Requirements

Software Requirement specifies a need to be fulfilled by the software product. [come from **stakeholders**]

- Identifying requirements is often not easy
- can be divided to **FR** and **NFR**
- can be **prioritized** based on the importance and urgency [group requirements into priority categories]
- can be **discarded** if they are considered 'out of scope'

Brownfield project: develop a product to replace/update an **existing software product**
Greenfield project: develop a totally new system from **scratch**

Stakeholder: An individual or an organization that is involved or potentially affected by the software project. **Example:** users, sponsors, developers, interest groups, government agencies

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

- NFRs are **easier to mis**s [stakeholders tend to think of functional requirements first]
- Sometimes NFRs are critical to success of the software

Scope

- **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.
- **Others:** 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, and more ...

Well-defined requirements

For individual requirements

- Unambiguous
- Testable (verifiable)
- Clear (concise, terse, simple, precise)
- Correct
- Understandable
- Feasible (realistic, possible)
- Independent
- Atomic **[Not divisible any further]**
- Necessary
- Implementation-free (i.e. abstract)

For set of requirements as a whole should be

- Consistent
- Non-redundant
- Complete

In a **brainstorming session** there are no "bad" ideas. The aim is to generate ideas; not to validate them.
User Surveys: To solicit responses and opinions from a large number of stakeholders regarding a current product or a new product.

Observation:

- Observing users in their natural work environment can **uncover product requirements**.
- Usage **data of an existing system** can also be used to gather information about how an existing system is being used. e.g. to find the situations where the user makes mistakes when using the current system.

Interviews: Interviewing **stakeholders** and **domain experts** can produce useful information about project requirements.

Focus groups: A kind of **informal interview** within an interactive group setting.

Prototype: A mock up, a scaled down version, or a partial system constructed [E.g., UI prototype using Wireframe D]

- to get users' feedback.
- to validate a technical concept
- to give a preview
- to compare multiple alternatives on a small scale before committing fully to one alternative.
- for early field-testing under controlled conditions.
- for discovering as well as specifying requirements
- can **uncover requirements**

Prose: A textual description to describe requirements.

- Especially **useful** when **describing abstract ideas** such as the vision of a product.
- **Avoid using lengthy prose** to describe requirements; they can be **hard to follow [understand]**.

Feature list: A list of features of a product

- grouped according to some criteria such as **aspect, priority, order of delivery**, etc.
- can be organized hierarchically.

User story: Short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a **user** or **customer** of the system

- capture user requirements in a way that is convenient for **scoping, estimation**, and **scheduling**
- can capture non-functional requirements
- handy for recording requirements during **early stages** of requirements gathering
- differ from traditional requirements specifications mainly in the level of detail.
- only provide enough details to make a low risk estimate of **how long** the user story will take to implement. [When implement, meet customer to discuss more detail]
- It is **fine to add more details** such as conditions, priority, urgency, effort estimates

Writing a user story

- **format:** As a {user type/role} I can {function} so that {benefit} [{benefit} can be omitted if it is **obvious**.]
  Invalid Example: As a developer, I …, so that …..
  [not written from the perspective of the user/customer.]
- write using a physical medium or a digital tool [Trello]
- can write user stories at various levels.
- can add conditions of satisfaction to a user story

> *As a* lecturer, *I can* view the forum post count of each student *so that* I can identify the activity level of students in the forum.
>
> Conditions:
>
> ☑ Separate post count for each forum should be shown
> ☐ Total post count of a student should be shown
> ☑ The list should be sortable by student name and post count

Other useful info that can be added includes (but not limited to)

- **Priority:** how important the user story is
- **Size:** the estimated effort to implement the user story
- **Urgency:** how soon the feature is needed

Epics (or themes): High-level user stories, cover bigger functionality. You can break down these epics to multiple user stories of normal size.

> [Epic] As a lecturer, I can monitor student participation levels
>
> - As a lecturer, I can view the forum post count of each student so that I can identify the activity level of students in the forum
> - As a lecturer, I can view webcast view records of each student so that I can identify the students who did not view webcasts
> - As a lecturer, I can view file download statistics of each student so that I can identify the students who did not download lecture materials

Tips

- Clear your mind even you alr. have some idea of product
- 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

Use Case: describes an interaction between the **system** and the **user(its actors)** for a specific functionality of the system.

- capture the functional requirements of a system
- **can not** capture NFRs, i.e., performance requirements
- can be expressed at different levels of abstraction [various levels of detail, Not just highest]
- **use case diagrams** [UML] that can illustrate use cases of a system visually
- no strict prefix requirement [i.e., no need to be UC only]

Use case in Testing

- **Use cases can be used for system testing and acceptance testing.**
- **To increase the E&E of testing, high-priority use cases are given more attention.**

Actor:

- A use case can involve multiple actors.
- An actor can be involved in many use cases.
- A single person/system can play many roles.
- Many persons/systems can play a single role.
- Use cases can be specified at various levels of detail.

Writing use case steps

- The main body is a sequence of steps that describes the interaction between the system and the actors.
- Describes **only the externally** visible behavior, **not internal** details, of a system
- A step **gives the intention** of the actor (not the mechanics). **UI details are usually omitted**. [leave as much flexibility to the UI designer as possible]
- numbering style is **not** a universal rule but a **widely used convention**
  Note: extension marked as ***a**. can happen at any step

Main Success Scenario (MSS)

- describes the most straightforward interaction for a given use case
- MSS should be **self-contained** [give us a **complete** usage scenario].
- assumes that **nothing** goes wrong
- Also called the **Basic Course of Action** or the **Main Flow of Events** of a use case.

Extensions: "add-on"s to the MSS that describe **exceptional / alternative** flow of events.

- scenario that can happen if certain things are not as expected by the MSS
- appear **below** the MSS
- not useful to mention events such as power failures or system crashes as extensions [system cannot function beyond such **catastrophic failures**]

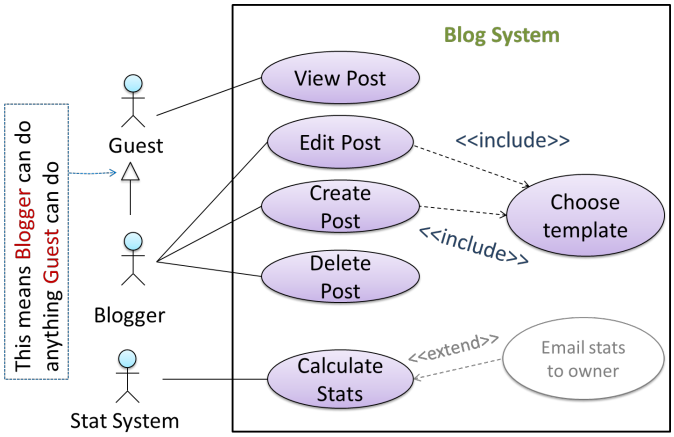
Inclusion: include **another use case** [use Underline Text]

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 diagram

[↑ Also means all use cases available to Actor A are also available to Actor B]



- brief summary, do not over-complicate
- recommended not to use System as a user
- less detailed than textual use cases

Details

- **Inclusion:** dotted arrow and an <<include>> annotation [MSS->inclusion]
- **Extension:** dashed line arrow and <<extend>> arrows [extension->MSS]
- Inclusion and Extension arrow direction are **different**
- can use **actor generalization** in use case diagrams using a symbol similar to that of UML notation for **inheritance**.

Pros:

- simple notation and plain English descriptions, easy understood
- decouple user intention from mechanism → more freedom to optimize functionality
- Identifying all possible extensions encourages us to consider all situations
- Separating typical scenarios from special cases encourages us to optimize the typical scenarios.

Main Cons: Not good for capturing requirements that do not involve a user interacting with the system → should **not** be used as the **sole** means to specify requirements.

Glossary: A glossary serves to ensure that all stakeholders have a common understanding of the noteworthy terms, abbreviations, acronyms etc.

Supplementary requirements section: used to capture requirements that do not fit elsewhere. Typically, **this is where most Non-Functional Requirements will be listed**.

Project Management

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

RCS: Revision control software, store the history of the working directory as a series of **commits**.

- can be done in centralized way or distributed way

Centralized RCS (CRCS)

- uses a central remote repo that is shared by the team
- interact directly with this central repository
- CVS and SVN support **only** this model

Distributed RCS (DRCS)

- allows multiple remote/local repos **working together**
- workflow can vary from team to team
- Git and Mercurial support this

Pros of use an **automated** revision control tool for a project

- track the history and evolution of your project
- makes it easier for you to collaborate
- helps you to recover from mistakes
- helps you to work simultaneously on, and manage the drift between, multiple versions of your project.

Repository: database that stores the revision history tracked by an RCS software

Working directory: root directory revision-controlled by Git.

Committing saves a snapshot of the current state of the tracked files in the revision control history

Each **commit** in a repo

- is a **recorded point** in the history of the project
- is **uniquely** identified by an **auto-generated hash**
- can **tag** a specific commit with a more easily identifiable name
- **diff** to see what changed between two points of the history
- To restore the state of the working directory at a point in the past, you can **checkout** the commit in concern.

Remote Repositories: repos that are hosted on remote computers

- **pull (or fetch)** from one repo to another, to receive new commits in the second repo
- A repo can work with any number of other repositories as long as they have a shared history
- A **fork** is a remote copy of a remote repo.

Branching is the process of evolving multiple versions of the software in **parallel**.

Merge conflicts happen when you try to merge two branches that had changed the **same part** of the code and the RCS **cannot decide which changes to keep**.

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.

WBS: **Work Breakdown Structure** depicts info. about **tasks** and their **details** in terms of **subtasks**.

- effort is measured in **man hour/day/month**
- All tasks should be **well-defined** [clear as to when the task will be considered done.]

Milestone: **end of a stage** which indicates significant progress.

- Each intermediate product release is a milestone.
- If not practical to have a very detailed plan→use a high-level plan for the whole project and a detailed plan for the **next few milestones**.

Buffer: time set aside to absorb any unforeseen delays

Do not inflate task estimates to create hidden buffers; have explicit buffers instead.

Reason: With explicit buffers, it is easier to detect incorrect effort estimates which can serve as feedback to improve future effort estimates.

Issue trackers (bug trackers): used to track task assignment and progress. [GitHub, SourceForge, and BitBucket]

Team Structures

SDLC process models

Software Development Lifecycle (SDLC): different stages of software development. such as requirements, analysis, design, implementation and testing.

Software Development Lifecycle Models (software process models): approaches that describe different ways to go through the SDLC.

Sequential Model [waterfall model]

- views software development as a **linear process**
- When one stage of the process is completed, it **produces some artifacts to be used** in the next stage.
- A strict sequential model project **moves only** in the **forward** direction
- can work well for a project that produces software to solve a **well-understood problem**
- **real-world** projects **often** tackle problems that are not well-understood at the beginning [**unsuitable** for this model]

Iterative Model

- advocates producing the software by going through several iterations.
- produces a new version of the product **each iteration**
- can be done in **breadth-first** or **depth-first** approach or **mixture**.

Breadth-first approach

- an iteration **evolves all major** components and all functionality areas in parallel
- **most** features and **most** components will be updated in each iteration
- producing a **working product** at the end of each iteration.

Depth-first approach

- an iteration focuses on **fleshing out only some** components or some functionality area.
- early iterations might not produce a working product.

<u>Agile processes</u>

eXtreme Programming (XP)

- stresses **customer satisfaction**
- aims to **empower developers** to confidently respond to changing customer requirements
  [even late in the lifecycle]
- emphasizes **teamwork**
- aims to improve a software project in five essential ways: communication, simplicity, feedback, respect, and courage
- has a set of **simple rules**
- Pair programming, CRC cards, project velocity, and standup meetings are some interesting topics related to XP

Scrum: a process **skeleton** that contains sets of practices and predefined roles. The main roles in Scrum are:

- **Scrum Master:** maintains the processes
- **Product Owner**: represents the stakeholders and the business
- **The Team**, a cross-functional group who do the actual analysis, design, implementation, testing, etc.

A Scrum project is divided into iterations called **Sprints**

- basic unit of development
- tend to last between one week and one month
- are a timeboxed (i.e. restricted to a specific duration) effort of a constant length.
- preceded by a planning meeting
- team creates potentially deliverable product increment during each sprint

Key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need

Daily Scrum

- is another key scrum practice
- each team member answers the following three questions during the daily scrum
  - What did you do yesterday?
  - What will you do today?
  - Are there any impediments in your way?
- meeting [Morning] is not used as a problem-solving or issue resolution meeting.
- Issues raised are taken offline and usually dealt with by the relevant subgroup immediately after the meeting.

Debugging and Refactoring

IDEs: Integrated Development Environments
Debugging: process of discovering defects in the program.
Bad Approach

- By inserting temporary print statements
- By manually tracing through the code

Recommended Approach: Using a debugger

Refactor: Improving a program's **internal structure** in **small steps** without modifying its external behavior

- is not rewriting [done in small steps]
- is not bug fixing [alter the external behavior]
- hidden bugs become easier to spot
- improve performance [sometimes, simpler faster than complex]
- too much refactoring: benefits no longer justify the cost: some are '**opposites**' of each other

Refactoring, even if done with the aid of an IDE, may still **result in regressions**.
Each small refactoring MUST followed by **regression testing**.
Commonly used Refactoring

- Consolidate Conditional Expression
- Decompose Conditional
- Inline Method **[inverse of Extract Function]**
- Remove Double Negative
- Replace Magic Literal
- Replace Nested Conditional with Guard Clauses
- Replace Parameter with Explicit Methods
- Reverse Conditional
- Split Loop
- Split Temporary Variable

Identify refactoring opportunities by **code smells**.
Code Smell: a surface indication that usually **corresponds to a deeper problem** in the system.

- a smell is something that's quick to spot
- smells don't always indicate a problem.

Code Smell Data Class: a class with **all data** and **no behavio**r
$\Longrightarrow$ Explore if refactoring it to move the corresponding behavior into that class is appropriate

Periodic refactoring: good way to pay off the technical debt a codebase has accumulated.
Cruft: deficiencies in internal quality that make it harder than it would ideally be to modify and extend the system further.

Documentation Principle

- **Developer-as-user:** there is a need for how such components are to be used
- **Developer-as-maintainer:** there is a need for designed, implemented and tested
- There **isn't one** thing called documentation, there are **four**: tutorials, how-to guides, explanation and technical reference.
- A writer-friendly source format is also **desirable as non-programmers**
- When writing project documents, a **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]
- It is **not a good idea to have separate sections** for each type of artifact
  - Aim for '**just enough**' **developer doc**umentation.
  - Minimize that overhead.
- Documentation should complement the code and should provide only just enough guidance to get started.
- Anything that is **already clear** in the **code** need **not** be **described in words**
- Focus on **providing higher level information** that is not readily visible in the code or comments.
- Describe the **similarities in one place** and emphasize only the **differences in other places**.
- JavaDoc is a tool for **generating API documentation** in HTML format from comments in the source code

Error Handling

Exception: Error occurs in the execution→the code being executed creates an **exception object**→hands it off to the **runtime** system.
contains information about the error, including its type and the state of the program when the error occurred.
Throwing: Creating an **exception object** and handing it to the runtime system
↓
Call Stack: **Runtime** system attempts to find something to handle it.
↓
Searches [in reverse order] the call stack for a method that contains **exception handler** that can handle the exception.
↓
Catch: The exception handler chosen.
Assertions: define assumptions about the program state so that the **runtime can verify** them.

- **assertion failure** indicates a possible bug in the code
- can be disabled without modifying the code. (Java disables assertions **by default**)
- recommend to use liberally in the code.
  [impact on performance is low]
- Do not use to do work [assertions **can be disabled**]
- suitable for verifying assumptions about Internal Invariants, Control-Flow Invariants, Preconditions, Postconditions, and Class Invariants.

Java assert vs JUnit assertions:

- Both check for a given condition
- JUnit assertions are more powerful and customized for testing.
- JUnit **assertions** are **not disabled** by default
- Use JUnit **assertions** in **test** code
- Use Java **assert** in **functional** code.

Exceptions	Assertions
an unusual condition created by the user or the environment	programmer made a mistake in the code
• complementary ways of handling errors • different purposes	

Logging: deliberate recording of certain information during a program execution for future reference.

- **useful for troubleshooting problems**. [records some system information regularly]
- Most programming environments come with logging systems that allow sophisticated forms of logging.
- possible to log information in other ways
  e.g. into a database or a remote server.

Defensive programmer codes under the assumption "if you leave room for things to go wrong, they will go wrong".
not necessary to be 100% defensive all the time
[less prone to be misused or abused→complicated and slow]

Integration

Combining parts of a software product to form a whole.
two general approaches

1. **Late and one-time:** wait till all components are completed and integrate all finished components near the end of the project. **[not recommended]**
2. **Early and frequent:** integrate early and evolve each part in parallel, in small steps, **re-integrating frequently**.

Walking Skeleton [associate with E&f integration]

- has all the high-level components needed for the first version in their minimal form, compiles, and runs
- may not produce any useful output yet

Big-bang integration: integrating **too many changes** at the same time. [**not recommended**]
Incremental integration: integrate **a few** components at a time.

Build automation tools automate the steps of the build process, usually by means of build scripts.

- Some of these build steps such as compiling, linking and packaging, are already automated in most modern IDEs.
- Some popular build tools relevant to Java developers: Gradle, Maven, Apache Ant, GNU Make
- Some other build tools: Grunt (JavaScript), Rake (Ruby)

Some build tools (Maven, Gradle) also serve as **dependency management** tools. [Modern software projects often depend on third party libraries that evolve constantly]

Continuous integration (CI)

- extreme application of build automation.
- integration, building, and testing happens **automatically** after each code change.

Continuous Deployment (CD)

- natural extension of CI
- changes are not only integrated continuously, but also deployed to end-users at the same time.

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

Reuse

a major theme in SWE practices.

- enhance robustness of a new software system
- reduce manpower and time requirement

costs of reuse

- may be an overkill [increasing size/degrade performance]
- may not be mature/stable enough to be used in an important product
- has the risk of dying off left dependency no longer maintained.
- license restriction
- might have bugs, missing features, or security vulnerabilities
- Malicious code can sneak into your product via compromised dependencies.

Application Programming Interface (API): specifies the interface through which other programs can interact with a software component. [contract between component and its clients]

- **A class has an API:** a collection of **public methods** that you can invoke to make use of the class.
- The **GitHub API** is a collection of web request formats that the GitHub server accepts and their corresponding responses. [interacts with GitHub through that API]

When developing large systems, if you define the API of each component early, the development team can develop the components in **parallel**

Library: collection of modular code that is general and can be used by other programs.

- **Java classes you get with the JDK are library** classes that are provided in the default Java distribution.

Use a library

- **Read the doc.** to confirm its functionality fits needs
- **Check the license**
- **Download the library and make it accessible** to your project. [Alternatively, you can configure your **dependency management tool** to do it for you.]
- **Call the library API** from your code where you need to use the library's functionality.

Framework: a **reusable implementation of a software** (**or part** thereof) providing generic functionality that can be selectively customized to produce a specific application.

- Some frameworks provide a complete implementation of a default behavior which makes them immediately usable.
- Some frameworks cover **only a specific** component or an **aspect.**
- A framework facilitates the adaptation and customization of some desired functionality.
- use a technique called **inversion of control**, aka the "**Hollywood principle**"

Example: Eclipse

- is an **IDE framework**
- is a fully functional Java IDE out-of-the-box.
- Eclipse plugin system can be used to create an IDE for different programming languages

Example

- JavaFX is a framework for creating Java GUIs
- Tkinter is a GUI framework for Python
- JUnit (Java) is a framework for testing

- Frameworks for web-based applications: Drupal (PHP), Django (Python), Ruby on Rails (Ruby), Spring (Java)
- Frameworks for testing: JUnit (Java), unittest (Python), Jest (JavaScript)

Libraries	Frameworks
meant to be used 'as is'	customized/extended
code calls the library code	framework code calls your code

Platform: provides a runtime environment for applications

- an **operating system can be called a platform**
- Two well-known examples of platforms are JavaEE and .NET [both are **platform** and **framework**]

Software Design Pattern

Design:

- process of **transforming the problem into a solution**
- the **solution is also called design**

Two main aspects

Product/external design: designing the external behavior of the product to meet the users' requirements. Done by product designers.

Implementation/internal design: designing how the product will be implemented to meet the required external behavior. Done by software architects and software engineers.

Abstraction

- **Guiding principle:** Only consider details that are relevant to the current perspective or the task at hand.
- Large amounts of intricate details is impossible to deal with at the same time→Need abstraction!
- **Data abstraction**: abstracting away the lower level data items and thinking in terms of bigger entities
- **Control abstraction:** 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 computer.
- can be applied **repeatedly** to obtain progressively higher levels of abstraction.
- not limited to just data or control abstractions. [**general concept**]

Examples

- An **OOP class** is an abstraction over related **data** and **behaviors**.
- An **architecture** is a higher-level abstraction of the **design of a software**.
- **Models** (e.g., UML models) are abstractions of some aspect of **reality**.

Coupling: measure of the degree of dependence. High coupling (aka tight coupling or strong coupling) is **discouraged**

- **Maintenance is harder** because a change in one module could cause changes in other modules coupled to it (i.e. a ripple effect).
- **Integration is harder** because 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.

A is coupled to B if a change to B can **potentially (but not necessarily always)** require a change in A.

Example:

- A has access to the internal structure of B (this results in a **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 para. or a return value
- A inherits from B
- A and B are required to follow the same data format or communication protocol

Cohesion: measure of how strongly-related and focused the various responsibilities of a component are. Higher cohesion is better. (keeps related functionalities together while keeping out all other unrelated things)

Disadvantages of **low cohesion** (aka weak cohesion):

- **Lowers the understandability** of modules (difficult to express functionalities at a higher level)
- **Lowers maintainability** because a module can be modified due to unrelated causes or many modules may need to be modified to achieve a small change in behavior.
- **Lowers reusability** of modules because they do not represent logical units of functionality.

Cohesion can be present in many forms.

- Code related to a single concept is kept together
- Code **invoked close together** in time is kept together
- Code manipulates same data structure is kept together

Multi-level design

- **Smaller system**: can be shown in one place.
- **Bigger system:** needs to be done/shown at multi-levels.

Top-down:

- Design the high-level design first
- Flesh out the lower levels later

Especially useful when designing big and novel systems where the **high-level design needs to be stable** before lower levels can be designed.

Bottom-up:

- Design lower level components first
- Put them together to create higher-level systems later.

Usually **Not Scalable for bigger systems**.

When:

- designing a variation of an existing system
- Re-purposing existing components to build new system.

Mix:

- Design the top levels using the top-down approach
- Use bottom-up approach when designing bottom levels

AddressBook

- Level2 has a single-level design.
- Level3 has a multi-level design.

Agile design

- Emergent, **not defined up front.**
- **Design will emerge over time**, evolving to fulfill new requirements and take advantage of new technologies as appropriate.
- Although you will often do some initial architectural modeling at the very beginning of a project, this will be **just enough to get your team going.**
- **Does not produce a fully documented set** of models before you may begin coding

Software Architecture (by software architect)

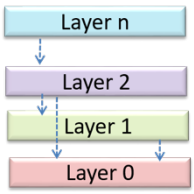
- consists of a set of interacting components
- forms the basis for the implementation

Architecture diagrams are **free-form** diagrams.

- **No** universally adopted **standard** notation
- **Any symbols** reasonable may be used
- **Minimize** the variety of symbols.
- **Avoid** the indiscriminate use of double-headed arrows to show interactions between components. [Some important will be no longer captured.]
- **Follow** various high-level styles (**architectural patterns**)
- Most applications use a **mix** of architectural styles.

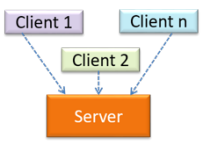
n-tier style:

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

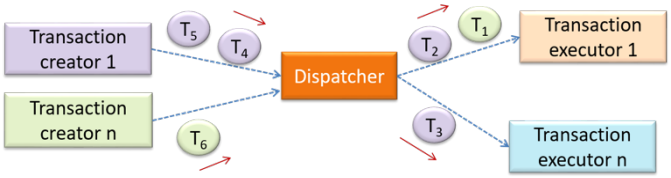


Client-server style has

- at least one component playing the role of a **server**
- at least one **client** component accessing the services of the server.



Transaction processing style: Divides the workload of the system down to a number of transactions → given to a dispatcher that controls the execution of each transaction. Task queuing, ordering, undo etc. are handled by the dispatcher.

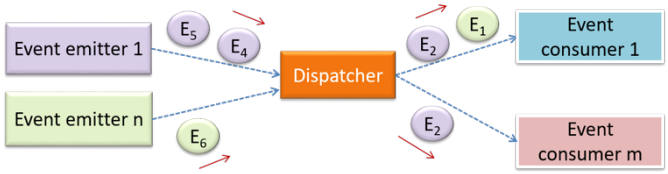


Service-oriented architecture (SOA) style

- Builds applications by combining functionalities packaged as programmatically accessible services.
- Aims to achieve interoperability between **distributed services**.
- May not even be implemented using the same programming language.

Event-driven style: Controls the flow of the application by detecting events from event emitters and communicating those events to interested event consumers.

This architectural style is often used in GUIs. [button clicked]



Design pattern: An **elegant reusable** solution to a commonly recurring problem within a given context in software design. A term popularized by the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software by the so-called "Gang of Four" (GoF)*

Format

- **Context:** The situation or scenario where the design problem is encountered.
- **Problem:** The main difficulty to be resolved.
- **Solution:** The core of the solution. It is important to note that the solution presented **only includes the most general details**, which may **need further refinement** for a specific context.
- **Anti-patterns (optional):** Commonly used solutions, which are **usually incorrect and/or inferior** to the Design Pattern.
- **Consequences (optional):** Identifying the pros and cons of applying the pattern.
- **Other useful information (optional):** Code examples, known uses, other related patterns, etc

Singleton pattern

- **Context:** Certain classes should have no more than just one instance These single instances are commonly known as **singletons.**
- **Problem:** A normal class can be instantiated multiple times by invoking the constructor.

Solution:

- Make the **constructor** of the singleton class **private**
- Provide a public class-level method to access the single instance.

```
1  class Logic {
2      private static Logic theOne = null;
3
4      private Logic() {
5          ...
6      }
7
8      public static Logic getInstance() {
9          if (theOne == null) {
10             theOne = new Logic();
11         }
12         return theOne;
13     }
14 }
```

Pros:

- easy to apply
- effective in achieving its goal with minimal extra work
- provides an **easy way to access the singleton object** from anywhere in the codebase

Cons:

- The singleton object acts like a global variable that increases coupling across the 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 of the others.

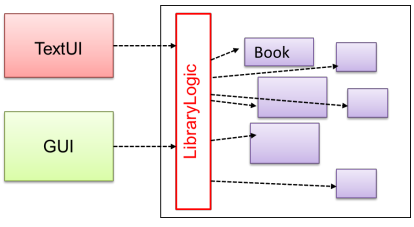
Facade pattern

Context: Components need to **access** functionality deep inside **other components**.

Problem: Access to the component should be allowed without exposing its internal details.

Solution:

Include a Façade class that sits **between** the component internals and users of the component such that **all access to the component happens through the Facade class**.



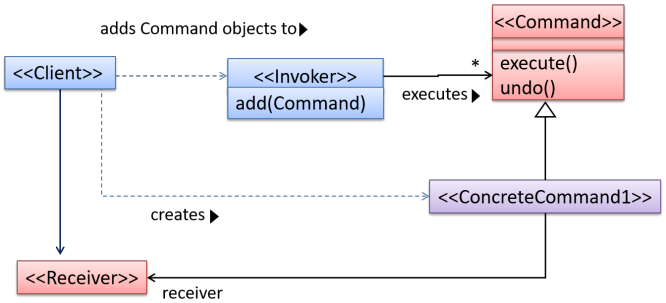
Command pattern

Context: Execute a number of commands, each doing a different task.

Problem: It is preferable that some part of the code executes these commands **without having to know each command type**.

Solution: Have a **general <<Command>> object** that can be passed around, stored, executed, etc without knowing the type of command (i.e. via **polymorphism**).

General Form



- The <<Client>> creates a <<ConcreteCommand>> object passes it to the <<Invoker>>
- <<Invoker>> object treats all commands as a general <<Command>> type.
- <<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().
- <<ConcreteCommand>> object may have to be linked to any <<Receiver>> of the command before it is passed to the <<Invoker>>.

Model view controller (MVC) pattern

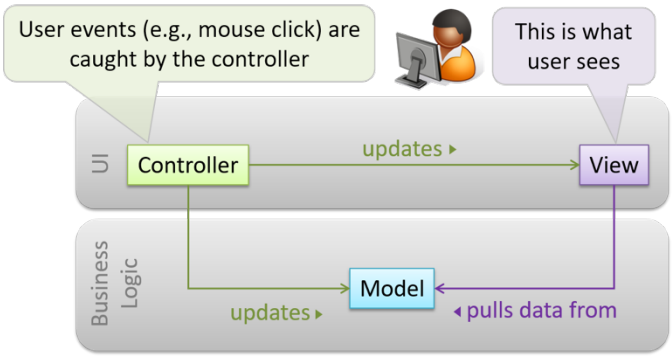
Context: Most applications support storage/retrieval of information, displaying of information to the user (often via multiple UIs having different formats), and changing stored information based on external inputs.

Problem: The high coupling that can result from the interlinked nature of the features described above.

Solution: **Decouple** data, presentation, and control logic of an application by separating them into three different components: **Model, View** and **Controller**.

- **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/changes** the model/view when necessary.
- **Model: Stores** and **maintains** data. **Updates** the view if necessary.

Typically, the **UI** is the combination of **View** and **Controller.** *Note that in a simple UI where there's only one view, Controller and View can be combined as one class.*

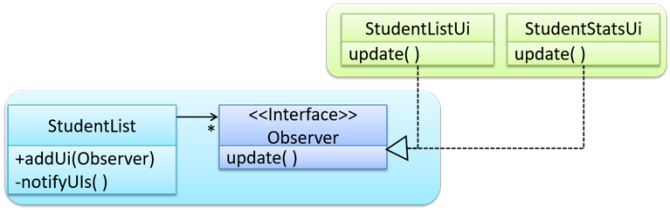


Observer pattern [**e.g., JavaFX**]

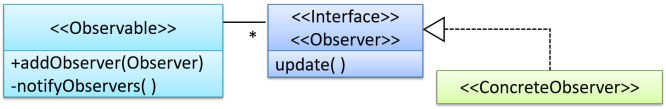
Context: An object (possibly more than one) is interested in being **notified** when a change happens to another object.

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

Solution: Force the communication through an **interface** known to both parties.



General Form



- <<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>>
- 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.

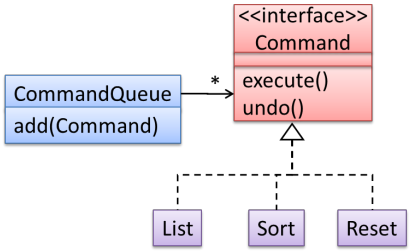
Principle

Single responsibility principle (SRP)

- A class should have one, and only one, reason to change. **[about classes not methods]**
- If a class has only one responsibility, it needs to change only when there is a change to that responsibility

Open-Closed Principle: A module should be **open for extension but closed for modification**.

- modules should be written so that they can be **extended, without requiring them to be modified**
- aims to make a code entity **easy to adapt and reuse** without needing to modify the code entity itself.
- often requires separating the specification (i.e. interface) of a module from its implementation.



Liskov substitution principle (LSP)

- a subclass should not be **more restrictive** than the behavior specified by the superclass.
- If class B is substitutable for parent class A → should pass all test cases of parent class A. Otherwise, not substitutable and violate LSP.

SOLID Principles [five OOP principles]

- **S**ingle Responsibility Principle (SRP)
- **O**pen-Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

Separation of concerns principle (SoC)

- To achieve **better modularity**, separate the code into distinct sections, such that each section addresses a separate concern.
- **reduces functional overlaps** among code sections
- **limits the ripple effect** when changes are introduced to a specific part of the system.
- can be applied at the class level, as well as at higher levels
- lead to **higher cohesion** and **lower coupling**.

Law of Demeter (LoD)

- An object should have **limited knowledge** of another object.
- An object should **only** interact with objects that are **closely related** to it.
- Also called **"Don't talk to strangers"** and "**Principle of least knowledge**"
- aims to **prevent objects from navigating** the internal structures of other objects.

Example: 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)
- Objects from the **direct association** of O

OOP

Object-Oriented Programming (OOP)

- is a **programming paradigm**
- views the world as a **network of interacting objects**
- **OOP solutions** try to create a similar object network inside the computer's memory
- **does not** demand that the virtual world object network follow the real world exactly

Programming paradigm guides programmers to **analyze** programming problems, and **structure** programming solutions, in a specific way.

Paradigm	Programming Languages
Procedural Programming paradigm	C
Functional Programming paradigm	F#, Haskell, Scala
Logic Programming paradigm	Prolog

Java

- is primarily an OOP language
- supports **limited forms** of functional programming
- can be used to (not recommended) write procedural code

JavaScript and **Python support**

- functional
- procedural
- OOP

An **Object** in OOP

- has both **state** (data) and **behavio**r (operations on data), similar to objects in the real world
- has an **interface** and an **implementation**
- interact by sending **messages**
- is an **abstraction mechanism**
  [allows us to abstract away the lower level details and work with bigger granularity entities]
- is an **encapsulation** of some data and related behavior in terms of **packaging aspect** and **information hiding aspect**.

The packaging aspect: An object packages data and related behavior together into **one self-contained** unit.

The information hiding aspect: The data in an object is **hidden from the outside** world and are only accessible using the object's interface.

Class: contains instructions for creating a specific kind of objects

Class-level members: Class-level attributes and methods

Enumeration

- is a fixed set of values
- can be considered as a **data type**.
- useful when using a regular data type such as int or String would allow **invalid values** to be assigned to a variable.

Associations: connections between objects

- **main connections** among the classes in a class diagram.
- can **change** over time
- can be **generalized** as associations between the corresponding classes
- implemented by using **instance level variables**
- can be shown as an **attribute** instead of a line.
  `name: type [multiplicity] = default value`
  Show each association as **either an attribute or a line** but **not both**. A line is preferred as it is easier to spot.

Association class

- represents **additional information** about an association
- is a **normal class** but plays a special role from a design point of view.
- can be implemented as a **normal class** with variables to represent the endpoint of the association it represents.

Navigability tells us if an object taking part in association knows about the other.

- unidirectional or bidirectional
- The **arrowhead** (not the entire arrow) denotes the navigability. The line denotes the association
- is an **extra annotation** added to an association line

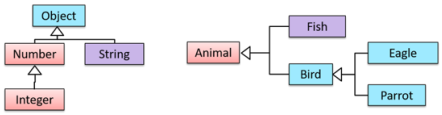
Multiplicity: is the aspect of an OOP solution that dictates how many objects take part in each association

- A normal instance-level variable gives us a 0..1 multiplicity (also called optional associations)
- A variable can be used to implement a 1 multiplicity too (also called compulsory associations).

Inheritance allows you to define a new class based on an existing class.

- A superclass is said to be more general than the subclass.
- implies the derived class can be considered as a subtype of the base class, resulting in an **is a** relationship.
- Inheritance relationships through a chain of classes can

result in **inheritance hierarchies** (aka **inheritance trees**).



- UML notes can augment UML diagrams with additional information. These notes can be shown connected to a particular element in the diagram or can be shown without a connection.
- **Multiple Inheritance** is when a class inherits directly from multiple classes. Multiple inheritance among classes is allowed in some languages (e.g., Python, C++) but **not in other languages (e.g., Java, C#)**.

Method overriding

- same name
- same type signature
- same (or a subtype of the) return type
- **overridden methods** are resolved using **dynamic binding**, and therefore resolves to the implementation in the actual type of the object.

Method overloading: indicate that multiple operations do similar things but take different parameters.

- same method name
- different method signatures
- possibly different return types.
- **overloaded methods** are resolved using **static binding**

Type signature: type sequence of the parameters. The return type and parameter names are **not** part of the type signature. However, the parameter **order is significant**.

Substitutability: Ability to substitute a **child class object** where a parent class object is expected.

- instance of a subclass is an instance of the superclass
- inheritance allows substitutability

Dynamic binding (Late binding): method calls in code are resolved at **runtime**, rather than at compile time.
Static binding (Early binding): method call is resolved at **compile time.**

Polymorphism: allows you to write code targeting superclass objects, use that code on subclass objects, and achieve possibly different results based on the actual class of the object. **Achieve polymorphism:**

- substitutability
- operation overriding
- dynamic binding

Composition is an association that represents a strong whole-part relationship.

- When the whole is destroyed, parts are destroyed too
- **Cannot be cyclical links**.
- Whether a relationship is a composition can depend on the context. In other words, two objects may have different relationship in different context.
- A common use of composition is when parts of a big class are carved out as smaller classes
- Cascading deletion alone is **not sufficient** for composition.
- Identifying and keeping track of composition relationships in the design has benefits
- Composition is **implemented** using a normal variable.

Aggregation: a **container-contained** relationship.
(a weaker relationship than composition)

- Containee object can exist even after the container object is deleted.
- Martin Fowler's famous book UML Distilled **advocates omitting** the aggregation symbol altogether because using it adds more **confusion** than clarity.

Dependency is a need for one class to depend on another without having a direct association in the same direction.

- We are specifically focusing on non-obvious dependencies here
- An association is a relationship resulting from one object keeping a reference to another object.
- we need not show that as a dependency arrow in the class diagram if the association is already indicated in the diagram. [does not add any value to the diagram]
- Use a dependency arrow to indicate a dependency only if that dependency is not already captured by the diagram in another way

Abstract Class: You can declare a class as abstract when a class is merely a representation of commonalities among its subclasses.

- A class that has an abstract method becomes an abstract class

MISC

What is the difference between a Class, an Abstract Class, and an Interface?

Interface: behavior specification with no implementation.
Class: behavior specification + implementation.
Abstract Class: behavior specification + a possibly incomplete implementation.

Testing and Test Case Design

Quality Assurance (QA)

- the process of ensuring that the software being built has the required levels of quality.
- = Validation + Verification

Validation: are you building the right system [requirements correct]
Verification: are you building the system right [implemented correctly]
E.g., ~~It is very important to clearly distinguish between validation and verification.~~ [Whether something belongs under validation or verification is not that important. **What is more important is that we do both.**]

Code review: Systematic examination of code with the intention of finding where the code can be improved.

- **Pull Request reviews** [GitHub | BitBucket]
- **In pair programming** implicit review of the code by the other member when working on the same code.
- **Formal inspections:** Members of the inspection team play various roles
  - **author** who create the artifact
  - **moderator**: planner and executor of meeting
  - **secretary**: recorder of the findings of the inspection
  - **inspector/reviewer** inspects/reviews the artifact

Static analysis: analysis of code without actually executing the code.

- can find useful information
  [unused variables
  unhandled exceptions
  style errors
  statistics]
- Most modern IDEs has inbuilt static analysis capabilities
- Higher-end static analyzers can perform more complex analysis such as locating potential bugs, memory leaks, inefficient code structures, etc.
  [CheckStyle, PMD, FindBugs]

Linters: a subset of static analyzers aiming to **locate areas** where the code can be made **'cleaner'**.

Dynamic Analysis: requires the code to be executed to gather additional information about the code

- Can used for measure **test converge.**

Formal verification

- uses **mathematical techniques to prove** the **correctness** of a program.
- can be used to **prove the absence of errors**
- more commonly used in **safety-critical software** such as flight control systems.

Disadvantages

- Only proves the compliance with the specification, **but not the actual utility** of the software.
- It requires highly specialized notations and knowledge [expensive technique to administer]

Testing can only prove the **presence** of errors

SUT: software under test
Testability: How easy it is to test an SUT. The higher the testability, the easier it is to achieve better quality software.
Regression: When modify a system, may result in some unintended and undesirable effects on the system.
Regression testing: Re-testing of the software to detect regressions. Can not be automated but automation is highly recommended.

Developer testing: Testing done by developers themselves
Pros:

- 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

Cons:

- **Only test situations that he knows** to work (i.e. test it too 'gently').
- May be 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**).
- **Misunderstood** what the SUT is supposed to do in the first place.
- A developer may **lack the testing expertise**.

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

- Requires the unit to be tested in isolation → bugs in the dependencies cannot influence the test

Stub:

- has the same interface as the component it replaces
- implementation is so simple [unlikely to have bugs]
- mimics the responses of the component, but only for a limited set of predetermined inputs.
- does not know how to respond to any other inputs
- mimicked responses are hard-coded rather than computed or retrieved from elsewhere [database]

Why

- Stubs can isolate the SUT from its dependencies.
- Use a hybrid of unit+integration tests to **minimize** the need for stubs.

Integration testing: testing whether different parts of the software **work together** (i.e. integrates) as expected.

- **Not simply a case of repeating** the unit test cases
- using the actual dependencies (instead of stubs)
- Additional test cases focus on the interactions between the parts.

System testing: take the **whole system** and test it against the system specification.

- done by a testing team (also called a QA team).
- based on the specified **external behavior** of the system
- Sometimes, system tests **go beyond the bounds** defined in the specification. This is **useful** when testing that the system fails 'gracefully' when pushed beyond its limits.
- includes testing against **NFRs** too.

Alpha testing is performed by the **users**, under controlled conditions set by the software development team.

Beta testing is performed by a **selected subset of target users** of the system in their natural work setting.

Open beta release: Release of not-yet-production-quality-but-almost-there software to the general population.

Dogfooding: When creators use their own product

Scripted testing: First write a set of test cases based on the expected behavior of the SUT, and then perform testing based on that set of test cases.

- More systematic, and hence, likely to discover more bugs given sufficient time

Exploratory testing: Devise test cases on-the-fly, creating new test cases based on the results of the past test cases.

- Simultaneous learning, test design, and test execution
- Also known as reactive testing, error guessing technique, attack-based testing, and bug hunting.
- Success depends on tester's prior experience and intuition.
- may allow us to detect some problems in a **short time**
- it is **not prudent to use exploratory testing as the sole means of testing a critical system.** [Use a mixture of Scripted Testing and Exploratory Testing]

Acceptance testing (aka User Acceptance Testing (**UAT**): test the system to **ensure** it **meets the user requirements**.)

- Involves testing the whole system
- **comes after system 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 **smaller** projects, the developers may do system testing as well, in addition to developer testing.
- System testing is **more extensive** than accept. Testing

Test Driver: code that 'drives' the SUT for the purpose of testing i.e. invoking the SUT with test inputs and verifying if the behavior is as expected.

- JUnit is a tool for automated testing of Java programs.

GUI Testing [TestFX | Visual Studio |Selenium (For Web)]

- 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.

Test coverage: Metric used to measure the extent to which testing exercises the code.

- **Function/method coverage:** base on function executed
- **Statement coverage:** based on the # of lines of code
- **Decision/branch coverage :** based on the decision points exercised
- **Condition coverage:** based on the **boolean sub-expressions**, each evaluated to **both true and false (Need to cover 2 conditions)** with different test cases. Condition coverage is not the same as the decision coverage.

- **Path coverage** measures coverage in terms of possible paths through a given part of the code executed. 100% means all possible paths have been executed. A commonly used notation for path analysis is called the **Control Flow Graph (CFG)**.
  Eg. enter -> 2 -> 3 -> 2 -> 3 -> exit
- **Entry/exit coverage:** measures coverage in terms of possible calls to and exits from the operations in the SUT.

Dependency injection: Process of 'injecting' objects to **replace current dependencies** with a different object. This is often used to **inject stubs to isolate the SUT** from its dependencies so that it can be tested in isolation.
Test-Driven Development(TDD) advocates writing the tests before writing the SUT, while evolving functionality and tests in small increments.

Test Cases Design

- Except for trivial SUTs, exhaustive testing is not practical
- Every test case adds to the cost of testing.
- Testing should be **effective** i.e., it finds a high percentage of existing bugs
  [Determine by absolute # of bugs detected]
- Testing should be **efficient** i.e., it has a high rate of success (bugs found/test cases)
  [Determine by **#bugs / #test cases**]
- For testing to be E&E, each new test you add should be targeting a potential fault that is not already targeted by existing test cases.

Positive test case: designed to produce an expected/valid behavior. E.g., Integer i == new Integer(50)

Negative test case: designed to produce a behavior that indicates an invalid/unexpected situation, such as an error message. E.g., Integer i == null

T**hree types**: based on how much of the **SUT's internal details** are considered when designing test cases:

- **Black-box (specification-based or responsibility-based) approach:** test cases are designed **exclusively** based on the SUT's specified external behavior.
- **White-box (glass-box or structured or implementation-based) approach:** test cases are designed **based on** what is known about the SUT's implementation.
- **Gray-box approach:** test case design uses some **important information** about the implementation.

Equivalence partition: A group of test inputs that are likely to be processed by the SUT in the same way.

- An EP may not have adjacent values.

By dividing possible inputs into **EPs** you can,

- avoid testing too many inputs from one partition **increases the efficiency** by reducing redundant cases.
- ensure all partitions are tested.
  increases the effectiveness by ↑ chance of finding bugs.

Example

> Consider a Java method `isPrime(int i)` that returns true if i is a prime number.
>
> 'All non-int values' is a possible EP for testing this method.

False. As Java is strongly-typed, it is not even possible to use non-int values to test the method.

Boundary Value Analysis (BVA): test case design heuristic that is based on the observation that bugs often result from incorrect handling of boundaries of equivalence partitions.

- values near boundaries (i.e. boundary values) are **more likely** to find bugs, but not just test boundary values!
- Boundary values are sometimes called corner cases.
- Choose 3 values around the boundary to test: **boundary value, just below and just above** the boundary.

Combining test inputs: Testing all possible combinations is **effective** but **not efficient**. Here lists 4 types:

- **All combinations strategy** generates test cases for each unique combination of test inputs.
- **At least once strategy** includes each test input at least once.
- **All pairs strategy** creates test cases so that for any given pair of inputs, all combinations between them are tested.
  Variation: test all pairs of inputs but only for inputs that could influence each other.
- **Random strategy** generates test cases using one of the other strategies and then picks a subset randomly (presumably because original set of test cases is too big).

Heuristic:
1. Each Valid Input at Least Once in a Positive Test Case
2. 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.]

- This is not to say never have more than one invalid input in a test case.
- If you can afford more test cases, also testing with combinations of invalid inputs.

Model and Diagrams 1

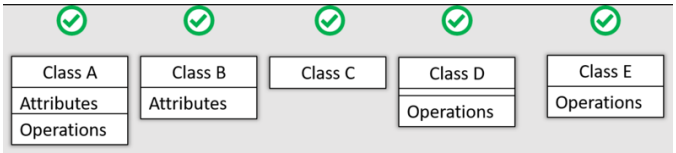
Model

- provides a simpler view of a complex entity
- captures only a selected aspect
- are **abstractions**.
- Multiple models of the same entity may be needed to capture it fully.
-

Usage

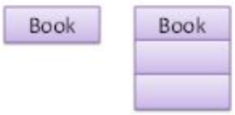
- To analyze a complex entity in software development.
- To communicate information among stakeholders.
  [As a visual aid in discussions and documentation]
- As a blueprint for creating software

Class Diagram: Describe the **structure** (**not behavior**)



Example: These two are not same

- the first one omits the attributes
- the second one has empty attributes and operations



Visibility: (Not Accessibility)

- no default visibility in UML.
- not show the visibility means unspecified

Model-driven development (MDD):

- Model-driven engineering
- an approach to software development that strives to exploit models as blueprints

Unified Modeling Language (UML): is a graphical notation to describe various aspects of a software system.

Object structures

- can change over time based on a set of rules **set by the designer** of that software. [Not Random]
- Rules that object structures need to follow can be illustrated as a **class structure**

Domain modeling is modeling the problem domain. Useful in understanding the problem domain. Can be done using,

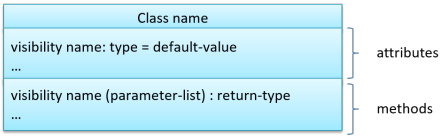
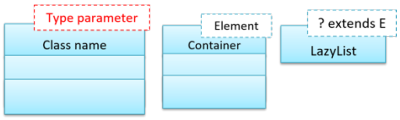
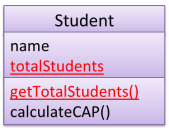
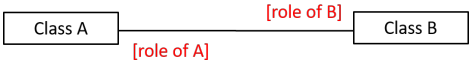
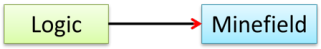
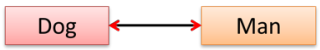
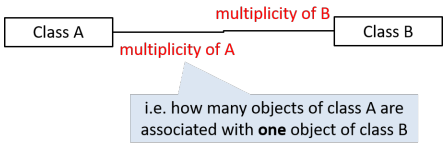
- **a domain-specific modeling notation** if such a notation exists (e.g., a modeling notation specific to the banking domain might have elements to represent loans, accounts, transactions etc.)
- **a general purpose modeling notation**, such as **UML**
- other **general purpose notations** (e.g., **organization chart** to model the employee hierarchy of a company).

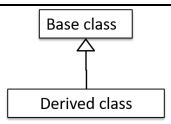
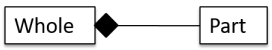
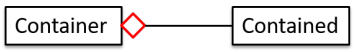
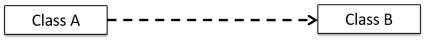
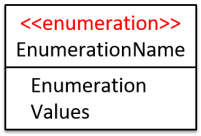
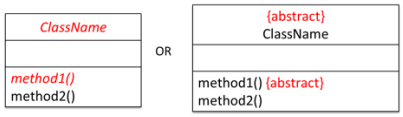
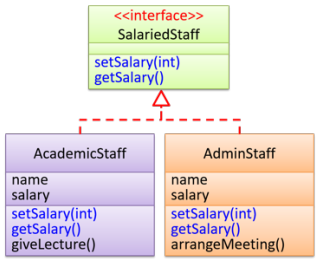
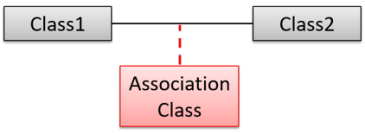
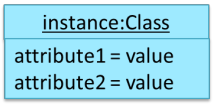
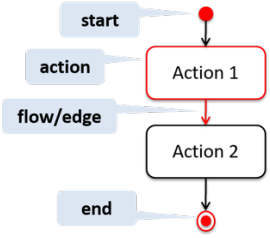
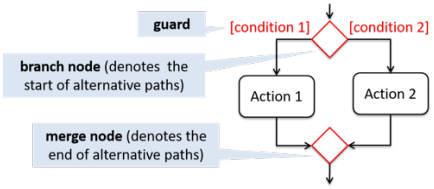
Conceptual Class Diagrams(CCDs): UML model captures class structures in the problem domain.

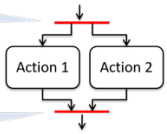
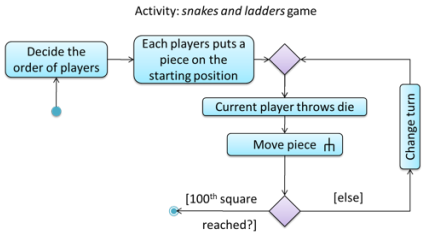
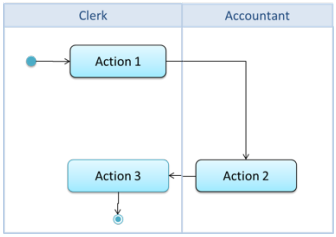
- a **lighter version** of class diagrams
- sometimes also called **OO domain models** (OODMs) [somewhat misleading]
  CCDs are only **one type of domain models** that can model an OOP problem domain.

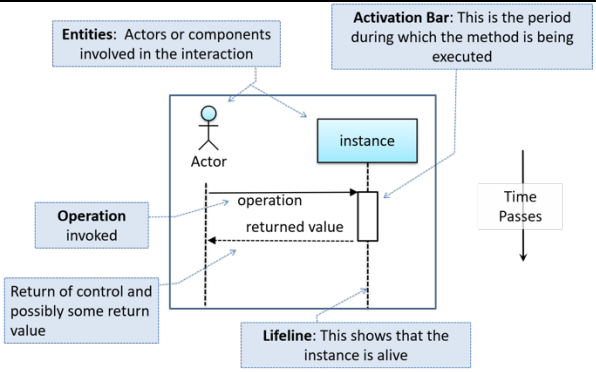
A UML **sequence diagram** captures the **interactions** between multiple entities for a given scenario.

UML Diagram Notation

	Example
Class	
visibility	+ : public - : private # : protected ~ : package private
Generic	
static (Class Level)	Use an underline to indicate static 
Association	Use a solid line to show an association 
Association labels [Optional]	Describe the meaning of the association [Arrow head indicates the direction in which the label is to be read] 
Association Role [Optional]	
Navigability	Logic has ref. to Minefield  Dog and Man have ref. to each other 
Multiplicity	 Commonly used multiplicities: • 0..1 : **optional**, can be linked to 0 or 1 objects. • 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 e.g., 2..5, 1..* (one or more), *..5 (up to five)
Inheritance	Does not matter triangle is filled or empty.

	
Composition	Use a solid diamond symbol 
Aggregation	Uses a hollow diamond 
dependencies	
Enumeration	
Abstract	*italics* or {abstract} (preferred) 
Interface and Implement	
Association class	
Object	 • class name and object name are underlined • no compartment for methods • Attributes compartment can be omitted • Object name can be omitted
Activity Diagram	
Alternate Paths [AD]	

	Some acceptable simplifications • Omitting the merge node if it doesn't cause any ambiguities. • Multiple arrows can start from the same corner of a branch node. • Omitting the [Else] condition.
Parallel Paths [AD]	**Fork** (denotes the start of parallel paths - many outgoing edges) **Join** (denotes the end of parallel paths – many incoming edges) 
rake notation [AD]	Indicate a part of activity is describe separately. 
swimlane diagrams [AD]	Show who is doing which action 

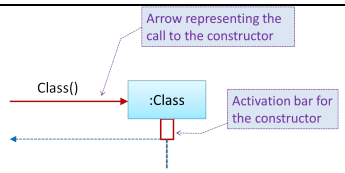
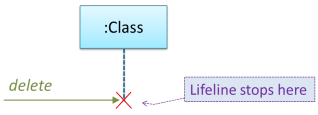
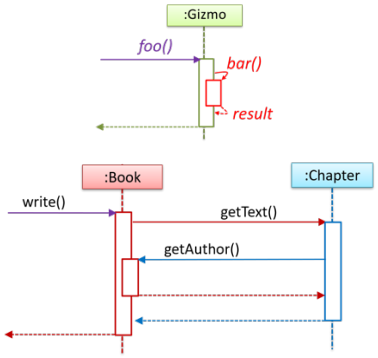
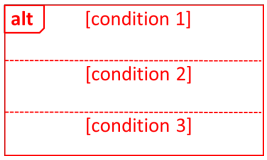
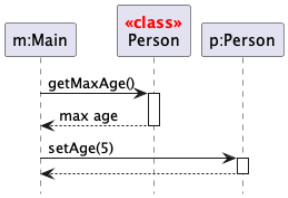
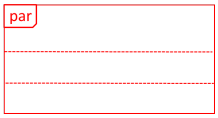


- Arrows representing method calls should be solid arrows while those representing method returns should be dashed arrows.
- Class/object name is not underlined in sequence diagrams.
- Arrows showing **synchronous** (i.e., the caller method is blocked from doing anything else until the called method returns) should use filled arrowheads (e.g., →).
- **Asynchronous** method calls (i.e., the caller method does not have to wait till the called method returns) are shown using lined arrowheads (e.g., →).

Optional elements

- activation bars
- return arrows
- method parameters use [foo(...) instead of foo(int size)]

Loops	loop [condition]

Object Creation	 • The arrow that represents the constructor arrives at the side of the box representing the instance. • The activation bar represents the period the constructor is active.
Object Deletion	Uses an X at the end of lifeline of an object to • show its deletion • indicate the point at which the object becomes ready to be **garbage-collected** 
Self-invocation	
alternative paths	 • No more than one altern. be executed • Acceptable for none to be executed
optional paths.	
static in [SD]	
parallel paths	
Reference Frames	Allow a segment of the interaction to be omitted and shown as a separate sequence diagram. - break complicated diagrams into multiple parts - omit details not interested in showing ref reference frame name sd reference frame name

CS2103T Code Standard

Name

- **Names** representing packages should be in all **lower case**.
- **Class/enum** names must be nouns and written in **PascalCase**.
- **Variable** names must be in **camelCase**.
- **Constant** names must be all **uppercase** using underscore to separate words (aka SCREAMING_SNAKE_CASE).
- Names representing **methods** must be verbs and written in **camelCas**e.
- **Abbreviations and acronyms** should **not be uppercase** when used as a part of a name.

Good

```
exportHtmlSource();
openDvdPlayer();
```

Bad

```
exportHTMLSource();
openDVDPlayer();
```

- Variables with **a large scope** should have **long names**, variables with **a small scope** can have **short names**.
- Boolean variables/methods should be named to sound like Booleans (use a prefix such as is, has, was, etc.)
- Plural form should be used on **names** representing a collection of objects.
- Iterator variables can be called i, j, k etc.
- Associated constants should have a common prefix.

Layout

- Basic indentation should be **4 spaces (not tabs)**
- Line length should be **no longer than 120 chars (hard limit)**. Try to keep line length shorter than **110 chars (soft limit)**.
- Indentation for wrapped lines should be **8 spaces [NOT 2 TABS]** (i.e. twice the normal indentation of 4 spaces) more than the parent line.
- Place line break to **improve readability**
  - Break after a comma.
  - Break before an operator.

```
totalSum = a + b + c
        + d + e;
setText("Long line split"
        + "into two parts.");
method(param1,
        object.method()
                .method2(),
        param3);
```

  - A method or constructor name **stays attached to the open parenthesis (** that follows it.
  - Prefer higher-level breaks to lower-level breaks. In the example below, the first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

Good

```
someMethodWithVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongName(
        int anArg, Object anotherArg);
```

Bad

```
someMethodWithVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongName
        (int anArg, Object anotherArg);
```

  - Prefer higher-level breaks to lower-level breaks.

Good

```
longName1 = longName2 * (longName3 + longName4 - longName5)
        + 4 * longname6
```

Bad

```
longName1 = longName2 * (longName3 + longName4
        - longName5) + 4 * longname6;
```

- Use **K&R style** brackets (aka [Egyptian style](#)).

Good

```
while (!done) {
    doSomething();
    done = moreToDo();
}
```

Bad

```
while (!done)
{
    doSomething();
    done = moreToDo();
}
```

- The while and the do-while statements should have the following form

```
while (condition) {
    statements;
}
```

```
do {
    statements;
} while (condition);
```

- The switch statement: No indentation for case clauses.
- The explicit //Fallthrough comment should be included whenever there is a case statement **without a break** statement.

```
switch (condition) {
case ABC:
    statements;
    // Fallthrough
```

- A try-catch statement should have the following form:

```
try {
    statements;
} catch (Exception exception) {
    statements;
} finally {
    statements;
}
```

- **White Space**
  - Operators should be surrounded by a space character.
  - Java reserved words should be followed by a white space.
  - Commas should be followed by a white space.
  - Colons should be surrounded by white space when used as a binary/ternary operator.
  - Does not apply to switch x:. Semicolons in for statements should be followed by a space character.

Good	Bad
`a = (b + c) * d;`	`a=(b+c)*d;`
`while (true) {`	`while(true){`
`doSomething(a, b, c, d);`	`doSomething(a,b,c,d);`
`for (i = 0; i < 10; i++) {`	`for(i=0;i<10;i++){`

- Logical units within a block should be separated by one blank line.

```
// Create a new identity matrix
Matrix4x4 matrix = new Matrix4x4();

// Precompute angles for efficiency
double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

// Specify matrix as a rotation transformation
matrix.setElement(1, 1,  cosAngle);
matrix.setElement(1, 2,  sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2,  cosAngle);

// Apply rotation
transformation.multiply(matrix);
```

Package and Import Statements

- Put every class in a package. Every class should be part of some package.
- The **ordering** of import statements must be consistent.
- Imported classes should always be **listed explicitly.**

Good

```
import java.util.List;
import java.util.ArrayList;
import java.util.HashSet;
```

Bad

```
import java.util.*;
```

Types: Array specifiers must be attached to the type not the variable.

Good

```
int[] a = new int[20];
```

Bad

```
int a[] = new int[20];
```

Variables

- Variables should be initialized where they are declared and they should be declared in the smallest scope possible.

Good

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        sum += i * j;
    }
}
```

Bad

```
int i, j, sum;
sum = 0;
for (i = 0; i < 10; i++) {
    for (j = 0; j < 10; j++) {
        sum += i * j;
    }
}
```

- Class variables should **never be declared public** unless the class is a data class with no behavior. This rule does not apply to constants.

Bad

```
public class Foo{

   public int bar;

}
```

Loops: The loop body should be **wrapped by curly brackets** irrespective of how many lines there are in the body.

Conditionals: The conditional should be put on a separate line.

Comments

- **MUST** write descriptive header comments for **all** public classes / methods. But they can be omitted for the following cases:
  - Getters/setters
  - When overriding methods (**provided the parent method's Javadoc** applies exactly as is to the overridden method)

```
/**
 * 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 {
    //...
}
```

- The opening /** on a separate line.
- Write the first sentence as a short summary [Javadoc automatically places it in the method summary table (and index)]
- In method header comments, the first sentence should start in the form **Returns ..., Sends ..., Adds** ... etc. (**not Return** or Returning etc.)
- Subsequent * is aligned with the first one.
- **Space** after each *.
- **Empty line** between description and parameter section.
- **Punctuation** behind each parameter description.
- **No blank line** between the documentation block and the method/class.
- **@return can be omitted** if the method does not return anything or the return value is obvious from the rest of the comment.
- **@params can be omitted** if all parameters of a method have **self-explanatory names** or already explained in the main part of the comment.
- When writing Javadocs for overridden methods, the **@inheritDoc tag** can be used to reuse the header comment from the parent method but with further modifications.

Code quality

Among various dimensions of code quality, such as run-time efficiency, security, and robustness, one of the most important is **readability** (aka understandability).

- **Avoid long methods** [the bigger the haystack, the harder it is to find a needle]
- **Avoid deep nesting** (**Arrowhead style code**) [the deeper the nesting, the harder it is for the reader to keep track of the logic]

Avoid complicated expressions

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

Bad

```
return 3.14236;
...
return 9;
```

Good

```
static final double PI = 3.14236;
static final int MAX_SIZE = 10;
...
return PI;
...
return MAX_SIZE - 1;
```

Bad

```
return "Error 1432"; // A magic string!
```

- Make the code **as explicit as possible**, even if the language syntax allows them to be implicit.
- Use **enumerations** when a **certain variable** can take only a small number of finite values
- Lay out the code to **adheres to the logical structure**.

Bad

```
statement A1
statement A2
statement A3
statement B1
statement C1
statement B2
statement C2
```

Good

```
statement A1
statement A2
statement A3

statement B1
statement B2

statement C1
statement C2
```

Avoid things that would make the reader go 'huh?'

- unused parameters in the method signature
- similar things that look different
- different things that look similar
- multiple statements in the same line
- data flow anomalies such as, pre-assigning values to variables and modifying it without any use of the pre-assigned value

KISS

- Do not try to write 'clever' code. "Keep it simple, stupid"
- Choose 'clever' sol. only if additional cost of is justifiable.

Avoid Premature Optimizations

- You may not know which parts are the real performance bottlenecks.
- Optimizing can complicate the code
- Hand-optimized code can be harder for the compiler to optimize
- there are also cases in which optimizing takes priority over other things

Single Level of Abstraction Principle (SLAP)

- Avoid having multiple levels of abstraction within a code fragment.

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

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

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

```
readData();
processData();
displayResult();
```

- Ensure that the code is written at the **highest level of abstraction** possible.

Bad (all statements are at a low levels of abstraction)

```
low-level statement A1
low-level statement A2
low-level statement A3
low-level statement B1
low-level statement B2
if condition X :
    low-level statement C1
    low-level statement C2
```

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

```
high-level step A
high-level step B
if condition X:
  high-level step C
```

- Sometimes **possible to pack two levels of abstraction** into the code

Happy path: execution path taken when everything goes well

- should be clear and prominent in your code
- **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.

Bad

```
if (!isUnusualCase) {  //detecting
    if (!isErrorCase) {
        start();    //main path
        process();
        cleanup();
        exit();
    } else {
        handleError();
    }
} else {
    handleUnusualCase(); //handling
}
```

Good

```
if (isUnusualCase) { //Guard Clause
    handleUnusualCase();
    return;
}

if (isErrorCase) { //Guard Clause
    handleError();
    return;
}

start();
process();
cleanup();
exit();
```

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
```

Naming Well

- nouns for classes/variables
- verbs for methods/functions.

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

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

Good

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

- **Avoid foreign language words, slang, and names** that are only meaningful within specific contexts/times
- multiple words 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`

- Avoid Misleading Name

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`

Avoid Unsafe ShortCuts

- Always **include a default branch** in case statements. [all possible outcomes have been considered at the branching point]
- use the **default branch for the intended default action** and not just to execute the last option
- no default action, you can use the **default branch 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");
```

Variables

- Use **one variable for one purpose**
- Do **not reuse formal parameters** as local variables

Bad

```
double computeRectangleArea(double length, double
    length = length * width;  // parameter reused
    return length;
}
```

Good

```
double computeRectangleArea(double length, double
    double area;
    area = length * width;
    return area;
}
```

- **Avoid empty catch statements: at least give a comment** to explain why the catch block is left empty.

Code Duplication and Scope

- Get rid of unused code the moment it becomes redundant.
- **Code duplication**, especially when you copy-paste-modify code, often indicates a **poor quality**
  [guideline is closely related to the **DRY Principle**]
- **Minimize global variables**.
- Define variables in the **least possible scope.**

DRY (Don't Repeat Yourself) Principle: Every piece of knowledge must have a **single, unambiguous, authoritative** representation within a system

Comment

- Do not repeat in comments information that is already obvious from the code.
- Comments should **explain the WHAT and WHY** aspects of the code, rather than the HOW aspect.

WHAT: The specification of what the code is supposed to do.
WHY: The rationale for the current implementation.
HOW: The explanation for how the code works.

Git convention

- Every commit must have a well-written commit message subject line.
- Try to limit the subject line to 50 characters (**hard limit: 72 chars**)
- Use the **imperative** mood in the subject line.
  ["Add" instead of "Added" or "Adds"]
- **Capitalize** the first letter of the subject line.
- **Do not end t**he subject line **with a period.**
- You can use `scope: change` format or `category: change`
- Commit messages for **non-trivial commits** should **have a body** giving details of the commit.
- Separate subject from body with a blank line.
- **Wrap the body at 72 characters**.
- Use blank lines to separate paragraphs.
- Use bullet points as necessary.
- **Explain WHAT, WHY, not HOW.**
- If your description starts to **get too long**, that's a sign that you probably **need to split up your commit** to finer grained pieces.