerations are usually a simple set of fixed values, Java enumerations can have behaviors too

Example: An enumeration to represent days of a week (in `Day.java` file)

```
public enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

```
Day today = Day.MONDAY;
Day[] holidays = new Day[]{Day.SATURDAY, Day.SUNDAY};

switch (today) {
case SATURDAY:
case SUNDAY:
   System.out.println("It's the weekend");
   break;
default:
   System.out.println("It's a week day");
}
```

Example: An enumeration to represent priority colors

```
public enum Priority {
    HIGH, MEDIUM, LOW
}
```

```
public class Main {
   public static void describe(String color, Priority p) {
       switch (p) {
           case LOW:
               System.out.println(color + " indicates low priority");
               Break;
           case MEDIUM:
               System.out.println(color + " indicates medium priority");
               break;
           case HIGH:
               System.out.println(color + " indicates high priority");
               break;
       }
   }

   public static void main(String[] args) {
       describe("Red", Priority.HIGH);
```

```
        describe("Orange", Priority.MEDIUM);
        describe("Blue", Priority.MEDIUM);
        describe("Green", Priority.LOW);
    }
}

// output
Red indicates high priority
Orange indicates medium priority
Blue indicates medium priority
Green indicates low priority
```

FILE ACCESS [W3.4c] C++ to Java → Miscellaneous Topics → File access

- `java.io.File` class to represent a file object

Create a `File` object to represent a file `fruits.txt` that exists in the `data` directory **relative to the current working directory** and uses that object to print some properties of the file

```
import java.io.File;

public class FileClassDemo {
    public static void main(String[] args) {
        File f = new File("data/fruits.txt");
        System.out.println("full path: " + f.getAbsolutePath());
        System.out.println("file exists?: " + f.exists());
        System.out.println("is Directory?: " + f.isDirectory());
    }
}
```

```
// output

full path: C:\sample-code\data\fruits.txt
file exists?: true
is Directory?: false
```

`Scanner` object to read (and print) contents of a text file line-by-line

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileReadingDemo {

    private static void printFileContents(String filePath) throws
FileNotFoundException {
```

```
        File f = new File(filePath);
                                // create a File for the given file path
        Scanner s = new Scanner(f);
        while (s.hasNext()) {
            System.out.println(s.nextLine());
        }
    }

    public static void main(String[] args) {
        try {
            printFileContents("data/fruits.txt");
        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        }
    }
}
```

```
// output

5 Apples
3 Bananas
6 Cherries
```

`java.io.FileWriter` object to write to a file

```
import java.io.FileWriter;
import java.io.IOException;

public class FileWritingDemo {

    private static void writeToFile(String filePath, String textToAdd)
throws IOException {
        FileWriter fw = new FileWriter(filePath);
        fw.write(textToAdd);
        fw.close();                 // complete writing operation
    }

    public static void main(String[] args) {
        String file2 = "temp/lines.txt";
        try {
            writeToFile(file2, "first line" + System.lineSeparator() +
"second line");
        } catch (IOException e) {
            System.out.println("Something went wrong: " + e.getMessage());
        }
    }
}
```

```
// Contents of the temp/lines.txt:

first line
second line
```

`FileWriter` object that appends to the file (instead of overwriting the current content)

```
private static void appendToFile(String filePath, String textToAppend)
throws IOException {
    FileWriter fw = new FileWriter(filePath, true);
                                    // create a FileWriter in append mode
    fw.write(textToAppend);
    fw.close();
}
```

`java.nio.file.Files` : Utility class that provides several useful file operations

- `java.nio.file.Paths` file generates Path objects that represent file paths

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public class FilesClassDemo {

    public static void main(String[] args) throws IOException{
        Files.copy(Paths.get("data/fruits.txt"),
Paths.get("temp/fruits2.txt"));
        Files.delete(Paths.get("temp/fruits2.txt"));
    }
}
```

PACKAGES `[W3.4d] C++ to Java → Miscellaneous Topics → Packages`

- Organize **types** (i.e., classes, interfaces, enumerations, etc.) into **packages**
- One package statement at the first line of every source file in that package
  - If you do not use a package statement, your type doesn't have a package
- Package of a type should match the folder path of the source file

`Formatter` class (in <source folder>/seedu/tojava/util/Formatter.java file) is in the package `seedu.tojava.util`. When it is compiled, the `Formatter.class` file will be in the location <compiler output folder>/seedu/tojava/util

```
package seedu.tojava.util;

public class Formatter {
    public static final String PREFIX = ">>";
```

```
    public static String format(String s){
        return PREFIX + s;
    }
}
```

- Package names are written in all lower case

To use a public package member from outside its package, do one of the following:

- Use the **fully qualified name** to refer to the member
- **Import** the package or the specific package member
    - NOTE: Importing a package does not import its sub-packages

```
package seedu.tojava;

// imports the class StringParser in the seedu.tojava.util package
import seedu.tojava.util.StringParser;

// imports all classes in the package
import seedu.tojava.frontend.*;

public class Main {

    public static void main(String[] args) {

        // Use fully qualified name to access the Processor class
        String status = seedu.tojava.logic.Processor.getStatus();

        // Using the StringParser previously imported
        StringParser sp = new StringParser();

        // Using classes from the tojava.frontend package
        Ui ui = new Ui();
        Message m = new Message();

    }
}
```

Static import → Import static members of a type

- So imported members can be used without specifying the type name
- Example
    - Import the constant PREFIX and the method format() from the seedu.tojava.util.Formatter class
        - **import static** seedu.tojava.util.Formatter.PREFIX;
        - **import static** seedu.tojava.util.Formatter.format;
    - the class can use PREFIX and format()
        - Instead of Formatter.PREFIX and Formatter.format

USING JAR FILES `[W3.4e] C++ to Java → Miscellaneous Topics → Using JAR files`

- Java applications are typically delivered as JAR (Java Archive) files
  - `java -jar` foo.jar launches the foo.jar file

JavaFX `[W4.4a] C++ to Java → Miscellaneous Topics → JavaFX`

- JavaFX is a technology for building Java-based GUIs
- https://se-education.org/guides/tutorials/javaFx.html

Varags - Variable Arguments

- Variable Arguments (Varargs)
  - A syntactic sugar type feature that allows writing a method that can take a variable number of arguments.

The `search` method below can be called as `search()`, `search("book")`, `search("book", "paper")`, etc.

```
1 public static void search(String ... keywords){
2    // method body
3 }
```