# Chapter 1: intro.

## Software engineering.

* The process of taking an idea and turning it into a program.
* OR an organized, analytical approach to the design, development, use, and maintenance of software.

### Advantages of following SWE techniques:

* Programmers understand what the user needs.
* Programmers understand what the user wants.
* The app has less bugs.
* Apps can be enhanced over time.
* The app is effective, usable, and maintainable.
* Helps in scheduling & producing the product.
* Gives more flexibility.

### SWE steps:

1. Requirement gathering.

* Finding out what the customer wants & needs.

1. High level design (HLD).

Includes decisions about:

* What platform to use.
* What data design to use.
* How it interfaces with other systems.

Project should be broken into:

* Database.
* Classes.
* User interface.
* External interfaces.

1. Low level design (LLD).

It includes:

* How pieces of the project should work.
* doesn’t need to include implementation details.

1. Development.

* Programmers refine LLDs until they know how to implement them.
* They should also work on the previous steps (req gathering, HLD, LLD).
* They should test code as they write it.

1. Testing.

* Pieces might not work properly together.
* Programmers should test their own code.
* Testers should write code tests.
* Pieces that pass those tests can be integrated into the project.
* Testing should be done after each step of the development.
  + Because ” The longer a bug stays undetected, the harder it’s fixed”.

Testing techniques:

* Exhaustive testing.
  + Testing with every possible input.
* Black-box testing.
  + You know what a method does, but not how.
* White-box testing.
  + You know how a method works and you pick difficult test cases.
  + Some people skip some important test cases.
* Gray-box testing.
  + Combination of both (black & white).

1. Deployment.

Difficult, time-consuming, and expensive.

May involve:

* New computers for backend database.
* New network.
* New computers for the users.
* User training.
* Parallel use (using new & old system).
* Special data maintenance.
  + To synchronize between old & new database.
* Massive bug fixing.

1. Maintenance.

* Fixing user discovered bugs.
* Adding user suggestions (enhancements).

1. Wrap-up.

* Evaluate the project.
  + Figure what went wrong & right.
  + Figure how to prevent the things that went wrong.
  + Figure how to increase the occurrence of the things that went right.

1. Everything all at once.

* Working on as much of the project as use can at any given moment.

## Project Documentation.

* There is a lot of different documentations in software projects.
* Some projects allow requirement changes.
  + This makes finding recent versions harder.
* Simultaneous editing may occur (race condition problem).
  + The last edit request will overwrite the other requests.
* To solve these problems, you need a good documentation system.
  + They must support one of the following:
    - Allow document sharing.
    - Allow only one user to edit at a time.
    - Allow to fetch most recent version.
    - Allow to fetch a specific version.
    - Allow to search documents (by tags, keywords, etc.).
    - Allow to compare two versions.

### E-mails:

* Create an email for your project.
* Every message should be copied to the email.
* Prefix your email about your project to make it easier to find.
* Add different classes to the email subject.

## Project management.

Ensures the goals are set, tracked, and met.

### Project management roles:

1. Executive support || sponsor (champion).

The Lack of a champion is one of the top reasons why software projects fail.

* Provides necessary resources (budgets, hardware, personnel).
* Makes “go/no-go” decisions.
* Decides when to cancel the project.
* Helps in navigating administrative obstacles.
* Defines the business case (evaluates benefits, cost, risk, etc.).
* Works with users & other stakeholders to persuade them to buy-in.
* Isolating the project from external distractions.
* Provides feedback to devs about finished features.
* Chooses between managers, users, devs, and others interested in the project.
* Gives guidance on high-level issues.

1. Project manager.

* The highest-ranking member of the project team.
* Works on all stages of development.
* Doesn’t need to be an expert in the users’ field || in programming.
* Their duties:
  + Helps define the project requirements.
  + Tracks/monitors project tasks.
  + Keeps users & champion up to date on the progress.
  + Provides an interface between customers & devs.
  + Manages resources (time, people, budget, hardware, software).
  + Manages delivery of services/products.
  + Responds to unexpected problems.
  + Manages risk.

### Project management tools:

These tools help study the project’s duration, look for potential bottlenecks, and schedule tasks.

1. PERT Charts.

* Stands for (Program Evaluation and Review Technique).
* A graph that uses nodes & links to show priority relationships among the project tasks.
* Types of PERT charts:
  + Activity on arrow (AOA): arrows represent tasks and nodes represent milestones.
  + Activity on node (AON): nodes represent tasks and arrows represent priority relationships (easier to build and understand).

AON steps:

1. List the tasks, predecessor tasks, and expected time for each task.
   1. You may add best & worst cases.
   2. Doesn’t need to include every predecessor combination.
   3. Unnecessary relationships don’t have to be removed.
2. Make rearranging tasks easy.
   1. Make an index card for each task (name, predecessor, expected time).
3. Place the start task in the Ready pile.
4. Place tasks in the Ready pile right to previous task.
5. Look through the tasks in the Pending pile and cross out the predecessors you added to the Ready pile.
6. Cards that have all their predecessors crossed out gets moved to the Ready pile.
7. Repeat steps 4-6 until you place the final task.
8. Draw arrows representing the predecessor relationships.

* You can add the start & finish tasks as a predecessor for any task (that doesn’t have predecessors).

Table

Description automatically generated

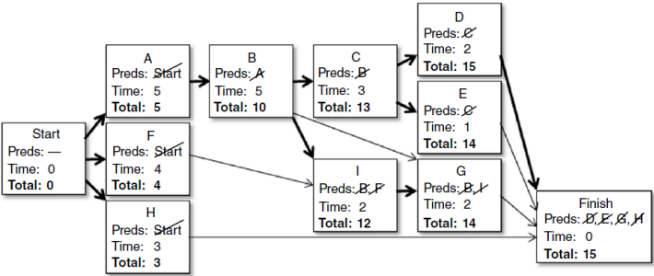
Text

Description automatically generated with medium confidence

Diagram

Description automatically generatedDiagram

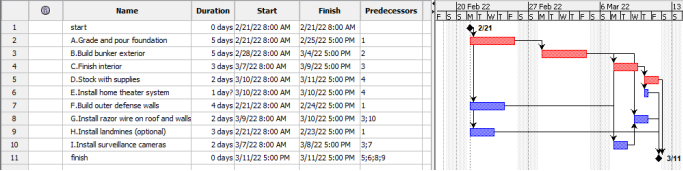
Description automatically generated with low confidence

Critical path methods.

PERT charts are used to find the critical paths through the network.

Critical path: the longest possible paths through the network.

* moving through every node and finding the longest path.
* There can be multiple paths with the same length.



1. Gantt Charts.

* a horizontal bar chart.
  + Bars represent task activities.
  + Bars’ lengths indicate tasks duration.
  + Bars are placed on a horizontally on a calendar to show their start & stop times.
* Shows a schedule for a collection of related tasks.
* Arrows show the relationships between tasks & their predecessors.
* Weekends extend the project’s total duration.

Steps:

1. List the tasks, their duration, and predecessors (on the left).
2. Cut out a thin rectangle for each task.
   1. rectangle width represents the duration.
   2. Write the tasks’ names on their rectangles.
3. Move from left to right placing each rectangle.
   1. Make sure the left edge lines up with the right edge of its predecessor.
4. Add arrows to show the predecessor relationships.
5. Scheduling software.

* Some tasks take longer than expected and re-drawing the whole schedule each time would be problematic.
  + Project scheduling tools can re-draw schedules.
  + These tools enable you to click & drag to connect two tasks.
  + You can change tasks’ duration.
  + Enable you to define other kinds of relationships.

1. Predicting times.

* Other methods depend on time estimated being accurate.
* Hardest part is predicting how long each task takes.
* Tasks are either:
  + Short & well known.
  + Long & highly uncertain.
* Steps to Minimize risk:
  + Get experience (find someone with experience).
  + Break tasks into simpler pieces (smaller = more predictable).
  + Look for similarities (between unknown & already performed tasks).
  + Expect the unexpected (expand each task estimate by an amount).
  + Track progress.
    - Act if a task isn’t going as planned.
    - Don’t ignore problems & don’t hope to make up for lost time.
    - Don’t pile extra devs on a delayed task (it will be delayed more).

### Risk management.

* Identifying possible risks, determining their potential impact & finding possible solutions.
* You should determine (for each task):
  + Likelihood.
    - How much do you know about this task.
  + Importance.
    - Can users live without this feature.
    - Can you cancel this feature or add it later (update).
  + Consequences.
    - Will problems with this task affect other tasks.
  + Solutions.

For each solution consider:

* + - Difficulty.
    - Impact (does it affect usability || extra work for the users).
    - Pros (advantages).
    - Cons (disadvantages).

# Chapter 2: requirement gathering.

Requirements: features that your app must provide.

* Used to verify that the finished app does what it’s supposed to.

## Requirement properties.

* Clear.
  + Needs to be clear, brief, easy to understand.
  + Common abbreviations & terms are okay.
* Unambiguous.
  + Avoid requirements that have more than one meaning.
* Consistent.
  + Requirements can’t contradict each other.
  + Requirement shouldn’t provide a lot of constraints.
  + “Fast, good, cheap. Pick two”.
    - Quick app with high quality & high cost (Fast & good & !cheap).
    - Quick app with low quality & low cost (Fast & !good & cheap).
    - High quality & low cost but takes long time ( !Fast & good & cheap).
  + Rewrite older requirements if needed.
* Prioritized.
  + Prioritize requirements with high cost as low priority.
  + “Must” & “Should” features should be assigned costs.

MOSCOW method:

* + Required features (M-Must).
    - Could you remove the requirement?
    - Will the app be usable without that feature?
    - Will the users give bad reviews (0 || 1 stars)?
  + Important features (S-Should).
    - Does the feature enhance the product?
    - If it was removed, will users ask for it?
    - Is it common in similar apps? Will reviews be bad (2 || 3 stars)?
  + Desirable features (C-Could).
    - Is the feature useful to the users?
    - Is it special (other apps don’t have it)?
    - Will it increase reviews (4 || 5 stars)?
  + Optional features (W-Won’t).
    - Is it unnecessary, confusing, stupid?
    - Is it rarely used?
    - Does it add nothing to the app?
* Verifiable.
  + Requirements must be limited & precisely defined to be verifiable.
  + Requirements can’t be open ended.
  + Words to avoid:
    - Comparatives (faster, better, more, etc.) replaced with quantities.
    - Imprecise adjectives (fast, robust, user-friendly, etc.).
    - Vague commands (minimize, improve, etc.) replaced with numbers.

## Requirement Categories.

### Audience-oriented requirements.

* Focus on audiences & their different POVs.

Audiences:

* Business requirements.
  + Lays the project’s high-level goals.
  + Explains the customers’ hopes.
  + Hopes might be too far-fetched.
* User (stakeholders) requirements.
  + How end users will use the project.
  + Requirements are very detailed.
  + Specify what users want but not how.
  + Sometimes it’s too detailed and it makes the designers have no choice.
* Functional requirements.
  + Detailed statements that include things the user won’t see directly.
* Non-functional requirements.
  + Focuses on the quality of the app’s behavior.
  + OR constraints on how it produces results.
  + Specify:
    - Application’s performance.
    - Reliability.
    - Security characteristics.
* Implementation requirements.
  + Temporary features.
  + Needed to transition to a new system (how to move from one system to another).
  + Hiring new people || training users.

### FURPS.

* Functionality.
  + What the application should do.
  + Describe the system’s general features (What is does, how it interfaces with other systems, Security).
* Usability.
  + What the program should look like.
  + Describe user-oriented features (app’s general appearance, ease of use, navigation, responsiveness).
* Reliability.
  + How reliable the system should be.
  + Indicate when system is available, how often it can fail, how accurate is the system).
* Performance.
  + How efficient the system should be.
  + Describe app speed, memory usage, disk usage, database capacity.
* Supportability.
  + How easy it is to support the app.
  + Include how easy it is to maintain, how easy to test the code, how flexible is the app.

### FURPS+.

An extended version of FURPS.

* Design constraints.
  + Hardware platform.
  + Software platform.
  + Network characteristics.
  + Database.
* Interface requirements.
  + Constraints on the system’s interfaces (with other systems).
  + Identify which other systems will exchange data with our system.
  + Describe (interaction kinds, when interactions occur, data format).
* Physical requirements.
  + Constraints on the hardware & physical devices.

### Common requirements.

* Screens (what type is needed).
* Menus.
* Navigation (how the user navigates through different parts).
* Workflow (how does data move through the system).
* Login (how login info is stored & validated, password rules).
* User types (different privileges).
* Audit tracking & history (does the sys track who changed data).
* Archiving (archiving old data, moving data for analysis).
* Configuration/settings (config screens for admins).

## Gathering requirements.

How to extract the requirements from the customers.

Techniques:

* Listen to customers (& users).
* Use Five Ws & H.
  + Who will use the software.
  + What the customer needs the app to do.
  + When is the app needed.
  + Where is the app used.
  + Why the customers need the app.
  + How does the customer want the app to function/look.
* Study cases.
  + Studying users as they work to learn about their needs & how they currently deal with those needs.
  + Better than interviewing because the users will often leave/forget important details.
* Copy existing systems.
  + Use existing behaviors as requirements for the new system.

## Refining requirements using brainstorming.

* Brainstorm exercises are recommended for converting collected goals into innovative requirements.
* Never judge/critique/review someone’s idea (they may stop participating).
* Steps:
  + Gather as many ideas as possible.
  + Assemble a large list of possible ideas.
  + Examine the list more closely to see which deserves further work.
  + Get a diverse group of participants who have interest in the project (programmers, users, customers, designers, etc.).
  + Get as many viewpoints as possible.
  + Brainstorming becomes less effective if the group is larger than 10 || 12.
* Brainstorming Rules:
  + Focus on quantity.
    - More ideas = higher chance of finding creative solution.
  + Withhold criticism.
    - Criticism can make people stop contributing & bad ideas can lead to better ideas.
  + Encourage unusual ideas.
    - You can tone down unusual ideas but it’s harder to do the opposite.
  + Combine & improve.
    - Form new ideas by combining other ideas || using an idea to modify other ideas.

## Recording requirements.

* Requirements should be written as a sequence of English instructions.

### Unified Modeling Language (UML).

* Uses several kinds of diagrams to represent different pieces of the system.
  + Can represent program items, behaviors, and data flows.
* Specifies how parts of the system should work.

### User Stories.

* A short story explaining how the system will let the user do something.

### Use Cases.

* A description of a series of interactions between actors.
* Actors can be users || application parts.
* Has a larger scope than a user story.
* Follow a template more than user stories.
* Use case template:
  + Title: name of the goal (includes action name + main actor).
  + Main success scenario: a sequence of steps describing the most normal scenario to achieve goal.
  + Extensions: sequence of steps describing other scenarios.

### Prototypes.

* A mockup of some or all of the application.
* Gives the customer a feel of how the app will look like & how it will behave.
* Used to get text descriptions from the user (user stores || use cases).
* Different prototypes:
  + Non-functional prototype.
    - Elements (buttons, menus, etc.) don’t do anything.
  + Functional prototype.
    - Looks & acts close to how the finished app will.
  + Throwaway prototype.
    - Any prototype that starts from scratch.
  + Evolutionary prototype.
    - A prototype that starts with simple code & fake data that then get replaced by quality code & real data.
    - Eventually becomes the finished application.

## Validation & Verification.

Validation: are we doing the right things?

Verification: are we doing the things right?

# Chapter 3: High-Level Design (HLD).

* HLD provides a view of the system (abstract level).
* Divides the system into small pieces (pieces are given to different teams).
* They don’t focus on how things will work.
* Shows how pieces will fit & interact with each other.
* They specify assumptions about the environment (hardware & software).

## What HLD specifies:

### Security:

* HLD should sketch all security needs.
* All forms of security interact with each other.
* Security needs:
  + Operating system security.
    - Login procedure type (password, pin, …).
    - Password expiration policies.
    - Password standards.
  + Application security.
    - Users should have the right access level.
    - Some apps rely on the OS’s security.
  + Data security.
    - Make sure data doesn’t reach hackers (ex. credit card info).
  + Network security.
    - Protect data against cyber hackers.

* + Physical security.
    - Protect devices in case they are stolen.

### Hardware.

* HLD specifies what hardware will be used.
* Hardware specifications:
  + Printers.
  + Network components.
  + Servers.
  + Specialized hardware (scales, microscopes, …).
  + Audio & video hardware.

### User interface (users & app).

* Indicate the navigation styles.
* Specify the forms used & verify if they do the required tasks.
* Include all the needed forms (from user stories & use cases).
* May include:
  + Special features (clickable maps, important tables).
  + Methods for specifying system settings (sliders, scroll bars).
  + Appearance (Color schemes, Logo placement, Form templates).

### Internal interfaces (among app modules).

* After dividing the system, specify how pieces/modules will interact.
  + Specifications must be clear & unambiguous.
  + Allows teams to work independently.

### External interfaces.

* Easier to specify than internal interfaces.
  + You don’t control both ends (you must meet the other end’s requirements).

### Architecture.

* The architecture describes how modules fit together.
* Some architectures attempt to simplify development.

Architecture types:

1. Monolithic.

* A single program does everything.
* Disadvantages:
  + System pieces are tied closely together.
  + Not flexible .
* Advantages:
  + A picture containing text, font, screenshot, line

    Description automatically generatedDoesn’t need complicated communication.

1. Client/server (multi-tier architecture).

* Separates parts that provide functions (servers) from the ones that need them (clients).
* A picture containing diagram, sketch, white, line

  Description automatically generatedDoesn’t allow multiple users to use the app (Dis-adv).
* Two-tier architecture: solves the disadvantage of the client/server archt.
  + Clients are separated from the server.
  + A diagram of a network

    Description automatically generated with low confidenceClients & servers communicate through a network.
* Three-tier architecture: a middle tier is inserted between both tiers.
  + Can run on the same device as the server.

1. Component-based (CB).

* Used in component-based software engineering (CBSE).
* The system is a collection of loosely coupled components.
* Separates pieces of code (same as multi-tier client/server).
  + All pieces are in the same program.

1. Service oriented (SO).

* Close to CB but pieces are implemented as services.
* Service: a self-contained program that provides a service for clients.

1. Data-centric (DC).

* Uses data in some central way.
* Data-centric designs:
  + Store data in relational database systems.
  + Using procedures to perform calculations & implement business logic.

1. Event-driven (ED).

* Pieces respond to events as they occur.

1. Rule-based (RB).

* Uses a collection of rules to decide what to do next.
* Sometimes referred to as expert systems || knowledge-based systems.
* Needs the programmer to identify the necessary rules for each job.
* Has trouble handling unexpected situations.

1. Distributed.

* Pieces run on different processors & can run concurrently.
* Processors are on different computers on the same network.
  + Processors can be different cores on a single computer.
* SO, CB, & multi-tier are often distributed.
* Disadvantages:
  + Extremely confusing.
  + Hard to debug.
  + Processor competition (Race condition).

1. Mix & match.

* Using different architectures in the same system.

### Reports.

* business apps might include reports (about costumers, products, users, …).

### Other outputs.

* include:
  + printouts.
  + Webpages.
  + Data files.
  + Image files, Audio, Video.
  + Output to special devices.
  + Email or text messages.

### Database.

* Decide what kind of database is needed & how data is going to be stored.
* Sketch out the tables for relational databases.

Database normalization:

* The process of decomposition (breaking tables into small tables & linking them).
* Advantages:
  + Avoids redundant data in tables.
  + Ensures that separate tables data can be maintained & updated separately.
  + Facilitates a distributed database.
* Steps:
  + Remove multiple value cells.
  + Move repeating values into different tables.
  + Create new tables to reduce redundancy.

Relationship types:

* One-to-one relationship:
  + one record (table 1) is related to one record (table 2).
* One-to-many relationship:
  + One record (table 1) is related to many records (table 2).
* Many-to-one relationship:
  + Many records (table 1) are related to one record (table 2).
* Many-to-many relationship:
  + Many records (table 1) are related to many records (table 2).

Database issues:

* Audit trails.
  + Audit trail: keeps track of each user who modifies a record.
  + Not needed in every application.
  + Simple version:
    - Create history table.
    - Contain user’s name, link to the record, and the data.
  + Advanced version:
    - Stores copies of the original data in the original table.
    - Might have notes section (to explain the changes).
* User access.
  + Providing users with different access levels (to the data).
  + Simple version:
    - Create an access level table.
    - List the users & their access levels.
  + Some databases have it built in.
* Database maintenance.
  + Removing old data (increases responsiveness).
  + Re-indexing key tables (restores peak performance).
  + Designing a database backup & recovery scheme.
  + Move older data into a secondary database (improves responsiveness & the data can be used for analysis).

### Configuration data.

* Provide configuration screens to users.
* Reduces the need to write new code.
* Provide different access levels to these configurations.

A diagram of a customer

Description automatically generated with low confidence

### Data flows & states.

* Specify the data flow between different processes.
* Events might make the data take different paths.

### Training.

* Describe how training should be (online, using manuals, with instructors).

## Unified Modeling Language (UML):

* A picture containing diagram, plan, technical drawing, schematic

  Description automatically generatedA general-purpose modeling language.
* Used to represent different system pieces.

### Structure diagrams:

* Describes things/objects that will be in the system.

Types:

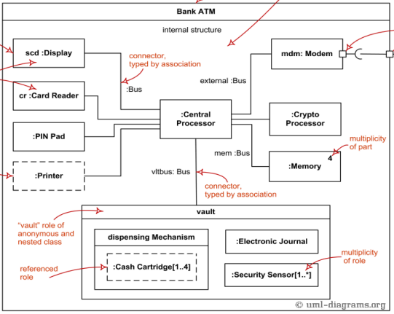
* Class diagram:
  + Most basic structure diagram.
  + A picture containing text, screenshot, font, line

    Description automatically generatedDescribes things/objects that make the system using classes, attributes, methods, and relationships.
  + Class: a collection of related properties.
    - Class name is a noun.
  + Objects: an instance of a class.
  + A picture containing text, font, white, screenshot

    Description automatically generatedMethod: a routine that makes an object do something.
  + Access modifiers symbols: a symbol that is used to represent access modifiers.
  + Relationships are represented using a line, an arrow, and a label.
    - Label is a verb.
  + A picture containing text, screenshot, font, number

    Description automatically generatedMultiplicity: a number placed at both ends of a relationship that indicates now many objects are involved in the relationship.
  + Can get complicated and devs separate it into different diagrams.
* Object diagram:
  + A picture containing text, font, white, screenshot

    Description automatically generatedShows how objects interact with each other and what values they contain at specific times.
* A diagram of a component

  Description automatically generated with low confidencecomponent diagram:
  + Shows how components form larger parts of the system.
  + Each component provides at least one interface.
    - Used by other interfaces to interact with it.
  + Complex components: components that include other components.
  + Components/symbols (go to page 76 for a full list):
    - Structured classifier: represents a class.
    - Part: represents a role played at runtime by one or more instances.
    - Port: an interaction point used to connect classifiers with parts & the environment.
    - Connector: binds two or more entities together.
    - Collaboration: more abstract than a classifier.
* composite diagram:
  + Shows the class’s internal structure & the collabs that the class allows.
* A picture containing text, font, diagram, screenshot

  Description automatically generatedpackage diagram:
  + Describes relationships among the system packages.
* deployment diagram:
  + A diagram of a computer server

    Description automatically generated with low confidenceDescribes the deployment of artifacts on nodes that can execute those artifacts.
    - Application artifacts (files, scripts, exe files, …).
    - Framework artifacts.
* profile diagram:
  + allows the designer to extend the basic UML notations (adding labels & comments).

### Behavior diagrams.

A picture containing text, screenshot, diagram, font

Description automatically generatedActivity diagram.

* Represents workflows for activities/tasks.
* A picture containing text, font, screenshot

  Description automatically generatedUses symbols & arrows to show the workflow.

Use case diagram.

* Represents user interactions with the system.
* A picture containing text, diagram, screenshot, line

  Description automatically generatedRepresents users with stick figures.
* <<include>> indicates that a task includes subtasks.
* <<extend>> a subtask that might occur based on the main task.
* Use dashed arrows with <<include>> & <<extend>>.

State machine diagram.

* A picture containing line, diagram, font, text

  Description automatically generatedShows the state of an object after each event.
* Representation:
  + States = rounded rectangles.
  + Transitions = arrows.
    - Labels are used to show the cause of the transition.
  + Starting state = black circle.
  + Stopping state = circled black circle.

Interaction diagram.

* A subset of activity diagrams.
* Include:
  + Sequence diagrams.
  + Communication diagrams.
  + Timing diagrams.
  + Interaction overview diagrams.

A diagram of a movie

Description automatically generated with medium confidence

Sequence diagram.

* Shows how objects/actors collab in a specific scenario.
  + Includes the interactions & their order.
* Represents the collab as a series of messages.
* Focuses on the message sequence.
* Representation:
  + Objects = rectangles.
  + Actors = stick figures (with class name and/or actor name).
  + Execution = white || gray rectangle on top of lifeline.
  + Synchronous messages = arrows with solid heads.
  + asynchronous messages = arrows with open heads.
  + Return messages = dashed arrows with open heads.
* Lifeline: a vertical dashed line that shows the active time of objects.
* execution: represents the actor/object doing something.

Communication diagram.

* Shows the communications between objects during a collab.
* Focuses on the objects.
* A diagram of a person

  Description automatically generated with low confidenceRepresentation:
  + Lines to connect objects.
  + Messages = labeled arrows.
    - Messages are numbered.

A picture containing text, diagram, line, parallel

Description automatically generatedTiming diagram.

* Shows state changes of multiple objects over time.
* Looks Like the sequence diagram (rotate the sequence 45 degree).
* Gives a sense of how long objects in a scenario will take.
* Advanced versions show multiple objects stacked over each other.

A picture containing text, diagram, plan, parallel

Description automatically generatedInteraction overview diagram.

* An activity diagram where nodes can contain other diagrams.
* Shows more details for nodes with complicated tasks.

# Chapter 4: Low-Level Design (LLD).