Why Do Projects Fail:

- Some of the most common reasons projects fail are:
    - Unrealistic or unarticulated project goals.
    - Inaccurate estimates of needed resources.
    - Badly defined system requirements.
    - Poor reporting of the project's status.
    - Unmanaged risks.
    - Poor communication among customers, developers, and users.
    - Use of immature technology.
    - Inability to handle the project's complexity.
    - Sloppy development practices.
    - Poor project management.
- Software project failures have a lot in common with airplane crashes. Just as pilots never intend to crash, software developers don't aim to fail. When a commercial plane crashes, investigators look at many factors, such as the weather, maintenance records, the pilot's disposition and training, and cultural factors within the airline. Similarly, we need to look at the business environment, technical management, project management, and organizational culture to get to the roots of software failures.
- Chief among the business factors are competition and the need to cut costs. Increasingly, senior managers expect IT departments to do more with less and do it faster than before. They view software projects not as investments but as pure costs that must be controlled.
- A lack of upper-management support can also hinder an IT undertaking. This runs the gamut from failing to allocate enough money and manpower to not clearly establishing the IT project's relationship to the organization's business.
- Frequently, IT project managers eager to get funded resort to a form of liar's poker, overpromising what their project will do, how much it will cost, and when it will be completed. Many, if not most, software projects start off with budgets that are too small. When that happens, the developers have to make up for the shortfall somehow, typically by trying to increase productivity, reducing the scope of the effort, or taking risky shortcuts in the review and testing phases. These all increase the likelihood of error and, ultimately, failure.
- Sloppy development practices are a rich source of failure, and they can cause errors at any stage of an IT project.
- Project managers also play a crucial role in software projects and can be a major source of errors that lead to failure. The most important function of the IT project manager is to allocate resources to various activities. Beyond that, the project manager is responsible for project planning and estimation, control, organization, contract management, quality management, risk management, communications, and human resource management. Bad decisions by project managers are probably the single greatest cause of software failures today. Poor technical management, by contrast, can lead to technical errors, but those can generally be isolated and fixed. However, a bad project management decision, such as hiring too few programmers or picking the wrong type of contract, can wreak havoc.

Software Process:

- It is:
  - A structured set of activities, used by a team to develop software systems.
  - The standards, practices, and conventions of a team.
  - A description of how a team performs its work.
- Some synonyms of software process are:
  - Software Development Process
  - Software Development Methodology
  - Software Development Life Cycle
- In a nutshell, it is a structured description of how a software development team goes through.
- **Software process** is a set of related activities that leads to the production of the software. These activities may involve the development of the software from the scratch, or modifying an existing system.
- Over time, people develop new software process models.
- A **software process model** represents the order in which the activities of software development will be undertaken. It describes the sequence in which the phases of the software lifecycle will be performed.

Software Process Model vs. Software Process:

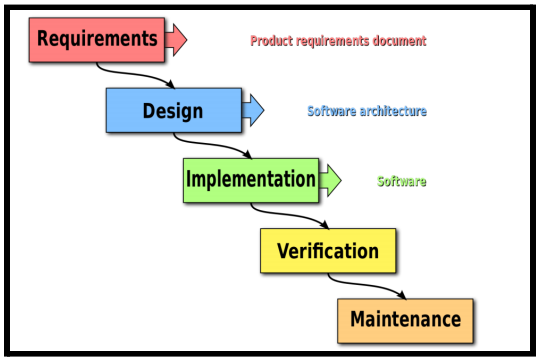
- Software process is a coherent set of activities for specifying, designing, implementing and testing software systems.
- A software process model is an abstract representation of a process that presents a description of a process from some particular perspective. When we describe and discuss about process models, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc and the ordering of these activities.
- Software process descriptions may include:
  - **Products:** The outcomes of a process activity.
  - **Roles:** Reflect the responsibilities of the people involved in the process.
  - **Pre and post-conditions:** Are statements that are true before and after a process activity has been enacted or a product produced.
- There are many different software processes but all involve:
  - **Specification:** Defining what the system should do.
  - **Design and Implementation:** Defining the organization of the system and implementing the system.
  - **Validation:** Checking that it does what the customer wants.
  - **Evolution:** Changing the system in response to changing customer needs.

Plan-Driven and Agile Processes:

- **Plan-driven processes** are processes where all of the process activities are planned in advance and progress is measured against this plan.
- In **agile processes**, planning is incremental which makes it easier to change the process to reflect changing customer requirements.
- In practice, most practical processes include elements of both plan-driven and agile approaches.
- There are no right or wrong software processes.

Waterfall Method:

- The **waterfall model** is a sequential (non-iterative) design process, used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of:
    1. Requirements analysis
    2. Design
    3. Implementation
    4. Testing (verification)
    5. Maintenance



- The result of each phase is one or more documents that should be approved and the next phase shouldn't be started until the previous phase has completely been finished. I.e. Each phase is carried out completely, for all requirements, before proceeding to the next. Furthermore, this process is strictly sequential. No backing up or repeating phases.
- The waterfall model should only be applied when requirements are well understood and unlikely to change radically during development as this model has a relatively rigid structure which makes it relatively hard to accommodate change when the process is underway.
- Pros:
    - Time spent early in the software production cycle can reduce costs at later stages.
    - Suitable for highly structured organizations.
    - It places emphasis on documentation which will contribute to corporate memory.
    - It provides a structured approach. The model itself progresses linearly through discrete, easily understandable and explainable phases and thus is easy to understand.
    - It also provides easily identifiable milestones in the development process.
    - It is well suited to projects where requirements and scope are fixed, the product itself is firm and stable, and the technology is clearly understood.
    - Simple, easy to understand and follow.
    - Highly structured.
    - After specification is complete, low customer involvement is required.
- Cons:
    - Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements. Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process. However, few business systems have stable requirements.
    - The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites. In these circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

Alternative Methodologies:

- As companies began to realize that the waterfall method was failing them (large projects were failing completely or going way over budget) alternatives were sought.
- A gradual trend was in methods that used an incremental development of product using iterations (almost like smaller waterfalls).
- Iterations could be only a few weeks, but still included the full cycle of analysis, design, coding, etc.
- These various lightweight development methods were later referred to as **agile methodologies**.
- As our models evolve, they encourage software development teams to:
    - Be more flexible and adaptive to changing requirements.
    - Collect feedback from users more frequently.
    - Release code more frequently.
- And then came the term Agile.
- **Agile** is neither a process nor a model but is a term that describes a process, model, or a team. Essentially, it means “Flexible and adaptive process/team, suitable for projects with constantly changing requirements".

Agile Manifesto:

- We value:
    - **Individuals and interactions** over **processes and tools**.
    - **Working software** over **comprehensive documentation**.
    - **Customer collaboration** over **contract negotiation**.
    - **Responding to change** over **following a plan**.
- There is value to the items on the right, but the left is valued more.

Agile:

- Agility is flexibility. It is a state of dynamic, adapted to the specific circumstances.
- Agile refers to a number of different frameworks that share these values.
  I.e. Agile is an umbrella term for a set of methods and practices based on the values and principles expressed in the Agile Manifesto that is a way of thinking that enables teams and businesses to innovate, quickly respond to changing demand, while mitigating risk.
- Examples of agile frameworks are:
    - Test Driven Development (TDD)
    - Extreme Programming (XP)
    - Scrum
    - Lean Software

Test Driven Development (TDD):

- A concept that started in the late 90s.
- Used by many Agile teams.
- The idea is to write the tests, before you write the code.
- The tests are the requirements that drive the development.
- A software development approach in which test cases are developed to specify and validate what the code will do. In simple terms, test cases for each functionality are created and tested first and if the test fails then the new code is written in order to pass the test and make code simple and bug-free.
- TDD ensures that your system actually meets requirements defined for it. It helps to build your confidence about your system.
- Traditionally, TDD means:
    - Write a failing test.
    - Write the (least amount of) code to pass the test.
    - Repeat.
    - Every now and then refactor/cleanup code.

- However, in practice, each team decides when and where it makes sense for tests to drive development.
- Most teams borrow some of the concepts of TDD, such as the fact that tests are used as specification/documentation and the fact that we should automate tests in fragile/crucial areas of your system.
- Software development teams can adopt TDD with different types of testings such as:
    - **Unit Test**
    - **Integration:** Test that all the different units in the system play nicely together.
    - **Acceptance:** Specify customer's requirement.
    - **Regression:** Verify we didn't break anything that was working before. Automated unit tests can be used as regression tests.
- Advantages of TDD:
    - **Early bug notification:**
        - Using TDD, over time, a suite of automated tests is built up that you and any other developer can rerun at will.
    - **Better designed, cleaner and more extensible code:**
        - TDD helps developers understand how the code will be used and how it interacts with other modules.
        - TDD allows you to write smaller modules with each having a single responsibility rather than monolithic modules with multiple responsibilities. This makes the code simpler to understand.
        - TDD forces you to write the bare minimum of production code needed to pass the tests.
    - **Confidence to refactor:**
        - If you refactor code, it might break. By having a set of automated tests, you can fix those bugs before release.
    - **Good for teamwork:**
        - In the absence of any team member, other team members can easily pick up and work on the code. It also aids knowledge sharing, thereby making the team more effective overall.
    - **Good for developers:**
        - Though developers have to spend more time writing TDD test cases, it will take a lot less time debugging and developing new features. You will write cleaner, less complicated code.

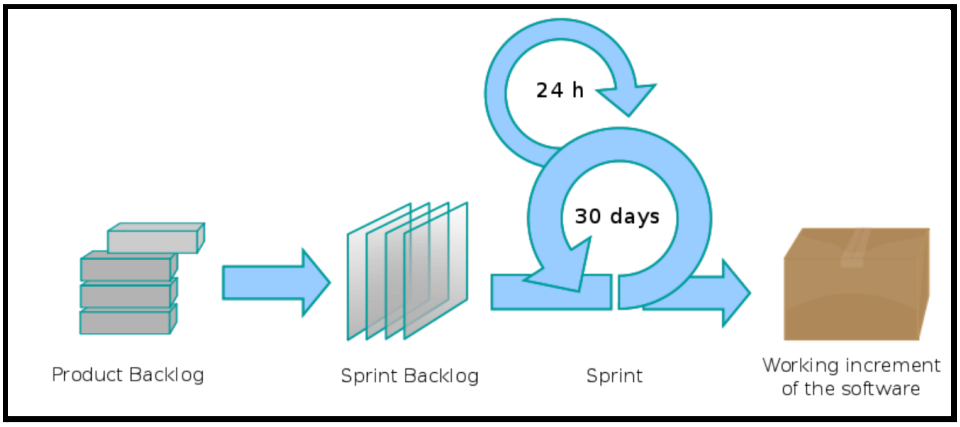
Extreme Programming (XP):

- XP is a model that was getting a lot of hype in the late 90s.
- XP is an Agile model, consisting of many rules/practices, one of which is TDD.
- XP is a very detailed model, but in practice, most teams adopt a subset of its rules.
- Some highlights of XP:
    - Iterative incremental model.
    - Better teamwork.
    - Customer's decisions drive the project.
    - Dev team works directly with a domain expert.
    - Accept changing requirements, even near the deadline.
    - Focus on delivering working software instead of documentation.
- A key assumption of XP is that the cost of changing a program can be held mostly constant over time. This can be achieved with:
    - Emphasis on continuous feedback from the customer
    - Short iterations
    - Design and redesign
    - Coding and testing frequently

- Eliminating defects early, thus reducing costs
- Keeping the customer involved throughout the development
- Delivering working product to the customer

- Extreme Programming involves:
  - Writing unit tests before programming and keeping all of the tests running at all times. The unit tests are automated and will eliminate defects early, thus reducing the costs.
  - Starting with a simple design just enough to code the features at hand and redesigning when required.
  - **Pair programming** which is when two programmers sit at one screen, taking turns to use the keyboard. While one of them is at the keyboard, the other constantly reviews and provides inputs.
  - Integrating and testing the whole system several times a day.
  - Putting a minimal working system into the production quickly and upgrading it whenever required.
  - Keeping the customer involved all the time and obtaining constant feedback.

Scrum:



- **Scrum** is a flexible, holistic product development strategy where a development team works as a unit to reach a common goal. Scrum is mainly about the management of software development projects.
- **Sprint:** The actual time period when the scrum team works together to finish an increment. Two weeks is a pretty typical length for a sprint, though some teams find a week to be easier to scope or a month to be easier to deliver a valuable increment. During this period, the scope can be re-negotiated between the product owner and the development team if necessary. This forms the crux of the empirical nature of scrum. Key features of sprints:
  - It is a basic unit of development in a scrum.
  - It is of fixed length, typically from one week to a month.
  - Each sprint begins with a **sprint planning meeting** to determine the tasks for the sprint and estimates are made.
  - During each sprint a potentially deliverable product is produced.
  - Features are pulled from a **product backlog**, a prioritized set of high level work requirements.
- **Sprint planning:** The work to be performed during the current sprint is planned during this meeting by the entire development team. This meeting is led by the **scrum master** and is where the team decides on the sprint goal. Specific **user stories** are then added to the sprint from the product backlog. These stories always align with the goal and are also agreed upon by the scrum team to be feasible to implement during the sprint. At the

end of the planning meeting, every scrum member needs to be clear on what can be delivered in the sprint and how the increment can be delivered.

- **User Story:** An informal, general explanation of a software feature written from the perspective of the end user or customer. The purpose of a user story is to articulate how a piece of work will deliver a particular value back to the customer. User stories are a few sentences in simple language that outline the desired outcome. They don't go into detail. Requirements are added later, once agreed upon by the team.
- **Product Backlog:** The master list of work that needs to get done maintained by the product owner or product manager. This is a dynamic list of features, requirements, enhancements, and fixes that acts as the input for the sprint backlog. It is, essentially, the team's "To Do" list. The product backlog is constantly revisited, re-prioritized and maintained by the Product Owner because, as we learn more or as the market changes, items may no longer be relevant or problems may get solved in other ways.
- **Sprint Backlog:** The list of items, user stories, or bug fixes, selected by the development team for implementation in the current sprint cycle. Before each sprint, in the sprint planning meeting the team chooses which items it will work on for the sprint from the product backlog. A sprint backlog may be flexible and can evolve during a sprint.
- **Increment/Sprint Goal:** The usable end-product from a sprint.
- **Daily Scrum/Daily Standup:** A short meeting that happens at the same place and time each day. At each meeting, the team reviews work that was completed the previous day and plans what work will be done in the next 24 hours. This is the time for team members to speak up about any problems that might prevent project completion.
  Some features of the daily scrum:
    - No more than 15 minutes.
    - Meetings must start on-time, and happen at the same location.
    - Each member answers the following:
        - What have you done since yesterday?
        - What are you planning on doing today?
        - Are there any impediments or stumbling blocks?
    - A scrum master will handle resolving any impediments outside of this meeting
- **Scrum Master:** The person on the team who is responsible for managing the process, and only the process. They are not involved in the decision-making, but act as a lodestar to guide the team through the scrum process with their experience and expertise. The scrum master is the team role responsible for ensuring the team follows the processes and practices that the team agreed they would use.
- In traditional agile development software is brought to release level every few months. **Releases**, which are sets of sprints, are used to produce shippable versions of software products.
- A key feature of scrum is that during a project a customer may change their minds about what they want/need. We need to accept that the problem cannot be fully understood or defined and instead allow teams to deliver quickly and respond to changes in a timely manner.

Why Use Agile:

- Demand for higher quality with lower cost.
- Post-mortems of software projects lead to a lot of knowledge gain about what went right/wrong during the development phase.
    - With the waterfall model, we do not have a clear way to predict the future, so these lessons are always in hindsight.
    - Smaller, iterative schedules mean problems can be identified/addressed much earlier in the cycle.

- Agile eliminates waste.
    - Making changes at the end of a production cycle is costly. With agile we are more likely to detect these changes early enough to reduce the costs.
    - Iterative design means we build a product in small steps.
        - Incrementally add features.
        - Software is in working condition at least every few weeks.
        - Allows people to test earlier in the development cycle.
        - Software improvements happen much earlier and can be fine-tuned rather then trying to modify the design at the end of a waterfall cycle.

Agile Team Management:

- From the bottom up.
- Teams are empowered to manage the smallest level of details, while leaving the higher levels to upper management.
- Teams, upon seeing the small amount of ownership they get from solving smaller problems, take on responsibility for larger problems.
    - Head off issues before they become major problems.
    - Individuals solve problems with their colleagues.

Agile Project Structure:

- An agile project consists of a series of iterations of development.
- Each interval usually lasts only two to four weeks.
- Developers implement features, called user stories, during each iteration that add value to the project.
- Each iteration contains a full development cycle:
    - Concept
    - Design
    - Coding
    - Testing
    - Deployment
- The project is reviewed at the end of each iteration.
- Results are used to direct future iterations.
- Every three to six iterations the project is built up to a release state, meaning that most major goals are accomplished.

Incremental Development Benefits:

- The cost of accommodating changing customer requirements is reduced as the amount of analysis and documentation that has to be redone is much less than what is required with the waterfall model.
- It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how much has been implemented.
- More rapid delivery and deployment of useful software to the customer is possible. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

Incremental Development Problems:

- The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- The system structure tends to degrade as new increments are added. Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

Is It True That Only Non-Agile Projects Fail:

- The Scott Ambler survey defines success as a solution being delivered and meeting its success criteria within the acceptable range defined by the organization, and failure as the project never delivering a solution.
- The Ambler report concluded that agile projects do not fail more than other projects. They succeed at the same level as other iterative methodologies.
- However, agile projects face a set of challenges and problems related to applying a different approach to project management. The top three reasons for agile project failure are:
    1. Inadequate experience with agile methods.
    2. Little understanding of the required broader organizational change.
    3. Company philosophy or culture at odds with agile values.

Selecting a Development Model:

- For organizations and projects, where experience can be used to plan a course of action with a good degree of certainty for a positive outcome, a traditional methodology may be more appropriate than an agile methodology. In this case, the plans can be developed up-front and then designed, developed, and tested without much variance.
- Agile methodologies are effective when the product details cannot be defined or agreed in advance with any degree of accuracy. This situation calls for the collaborative environment between the user and the developer. Agile methodologies are suited for a dynamic and changing environment.

Key Process Stages:

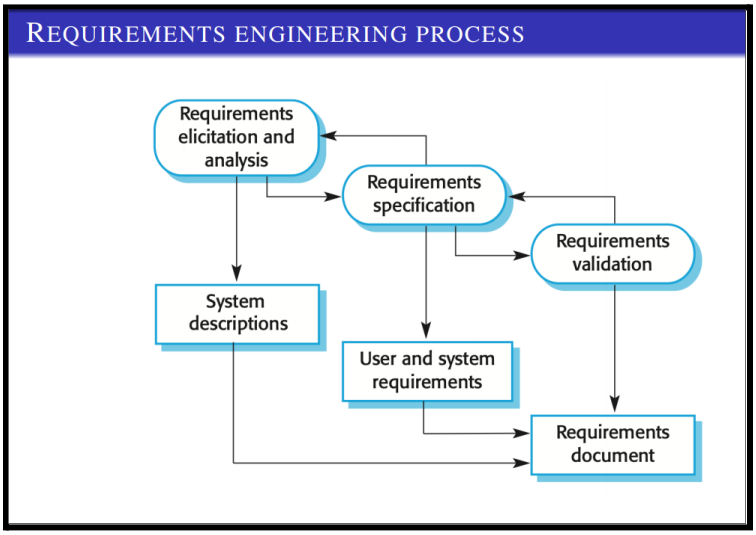
- The stages are:
    1. Requirements specification
    2. Software discovery and evaluation
    3. Requirements refinement
    4. Application system configuration
    5. Component adaptation and integration
- Real software processes are interleaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.
- The four basic process activities of specification, development, validation and evolution are organized differently in different development processes. For example, in the waterfall model, they are organized in sequence, whereas in the incremental development they are interleaved.

Software Specification:

- Is the process of establishing what services are required and the constraints on the system's operation and development.
- **Requirements engineering process** involves the following:
    - **Requirements elicitation and analysis:** Is the practice of researching and discovering the requirements of a system from users, customers, and other stakeholders. It is the various ways used to gain knowledge about the project domain and requirements. The various sources of domain knowledge include customers, business manuals, the existing software of the same type, standards and other stakeholders of the project. It answers the question "What do the system stakeholders require or expect from the system?"
    - **Requirements specification:** Defines the requirements in detail. This activity is used to produce formal software requirement models. During specification, more knowledge about the problem may be required which can again trigger the elicitation process.
      I.e. It is the process of writing down the **user and system requirements** into a

document. The requirements should be clear, easy to understand, complete and consistent.
The **user requirements** for a system should describe the functional and non-functional requirements so that they are understandable by users who don't have technical knowledge. You should write user requirements in natural language supplied by simple tables, forms, and intuitive diagrams.
The requirement document shouldn't include details of the system design and you shouldn't use any software jargon or formal notations.
The **system requirements** are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They shouldn't be concerned with how the system should be implemented or designed. The system requirements may also be written in natural language but other ways based on structured forms, or graphical notations are usually used.

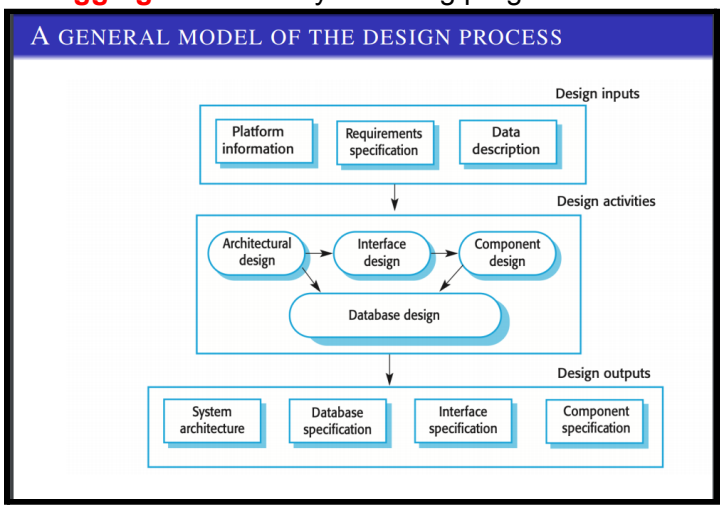
- **Requirements validation:** Checks the validity of the requirements. It's a process of ensuring that the specified requirements meet the customer needs.



Software Design and Implementation:

- Is the process of converting the system specification into an executable system.
- **Software design** is designing a software structure that realizes the specification.
  Some software design activities include:
  - **Architectural design:** Identifying and defining the overall structure of the system, the principal components, their relationships and how they are distributed.
  - **Database design:** Designing the system data structures and how they will be represented in a database.
  - **Interface design:** Defining the interfaces between system components.
  - **Component selection and design:** Searching for reusable components. If unavailable, you design how it will operate.
- **Software implementation** is taking your design and translating into an executable program.
  The software is implemented either by developing program(s) or by configuring an application system.
  Design and implementation are interleaved activities for most types of software system.
  Programming is an individual activity with no standard process.

Debugging is the activity of finding program faults and correcting these faults.



- The activities of design and implementation are closely related and may be interleaved.

Software Validation:

- **Verification and validation (V&V)** is intended to show that a system conforms to its specification and meets the requirements of the system customer.
  I.e. It is the process of checking that a software system meets specifications and that it fulfills its intended purpose.
- **Software validation** is a dynamic mechanism of testing and validating if the software product actually meets the exact needs of the customer or not. The process helps to ensure that the software fulfills the desired use in an appropriate environment. The validation process involves activities like unit testing, integration testing, system testing and user acceptance testing.
  I.e. Software validation checks if the code actually does what it is supposed to do.
- **Software verification** is a process of checking documents, design, code, and program in order to check if the software has been built according to the requirements or not. The main goal of the verification process is to ensure quality of software application, design, architecture etc. The verification process involves activities like reviews, walk-throughs and inspection.
  I.e. Software verification checks if the program is built according to the specifications and design.
- **System testing** is the process of testing an integrated system to verify that it meets the specified requirements. The purpose of this test is to evaluate the system's compliance with the specified requirements.
- Testing is the most commonly used V&V activity.

Building Models:

- **Introduction to Models:**
- **Forward engineering** is where we take the requirements and produce a model from it. This model can either be very high level or very low level.
  If we want a high level point of view, we usually sketch the components of the model.
  If we want a low level point of view, we need to add a lot of details.
- **Reverse engineering** is where we're given a code base and we extract the requirements from it. Creating higher level views can help us better understand the project.
- For reverse engineering, we would like to know the following information:
  1. **Structure of the code:**
     - This involves knowing the dependencies of the code and all the different components and how they couple together.
  2. **Behaviour of the code:**
     - This involves knowing how the code executes.
     - For more complex code, we may need to make state machine models.
  3. **Function of the code:**
     - This involves knowing what functions the code provides to the user.
- Modelling can guide your exploration.
  - It can help you figure out what questions to ask.
  - It can help to reveal key design decisions.
  - It can help you to uncover problems.
- Modelling can help us check our understanding.
  - We can use the model to understand its consequences and know if it has the properties we expect.
  - We can animate the model to help us visualize/validate software behaviour.
- Modelling can help us communicate.
  - Modelling provides useful abstractions that focus on the point you want to make without overwhelming people with detail.
  - Furthermore, both technical and non-technical people can understand.
- However, the exercise of modelling is more important than the model itself.
  Time spent perfecting the models might be time wasted.
- **Dealing with the Problem Complexity:**
- There are 4 ways to dealing with problem complexity:
  1. **Abstraction:**
     - Allows us to ignore details and look at the big picture.
     - We can treat objects as the same by ignoring certain differences.
     - **Note:** Every abstraction involves choice over what is important.
  2. **Decomposition:**
     - Allows us to break a problem into many independent pieces so that we can study each piece separately.
     - **Note:** The pieces are rarely independent.
  3. **Projection:**
     - Allows us to separate different points of view and describe them separately. I.e. Divide and conquer.
     - This is different from decomposition as it does not partition the problem space.
     - **Note:** Different views will be inconsistent most of the time.
  4. **Modularization:**
     - Allows us to choose structures that are stable over time to localize change.
     - **Note:** Some structures will make localizing changes easier and others will make it harder.

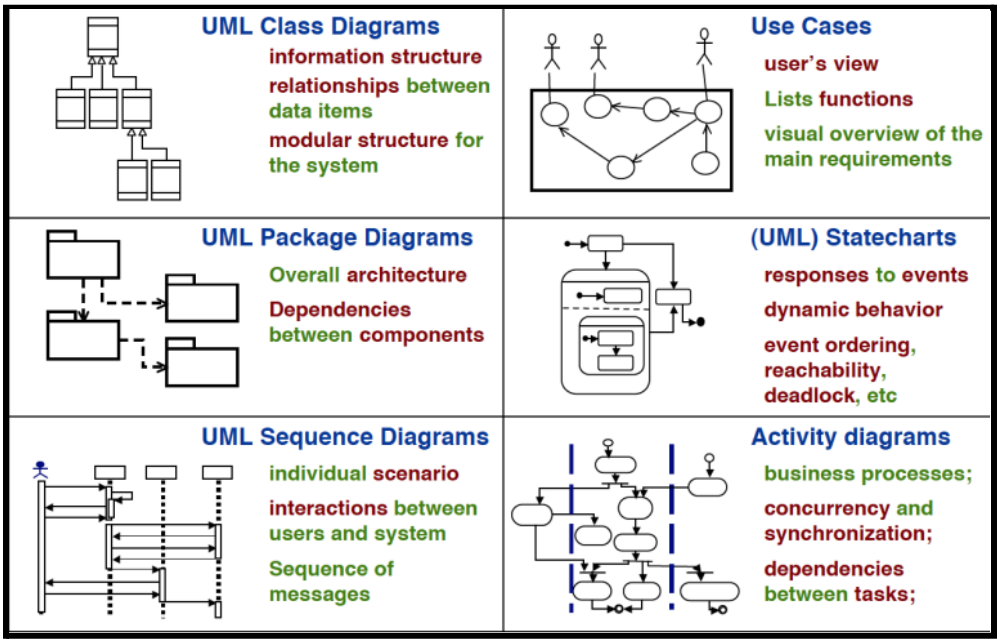
- **Design Phase:**
- **Design** is specifying the structure of how a software system will be written and function, without actually writing the complete implementation.
- During the design phase we transition from "what" the system must do to "how" the system will do it.
- The design phase involves answering the following questions:
    - What classes will we need to implement a system that meets our requirements?
    - What fields and methods will each class have?
    - How will the classes interact with each other?

Unified Modeling Language (UML):

- **Introduction to UML:**
- **Unified Modeling Language (UML)** allows us to express the design of a program before writing any code.
- It is language-independent.
- It is an extremely expressive language.
- UML is a graphical language for visualizing, specifying, constructing, and documenting information about software-intensive systems.
- UML can be used to develop diagrams and provide programmers with ready-to-use, expressive modeling examples. Some UML tools can generate program language code from UML. UML can be used for modeling a system independent of a platform language.
- UML is a picture of an object oriented system. Programming languages are not abstract enough for object oriented design. UML is an open standard and lots of companies use it.
- Legal UML is both a descriptive language and a prescriptive language. It is a descriptive language because it has a rigid formal syntax, like programming languages, and it is a prescriptive language because it is shaped by usage and convention.
- It's okay to omit things from UML diagrams if they aren't needed by the team/supervisor/instructor.
- **History of UML:**
- In an effort to promote object oriented designs, three leading object oriented programming researchers joined forces to combine their languages. They were:
    1. Grady Booch (BOOCH)
    2. Jim Rumbaugh (OML: object modeling technique)
    3. Ivar Jacobsen (OOSE: object oriented software eng)
- They came up with an industry standard in the mid 1990's.
- UML was originally intended as a design notation and had no modelling associated with it.
- **UML diagrams can help engineering teams:**
- Bring new team members or developers switching teams up to speed quickly.
- Navigate source code.
- Plan out new features before any programming takes place.
- Communicate with technical and non-technical audiences more easily.
- **Uses of UML:**

1. It can be used as a sketch to communicate aspects of the system.
- **Forward design:** Doing UML before coding.
- **Backward design:** Doing UML after coding as documentation.
2. It can be used as a blueprint to show a complete design that needs to be implemented. This is sometimes done with CASE (Computer-Aided Software Engineering) tools. One of these tools is visual paradigm.
3. It can be used as a programming language.
- Some UML tools can generate program language code from UML.

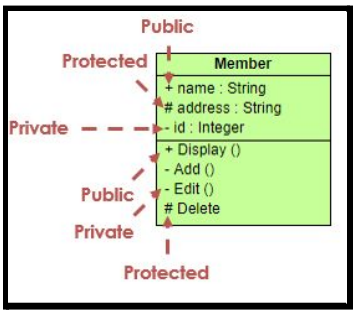
- **UML Notations:**

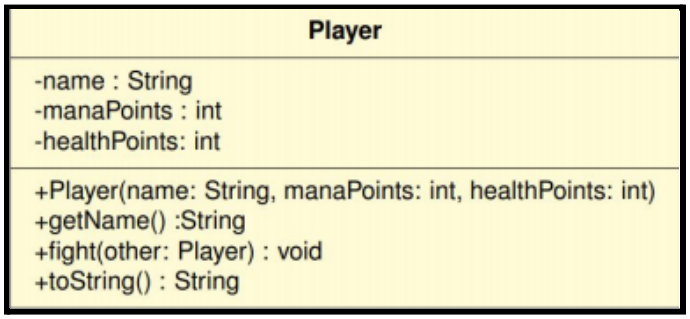


- **Class Diagram:** It displays the system's classes, attributes, and methods. It is helpful in recognizing the relationship between different objects as well as classes.
- **Object Diagram:** It describes the static structure of a system at a particular point in time. It can be used to test the accuracy of class diagrams. It represents distinct instances of classes and the relationship between them at a time.
- **Use Case Diagram:** It represents the functionality of a system by utilizing actors and use cases. It encapsulates the functional requirement of a system and its association with actors. It portrays the use case view of a system.
- **Package Diagram:** It is used to illustrate how the packages and their elements are organized. It shows the dependencies between distinct packages. It manages UML diagrams by making it easily understandable. It is used for organizing the class and use case diagrams.
- **Statechart:** It is a behavioral diagram. It portrays the system's behavior by utilizing finite state transitions. It models the dynamic behavior of a class in response to external stimuli.
- **Sequence Diagram:** It shows the interactions between objects in terms of messages exchanged over time. It delineates in what order and how the object functions are in a system. Time does not play a role.
- **Activity Diagram:** It models the flow of control from one activity to the other. With the help of an activity diagram, we can model sequential and concurrent activities. It visually depicts the workflow as well as what causes an event to occur. Time plays a role.
- **Note:** Statechart depicts the state transition. E.g. Finite state machine. Activity diagrams depict the various activities that occur. In activity diagrams, the time plays a role while in sequence diagrams, time does not play a role. An activity diagram is a sequence diagram but with timing.

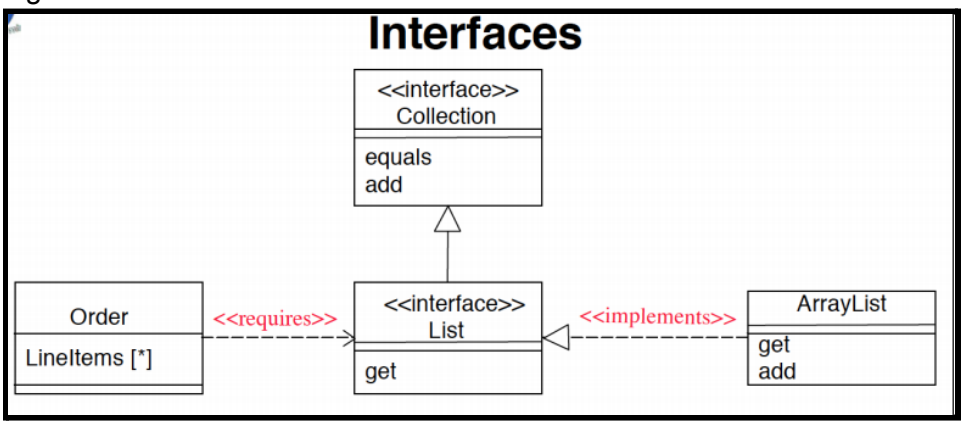
- **UML Class Diagram:**
- A class describes a group of objects with:
  - Similar attributes
  - Common operations
  - Common relationships with other objects
  - Common meaning
- A **class diagram** describes the structure of an object oriented system by showing the classes in that system and the relationships between the classes. A class diagram also shows the constraints, and attributes of classes.
  I.e.
  A UML class diagram is a picture of:
  - The classes in an object oriented system.
  - Their fields and methods.
  - Connections between the classes that interact or inherit from each other.
- Some things that are not represented in a UML class diagram are:
  - Details of how the classes interact with each other.
  - Algorithmic details, like how a particular behavior is implemented.
- **Note:** Coupling between classes must be kept low, while cohesion within a class must be kept high. Furthermore, we should respect the SOLID principles.
- UML class diagrams cans show:
  1. Division of responsibility
  2. Subclassing/Inheritance
  3. Visibility of objects and methods
  4. Aggregation/Composition
  5. Interfaces
  6. Dependencies
- Naming Convention:
  1. **Class name**
  - Use **<<interface>>** on top of interface names.
  - To show that a class is abstract, either italicize the class name or put **<<abstract>>** on top of the abstract class name.
  2. **Data members/Attributes**
  - The data members section of a class lists each of the class's data members on a separate line.
  - Each line uses this format: **attributeName : type**
    E.g. **name : String**
  - We must underline static attributes.
  3. **Methods/Operations**
  - The methods of a class are displayed in a list format, with each method on its own line.
  - Each line uses this format:
    methodName(param1: type1, param2: type2, ...) : returnType
    E.g. **distance(p1: Point, p2: Point) : Double**
  - We may omit setters and getters. However, don't omit any methods from an interface.
  - Furthermore, do not include inherited methods.
  - We must underline static methods.

- Visibility:
- − means that it is private.
- + means that it is public.
- # means that it is protected.
- ~ means that it is a package.
- / means that it is a **derived attribute**. A derived attribute is an attribute whose value is produced or computed from other information.
- **Note:** Everything except / is common for both methods and attributes.
- E.g.

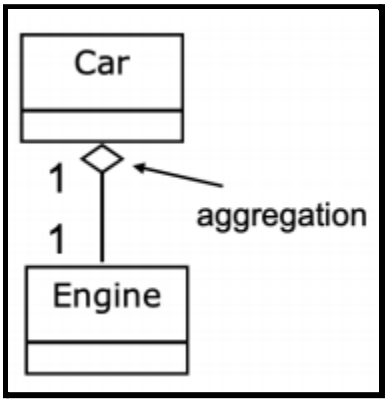




- Inheritance/Generalization and Realization Relationships:
- **Generalization/inheritance** is when a class extends another class while **realization** is when a class implements an interface.
- Generalization represents a “IS-A” relationship.
- Hierarchies are drawn top down with arrows pointing upward to the parent class. I.e. The parent class is above the child class and the arrow goes from the child class to the parent class.
- For a class, draw a solid line with a black arrow pointing to the parent class.
- For an abstract class, draw a solid line with a white arrow pointing to the parent abstract class.
- For an interface, draw a dashed line with a white arrow pointing to the interface.
- E.g.

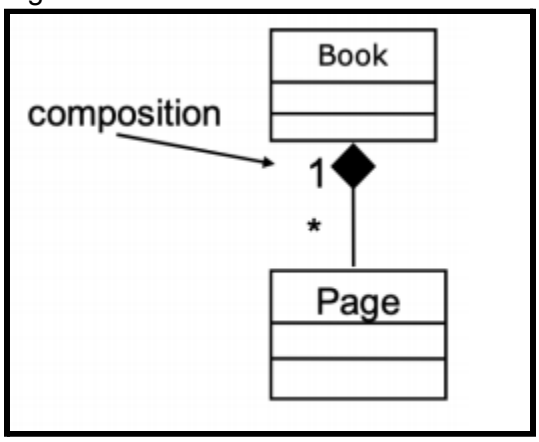


- Association:
- An association represents a relationship between two classes. It also defines the multiplicity between objects.
- Association can be represented by a line between the classes with an arrow indicating the navigation direction.
  Note: Sometimes, association can be represented just by a line between the classes. This means that information can flow in both directions.
- We need the following items to represent association between 2 classes:
  1. The multiplicity
  2. The name of the relationship
  3. The direction of the relationship
- Aggregation, composition and dependency are all types of association.
- Multiplicity:
- * means 0 or more.
- 1 means 1 exactly.
- 2..4 means 2 to 4, inclusive.
- 3..* means 3 or more.
- There are other relationships such as 1-to-1, 1-to-many, many-to-1 and many-to-many.
- Aggregation:
- A special type of association.
- Aggregation implies a relationship where the child class can exist independently of the parent class. This means that if you remove/delete the parent class, the child class still exists.
  I.e. Aggregation represents a “HAS-A” or “PART-OF” relationship.
  E.g. Say we have 2 classes, Teacher (the parent class) and Student (the child class). If we delete the Teacher class, the Student class still exists.
- Aggregation is symbolized by an arrow with a clear white diamond arrowhead pointing to the parent class.
- E.g.

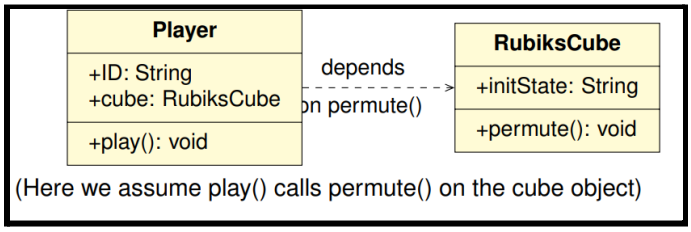


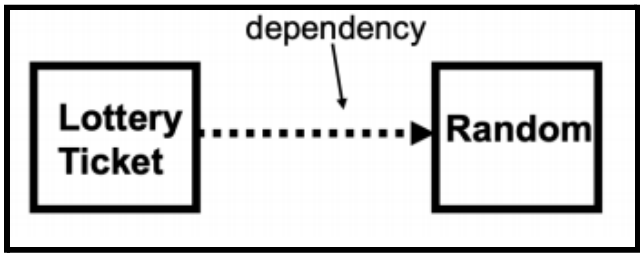
- Aggregation is considered as a weak type of association.
- Composition:
- A special type of association.
- It is a stronger version of aggregation where if you delete the parent class, then all the child classes are also deleted.
  I.e. Composition represents a “ENTIRELY MADE OF” relationship.
  E.g. Say we have 2 classes, House (the parent class) and Room (the child class). If we delete the House class, the Room class is also deleted.
- Composition is symbolized by an arrow with a black diamond arrowhead pointing to the parent class.

- E.g.

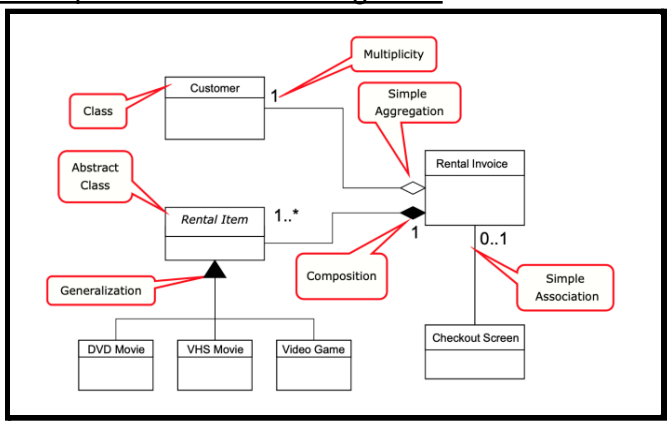


- Composition is considered as a strong type of association.
- Dependency:
- Is a special type of association.
- Dependency indicates a “uses” relationship between two classes. If a change in structure or behaviour of one class affects another class, then there is a dependency between those two classes.
- Dependency is represented by a dotted arrow where the arrowhead points to the independent element.
- E.g.

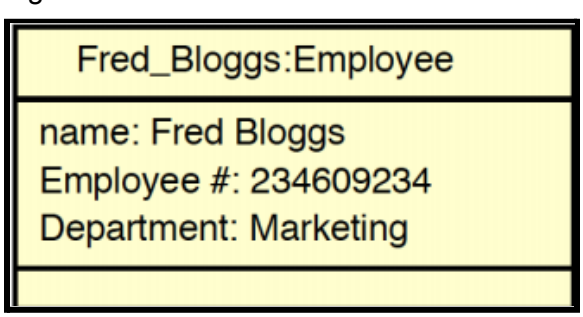


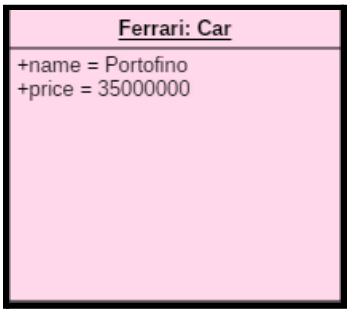


- Examples of UML class diagrams:



- How to draw class diagrams:
    1. Identify the objects in the problem and create classes for each of them
    2. Add attributes
    3. Add operations
    4. Connect classes with relationships
    5. Specify the multiplicities for association connections.
- **UML Object Diagram:**
- Object diagrams look very similar to class diagrams.
- Naming Convention:
  Object name: Type
  Attribute: Value (Sometimes, it's Attribute = Value)
  E.g.





- **Note:** 2 different objects may have identical attribute values.
- Purpose:
- It is used to describe the static aspect of a system.
- It is used to represent an instance of a class.
- It can be used to perform forward and reverse engineering on systems.
- It is used to understand the behavior of an object.
- It can be used to explore the relations of an object and can be used to analyze other connecting objects.

Coupling vs Cohesion:

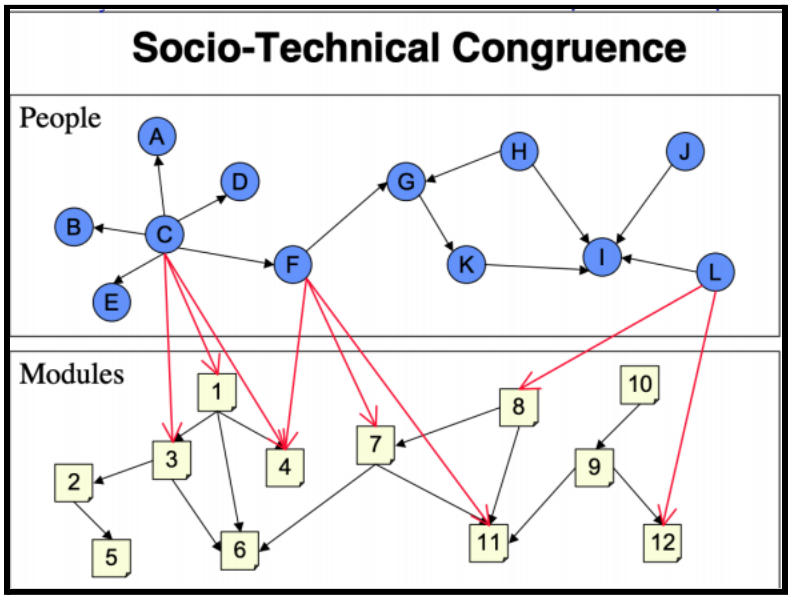
- **Coupling and Cohesion:**
- **Modules** are the building blocks of architectural applications and they often communicate with each other.
- A good architecture minimizes **coupling** between modules and maximizes the **cohesion** between modules.
- **Cohesion** refers to the degree to which the elements inside a module belong together.
- High-cohesion means that each class takes care of one thing, and one thing only. This references the **Single Responsibility Principle**.
- Low cohesion implies that a given module performs tasks which are not very related to each other and hence can create problems as the module becomes large.
- Modules with high cohesion tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.
- Think of building a physical robot. Many small parts (highly cohesive), versus a few mega parts (low cohesion, monolithic).
- **Coupling** refers to the interdependencies between modules.
  I.e. Components that are mutually dependent are also called coupled.
- A **loosely coupled** system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components.
- **Tight coupling/tightly coupled** is a type of coupling that describes a system in which hardware and software are not only linked together, but are also dependent upon each other.
- Loose coupling is important because it enables isolation and makes for future changes easier.
- We want to keep the coupling low and the cohesion high.
- **Measuring Coupling:**
- **Efferent coupling (ec):** Measures the number of classes on which a given class depends.
- **Afferent coupling (ac):** Measures how many classes depend on a given class.
- The **instability index** measures efferent coupling in relation to total coupling:
  $$\frac{ec}{ec + ac}$$
- The closer to zero the instability index is, the better.
- **Measuring Cohesion:**
- The cohesion of a class is the inverse of the number of method pairs whose similarity is zero.
- E.g. Assume a class has methods M1(p,q,r,s,t), M2(r,s,t), and M3(a,b,c). The lower case letters are instance variables participating in the methods. We see that for two methods (M1 and M2), their sets of participating variables intersect while the third (M3) doesn't.
  I.e. M1 and M2 share some of the same instance variables while M3 doesn't share any instance variables with M1 or M2.
  Hence the cohesion is 1/(2−0) = 1/2, which is small.

Conway's Law:

- **Conway's law** is an adage stating that organizations design systems which mirror their own communication structure.
- The law states: "The structure of a software system reflects the structure of the organization that built it."

Socio-Technical Congruence:

- Product development endeavors involve two fundamental elements:
    1. Technical: This involves processes, tasks, technology, etc.
       I.e. How people interact with the software.
    2. Social: This involves organizations and the individuals involved in the development process, their attitudes and behaviors.
       I.e. How people interact with each other.
- I.e.



- We can measure socio-technical congruence using a **coordination requirements matrix**.

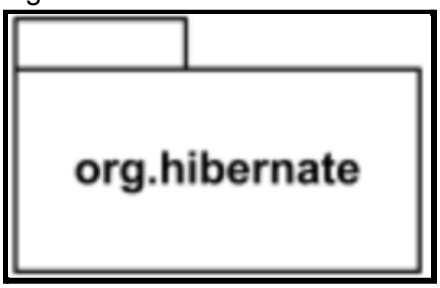
Software Architecture:

- The architecture of a system describes its major components, their relationships and how they interact with each other.
  A software architecture defines:
    1. The components of the software system.
    2. How the components use each other's functionality and data.
    3. How control is managed between the components.
- An example of a software architecture is the **client-server architecture**. The client-server architecture is a computing model in which the server hosts, delivers and manages most of the resources and services to be consumed by the client. This type of architecture has one or more client computers connected to a central server over a network or internet connection.
  Another example of a software architecture is the **3-layer architecture**. The 3-layer architecture has 3 main components:
    1. The presentation layer (UI)
    2. The application logic layer
    3. The database layer
- **Note:** MVC is not a 3-layer architecture. A fundamental rule in a 3-layer architecture is the presentation layer never communicates directly with the database layer. In a 3-layer architecture all communication must pass through the application logic layer. Conceptually the three-tier architecture is linear. However, the MVC architecture is triangular. The view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model.

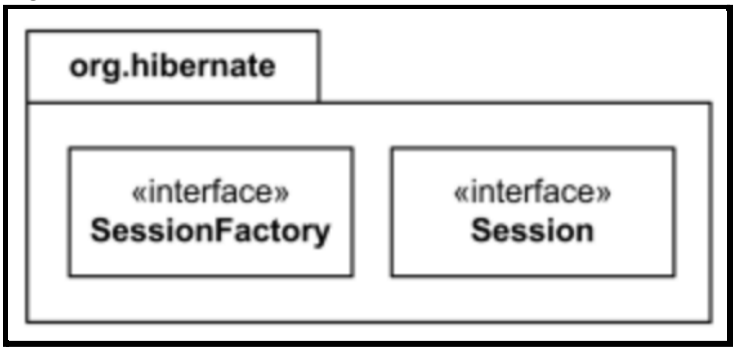
UML Packages:

- **Introduction:**
- A **package** is a namespace used to group together elements that are semantically related and might change together. It is a general purpose mechanism to organize elements into groups to provide a better structure for a system model.
- **UML package diagrams** are structural diagrams used to show the organization and arrangement of various model elements in the form of packages. A package is a grouping of related UML elements, such as diagrams, documents, classes, or even other packages. Each element is nested within the package, which is depicted as a file folder, and then is arranged hierarchically within the diagram. Package diagrams are most commonly used to provide a visual organization of the layered architecture within any UML classifier, such as a software system.
- Package diagrams are UML structure diagrams which show packages and dependencies between the packages.
  Note: Structure diagrams do not utilize time related concepts and do not show the details of dynamic behavior.
- If a package is removed from a model, so are all the elements owned by the package.
- A package could also be a member of other packages.
- A package in the Unified Modeling Language helps:
- To group elements.
- To provide a namespace for the grouped elements.
- Provide a hierarchical organization of packages.
- Benefits of UML package diagrams:
- They provide a clear view of the hierarchical structure of the various UML elements within a given system.
- These diagrams can simplify complex class diagrams into well-ordered visuals.
- They offer valuable high-level visibility into large-scale projects and systems.
- Package diagrams can be used to visually clarify a wide variety of projects and systems.
- These visuals can be easily updated as systems and projects evolve.
- **Terminology:**
- **Package:** A namespace used to group together logically related elements within a system. Each element contained within the package should be a packageable element and have a unique name.
- **Packageable element:** A named element, possibly owned directly by a package. These can include events, components, use cases, and packages themselves. Packageable elements can also be rendered as a rectangle within a package, labeled with the appropriate name.
- **Dependencies:** A visual representation of how one element or set of elements depends on or influences another. Dependencies are divided into two groups: access and import dependencies.
- **Access dependency:** Indicates that one package requires assistance from the functions of another package.
  I.e. One package requires help from functions of another package. (Making an API call for example)
- **Import dependency:** Indicates that functionality has been imported from one package to another.
  I.e. One package imports the functionality of another package. (Importing a package)
- **Notation:**
- A package is rendered as a rectangle with a small tab attached to the left side of the top of the rectangle. If the members of the package are not shown inside the package rectangle, then the name of the package should be placed inside.

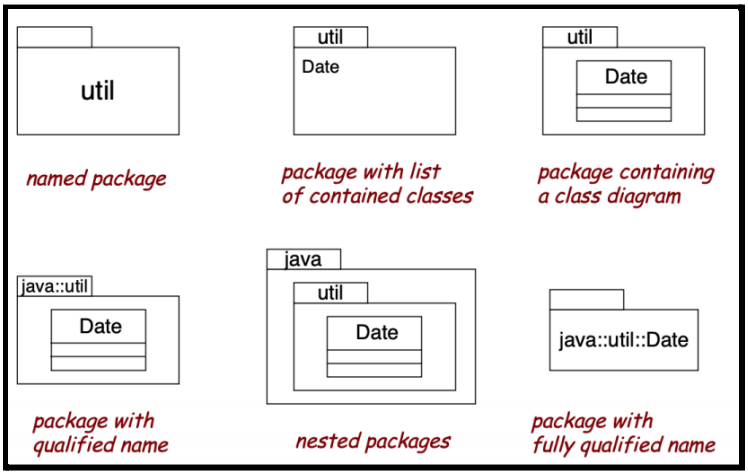
E.g.



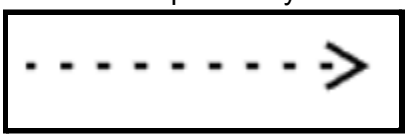
- The members/elements of the package may be shown within the boundaries of the package. If the names of the members of the package are shown, then the name of the package should be placed on the tab.
  E.g.



  Here, Package org.hibernate contains SessionFactory and Session.
- More examples:



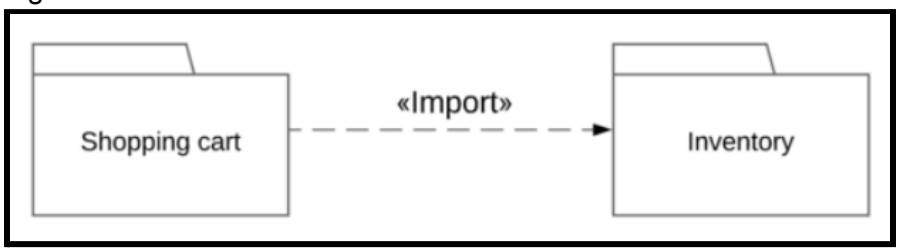
- To show a dependency between 2 packages, you draw a dotted arrow,



, between the 2 packages such that the arrow is pointing to the independent package.

- To show an access dependency, write <<Access>> on the dotted arrow.
  E.g.

- To show an import dependency, write <<Import>> on the dotted arrow.
  E.g.



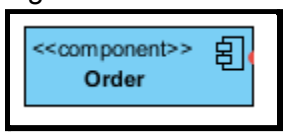
- **Criteria for Decomposing a System into Packages:**
- Different owners - who is responsible for working on which diagrams?
- Different applications - each problem has its own obvious partitions.
- Clusters of classes with strong cohesion - E.g. course, course description, instructor, student, etc.
- Or: Use an architectural pattern to help find a suitable decomposition such as the MVC Framework.
- **Other Guidelines for Packages:**
- Gather model elements with strong cohesion in one package.
- Keep model elements with low coupling in different packages.
- Minimize relationships, especially associations, between model elements in different packages.
- Namespace implication: An element imported into a package does not know how it is used in the imported package.
- We want to avoid dependency cycles.

Component Diagrams:

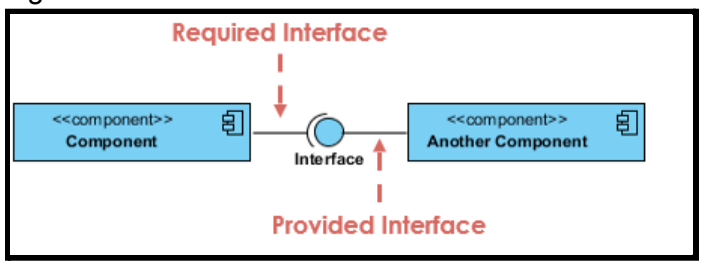
- **Introduction:**
- **Component diagrams** are used in modeling the physical aspects of object-oriented systems that are used for visualizing, specifying, and documenting component-based systems and also for constructing executable systems through forward and reverse engineering.
- A component diagram breaks down the actual system under development into various high levels of functionality.
- Each component is responsible for one clear aim within the entire system and only interacts with other essential elements on a need-to-know basis.
- Component diagrams can help your team:
  - Imagine the system's physical structure.
  - Pay attention to the system's components and how they relate.
  - Emphasize the service behavior as it relates to the interface.
- A component diagram gives a bird's-eye view of your software system. Understanding the exact service behavior that each piece of your software provides will make you a

better developer. Component diagrams can describe software systems that are implemented in any programming language or style.

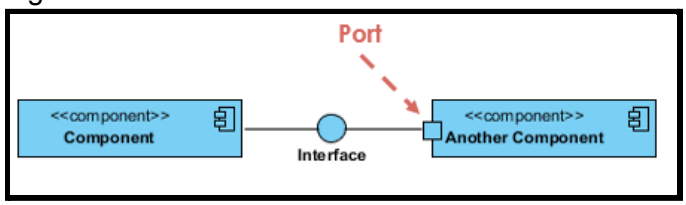
- **Notation:**
- **Component:** A rectangle with the component's name, stereotype text, and icon. A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.
  E.g.



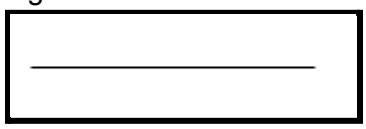
- **Interface:** There are 2 types of interfaces, **provided interface** and **required interface**.
  Provided interface: A complete circle with a line connecting to a component. Provided interfaces provide items to components.
  Required Interface: A half circle with a line connecting to a component. Required interfaces are used to provide required information to a provided interface.
  E.g.



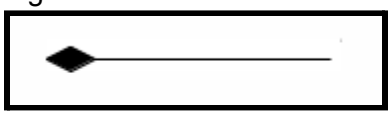
- **Port:** A square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component. Ports are used to hook up other elements in a component diagram.
  E.g.



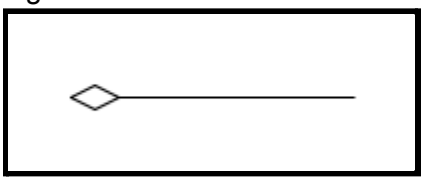
- **Association:** An association specifies a relationship that can occur between two instances. You represent an association using a straight line connecting 2 components.
  E.g.



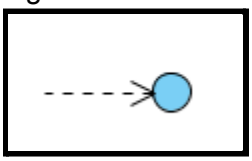
- **Composition:** Composition is a stronger form of aggregation that requires a part instance to be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Composition is a type of association. You can represent a composition using an arrow where the arrowhead is filled in and points to the parent class.
  E.g.



- **Aggregation:** Aggregation implies a relationship where the child class can exist independently of the parent class. This means that if you remove/delete the parent class, the child class still exists. It is a special type of association and a weak form of association. You can represent an aggregation using an arrow where the arrowhead is not filled in and points to the parent class.
  E.g.

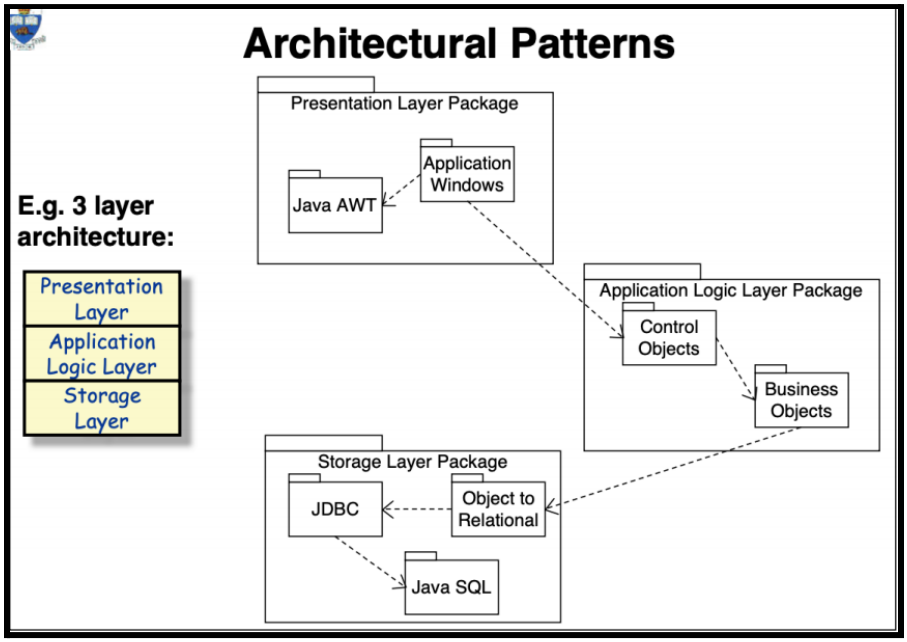


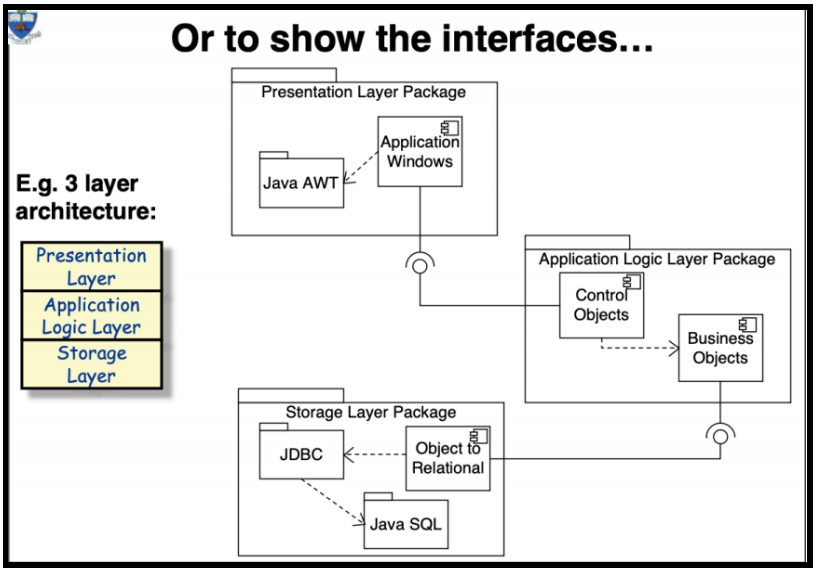
- **Dependency:** A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. It is denoted as a dotted arrow with a circle at the tip of the arrow.
  E.g.



Examples:

- The following 2 UML diagrams show the same thing, a 3-layer application, but the first is a package diagram while the latter is a component diagram.



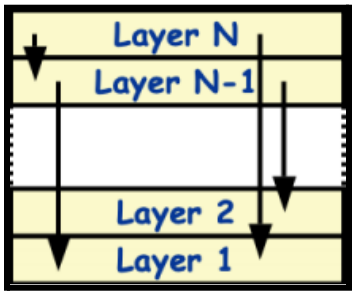


Layered Systems:

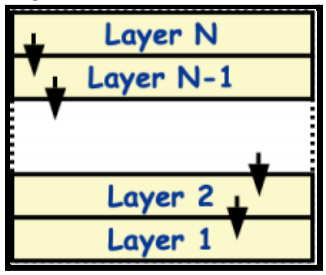
- Examples include operating systems and communication protocols.
- It supports increasing levels of abstraction during design.
- It supports enhancement (adding features and functionalities) and code reuse.
- It can define standard layer interfaces.
- A disadvantage is that it may not be able to define clean layers.

Open vs Closed Layered Architecture:

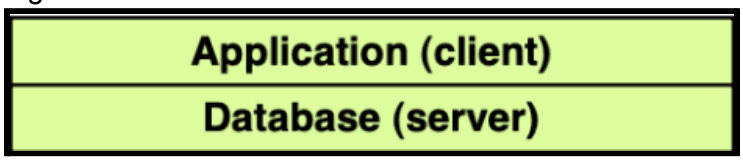
- **Open Layered Architecture:**
- A layer can use services from any lower layer.
- There is more compact code as the services of lower layers can be accessed directly.
- Breaks the encapsulation of layers, so there is an increase in dependency between layers.
- E.g.



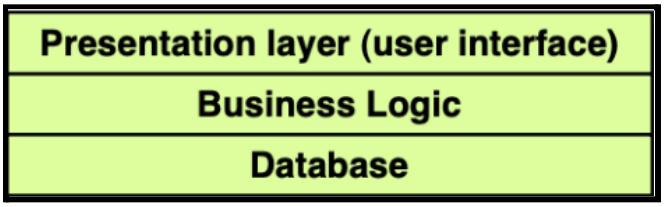
- **Closed Layered Architecture:**
- Each layer only uses services of the layer immediately below it.
- Minimizes dependencies between layers and reduces the impact of changes.
- E.g.



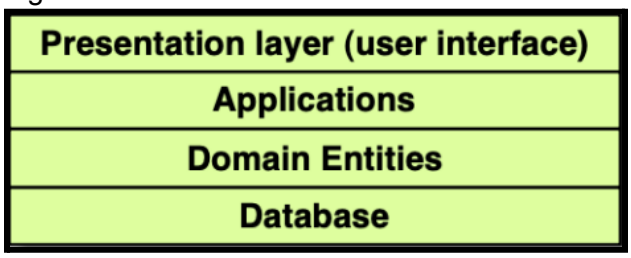
- With 2 layers, you have an application layer and a database layer. An example of a 2-layer architecture is a simple client-server model.
  E.g.



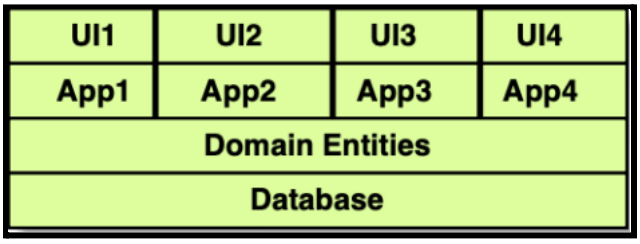
- With 3 layers, you have a presentation layer (UI), a business logic layer and a database layer. Here, we separated the business logic to make the UI and database layers more modifiable.
  E.g.



- With 4 layers, you have a presentation layer (UI), an application layer, a domain entity layer and a database layer. Here, we separated the application from the domain entity.
  E.g.

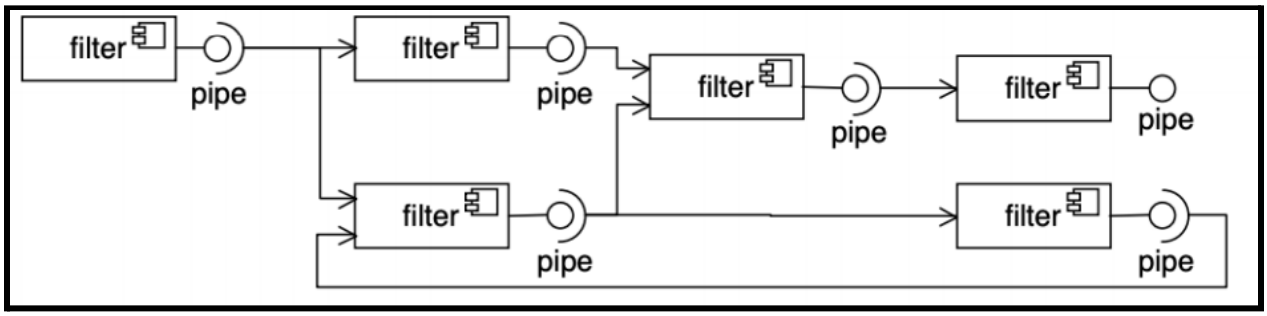


- With partitioned 4 layers, we have separate UIs for each application.
  E.g.



Pipe and Filters:

- Examples include Unix commands, compilers and signal processing.
- Filters don't need to know anything about what they're connected to.
- Filters can be implemented in parallel.
- The behaviour of the system is the composition of the behaviour of the filters.
- UML:



Object Oriented Architecture:

- Examples include abstract data types.
- Has encapsulation and abstraction.
- Can decompose problems into sets of interacting agents.
- Can be single or multi-threaded.
- A disadvantage is that objects must know the identity of the objects they wish to interact with.

Object Brokers:

- A variant of object oriented architecture.
- It adds a broker between the clients and servers.
- Clients no longer need to know which servers they are using.
- It can have many brokers and many servers.
- A disadvantage is that brokers can become bottlenecks which leads to degraded performance.

Event Based:

- Examples include debugging systems (listening for particular breakpoints), database management systems (data integrity checking) and graphical user interfaces.
- The announcers of events don't need to know who will handle the event.
- It supports re-use and evolution of systems and can add new agents easily.
- A disadvantage is that components have no control over ordering of computations.

Repositories:

- Examples include databases, blackboard expert systems and programming environments.
- Can choose where the locus of control is.
- Reduces the need to duplicate complex data.
- A disadvantage is that the blackboard can be a bottleneck.

Model-View-Controller (MVC):

- There is one central model with many viewers/views.
- Each view has an associated controller.
- The controller handles updates from the user of the view.
- Changes to the model are propagated to all the views.

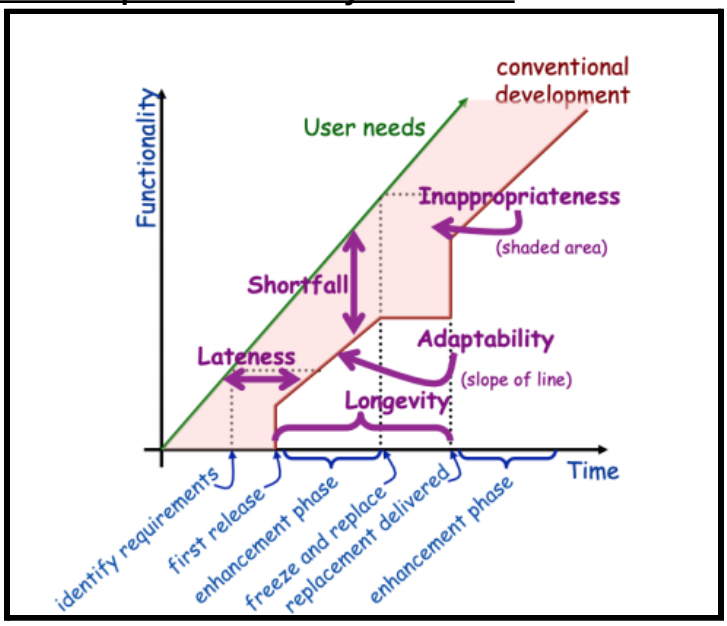
Program Types:

- **S-Type Programs (Specifiable):**
- The problem can be stated formally and completely.
- Acceptance: Is the program correct according to its specification?
- Evolution is not relevant as a new specification defines a new problem.
- S-Type software is one where specification is clear and detailed before the development even begins. Thanks to this detailed specification, it is clear what the solution should be and implementing it is trivial.
- S-programs are programs whose function is formally defined by and derivable from a specification.
- An S-program may be changed to improve its clarity or its elegance, to decrease resource usage when the program is executed, but any such changes must not affect the mapping between the input and output it achieves in execution.
- In S-programs, judgments about the correctness and the value of the programs relate only to its specification.
- E.g. Create a function that adds 2 numbers.
- **P-Type Programs (Problem-solving):**
- Here, we have an imprecise statement of a real-world problem.
- Acceptance: Is the program an acceptable solution to the problem?
- The software may continuously evolve as the solution is never perfect and can always be improved and the real world changes, so the problem changes.
- P-programs are programs for which the problem may be precisely formulated, but for which the solution must inevitably reflect an approximation of the real world.
- E.g. A program to play chess.
- **E-Type Programs (Embedded):**
- Here, the software becomes part of the world that it models.
- Acceptance: Depends entirely on opinion and judgment.
- This software is inherently evolutionary as changes in the software and world affect each other.
- E-programs are those programs that mechanize a human or societal activity.
- The program has become a part of the world it models.
- E.g. Operating Systems, business administration software, inventory management, etc.

Laws of Program Evolution:

- **Continuing Change:**
- Any software that reflects some external reality undergoes continual change or becomes progressively less useful.
- The change continues until it is judged that it is more cost effective to replace the system.
- **Increasing Complexity:**
- As software evolves, its complexity increases unless steps are taken to control it.
- **Fundamental Law of Program Evolution:**
- Software evolution is self-regulating with statistically determinable trends and invariants.
- **Conservation of Organizational Stability:**
- During the active life of a software system, the work output of a development project is roughly consistent, regardless of resources.
- **Conservation of Familiarity:**
- The amount of change in successive releases is roughly constant.

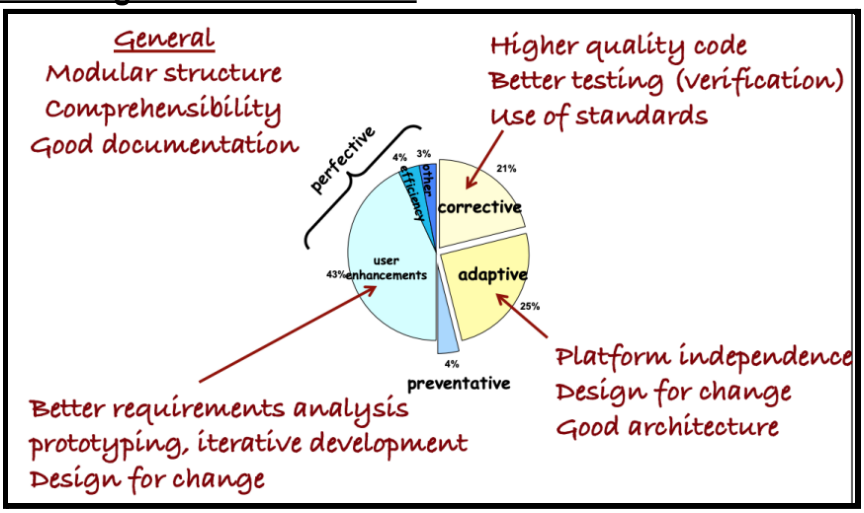
User Requirements Always Increase:



Software Geriatrics:

- **Causes of software aging:**
- Failure to update the software to meet changing needs. Customers will switch to a new product if the benefits outweigh the switching costs.
- Changes to software tend to reduce coherence and increase complexity.
- **Costs of software aging:**
- Owners of aging software may find it hard to keep up with the marketplace.
- Deterioration in space/time performance due to deteriorating structure.
- Aging software gets more buggy and each bug fix adds more errors than it fixes.
- **Ways of increasing longevity:**
- Design for change.
- Document the software carefully.
- Requirements and designs should be reviewed by those responsible for its maintenance.
- Software rejuvenation.

Reducing Maintenance Costs:



Factors That Drive The Cost of Maintaining Software:

- **Adaptive Maintenance:**
  - Accounts for 25% of the total maintenance cost.
  - Arises from modifying the software after its delivery to ensure that the product remains usable in a changing environment.
- **Corrective Maintenance:**
  - Accounts for 20% of the total maintenance cost.
  - Arises from resolving issues you identify during the initial deployment or release.
- **Perfective Maintenance:**
  - Accounts for 5% of the total maintenance cost.
  - Arises from improving software to make it perform efficiently.

Why Maintenance is Hard:

- Poor code quality
- Lack of knowledge of the application domain
- Lack of documentation
- Lack of glamour

Rejuvenation:

- **Reverse Engineering:**
- Includes re-documentation and design recovery.
- **Restructuring:**
- Includes refactoring (no changes to functionality) and revamping (only the ui is changed).
- **Re-Engineering:**
- Real changes are done to the code.
- It is usually done as a round trip:
  Design recovery → Design improvement → Re-implementation

Program Comprehension:

- During maintenance, programmers study the code about 1.5 times as long as the documentation and spend as much time reading code as editing it.
- Experts have many knowledge chucks.
- Experts follow dependency links while novices read sequentially.
- Much of the knowledge comes from outside the code.

Review of UML Diagrams:

- **Uses of UML:**
  - As a sketch:
  - Can be used to sketch a high level view of the system.
  - **Forward engineering:** Describes the concepts we need to implement.
  - **Reverse engineering:** Explains how parts of the code work.
  - As a blueprint:
  - Should be complete and describes the system in detail.
  - **Forward engineering:** Model as a detailed specification for the programmer.
  - **Reverse engineering:** Model as a code browser.
  - Tools provide both forward and reverse engineering to move back and forth between the program and the code.
  - As a programming language:
  - UML diagrams can be automatically compiled into working code using sophisticated tools, such as Visual Paradigm.
- **Things to Model:**
  - Structure of the code:
  - Code dependencies.
  - Components and couplings.
  - Behaviour of the code:
  - Execution traces.
  - State machine models of complex objects.
  - Function of the code:
  - What function does it provide to the user?

Interaction Diagrams:

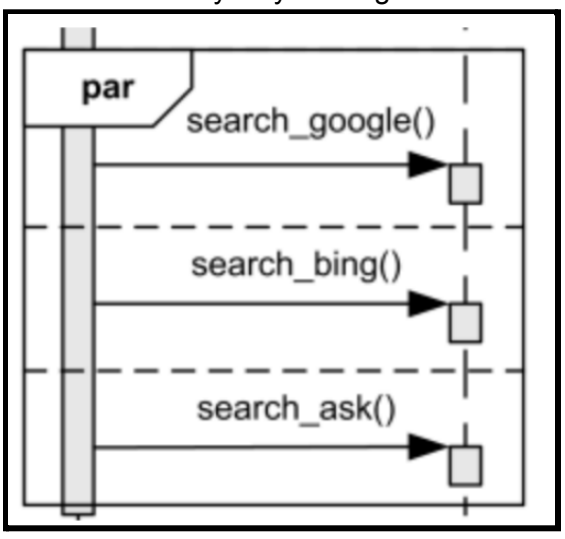
- **Interaction diagrams** describe how a group of objects collaborate in some behavior. They commonly contain objects, links and messages.
- Objects communicate with each other through function/method calls called **messages**.
- An interaction is a set of messages exchanged among a set of objects in order to accomplish a specific goal.
- Interaction diagrams:
  - Are used to model the dynamic aspects of a system.
  - Aid the developer in visualizing the system as it is running.
  - Are storyboards of selected sequences of message traffic between objects.
- After class diagrams, interaction diagrams are possibly the most widely used UML diagrams.
- A **lifeline** represents a single participant in an interaction. It describes how an instance of a specific classifier participates in the interaction. A lifeline represents a role that an instance of the classifier may play in the interaction.
- A **message** is the vehicle by which communication between objects is achieved. A function/method call is the most common type of message. The return of data as a result of a function call is also considered a message.
- A message may result in a change of state for the receiver of the message.
- The receipt of a message is considered an instance of an event.
- Interactions model the dynamic aspects of a system by showing the message traffic between a group of objects. Showing the time-ordering of the message traffic is a central ingredient of interactions.
- Graphically, a message is represented as a directed line that is labeled.
- The **sequence diagram** is the most commonly used UML interaction diagram. Typically a sequence diagram captures the behavior of a group of objects in a single scenario.

Other types of interaction diagrams include communication diagrams and timing diagrams. Out of those three, sequence diagrams are preferred for their simplicity.

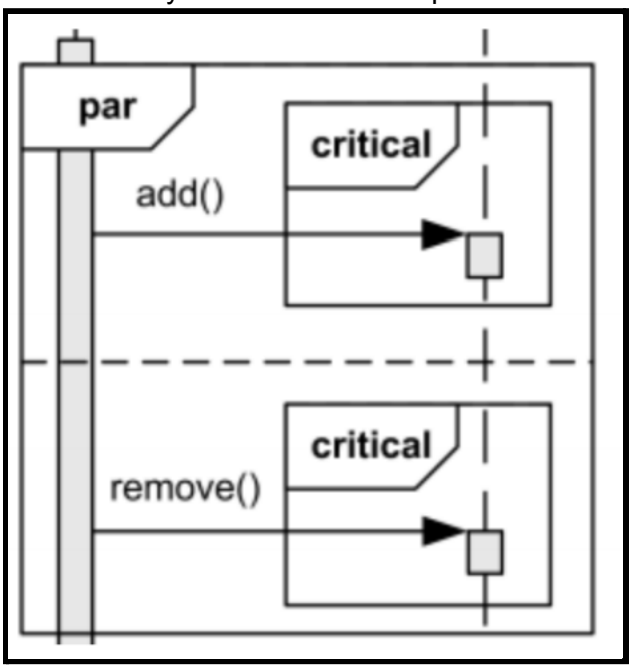
- Interaction Frame Operators:

Operator	**Name**	**Meaning**
Opt	Option	An operand is executed if the condition is true. (E.g. If-else)
Alt	Alternative	The operand, whose condition is true, is executed. (E.g. Switch)
Loop	Loop	It is used to loop an instruction for a specified period.
Break	Break	It breaks the loop if a condition is true or false, and the next instruction is executed.
Ref	Reference	It is used to refer to another interaction.
Par	Parallel	All operands are executed in parallel.
Region	Critical Region	Only 1 thread can execute this frame at a time.
Neg	Negative	Frame shows an invalid interaction.
Sd	Sequence Diagram	(Optional) Used to surround the whole diagram.

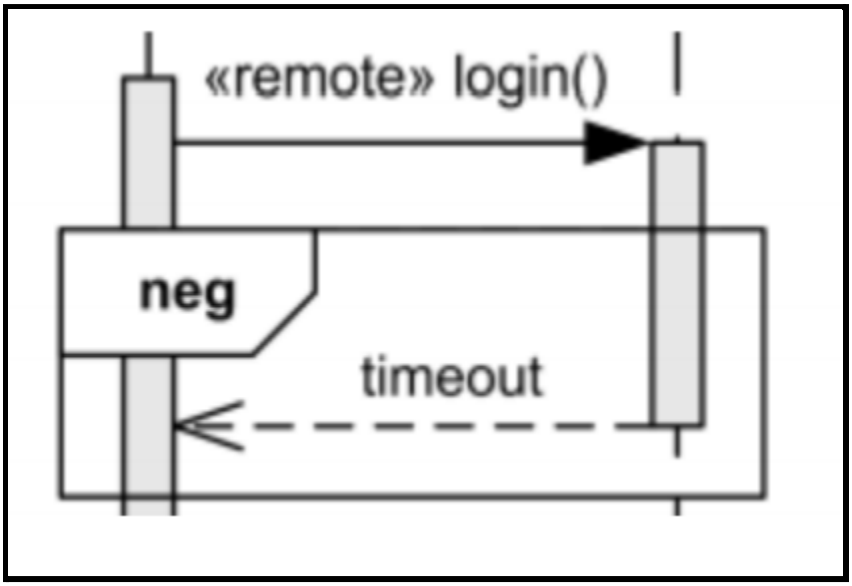
- **Parallel Example:** The interaction operator par defines potentially parallel execution of behaviors of the operands of the combined fragment. Different operands can be interleaved in any way as long as the ordering imposed by each operand is preserved.



- **Region Example:** The interaction operator region defines that the combined fragment represents a critical region. A critical region is a region with traces that cannot be interleaved by other occurrence specifications on the lifelines covered by the region.

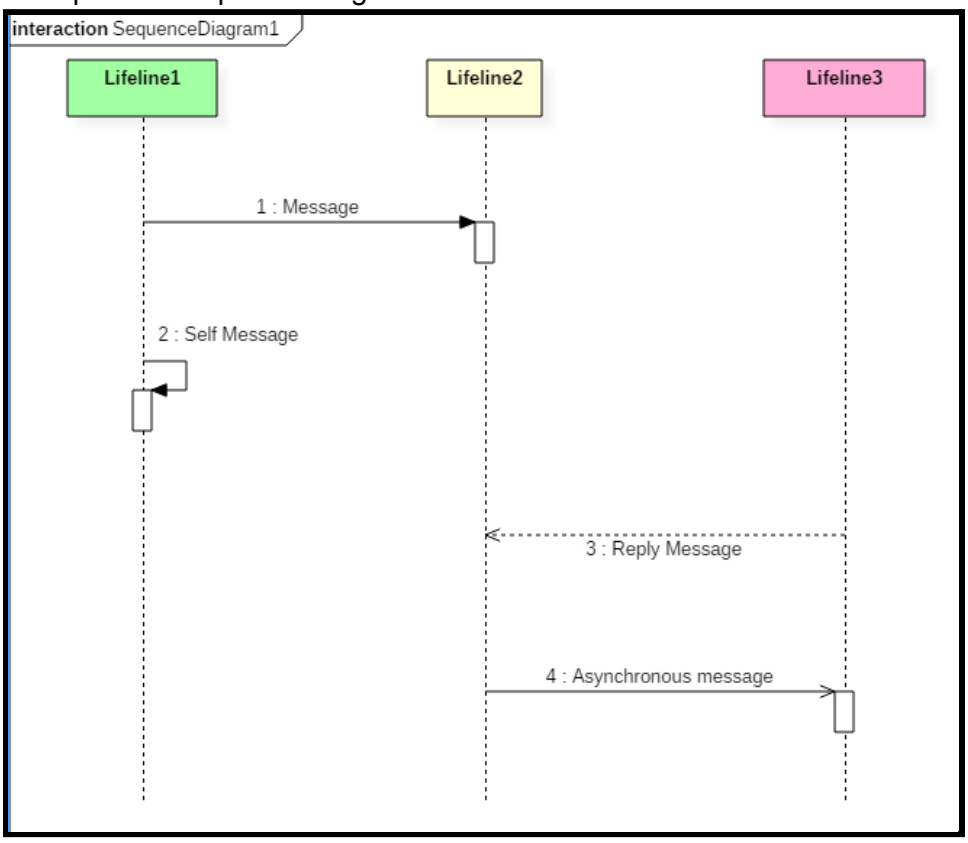


- **Negative Example:** The interaction operator neg describes a combined fragment of traces that are defined to be negative (invalid). Negative traces are the traces which occur when the system has failed. All interaction fragments that are different from the negative are considered positive, meaning that they describe traces that are valid and should be possible.

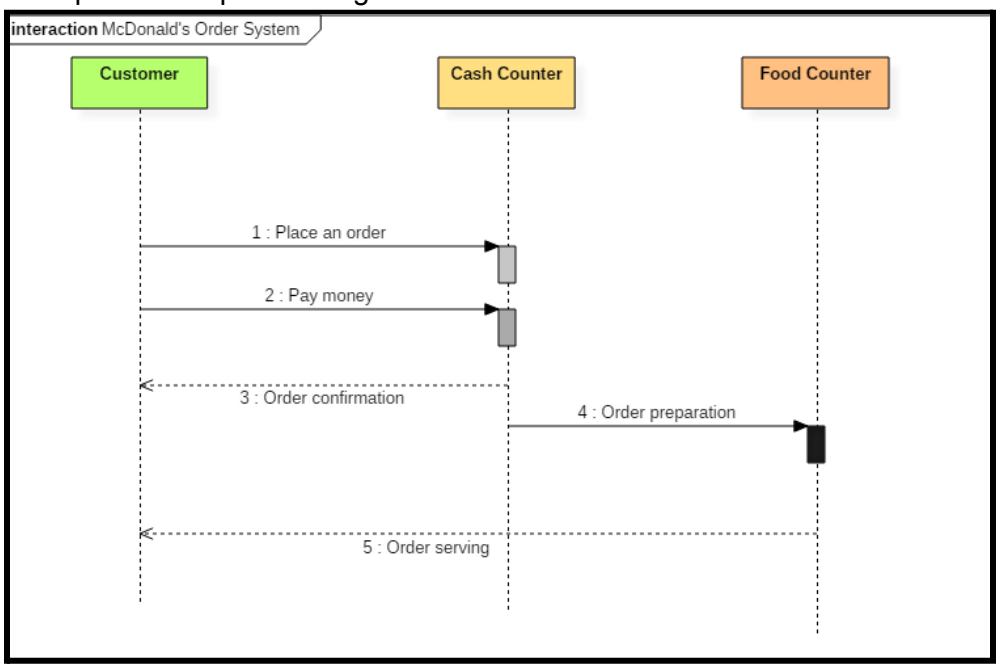


Sequence Diagrams:

- **Introduction:**
- A **sequence diagram** depicts interactions between objects in a sequential order. The purpose of a sequence diagram in UML is to visualize the sequence of a message flow in the system. The sequence diagram shows the interaction between two lifelines as a time-ordered sequence of events.
- A sequence diagram shows an implementation of a scenario in the system. Lifelines in the system take part during the execution of a system.
- In a sequence diagram, a lifeline is represented by a vertical bar.
- A message flow between two or more objects is represented using a vertical dotted line which extends across the bottom of the page.
- Sequence diagrams are built around an X-Y axis.
- Objects are aligned at the top of the diagram, parallel to the X axis.
- Messages travel parallel to the X axis.
- Time passes from top to bottom along the Y axis.
- Sequence diagrams most commonly show relative timings, not absolute timings.
- Links between objects are implied by the existence of a message.
- Example of a sequence diagram:



- Example of a sequence diagram:



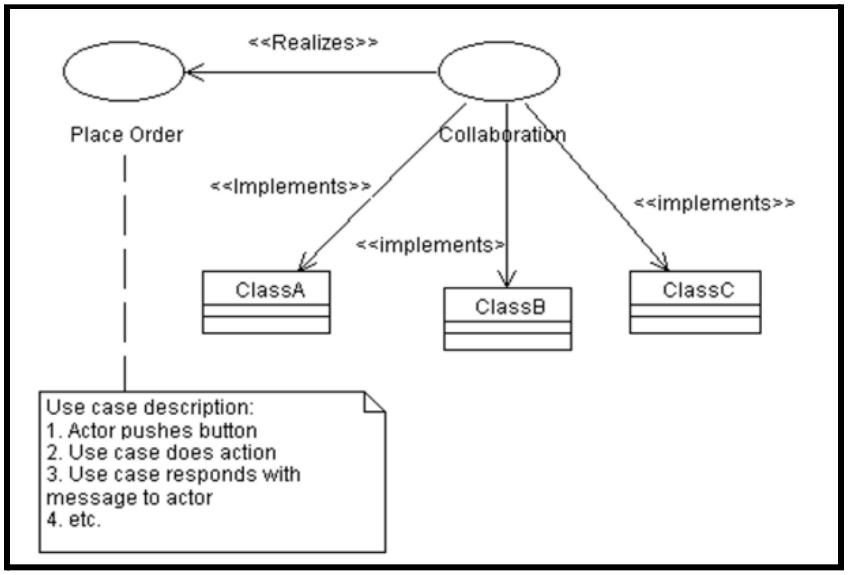
- **Benefits of a sequence diagram:**
- Sequence diagrams are used to explore any real application of a system.
- Sequence diagrams are used to represent the message flow from one object to another.
- Sequence diagrams are easy to maintain and generate.
- Sequence diagrams can be easily updated according to the changes within a system.
- Sequence diagrams allow both reverse and forward engineering.
- **Drawbacks of a sequence diagram:**
- Sequence diagrams can become complex when too many lifelines are involved in the system.
- If the order of message sequence is changed, then incorrect results are produced.
- Each sequence needs to be represented using different message notation, which can be a little complex.
- The type of message decides the type of sequence inside the diagram.
- **When to use sequence diagrams:**
    1. Comparing Design Options:
        - Shows how objects collaborate to carry out a task.
        - Graphical form shows alternative behaviours.
    2. Assessing Bottlenecks
    3. Explaining Design Patterns:
        - Enhances structural models.
        - Good for documenting behaviour of design features.
    4. Elaborating Use Cases:
        - Shows how the user expects to interact with the system.
        - Shows how the user interface operates.
- **Modelling Control Flow By Time:**
- Determine what scenarios need to be modeled.
- Identify the objects that play a role in the scenario.
- Lay the objects out in a sequence diagram left to right, with the most important objects on the left.
  Most important in this context means objects that are the principle initiators of events.

- Draw in the message arrows, top to bottom.
  Adorn the message as needed with detailed timing information.
- **Style Guide for Sequence Diagrams:**
  1. Spatial Layout:
     - Strive for left-to-right ordering of messages.
     - Put proactive actors on the left.
     - Put reactive actors on the right.
  2. Readability:
     - Keep diagrams simple.
     - Don't show obvious return values.
     - Don't show object destruction.
  3. Usage:
     - Focus on critical interactions only.
  4. Consistency:
     - Class names must be consistent with class diagram.
     - Message routes must be consistent with navigable class associations.

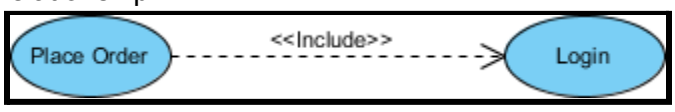
Use Case Diagrams:

- **Introduction:**
- A **use case diagram** is the primary form of system/software requirements for a new software program.
- Use cases specify the expected behavior (what), and not the exact method of making it happen (how).
- A key concept of use case modeling is that it helps us design a system from the end user's perspective. It is an effective technique for communicating a system's behavior in the user's terms.
- Use case diagrams are used to gather the requirements of a system including internal and external influences.
- A use case:
  - Specifies the behavior of a system or some subset of a system.
  - Is a set of scenarios tied together by a common user goal.
  - Does not indicate how the specified behavior is implemented, only what the behavior is.
  - Performs a service for some user of the system, called an **actor**.
- A use case represents a functional requirement of the system. A requirement:
  - Is a design feature, property, or behavior of a system.
  - States what needs to be done, but not how it is to be done.
  - Is a contract between the customer and the developer.
  - Can be expressed in various forms, including use cases.
- In brief, the purposes of use case diagrams are as follows:
  - Used to gather the requirements of a system.
  - Used to get an outside view of a system.
  - Identify the external and internal factors influencing the system.
  - Show the interaction among the requirements of the actors.
- An actor:
  - Is a role that the user plays with respect to the system. The user does not have to be human.
  - Is associated with one or more use cases.
  - Is most typically represented as a stick figure of a person labeled with its role name. Note that the role names should be nouns.
  - May exist in a generalization relationship with other actors in the same way as classes may maintain a generalization relationship with other classes.

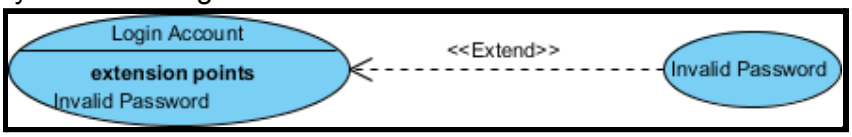
- **Note:** Use cases diagrams do not show the order in which the steps are performed to achieve the goals of each use case. It only shows the relationship between actors, systems and use cases.
- Use cases are a technique for capturing the functional requirements of a system. Use cases work by describing the typical interactions between the users of a system and the system itself, providing a narrative of how the system is used.
- Use case development process:
    1. Develop multiple scenarios.
    2. Distill the scenarios into one or more use cases where each use case represents a functional requirement.
    3. Establish associations between the use cases and actors.
- A use case is graphically represented as an oval with the name of its functionality written inside. The functionality is always expressed as a verb or a verb phrase.
- A use case may exist in relationships with other use cases much in the same way as classes maintain relationships with other classes.
- As stated earlier, a use case by itself does not describe the flow of events needed to carry out the use case. The flow of events can be described using informal text, pseudocode, or activity diagrams.
  I.e. You can attach a note to a use case to show the flow of the event. Be sure to address exception handling when describing the flow of events.
  E.g.



- **Relationships Between Use Cases:**
- A use case may have a relationship with other use cases.
- Generalization between use cases is used to extend the behavior of a parent use case.
- An **<<include>> relationship** between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base.
  Note: Sometimes **<<uses>>** is used instead of **<<include>>**.
  When a use case is depicted as using the functionality of another use case, the relationship between the use cases is named as an **<<include>>** or **<<uses>>** relationship.



- An **<<extend>> relationship** between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case.



- Extended behavior is optional behavior, while included behavior is required behavior. I.e. Extended means “may use” while include/uses means “will use”.
- Extend occurs when one use case adds a behaviour to a base use case while include occurs when one use case invokes another.
- **Actor Classes:**
- Identify classes of actors.
- Actors inherit use cases from the class.
- **Describing Use Cases:**
- For each use case, a flow of events document, written from the actor’s point of view, describes what the system must provide to the actor when the use case is executed.
- Typical contents:
    - How the use case starts and ends.
    - Normal flow of events.
    - Alternate flow of events.
    - Exceptional flow of events.
- Documentation style:
    - Activity Diagrams - Good for business process.
    - Collaboration Diagrams - Good for high level design.
    - Sequence Diagrams - Good for detailed design.
- **Finding Use Cases:**
- Noun phrases may be domain classes.
- Verb phrases may be operations and associations.
- Possessive phrases may indicate attributes.
- **For each actor, ask the following questions:**

1. What functions does the actor require from the system?
2. What does the actor need to do?
3. Does the actor need to read, create, destroy, modify or store information in the system?
4. Does the actor have to be notified about events in the system?
5. Does the actor need to notify the system about something?
6. What do these events require in terms of system functionality?
7. Could the actor’s daily work be simplified or made more efficient through new functions provided by the system?

Relationship with Customers:

- Customer Specific:
  - One customer with a specific problem.
  - The customer may be another company with a contractual agreement or a division within the same company.
- Market-based:
  - Selling the product to a general market.
  - In some cases, the product must generate customers.
  - The marketing team may act as a substitute customer.
- Community-based:
  - Intended as a general benefit to some community.
  - Examples include open-source tools and tools for scientific research.
- Hybrid:
  - A mix of the above.

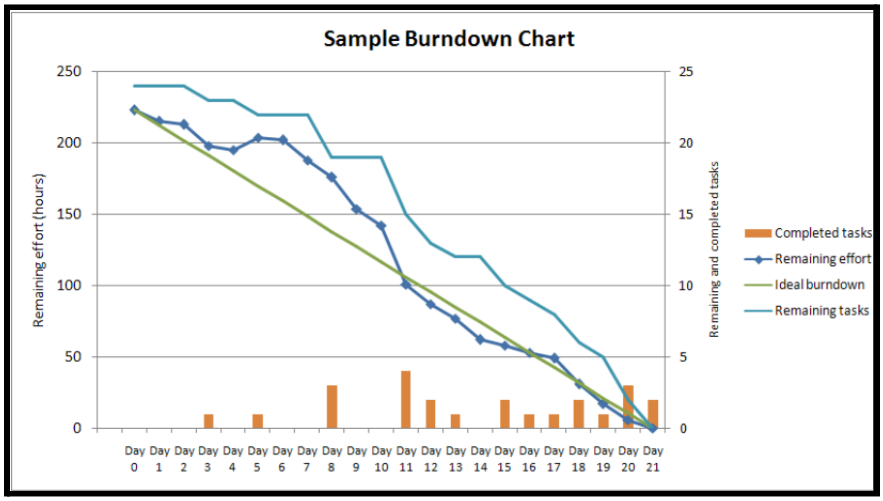
Project Planning:

- **Parts of Project Planning:**
- Given:
  - A list of customer requirements.
  - Examples include a set of use cases or a set of change requests.
- Estimate:
  - How long each one will take to implement (cost).
  - How important each one is (value).
- Plan:
  - Which requests should be included in the next release.
- Complication:
  - Customers care about other stuff, such as security and quality, too.
- **Key Principles of Management:**
- A manager can control 4 things:
  1. Resources - Can get more money, personnel, etc
  2. Time - Can vary the schedule and delay milestones
  3. Product - Can vary the amount of functionality
  4. Risk - Can decide which risks are acceptable
- Approach:
  - Understand the goals and objectives.
  - Understand the constraints.
  - Plan to meet the objectives within the constraints.
  - Monitor and adjust the plan.
  - Preserve a calm, productive, positive work environment.
- **Note:** You cannot control what you cannot measure.
- **Strategies:**
- Fixed Product:
  1. Identify the customer requirements.
  2. Estimate the size of software needed to meet them.
  3. Calculate the time required to build the software.
  4. Get the customer to agree to the cost and schedule.
- Fixed Schedule:
  1. Fix a date for the next release.
  2. Obtain a prioritized list of requirements.
  3. Estimate the effort for each requirement.
  4. Select requirements from the list until there's no more left or you don't think you can work on more tasks.

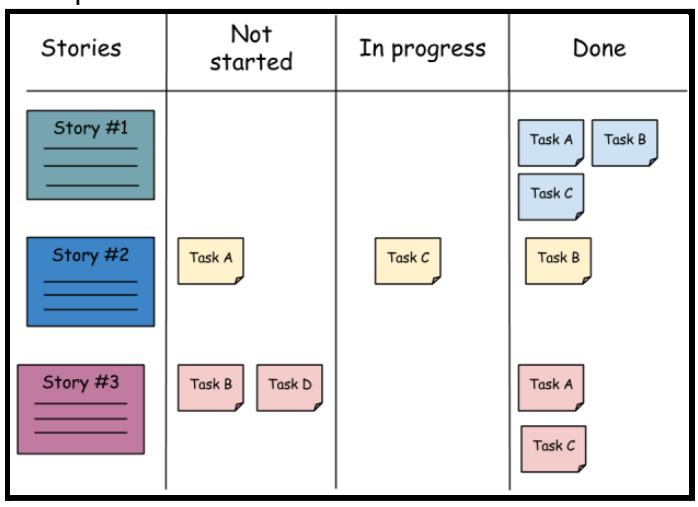
- Fixed Cost:
    1. Agree with the customer on how much they wish to spend.
    2. Obtain a prioritized list of requirements.
    3. Estimate the cost for each requirement.
    4. Select requirements from the list until the “cost” is used up.
- **Estimating Effort - Constructive Cost Model (COCOMO):**
- Predicts the cost of a project from a measure of size (lines of code).
- The basic model is: $E = aL^b$
  E = effort
  a & b = project specific folders
  L = lines of code
- Modelling process:
    1. Establish the type of project (organic, semidetached, embedded, etc). This gives sets of values for a and b.
    2. Identify the component modules and estimate L for each module.
    3. Adjust L according to how much code is reused.
    4. Compute E using the formula above.
    5. Adjust E according to the difficulty of the project.
    6. Compute time using $T = cE^d$, where c and d are provided for different project types, like a and b.
- **Estimating Size - Function Points:**
- Used to calculate the size of software from a statement of the problem.
- Tries to address the variability in lines of code estimates used in models such as COCOMO.
- Basic model is: $FP = a_1I + a_2O + a_3E + a_4L + a_5F$
- Each a_i is a weighting factor for their respective metric.
- I is the number of user inputs (data entry).
- O is the number of user outputs (reports, screens, error messages).
- E is the number of user queries.
- L is the number of files.
- F is the number of external interfaces.
- **Three-Point Estimating:**
- W = worst possible case
- M = Most likely case
- B = Best possible case
- $E = \sum_i \frac{Wi + 4Mi + Bi}{6}$
- **Story Points:**
- We need a common way to compare story sizes.
- It can be hard to find common ground between a programming story and a database management story.
- **Story points** are a relative measure of a feature’s size or complexity.
  They are not durations nor a commitment to when a story will be completed.
  Different teams have different velocities, so they may complete stories at different rates depending on experience.
- A good tool to do the estimation is **planning poker**. It is a series similar to the Fibonacci Series that can be a useful range for story points. Here, each number is almost the sum of the two preceding numbers: 0, 1, 2, 3, 5, 8, 13, 20, 40, 100.

- 0-points estimates are used for trivial tasks that require little effort, though too many zero-pointers can add up.
- Only use numbers within the set and avoid averages. We avoid averages because if a user story turns out to be harder than expected, then the people who picked a higher number will say “I told you” to the people who picked a lower number. The average does not convince other people.
- What happens is this:
    1. A feature is mentioned.
    2. Each person in the team takes a number from the set, but doesn't show/tell anyone else yet.
       They choose the number based on how difficult they think implementing the feature will be.
       A feature with point 0 means that it requires very little effort.
    3. After 3 seconds, everyone shows their number.
    4. If everyone or most people have the same number, it’s good.
    5. If everyone or most people have different numbers, then each person has to defend why they picked their number.
       Once the discussion has been carried out, there is a second round of voting.
    6. The process repeats until everyone agrees.
       If people consistently do not agree with one another, then the user story is not a good user story. This is because one of the features of a user story is that it must be estimable.
- Powers of 2 is also an effective tool to do the estimation.
- **Ideal Days:**
- Another unit of measure. It can be used as a transition for teams that are new to agile.
- It represents an ideal day of work with no interruptions (phone calls, questions, broken builds, etc.)
  However, it doesn’t mean an actual day of work to finish.
- Tasks are estimated in hours.
  An estimation is an ideal time (without interruptions/problems).
  Smaller task estimates are more accurate than large.
- After all tasks have been estimated, the hours are totaled up and compared against the remaining hours in the sprint backlog. If there is room, the PBI is added and the team commits to completing the PBI.
  If the new PBI overflows the sprint backlog, the team does not commit and
    - the PBI can be returned to the product backlog and a smaller PBI chosen instead or
    - we can break the original PBI into smaller chunks or
    - we can drop an item already in the backlog to make room or
    - the product owner can help decide the best course of action.
- **Tracking Progress:**
- Information about progress, impediments and sprint backlog of tasks needs to be readily available.
- How close a team is to achieving their goals is also important.
- Scrum employs a number of practices for tracking this information:
    1. Task cards
    2. Burndown charts
    3. Task boards
    4. War rooms (standups)

- **Burndown Charts:**
- Example of a burndown chart:



- Indicates how much work has been done in terms of how many user stories have been completed and when.
- Burndown charts are on Jira.
- On the beginning of the sprint, on the vertical axis, is the number of user stories you'd like to implement.
- At the end of the sprint, the number of user stories should be decreased to 0. That means all the stories have been implemented.
- A burndown chart is a graphical representation of work left to do versus time. It is often used in agile software development methodologies such as Scrum.
- Typically, in a burndown chart, the outstanding work is often on the vertical axis, with time along the horizontal. It is useful for predicting when all of the work will be completed.
- **Taskboard:**
- Example of a taskboard:



- Both taskcards and taskboards are on Jira.
- The leftmost column are the stories to be implemented and there are 3 columns describing the progress of the tasks.

- In scrum the **task board** is a visual display of the progress of the scrum team during a sprint. It presents a snapshot of the current sprint backlog allowing everyone to see which tasks remain to be started, which are in progress and which are done.
- Simply put, the task board is a physical board on which the user stories which make up the current sprint backlog, along with their constituent tasks, are displayed. Usually this is done with index cards or post-it notes.
- The task board is usually divided into the columns listed below.
  Stories: This column contains a list of all the user stories in the current sprint backlog.
  Not started: This column contains sub tasks of the stories that work has not started on.
  In progress: All the tasks on which work has already begun.
  Done: All the tasks which have been completed.

Risk Management:

- **Introduction to Risk:**
- **Risk** is the possibility of suffering loss.
- Risk itself is not bad; it's essential to progress.
- The challenge is to manage the amount of risk.
- **Risk Exposure (RE):**
- RE = Probability of risk occurring * Total loss if risk occurs
- Calculates the effective current cost of a risk and can be used to prioritize risks that require countermeasures.
  I.e. Can help find which countermeasures work best.
- Higher RE means more serious risk.
- **Risk Reduction Leverage (RRL):**
- RRL = $\frac{Reduction\ in\ Risk\ Exposure}{Cost\ of\ the\ countermeasure}$
- Calculates a value for the return on investment for a countermeasure and can be used to prioritize possible countermeasures.
- Higher RRL indicates more cost-effective countermeasures.
- **Risk Assessment:**
- Quantitative:
  - Measures risk exposure using standard cost and probability measures.
  - **Note:** Probabilities are rarely independent.
- Qualitative:
  - Create a risk exposure matrix.
  - E.g. for NASA

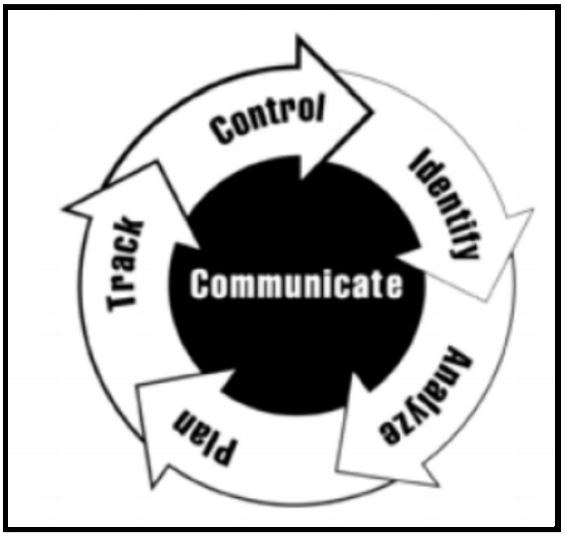
		Likelihood of Occurrence		
		Very likely	Possible	Unlikely
Undesirable outcome	(5) Loss of Life	Catastrophic	Catastrophic	Severe
	(4) Loss of Spacecraft	Catastrophic	Severe	Severe
	(3) Loss of Mission	Severe	Severe	High
	(2) Degraded Mission	High	Moderate	Low
	(1) Inconvenience	Moderate	Low	Low

- **Top Software Engineering Risks With Countermeasures:**

Risks	**Countermeasures**
Personnel Shortfalls	Use top talent. Team building. Training.
Unrealistic schedule/budget	Multisource estimation. Designing to cost. Requirements scrubbing.
Developing the wrong software function	Better requirements analysis. Organizational/Operational analysis.
Developing the wrong user interface	Prototypes, scenarios, task analysis.
Gold plating **Gold plating** is the phenomenon of working on a project or task past the point of diminishing returns. For example, after having met the requirements, the project manager or the developer works on further enhancing the product, thinking the customer will be delighted to see additional or more polished features, rather than what was asked for or expected. The customer might be disappointed in the results, and the extra effort by the developer might be futile.	Requirements scrubbing. Cost benefit analysis. Designing to cost.
Continuing stream of requirement changes	High change threshold. Information hiding. Incremental development.
Shortfalls in externally furnished components	Early benchmarking. Inspections, compatibility analysis.
Shortfalls in externally performed tasks	Pre-award audit. Competitive designs.
Real-time performance shortfalls	Targeted analysis. Simulations, benchmarks, models.
Straining computer science capabilities	Technical analysis. Checking scientific literature.

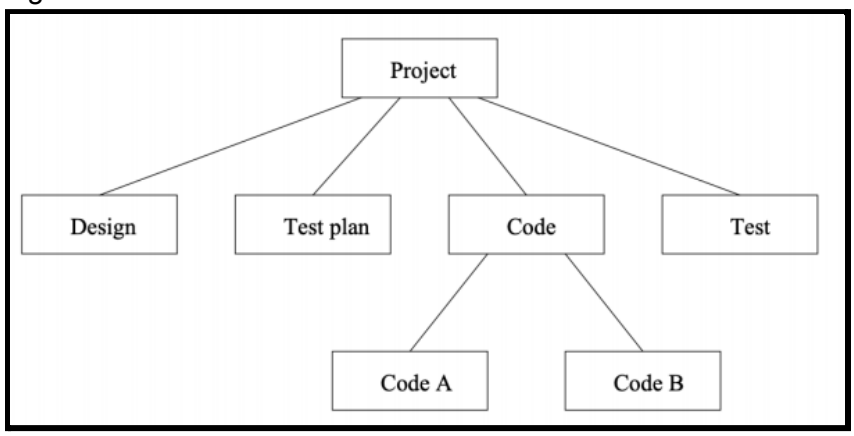
- **Principles of Risk Management:**
- Global perspective:
    - View software in the context of a larger system.
    - For any opportunity, identify both potential value and potential impacts of adverse results.
- Forward looking view:
    - Anticipate possible outcomes.
    - Identify uncertainty and manage resources accordingly.

- Open communications:
    - Free-flow information at all project levels.
    - Value the individual voice. Everyone has unique knowledge and insights.
- Integrated management:
    - Project management is risk management.
- Continuous process:
    - Continually identify and manage risks.
    - Maintain constant vigilance.
- Shared product vision:
    - Everyone understands the mission.
    - Focus on results.
- Teamwork:
    - Work cooperatively to achieve the common goal.
- **Continuous Risk Management:**
- Control:
    - Correct for deviations from the risk mitigation plans.
- Identify:
    - Search for and locate risks before they become problems.
    - Use systematic techniques to discover risks.
- Analyze:
    - Transform risk data into decision-making information.
    - For each risk, evaluate:
        - Probability
        - Impact
        - Timeframe
    - Classify and prioritize risks.
- Plan:
    - Choose risk mitigation actions.
- Track:
    - Monitor risk indicators.
    - Reassess risk.
- Communicate:
    - Share information on current and emerging risks.

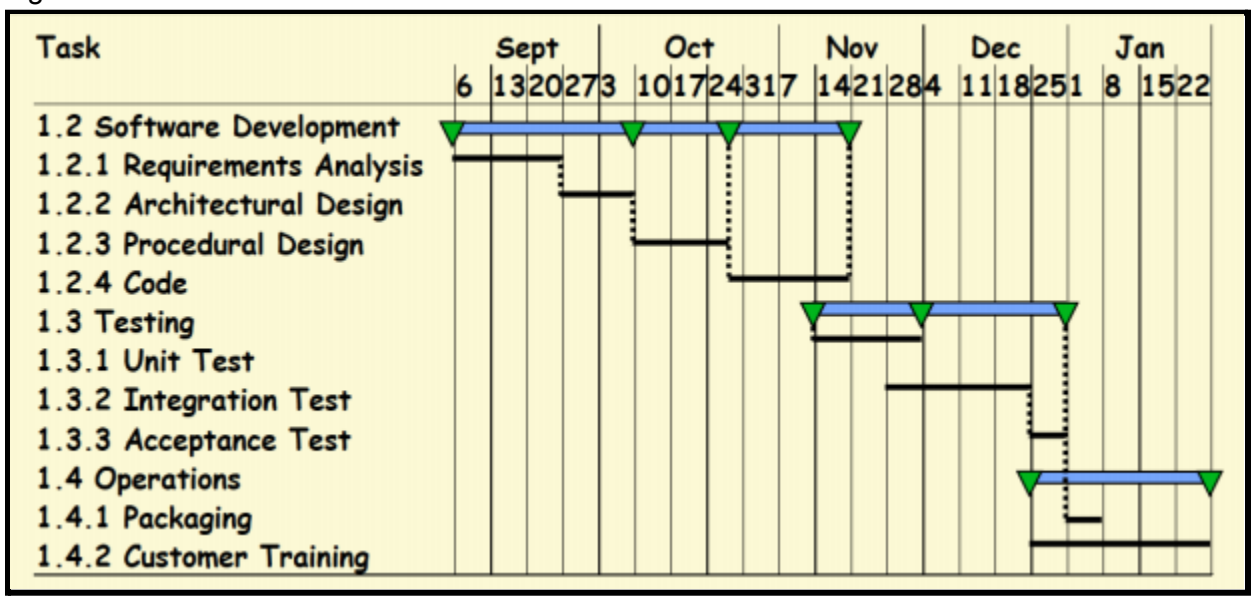


Project Management:

- **Project Management Tools:**
- Work Breakdown Structure:
- A **work breakdown structure (WBS)** is a hierarchical decomposition of the total scope of the work to be carried out by the project team to accomplish the project objectives and create the required deliverables.
- E.g.



- Gantt Charts:
- A **Gantt chart** is a bar chart that provides a visual view of the tasks scheduled over time. It is used for planning projects of all sizes, and it is a useful way of showing what work is scheduled to be done on a specific day. It can also help you view the start and end dates of a project in one simple chart.
- E.g.

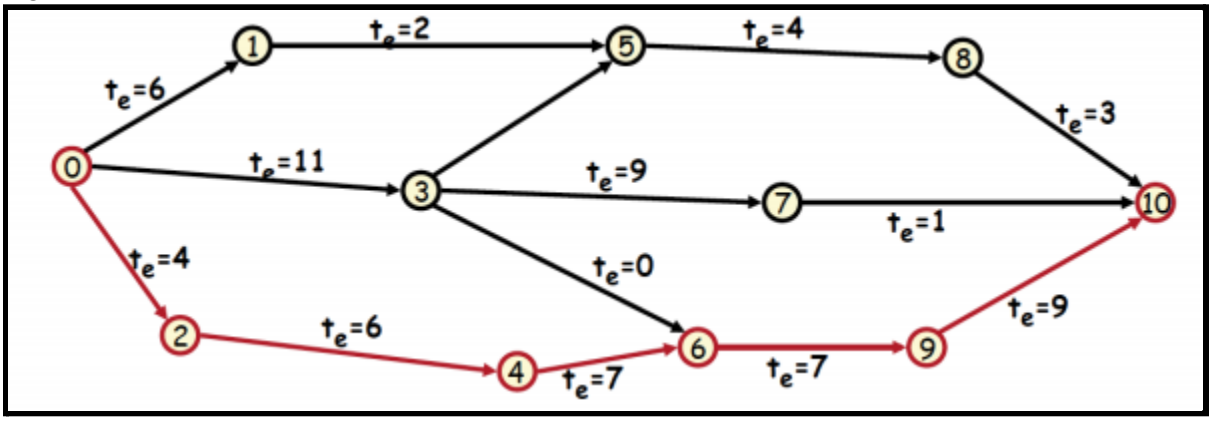


- Notations:
  - Bars show the duration of tasks.
  - Triangles show milestones.
  - Vertical lines show dependencies.

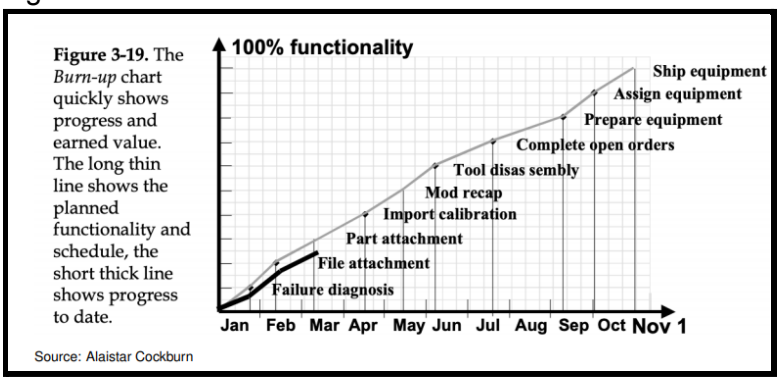
- PERT Charts:
- **PERT** stands for Program Evaluation and Review Technique. A PERT chart illustrates a project as a network diagram. The U.S. Navy created this tool in the 1950s as they developed the Polaris missile.
- Project managers create PERT charts to gauge the minimum time necessary to complete the project, analyze task connections, and assess project risk. PERT charts make it easy to visualize and organize complex projects by illustrating the dependencies between each step in the project.
- PERT charts are best utilized by project managers at the beginning of a project to ensure that it is accurately scoped. This tool gives users a birds-eye view of the entire project before it's started to avoid potential bottlenecks. While PERT charts can be used during the project's implementation to track progress, they are not flexible enough for teams to adapt them to small changes when team members are confronted with roadblocks.
- Advantages of using PERT charts
    - A PERT chart allows managers to evaluate the time and resources necessary to manage a project. This evaluation includes the ability to track required assets during any stage of production in the course of the entire project.
    - PERT analysis incorporates data and information from multiple departments. This combining of information encourages department responsibility and it identifies all responsible parties across the organization. It also improves communication during the project and it allows an organization to commit to projects that are relevant to its strategic positioning.
    - Finally, PERT charts are useful for what-if analyses. Understanding the possibilities concerning the flow of project resources and milestones allows management to achieve the most efficient and useful project path.
- Disadvantages of using PERT charts:
    - The use of a PERT chart is highly subjective and its success depends on the management's experience. These charts can include unreliable data or unreasonable estimates for cost or time for this reason.
    - PERT charts are deadline-focused and they might not fully communicate the financial positioning of a project. Because a PERT chart is labor-intensive, the establishment and maintenance of the information require additional time and resources. Continual review of the information provided, as well as the prospective positioning of the project, is required for a PERT chart to be valuable.
- A **critical path** is the sequential activities from start to the end of a user story. Although many user stories only have one critical path, some may have more than one critical paths depending on the flow logic used in the user story implementation.
- A critical path is determined by identifying the longest stretch of dependent activities and measuring the time required to complete them from start to finish.
- If there is a delay in any of the activities under the critical path, there will be a delay of the user story delivery.
- Key steps in a critical path method:
    1. **Activity specification:**
    - Break down a user story in a list of activities.
    2. **Activity sequence establishment:**
    - Need to ask three questions for each task of your list.
        1. Which tasks should take place before this task happens?
        2. Which tasks should be completed at the same time as this task?
        3. Which tasks should happen immediately after this task?

3. **Network diagram:**
   - Once the activity sequence is correctly identified, the network diagram can be drawn.
4. **Identify critical path:**
   - The critical path is the longest path of the network diagram. If an activity of this path is delayed, the user story will be delayed.

- Tasks on the critical path have to start as early as possible or else the whole project will be delayed. However tasks not on the critical path have some flexibility on when they are started. This flexibility is called the **slack time**.
- E.g.



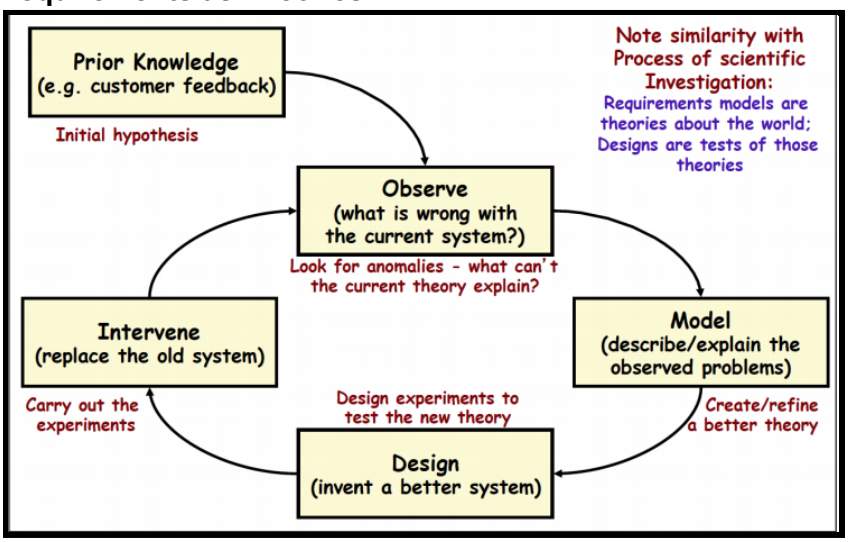
- Notation:
  - Nodes indicate milestones.
  - Edges indicate dependencies and are labelled with the time to complete it.
- Burn-up Charts:
- A **burnup chart** is a tool used to track how much work has been completed, and show the total amount of work for a project or iteration. Typically, in a burnup chart, the outstanding work is often on the vertical axis, with time along the horizontal. It is useful for predicting when all of the work will be completed.
- Burnup charts and burndown charts are quite similar and display much of the same information. Burndown charts are simple and easy for project members and clients to understand. A line representing the remaining project work slowly decreases and approaches zero over time. However, this type of chart doesn't show clearly the effects of scope change on a project. If a client adds work mid-project the scope change would appear as negative progress by the development team on a burndown chart.
- In contrast, scope changes are immediately evident on burnup charts. When new work is added the total work line will clearly show the increase in scope and total work.
- E.g.



- Meetings as a management tool:
- Meetings are expensive, so don't waste people's time and the company's money with unnecessary meetings. However, meetings are necessary for communications.
- Do's/Don'ts for meetings:
    - Do announce details in advance.
      I.e. The purpose of the meeting, the duration of the meeting, who should join, etc.
    - Do lay out a clear and concise agenda for the meeting.
    - Do identify a person who will lead the meeting.
    - Do identify a person who will take notes for the meeting.
    - Do not waste people's time by showing up unprepared (especially if you're in charge).
    - Do not let discussion get sidetracked.
    - Do not let one one or two people dominate the meeting.

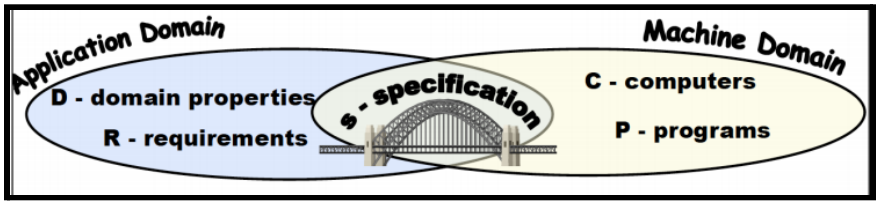
Requirement Analysis:

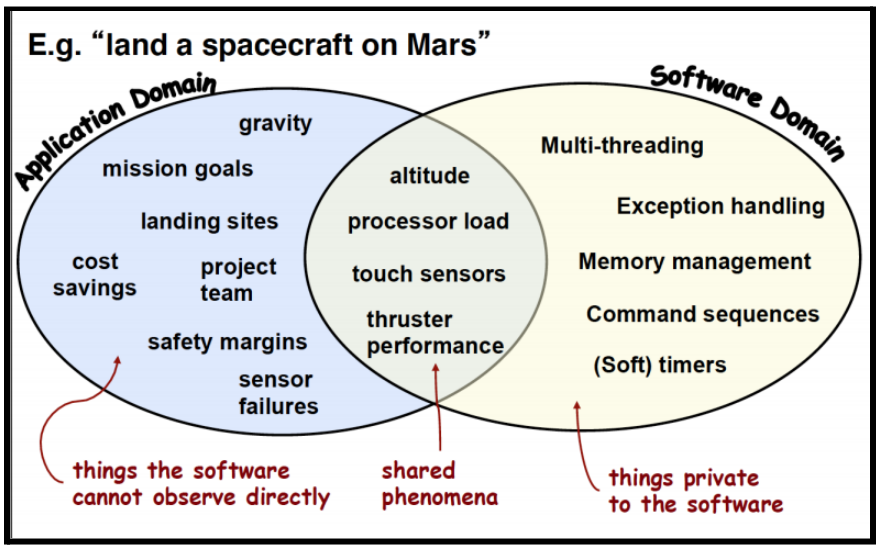
- **Requirements as Theories:**



- **Starting point:**
- Given a vague request for a new feature from users of your software:
    1. Identify the problem:
    - Find out what the goal is.
    2. Scope the problem:
    - Find out what new functionalities are needed.
    3. Identify solution scenarios:
    - Find out how users will interact with the software to solve the problem.
    4. Map onto the architecture:
    - Find out how the new functionalities will be met.
- Given a problem, requirements analysts must identify the following:
    - Which problem(s) need to be solved?
    - Where are the problem(s)?
    - Whose problem is it?
    - Why does it need solving?
    - When does it need solving?
    - What might prevent us from solving it?
    - How might a software system help?

- **Terminology:**
- **Application Domain:** Things the software cannot observe directly. Includes domain properties and requirements.
- **Domain Properties:** Things in the application domain that are true, whether or not we ever build the proposed system.
- **Requirements:** Things in the application domain that we wish to be made true, by delivering the proposed system.
- **Software Domain:** Things private to the software. Includes computers and programs.
- **Specification:** A description of the behaviours the program must have in order to meet the requirements.





- **Verification:** The program running on a particular computer satisfies the specification, and the specification, in the context of the given domain properties, satisfies the requirement.
- **Validation:** We discovered all the important requirements and properly understood the relevant domain properties.
- **Observations:**
- Analysis isn't necessarily a sequential process. Rewriting the problem statement can be useful at any stage of development.
- The problem statement will be imperfect. Models are approximations, not the actual thing. They will contain some inaccuracies and omit some information. We need to assess the risk that these will cause serious problems.
- Perfecting a specification may not always be cost effective.
- The problem statement should never be treated as fixed. Change is inevitable and therefore must be planned for.

- **Stakeholders:**
- **Stakeholder analysis** is identifying all the people who must be consulted during information acquisition.
- Type of stakeholders:
    - Users - Concerned with the features and functionalities of the new system.
    - Customers - Wants to get the best value for money invested.
    - Business analysts/Marketing team - Wants to make sure that we're doing better than the competition.
    - Training and user support staff - Want to make sure the new system is usable and manageable.
    - Technical authors - Will prepare user manuals and other documentation for the new system.
    - System analysts - Want to get the requirements right.
    - Designer - Want to build a perfect system or reuse existing code.
    - Project manager - Wants to complete the project on time, on budget and with all objectives met.

Goals:

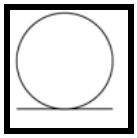
- A **goal** is a stakeholder's objective for the system.
- A **goal model** is a hierarchy of goals that relates the high-level goals to low-level system requirements.
- **Hard goals** describe the functions the system will perform.
  E.g. The system collects timetables from users.
- **Soft goals** describe the desired system qualities. Soft goals cannot be fully satisfied.
  E.g. The system should be reliable.
  E.g. The system should be of high quality.
- Goal elaboration:
    - "Why" questions explore higher goals (context).
    - "How" questions explore lower goals (operations).
    - "How else" questions explore alternatives.
- Relationships between goals:
    - A goal helps another goal. (+)
    - A goal hurts another goal. (-)
    - A goal makes another goal. (++)
    - A goal breaks another goal. (--)
- Approach for identifying stakeholder goals:
    - Focus on why a system is required.
    - Express the "why" as a set of stakeholder goals.
    - Use goal refinement to arrive at specific requirements.
    - Document, organize and classify goals.
    - Refine, elaborate and operationalize goals.

Robustness Analysis:

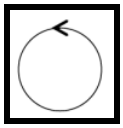
- **Law of Demeter:**
- The **Law of Demeter** states that a module should not have the knowledge on the inner details of the objects it manipulates. In other words, a software component or an object should not have knowledge of the internal working of other objects or components.
- The Law of Demeter reduces dependencies and helps build components that are loosely coupled for code reuse, easier maintenance, and testability.
- I.e.
  A method, M, of an object, O, can only call methods of:
  - O itself
  - M's parameters
  - Any objects created by M
  - O's direct component objects

  M cannot call methods of an object returned by another method call.
- **Note:** The programmer's rule of thumb is to use 1 dot.
  E.g. Instead of Customer.PayPalAccount.CreditCard.Subtract(total), use Customer.getPayment(total).
- **Robustness Analysis:**
- **Robustness analysis** is a way of analyzing your use case model. Robustness analysis provides an approach to the structuring of problem situations in which uncertainty is high, and where decisions can or must be staged sequentially. The specific focus of robustness analysis is on how the distinction between decisions and plans can be exploited to maintain flexibility. These are classified into **boundary objects**, **entity objects**, and **control objects**.
- **Boundary Objects:**
- Used by actors when communicating with the system.
- Represents the interfaces between the actors and the system.
- The "view" part of MVC architecture.

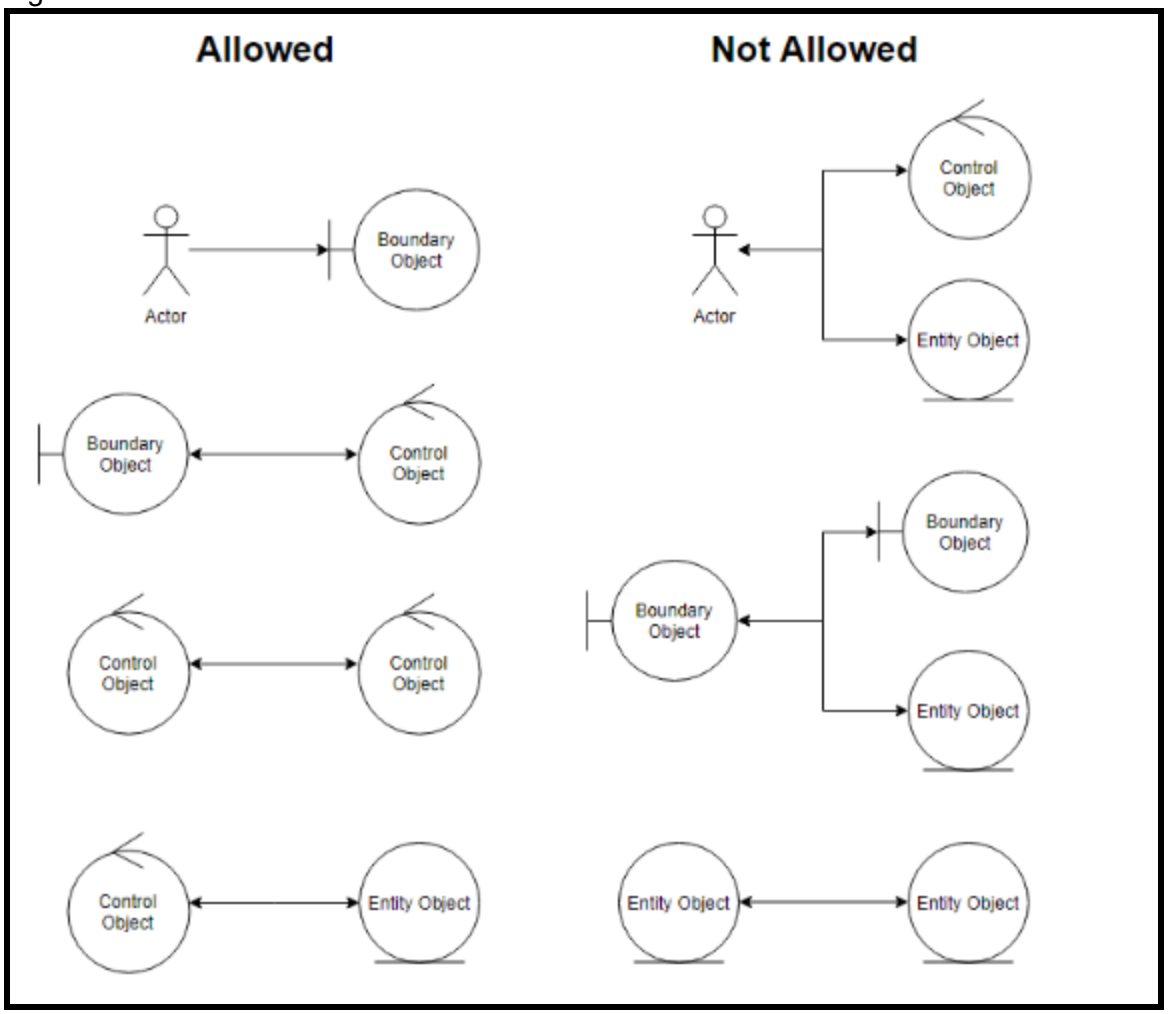
- **Entity Objects:**
- Usually objects from the domain model. They are objects representing stored data.
- Manages the information the system needs to provide the required functionality.
- The "model" part of the MVC architecture.



- **Control Objects:**
- The "glue" between boundary objects & entity objects. It captures business rules and policies and represents the use case logic and coordinates the other classes.
- The "controller" part of the MVC architecture.
- **Note:** It is often implemented as methods of other objects.



- **Robustness Diagram – 4 Connection Rules:**
- Boundary objects and entity objects are nouns while controllers are verbs. Nouns can't talk to other nouns, but verbs can talk to either nouns or verbs. Here are the four basic connection rules which should always be mind:
    1. Actors can only talk to boundary objects.
    2. Boundary objects can only talk to controllers and actors.
    3. Entity objects can only talk to controllers.
    4. Controllers can talk to boundary objects, entity objects, and other controllers, but not to actors.
- E.g.

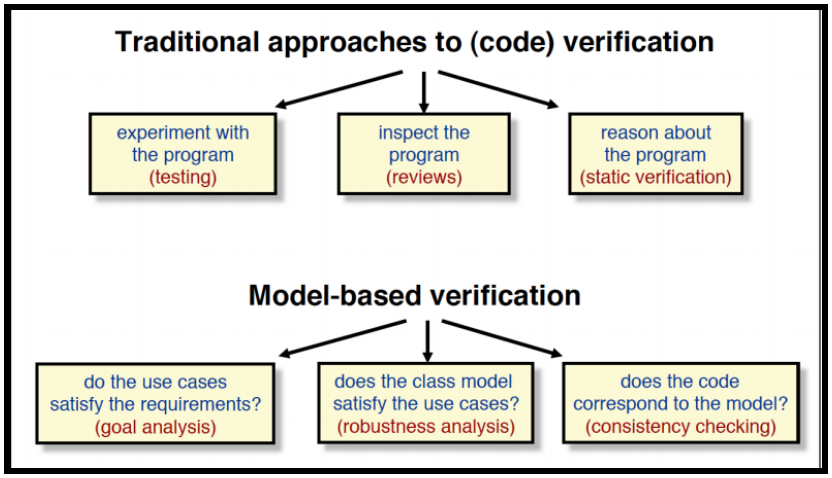


- **Why Use Robustness Analysis:**
    1. Bridges the gap between design and requirements.
    2. Sanity Check:
        - Tests the language in the use case description.
        - Nouns from the use case get mapped onto objects.
        - Verbs from the use case get mapped onto actions.
    3. Completeness Check:
        - Discover the objects you need to implement the use case.
        - Identify alternative courses of action.
    4. Object Identification:
        - Decide which methods belong to which objects.

- **Benefits of Robustness Analysis:**
    1. Forces a consistent style for use cases.
    2. Forces a correct 'voice' for use cases.
    3. Sanity and completeness check for use cases.
    4. Creates syntax rules for use case descriptions.
    5. Quicker and easier to read than sequence diagrams.
    6. Encourages use of Model-View-Controller (MVC) pattern.
    7. Helps build layered architectures e.g presentation layer, domain layer, repository layer.
    8. Checks for reusability across use cases before doing detailed design.
    9. Provides traceability between user's view and design view.
    10. Plugs semantic gap between requirements and design.
- **Constructing a Robustness Diagram:**
    1. Add a boundary element for each major UI element. Not at the level of individual widgets though.
    2. Add controllers:
        - One to manage each use case.
        - One for each business rule.
        - Another for each activity that involves coordination between several other elements.
    3. Add an entity for each business concept.

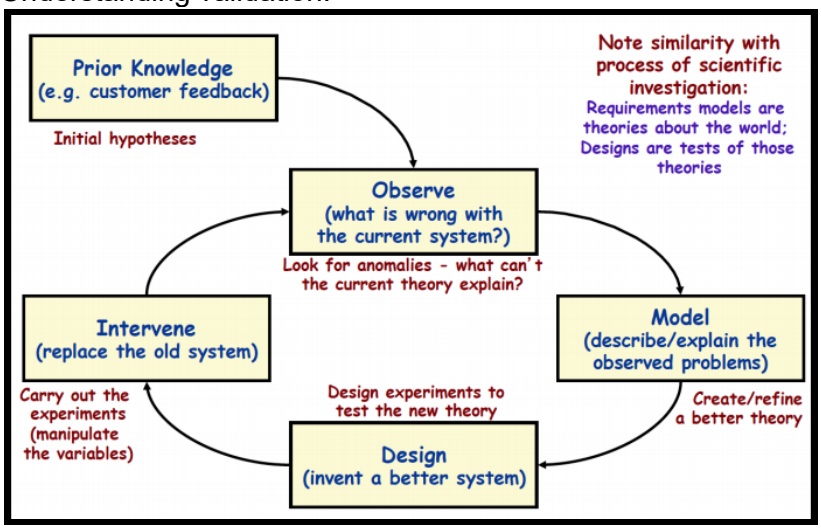
Verification and Validation (V&V):

- **Verification:**
- **Verification** is testing that your product meets the specifications/requirements you have written.
  I.e. Are we building the system right?
  It is ensuring that the software to be built is actually what the user wants.
- I.e.

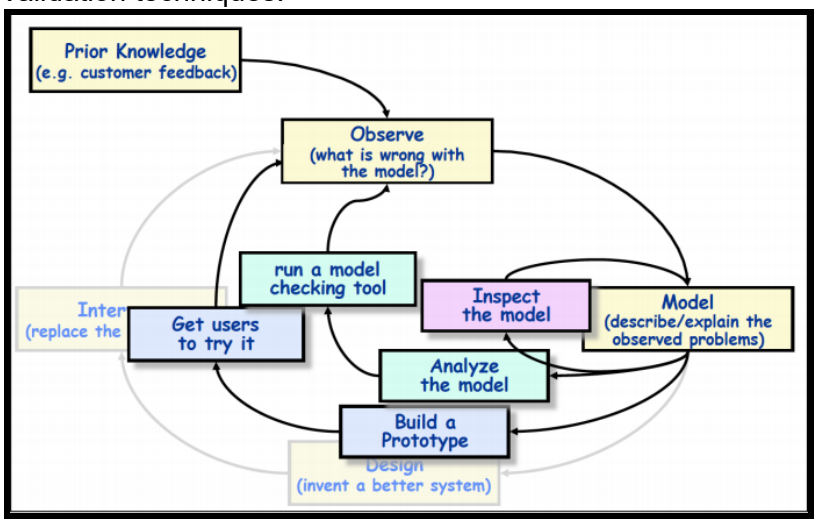


- **Validation:**
- **Validation** is testing how well you addressed the business needs that caused you to write those requirements. It is also sometimes called acceptance or business testing.
  I.e. Are we building the right system?
  It is ensuring that the software runs correctly.

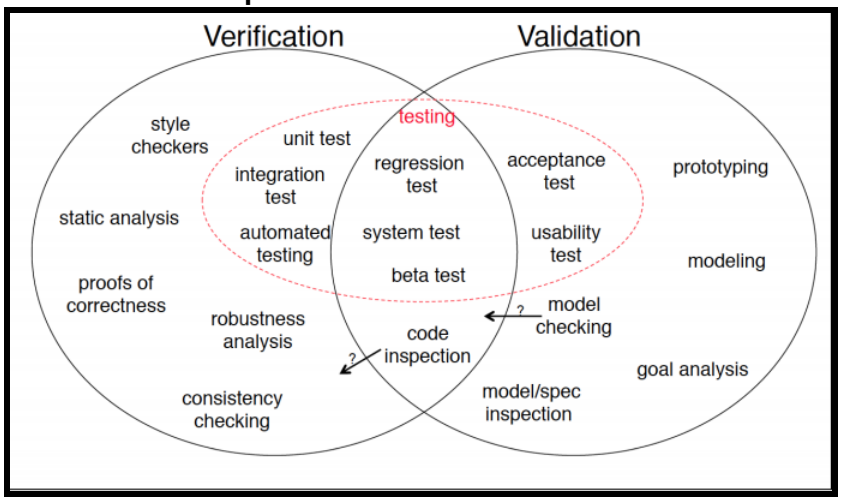
- Understanding validation:



- Validation techniques:



- **Choice of Techniques:**



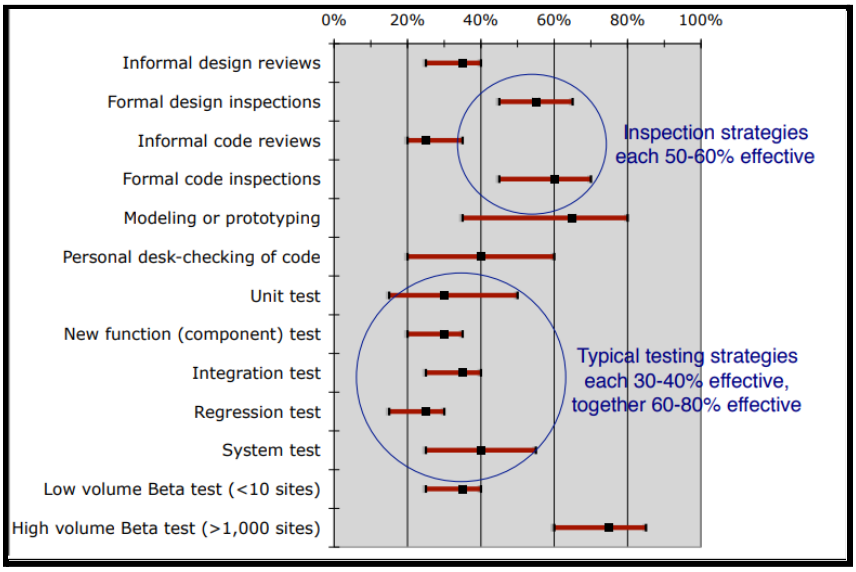
- **Roles for Independent V&V:**
- V&V is usually performed by an independent contractor.
- Independent verification and validation involves V&V done by a third party organization not involved in the development of the product. The main check performed is whether user requirements are met alongside ensuring that the product is structurally sound and built to the required specifications. Independent V&V fulfills the need for an independent technical opinion.
- V&V costs between 5% and 15% of development costs, but a study by NASA showed that its return on investment is fivefold. Bugs/errors are found earlier, making them easier to fix. The specifications are clearer. Developers are more likely to use better practices.
- 3 types of independence:
    1. Managerial Independence:
        - Separate responsibility from that of creating the software.
        - Can decide when and where to focus the V&V effort.
    2. Financial Independence:
        - Costed and funded separately.
        - No risk of diverting funds/resources when the going gets tough.
    3. Technical Independence:
        - The code is looked at by an outside party. There is little or no bias.
        - The tools and techniques used can be different.
- **Prototyping:**
- Presentation Prototypes:
    - Used for proof of concept and explaining design features.
- Exploratory Prototypes:
    - Used to determine problems, elicit needs, clarify goals, and compare design options.
    - It is informal and unstructured.
- Breadboards or Experimental Prototypes:
    - Used to explore technical feasibility or to test the suitability of a technology.
    - Typically there is no user/customer involvement.
- Evolutionary/Operational Prototypes:
    - Development is seen as a continuous process of adapting the system.
- **Usability Testing:**
- Real users try out the system or prototype and write down what problem(s) they observed. 3-5 users give the best return on investment.
- **Model Analysis:**
- Verification:
    - “Is the model well-formed?”
    - Are the parts of the model consistent with one another?
- Validation:
    - ‘What if’ questions:
        - Reasoning about the consequences of particular requirements.
        - Reasoning about the effect of possible changes.
        - Asking if the system will ever do the following tasks.
    - Formal challenges:
        - If the model is correct then the following property should hold.
    - Animation of the model on small examples
    - State exploration:
        - Using model checking to find traces that satisfy some property.

- **UML Consistency Check:**
- Use Case Diagrams:
  - Does each use case have a user?
  - Is each use case documented?
- Class Diagrams:
  - Does the class diagram capture all the classes mentioned?
  - Does every class have methods to set and get its attributes?
- Sequence Diagrams:
  - Is each class in the sequence diagram?
  - Can each message be sent?
    - Is there an association connecting sender and receiver classes on the sequence diagram?
    - Is there a method call in the sending class for each message sent?
    - Is there a method call in the receiving class for each message received?
- **Model Checkers:**
- Automatically check properties (expressed in Temporal Logic):
  -  means that p is true now and always (in the future).
  -  means that p is true eventually (in the future).
  -  means that whenever p occurs, it's always (eventually) followed by a q.
- The model may be:
  - Of the program itself (each statement is a 'state').
  - An abstraction of the program.
  - A model of the specification.
  - A model of the requirements.
- A model checker searches all paths in the state space with lots of techniques for reducing the size of the search.
- Model checking does not guarantee correctness.
  It only tells you about the properties that you ask about and may not be able to search the entire state space.
  However, it is good at finding many safety, liveness and concurrency problems
- **Inspections:**
- Management Reviews:
  - Used to provide confidence that the design is sound.
  - The audience are the management and sponsors (customers).
  - Examples include preliminary design reviews and critical design reviews.
- Walkthroughs/Scientific Peer Reviews:
  - Used by development teams to improve the quality of the product.
  - The focus is on understanding design choices and finding defects.
- Fagan Inspections:
  - A process management tool. (Always formal)
  - Used to improve the quality of the development process.
  - Collect defect data to analyze the quality of the process.
  - Written output is important.
  - Major role in training new staff and transferring expertise.

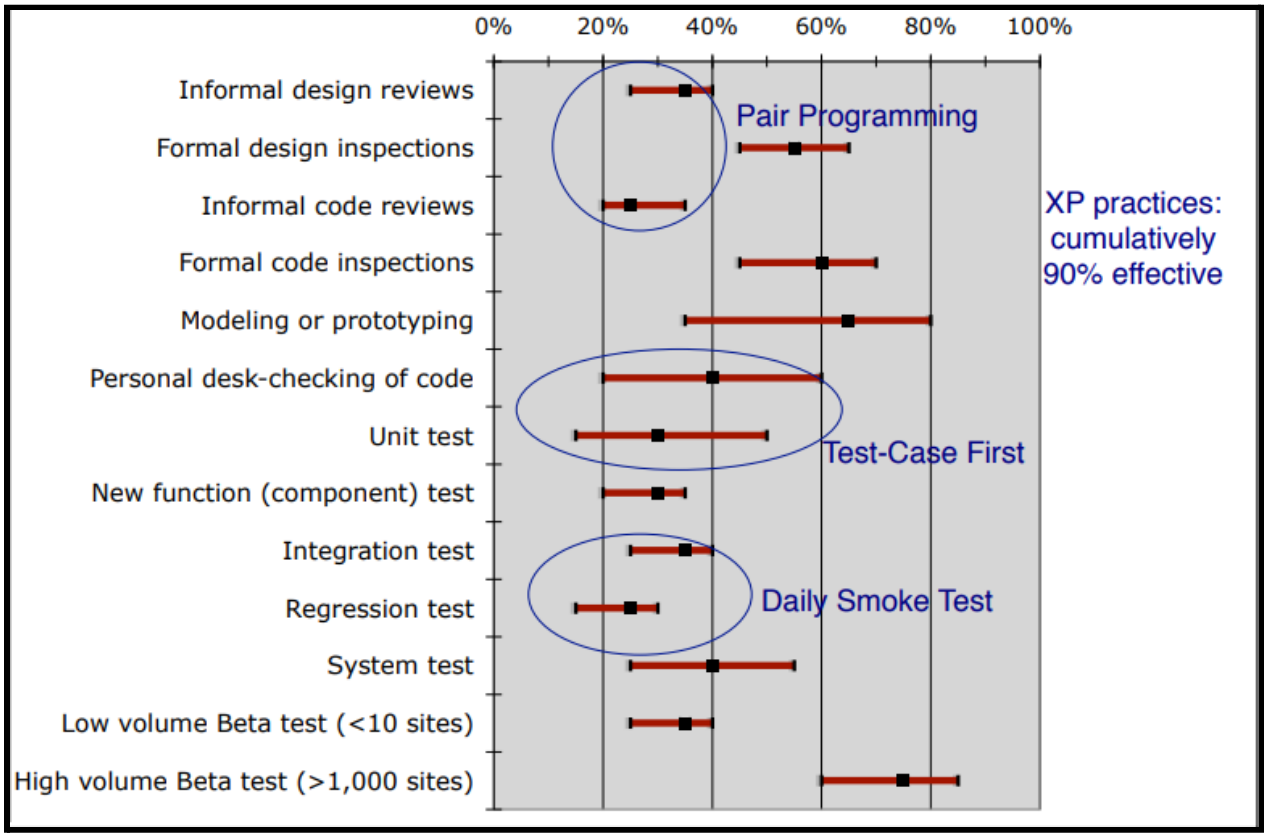
- Inspections are very effective. Code inspections are better than testing for finding defects. For models and specifications, inspections ensure that the domain experts carefully reviewed them.
- Key ideas of inspections:
    - Preparation: Reviewers individually inspect the code/diagram first.
    - Collection meeting: Reviewers meet to merge their defect lists. They note each defect, but don't spend time trying to fix it. The meeting plays an important role as reviewers learn from one another when they compare their lists and additional defects can be uncovered. Defect profiles from each inspection are important for process improvement.
- How to structure the inspection:
    - Checklist:
        - Uses a checklist of questions/issues.
        - The review is structured by the issues on the list.
    - Walkthrough:
        - One person presents the product step-by-step.
        - The review is structured by the product.
    - Round Robin:
        - Each reviewer gets to raise an issue. (Goes in a circle)
        - The review is structured by the review team.
    - Speed Review:
        - Each reviewer gets 3 minutes to review a chunk and then passes it to the next reviewer.
        - Good for assessing comprehensibility.
- Benefits of inspection:
    - For applications programming:
        - More effective than testing.
        - Most reviewed programs run correctly the first time.
    - Data from large projects:
        - Error reduction by a factor of 5 (10 in some reported cases).
        - Improvement in productivity by 14% to 25%.
        - Percentage of errors found by inspection: 58% to 82%.
        - Cost reduction of 50%-80% for V&V even including cost of inspection.
    - Effects on staff competence:
        - Increased morale, reduced turnover.
        - Better estimation and scheduling (more knowledge about defect profiles).
        - Better management recognition of staff ability.

Introduction to Testing:

- **Defects vs. Failures:**
- Many causes of defects in software include:
  - Missing requirement
  - Wrong specification
  - Requirements that were infeasible
  - Faulty system design
  - Wrong algorithms
  - Faulty implementation
- Defects may lead to failures but the failure may show up somewhere else. Tracking the failure back to a defect can be hard.
- Examples of program defects are:
  - **Syntax Faults:** Incorrect use of programming constructs.
  - **Algorithmic Faults:**
    - Branching too soon or too late.
    - Testing for the wrong condition.
    - Failure to initialize correctly.
    - Failure to test for exceptions. E.g. divide by 0
    - Type mismatch
  - **Precision Faults:**
    - Mixed precision.
    - Faulty/incorrect floating point conversion.
  - **Documentation Faults:** Design docs or user manual is wrong.
  - **Stress Faults:**
    - Overflowing buffers.
    - Lack of bounds checking.
  - **Timing Faults:**
    - Processes fail to synchronize.
    - Events happen in the wrong order (Race Condition).
  - **Throughput Faults:** Performance is lower than required.
  - **Recovery Faults:** Incorrect recovery after another failure.
  - **Hardware Faults:** Hardware doesn't perform as expected.
- **Effectiveness of defect detection strategies:**
- Defect Detection Effectiveness



- XP Practices

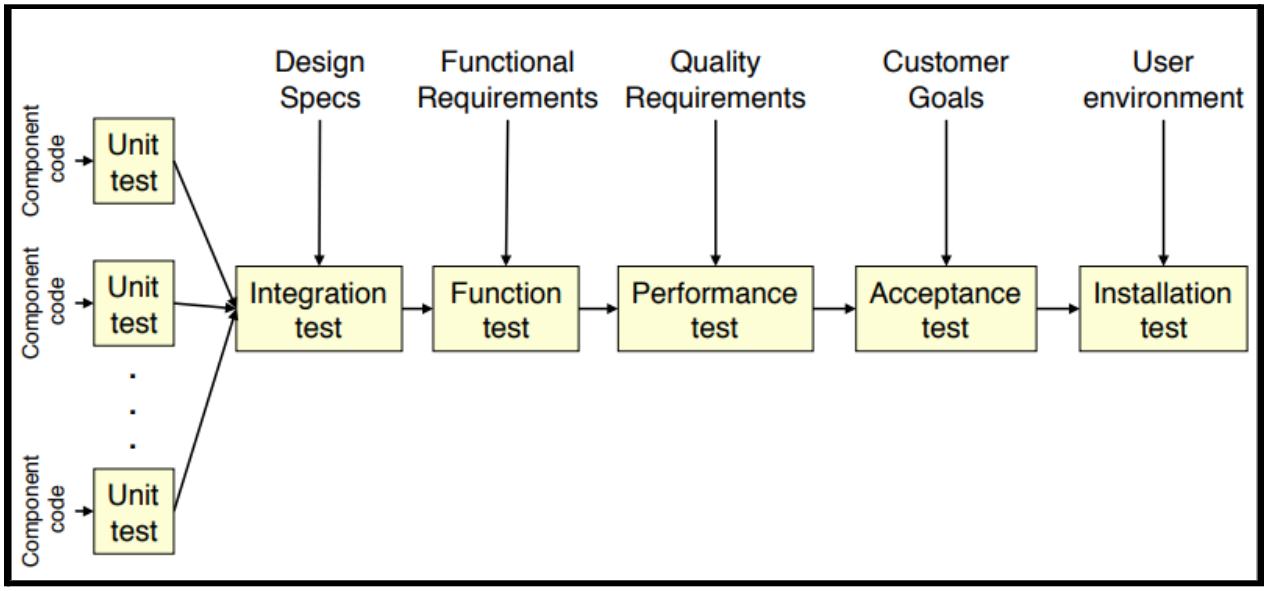


- Use a combination of techniques because:
  - Different techniques will find different defects.
  - Different people will find different defects.
  - Testing alone is only 60-80% effective.
  - The best organisations achieve 95% defect-removal.
  - Inspection, modeling, prototyping, and system tests are all important.
- Costs vary (This is from an IBM study):
  - On average, 3.5 hours are spent on each defect for inspection.
  - On average, 15-25 hours are spent on each defect for testing.
- Costs of fixing defects also vary:
  - It is 100 times more expensive to remove a defect after implementation than in design.
  - 1-step methods (inspection) are cheaper than 2-step methods (test+debug).
- Cost of Rework:
  - The industry average is 10-50 lines of delivered code per day per person.
  - Debugging + retesting is 50% of the effort in traditional software engineering.
- Removing defects early saves money. Testing is easier if the defects are removed first and high quality software will be delivered sooner at a lower cost.
- How not to improve quality: “Trying to improve quality by doing more testing is like trying to diet by weighing yourself more often.”
- **Basics of Testing:**
- Benefits of testing:
  - Find important defects to get them fixed.
  - Assess the quality of the product.
  - Help managers make release decisions.

    - Block premature product releases.
    - Help predict and control product support costs.
    - Check interoperability with other products.
    - Find safe scenarios for use of the product.
    - Assess conformance to specifications.
    - Certify that the product meets a particular standard.
    - Ensure that the testing process meets accountability standards.
    - Minimize the risk of safety-related lawsuits.
    - Measure reliability.
- Testing is more effective if you removed the bugs first.
- The goal of testing is unachievable. We cannot ever prove absence of errors. Furthermore, finding no errors probably means your tests are ineffective.
- The goal of testing is also counter-intuitive. It is the only goal in software engineering whose aim is to find errors/break the software. All other development activities aim to avoid errors/breaking the software.
- Testing does not improve software quality. Test results measure the quality of the existing code, but doesn't improve it. Test-debug cycles are the least effective way to improve quality of code.
- Testing requires you to assume your code is buggy. If you assume otherwise, you probably won't find them.
- Testing must be done appropriately based on the context and the requirements/specification.
- Good tests have:
    - **Power:** When a problem exists, the test will find it.
    - **Validity:** The problems found are genuine problems.
    - **Value:** Each test reveals things the clients will want to know.
    - **Credibility:** Each test is a likely operational scenario.
    - **Non-redundancy:** Each test provides new information.
    - **Repeatability:** Each test is easy and inexpensive to re-run.
    - **Maintainability:** Tests can be revised as the requirements/specifications/products are revised.
    - **Coverage:** The test cases exercise the product in a way not already tested for. This is similar to non-redundancy.
    - **Ease of evaluation:** The test results are easy to interpret.
    - **Diagnostic power:** The tests help pinpoint the cause of problems.
    - **Accountability:** You can explain, justify and prove you ran the test cases.
    - **Low cost:** The time and effort to develop and execute the test cases are low.
    - **Low opportunity cost:** Creating and running a test is a better use of your time than other things you could be doing.
- Types of testing:
    - **Unit test:** Testing an individual software component or module.
      I.e. Each unit is tested separately to check it meets its specification.
    - **Integration test:** Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.
      I.e. Units are tested together to check they work together.
      Integration testing is hard because it is much harder to identify equivalence classes, problems of scale may occur, and it tends to reveal specification errors rather than integration errors.
    - **Function test:** Testing that validates the software system against the functional requirements/specifications.

- **Performance test:** Testing the speed, responsiveness and stability of a computer, network, software program or device under a workload.
- **Acceptance test:** An acceptance test verifies whether the end to end flow of the system is as per the business requirements and if it is as per the needs of the end-user. The client accepts the software only when all the features and functionalities work as expected. It is the last phase of the testing, after which the software goes into production. It is also called User Acceptance Testing (UAT).
- **Installation test:** Testing to check if the software has been correctly installed with all the inherent features and that the product is working as per expectations.

I.e.



- Systematic testing depends on partitioning. We need to partition the set of possible behaviours of the system and choose representative samples from each partition and make sure we covered all partitions. Identifying suitable partitions is what testing is all about. We can use different test strategies and methods to do so.
- Coverage 1: Structural

```
boolean equal (int x, y) {
/* effects: returns true if
  x=y, false otherwise
*/
if (x == y)
  return(TRUE)
else
  return(FALSE)
}
```

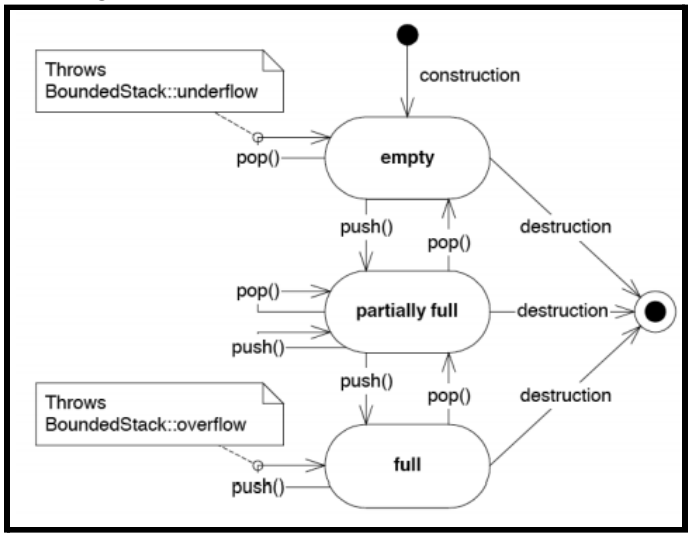
A naïve testing strategy is to pick random values for x and y and test ‘equal’ on them. However, we might never test the first branch of the ‘if’ statement, so we need enough test cases to cover every branch in the code.

- Coverage 2: Functional

```
int maximum (list a)
/* requires: a is a list of
   integers
   effects: returns the maximum
   element in the list
*/
```

A naïve testing strategy is to generate lots of lists and test 'maximum' on them. However, we haven't tested off-nominal cases such as empty lists, non-integers, negative integers, etc, so we need enough test cases to cover every kind of input the program might have to handle.

- Coverage 3: Behavioural



A naïve testing strategy is to push and pop things off the stack and check it all works. However, we might miss full and empty stack exceptions, so we need enough tests to exercise every event that can occur in each state that the program can be in.

- Other types of tests:
  - **Facility testing:** Does the system provide all the functions required?
  - **Volume testing:** Can the system cope with large data volumes?
  - **Stress testing:** Can the system cope with heavy loads?
  - **Endurance testing:** Will the system continue to work for long periods?
  - **Usability testing:** Can the users use the system easily?
  - **Security testing:** Can the system withstand attacks?
  - **Performance testing:** How good is the response time?
  - **Storage testing:** Are there any unexpected data storage issues?
  - **Configuration testing:** Does the system work on all target hardware?
  - **Installation testing:** Can we install the system successfully?
  - **Reliability testing:** How reliable is the system over time?
  - **Recovery testing:** How well does the system recover from failure?
  - **Serviceability testing:** How maintainable is the system?
  - **Documentation testing:** Is the documentation accurate, usable, etc.
  - **Operations testing:** Are the operators' instructions right?
  - **Regression testing:** Repeat all the testing every time we modify the system.

Testing Strategies:

- **Good Practices:**
- Write the test cases first. This forces you to think carefully about the requirements first and exposes requirements problems early.
- **Structural Coverage Strategies (White box testing):**
- **White box testing** is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester.
- Types of white box testing include:
  - **Structured Basis Testing:**
    - Structured basis testing gives you the minimum number of test cases you need to exercise every path.
    - Start with 1 test case for the straight path. Add 1 test case for each of these keywords: if, while, repeat, for, and, or add 1 test case for each branch of a case statement.
    - E.g.

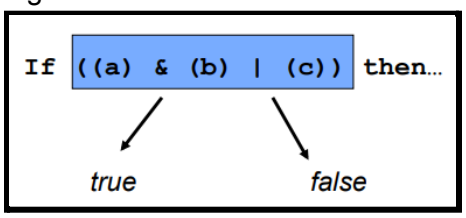
```
int midval (int x, y, z) {
/* effects: returns median
  value of the three inputs
*/
if (x > y) {
  if (x > z) return x
  else return z }
else {
  if (y > z) return y
  else return z } }
```

We will need 4 test cases.

  - **Statement Coverage:**
    - Statement coverage is a white box testing technique, which involves executing all the statements in the source code at least once.

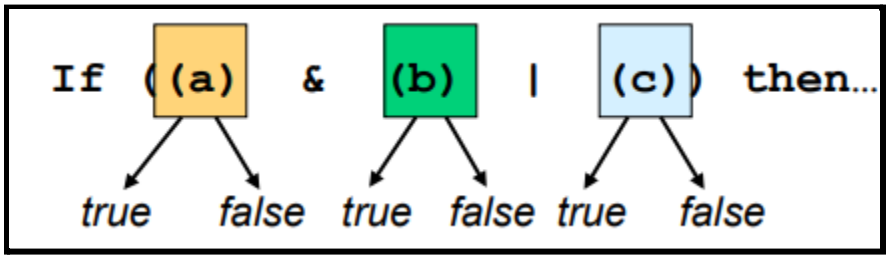
$$\textit{Statement Coverage} = \frac{\textit{Number of executed statements}}{\textit{Total number of statements}} \times 100$$

  - **Branch Coverage:**
    - Branch coverage is a white box testing method in which every outcome from a code module is tested. The purpose of branch coverage is to ensure that each decision condition from every branch is executed at least once.
    - E.g.



$$\textit{Branch Coverage} = \frac{\textit{Number of Executed Branches}}{\textit{Total Number of Branches}}$$

- **Condition/Decision Coverage:**
  - Condition coverage is a testing method used to test and evaluate the variables or sub-expressions in the conditional statement. The goal of condition coverage is to check individual outcomes for each logical condition. In this coverage, expressions with logical operands are only considered.
  - Condition coverage does not give a guarantee about full decision coverage.
  - E.g.



  - E.g.
    Suppose we have the code below.

```
if ((A || B) && C)
{
  << Few Statements >>
}
else
{
  << Few Statements >>
}
```

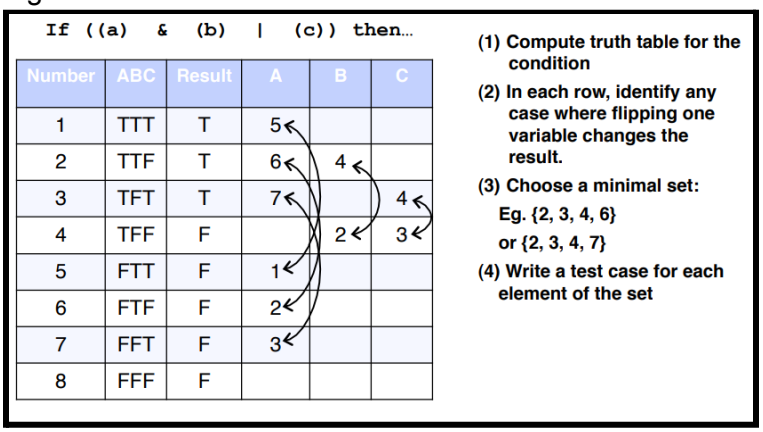
    The 3 following tests would be sufficient for 100% condition coverage testing.

A = true	B = not eval	C = false
A = false	B = true	C = true
A = false	B = false	C = not eval

$$Condition\ Coverage = \frac{Number\ of\ Executed\ Operands}{Total\ Number\ of\ Operands}$$

- **Modified Condition/Decision (MC/DC) Coverage:**
  - The modified condition/decision coverage (MC/DC) coverage is like condition coverage, but every condition in a decision must be tested independently to reach full coverage. This means that each condition must be executed twice, with the results true and false, but with no difference in the truth values of all other conditions in the decision. In addition, it needs to be shown that each condition independently affects the decision.
  - The Modified Condition/Decision Coverage enhances the condition/decision coverage criteria by requiring that each condition be shown to independently affect the outcome of the decision. This kind of testing is performed on mission critical application which might lead to death, injury or monetary loss.

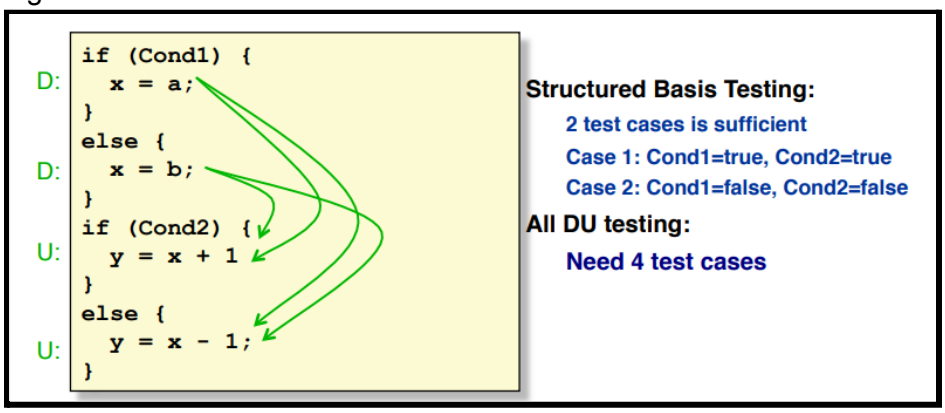
- E.g.



Number	ABC	Result	A	B	C
1	TTT	T	5		
2	TTF	T	6	4	
3	TFT	T	7		4
4	TFF	F		2	3
5	FTT	F	1		
6	FTF	F	2		
7	FFT	F	3		
8	FFF	F			

  - Advantages:
    - Linear growth in the number of conditions.
    - Ensures coverage of the object code.
    - Discovers dead code (operands that have no effect).
  - It is mandated by the US Federal Aviation Administration. In avionics, complex boolean expressions are common, and MC/DC coverage has been shown to uncover important errors not detected by other test approaches.
  - It's expensive. The total cost of aircraft development for Boeing 777 is $5.5 billion while the cost of testing to MC/DC criteria is approximately $1.5 billion.
- **Data Flow Coverage:**
  - Things that can happen to data:
    - **D**efined - Data is initialized but not yet used.
    - **U**sed - Data is used in a computation.
    - **K**illed - Space is released.
    - **En**tered - Working copy created on entry to a method.
    - **Ex**ited - Working copy removed on exit from a method.
  - Normal life of a variable: Defined once, used a number of times, killed.
  - Potential Defects:
    - **D-D:** A variable is defined twice.
    - **D-Ex, D-K:** A variable is defined but not used.
    - **En-K:** Destroying a local variable that wasn't defined.
    - **En-U:** Using a local variable before it's initialized.
    - **K-K:** Unnecessary killing a variable. This can hang the machine.
    - **K-U:** Using data after it has been destroyed.
    - **U-D:** Redefining a variable after it has been used.
  - Data flow testing helps us pinpoint any of the following issues:
    - A variable that is declared but never used within the program.
    - A variable that is used but never declared.
    - A variable that is defined multiple times before it is used.
    - Deallocating a variable before it is used.
  - To get the minimal set of tests to cover every D-U path, we need 1 test for each path from each definition to each use of the variable.

- E.g.



- **Boundary Checking:**
  - Every boundary needs 3 tests.
  - E.g. Suppose we have this line:
    if (x < 3)
    We need 3 test cases:
    One to test when x < 3.
    One to test when x == 3.
    One to test when x > 3.

- **Function Coverage Strategies (Black box testing):**
- **Black box testing** is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester.
- Generating Tests from Use Cases:
  1. Test the Basic Flow.
  2. Test the Alternate Flows.
  3. Test the Postconditions.
  4. Break the Preconditions.
  5. Identify options for each input choice.
- Classes of Bad Data:
  - Too little data or no data.
  - Too much data.
  - The wrong kind of data (invalid data).
  - The wrong size of data.
  - Uninitialized data.
- Classes of Good Data:
  - Nominal cases - middle of the road, expected values.
  - Minimum normal configuration.
  - Maximum normal configuration.
  - Compatibility with old data.
- Classes of input variables:
  - **Values that trigger alternative flows:**
    E.g. Invalid credit card information.
  - **Trigger different error messages:**
    E.g. The text is too long for the field.
    E.g. An email address with no “@”.
  - **Inputs that cause changes in the appearance of the UI:**
    E.g. A prompt for additional information.
  - **Inputs that cause different options in dropdown menus:**
    E.g. US/Canada triggers a menu of states/provinces.

- **Cases in a business rule:**
  E.g. No next day delivery after 6pm.
- **Border conditions:**
  E.g. If the password must be min 6 characters long, test passwords of 5,6,7 characters.
- **Check the default values:**
  E.g. When the cardholder's name is filled automatically.
- **Override the default values:**
  E.g. When the user enters a different name.
- **Enter data in different formats:**
  E.g. phone numbers: (416) 555 1234 vs 416-555-1234 vs 416 555 1234.
- **Test country-specific assumptions:**
  E.g. date order: 3/15/12 vs 15/3/12.

- Limits of Use Cases as Test Cases:
  - **Use case tests are good for:**
    User acceptance testing.
    "Business as usual" functional testing.
    Manual black-box tests.
    Recording automated scripts for common scenarios.
  - **Limitations of use case tests:**
    Likely to be incomplete.
    Use cases don't describe enough detail of use.
    Gaps and inconsistencies may occur between use cases.
    Use cases might be out of date.
    Use cases might be ambiguous.
  - **Defects you won't discover:**
    System errors (e.g. memory leaks).
    Things that corrupt persistent data.
    Performance problems.
    Software compatibility problems.
    Hardware compatibility problems.

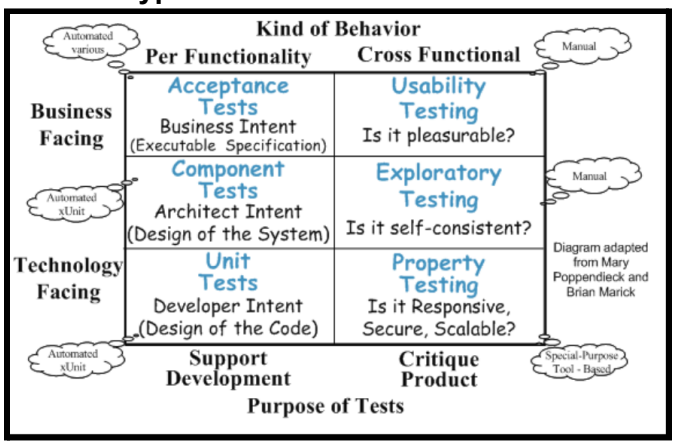
- **Stress Testing:**
- **Stress testing** is a type of software testing that verifies the stability and reliability of the software application. The goal of stress testing is measuring software on its robustness and error handling capabilities under extremely heavy load conditions and ensuring that software doesn't crash under crunch situations. It even tests beyond normal operating points and evaluates how software works under extreme conditions.
- **QuickTests:**
  - QuickTests are tests that don't cost much to design, are based on some estimated idea for how the system could fail and don't take much prior knowledge in order to apply.
  - **Explore the input domain:**
    1. Inputs that force all the error messages to appear.
    2. Inputs that force the software to establish default values.
    3. Explore allowable character sets and data types.
    4. Overflow the input buffers.
    5. Find inputs that may interact, and test combinations of their values.
    6. Repeat the same input numerous times.

- **Explore the outputs:**
    7. Force different outputs to be generated for each input.
    8. Force invalid outputs to be generated.
    9. Force properties of an output to change.
    10. Force the screen to refresh.
- **Explore stored data constraints:**
    11. Force a data structure to store too many or too few values.
    12. Find ways to violate internal data constraints.
- **Explore feature interactions:**
    13. Experiment with invalid operator/operand combinations.
    14. Make a function call itself recursively.
    15. Force computation results to be too big or too small.
    16. Find features that share data.
- **Vary file system conditions:**
    17. File system full to capacity.
    18. Disk is busy or unavailable.
    19. Disk is damaged.
    20. Invalid file name.
    21. Vary file permissions.
    22. Vary or corrupt file contents.

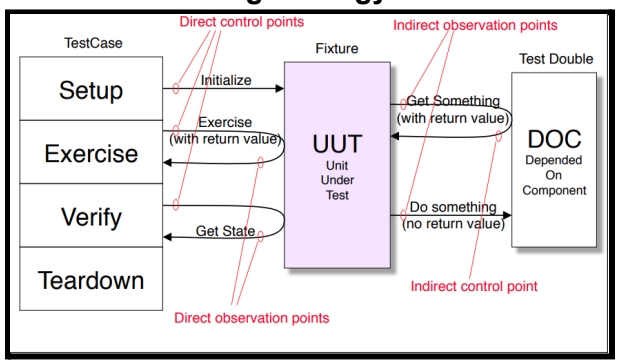
- **Interference Testing:**
    - Examples include:
        - Generate interrupts
        - Change the context
        - Cancel a task
        - Pause the task
        - Swap out the task
        - Compete for resources
- **A radical alternative - Exploratory Testing:**
- **Exploratory testing** is a style of software testing that emphasizes personal freedom and responsibility of the tester to continually optimize the value of their work by treating test-related learning, test design, and test execution as mutually supportive activities that run in parallel throughout the project.
- Exploratory testing is a type of software testing where test cases are not created in advance but testers check the system on the fly. They may note down ideas about what to test before test execution. The focus of exploratory testing is more on testing as a "thinking" activity.
- Exploratory testing is widely used in Agile models and is all about discovery, investigation, and learning. It emphasizes personal freedom and responsibility of the individual tester.
- Under scripted testing, you design test cases first and later proceed with test execution. On the contrary, exploratory testing is a simultaneous process of test design and test execution all done at the same time. It is unscripted.

Automated Testing:

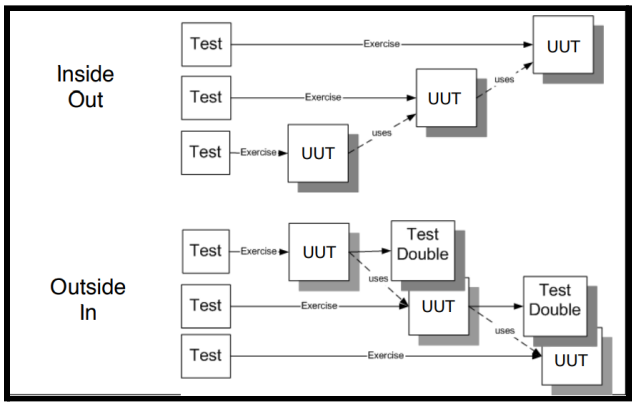
- **Different types of tests:**



- Where possible, automate your testing. By doing regression testing, tests can be repeated whenever the code is modified. This takes the tedium out of extensive testing and makes more extensive testing possible.
- In order to do automated testing, you'll need:
  - **Test drivers** which will automate the process of running a test set. It sets up the environment, makes a series of calls to the Unit-Under-Test (UUT), saves the results and checks if they were right and generates a summary for the developer.
  - **Test stubs** which will simulate part of the program called by the UUT. It checks whether the UUT set up the environment correctly, checks whether the UUT passed sensible input parameters to the stub and passes back some return values to the UUT.
- **Automated Testing Strategy:**

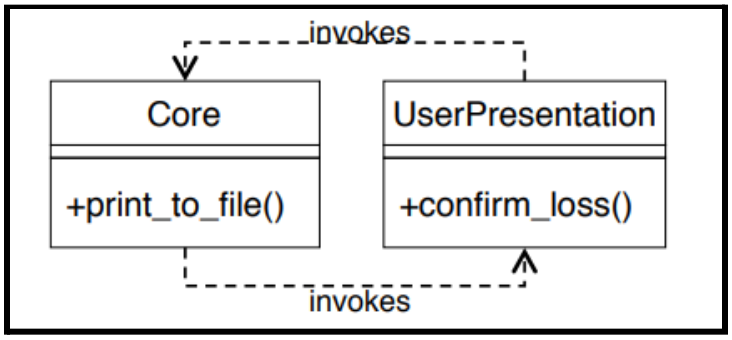


- Test Order

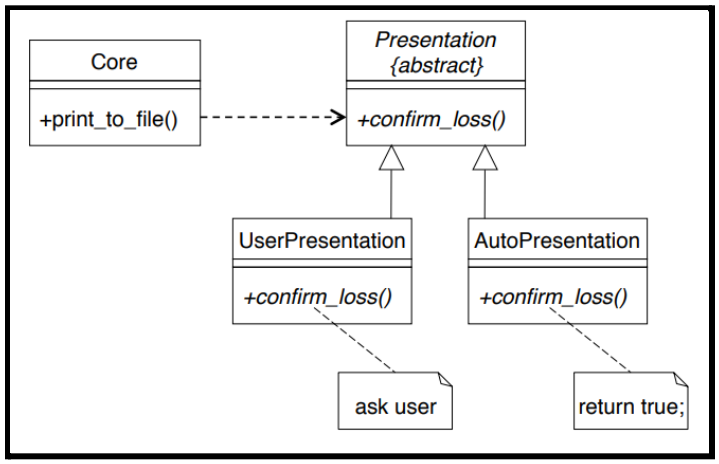


- **JUnit:**
- JUnit is a unit testing framework for Java.
- Assertion methods in JUnit:
    - **Single-Outcome Assertions:**
      E.g. fail;
    - **Stated Outcome Assertions:**
      E.g. assertNotNull(anObjectReference);
      E.g. assertTrue(booleanExpression);
    - **Expected Exception Assertions:**
      E.g. assert_raises(expectedError) {codeToExecute };
    - **Equality Assertions:**
      E.g. assertEqual(expected, actual);
    - **Fuzzy Equality Assertions:**
      E.g. assertEqual(expected, actual, tolerance);
- **Principles of Automated Testing:**
    - Write the test cases first.
    - Design for testability.
    - Use the front door first. This means test using public interfaces and avoid creating backdoor manipulations.
    - Communicate intent. Treat tests as documentation and make it clear what each test does.
    - Don't modify the UUT. Avoid test doubles and test-specific subclasses unless absolutely necessary.
    - Keep tests independent.
    - Isolate the UUT.
    - Minimize rest overlap.
    - Check one condition per test.
    - Test different concerns separately.
    - Minimize untestable code.
    - Keep test logic out of production code.
- **Challenges for automated testing:**
    - **Synchronization** - How do we know a window popped open that we can click in?
    - **Abstraction** - How do we know it's the right window?
    - **Portability** - What happens on a display with different resolution/size?
- **Techniques for testing the presentation layer:**
    - **Script the mouse and keyboard events:**
        - We can write a script that sends mouse and keyboard events. (E.g. "send_xevents @400,100")
        - However, this is not good practice/design because the script is write-only and fragile.
    - **Script at the application function level:**
        - E.g. Applescript: tell application "UMLet" to activate.
        - This is robust against size and position changes but fragile against widget renamings, and layout changes. Hence, this is still not good practice/design.
    - **Write an API for your application:**
        - We can use these APIs for testing.
        - E.g. Allow an automated test to create a window and interact with widgets.

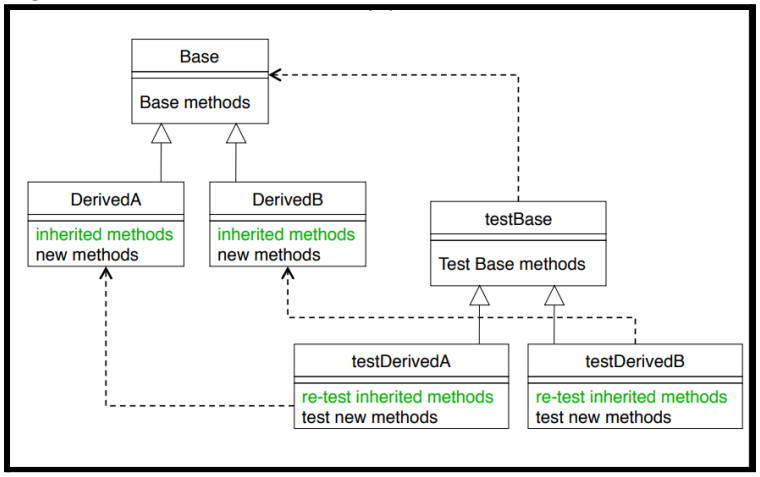
- **Circular Dependencies:**
- If you have circular dependencies in your code, you should refactor your code to remove them.
- E.g.
  Here, we have a circular dependency.



Because we have a circular dependency, we need to refactor the code. Here is the new code.



- **Testing Object Oriented Code:**
- Object oriented code can be hard to test. The best/most efficient way to test object oriented code is to have a parent test class for the parent class and to extend the parent test class for the subclasses.
  E.g.



E.g.
The test class for the parent class.

```
class FooTest {
        @Test
        public void testSomeMethodBar() {
         ...
        }

        @Test public void void someOtherMethodBaz(Baz baz) {
        ...
        }
}
```

The test class for the subclass class. Notice that this test class inherits from FooTest.

```
class EnhancedFooTest extends FooTest {
        @Test
        public void testSomeMethodBar() {
         ...
        }
}
```

- **When to stop testing:**
  - **Motorola's Zero-failure** testing model predicts how much more testing is needed to establish a given reliability goal.
  - **Failures = $ae^{-b(t)}$** where "a" and "b" are constants and "t" is the testing time.
  - The **reliability estimation process** gives the number of further failure free hours of testing needed to establish the desired failure density.
    Note: If a failure is detected during this time, you stop the clock and recalculate.
    Note: This model ignores operational profiles.
    Inputs needed:
    - **fd** = target failure density (e.g. 0.03 failures per 1000 LOC)
    - **tf** = total test failures observed so far
    - **th** = total testing hours up to the last failure

    Formula:

$$\frac{ln(fd/(0.5+fd)) * xh}{ln((0.5+fd)/(tf+fd))}$$

- **Fault Seeding:**
- **Fault seeding** is a technique for evaluating the effectiveness of a testing process. One or more faults are deliberately introduced into a code base, without informing the testers. The discovery of seeded faults during testing can be used to calibrate the effectiveness of the test process.
- The idea is that

$$\frac{\text{Detected seeded faults}}{\text{Total seeded faults}} = \frac{\text{Detected nonseeded faults}}{\text{Total nonseeded faults}}$$

and we can use this data to estimate test efficiency and to estimate the number of remaining faults.

Acceptance Testing:

- **Introduction to Acceptance Testing:**
- **Acceptance testing** is testing conducted to determine whether a system satisfies its acceptance criteria.
- There are two categories of acceptance testing:
    1. **User Acceptance Testing (UAT):** It is conducted by the customer to ensure that the system satisfies the contractual acceptance criteria before being signed-off as meeting user needs. This is the final test performed. The main purpose of this testing is to validate the software against the business requirements. This validation is carried out by the end-users who are familiar with the business requirements.
    2. **Business Acceptance Testing (BAT):** It is undertaken within the development organization of the supplier to ensure that the system will eventually pass the user acceptance testing. This is to assess whether the product meets the business goals and purposes or not.
- **The 3 main goals of accepting testing are:**
    1. Confirm that the system meets the agreed upon criteria.
    2. Identify and resolve discrepancies, if there are any.
    3. Determine the readiness of the system for cut-over to live operations.
- **The acceptance criteria are defined on the basis of the following attributes:**
    1. Functional Correctness and Completeness
    2. Accuracy
    3. Data Integrity
    4. Data Conversion
    5. Backup and Recovery
    6. Competitive Edge
    7. Usability
    8. Performance
    9. Start-up Time
    10. Stress
    11. Reliability and Availability
    12. Maintainability and Serviceability
    13. Robustness
    14. Timeliness
    15. Confidentiality and Availability
    16. Compliance
    17. Installability and Upgradability
    18. Scalability
    19. Documentation
- **Selection of Acceptance Criteria:**
- The acceptance criteria discussed are usually too general, so the customer needs to select a subset of the quality attributes.
- The quality attributes are then prioritized to the specific situation.
- Ultimately, the acceptance criteria must be related to the business goals of the customer's organization.

- Example of an acceptance test plan

1. Introduction
2. Acceptance test category For each category of acceptance criteria (a) Operation environment (b) Test case specification (i) Test case Id# (ii) Test title (iii)Test objective (iv) Test procedure
3. Schedule
4. Human resources

- **Acceptance Test Execution:**
- The acceptance test cases are divided into two subgroups:
  - The first subgroup consists of basic test cases.
  - The second subgroup consists of test cases that are more complex to execute.
- The acceptance tests are executed in two phases:
  - In the first phase, the test cases from the basic test group are executed.
  - If the test results are satisfactory then the second phase, in which the complex test cases are executed, is taken up.
  - In addition to the basic test cases, a subset of the system-level test cases are executed by the acceptance test engineers to independently confirm the test results.
- Acceptance test execution activity includes the following detailed actions:
  - The developers train the customer on the usage of the system.
  - The developers and the customer coordinate the fixing of any problem discovered during acceptance testing.
  - The developers and the customer resolve the issues arising out of any acceptance criteria discrepancy.
- The acceptance test engineer may create an Acceptance Criteria Change (ACC) document to communicate the deficiency in the acceptance criteria to the supplier.
- An ACC report is generally given to the supplier's marketing department through the on-site system test engineers.
- E.g. of an ACC document

1. ACC Number:	A unique number
2. Acceptance Criteria Affected:	The existing acceptance criteria
3. Problem/Issue Description:	Brief description of the issue
4. Description of Change Required:	Description of the changes needed to be done to the original acceptance criterion
5. Secondary Technical Impacts:	Description of the impact it will have on the system
6. Customer Impacts:	What impact it will have on the end user
7. Change Recommended by:	Name of the acceptance test engineer(s)
8. Change Approved by:	Name of the approver(s) from both the parties

- **Acceptance Test Report:**
- The acceptance test activities are designed to reach at a conclusion:
  - Accept the system as delivered.
  - Accept the system after the requested modifications have been made.
  - Do not accept the system.
- Usually some useful intermediate decisions are made before making the final decision.
- A decision is made about the continuation of acceptance testing if the results of the first phase of acceptance testing is not promising. If the test results are unsatisfactory, changes are made to the system before acceptance testing can proceed to the next phase.
- During the execution of acceptance tests, the acceptance team prepares a test report on a daily basis.
  I.e. During the execution of acceptance tests, a daily acceptance test report is made.
- E.g. of a daily acceptance test report

1. Date:	Acceptance report date
2. Test case execution status:	Number of test cases executed today Number of test cases passing Number of test cases failing
3. Defect identifier:	Submitted defect number Brief description of the issue
4. ACC number(s):	Acceptance criteria change document number(s), if any
5. Cumulative test execution status:	Total number of test cases executed Total number of test cases passing Total number of test cases failing Total number of test cases not executed yet

- At the end of the first and the second phases of acceptance testing an acceptance test report is generated.
- E.g. of a finalized acceptance test report

1. Report identifier
2. Summary
3. Variances
4. Summary of results
5. Evaluation
6. Recommendations
7. Summary of activities
8. Approval

- **Acceptance Testing in Extreme Programming:**
- In the XP framework, the user stories are used as the acceptance criteria.
- The user stories are written by the customer as things that the system needs to do for them.
- Several acceptance tests are created to verify the user story has been correctly implemented.
- The customer is responsible for verifying the correctness of the acceptance tests and reviewing the test results.
- A story is incomplete until it passes its associated acceptance tests.
- Ideally, acceptance tests should be automated, either using the unit testing framework, before coding.
- The acceptance tests take on the role of regression tests.

Static Analysis:

- Static analysis is a method of computer program debugging that is done by examining the code without executing the program.
- This process provides an understanding of the code structure and can help ensure that the code adheres to industry standards.
- Automated tools can assist programmers and developers in carrying out static analysis. The software will scan all code in a project to check for vulnerabilities while validating the code.
- Static analysis is generally good at finding coding issues such as:
    - Programming errors
    - Coding standard violations
    - Undefined values
    - Syntax violations
    - Security vulnerabilities
- Once the code is written, a static code analyzer should be run to look over the code. It will check against defined coding rules from standards or custom predefined rules. Once the code is run through the static code analyzer, the analyzer will have identified whether or not the code complies with the set rules. It is sometimes possible for the software to flag false positives, so it is important for someone to go through and dismiss any. Once false positives are waived, developers can begin to fix any apparent mistakes, generally starting from the most critical ones. Once the code issues are resolved, the code can move on to testing through execution.
- Example of static analysis tools include:
    - FindBugs
    - JLint
    - JSHint
- Different tools find different bugs.
- Benefits of using static analysis include:
    - It can evaluate all the code in an application, increasing code quality.
    - Automated tools are less prone to human error and are faster.
    - It will increase the likelihood of finding vulnerabilities in the code, increasing web or application security.
    - It can be done in an offline development environment.
- Drawbacks of using static analysis include:
    - False positives can be detected.
    - It will detect harmless bugs that may not be worth fixing.
    - A tool might not indicate what the defect is if there is a defect in the code.
    - Static analysis can't detect how a function will execute.

Quality:

- **Introduction to Quality:**
- Quality is value to some person.
- Quality is fitness to purpose.
- Quality is exceeding the customer's expectations.
- **Quality in Use:** The user's view of the quality of a system.
  I.e. What's the end-user's experience?
- **External Quality Attributes:** External quality determines the fulfillment of stakeholder requirements. It is about the functionality of the system.
  I.e. Does it pass all the tests?
- **Internal Quality Attributes:** Internal quality has to do with the way that the system has been constructed.
  I.e. Is it well-designed?
- **Process Quality:** Process quality focuses on the steps of manufacturing the product.
  I.e. Is it assembled correctly?
- **Quality Assurance (QA):**
- Verification and validation (V&V) focuses on the quality of the product.
- QA focuses on the quality of the processes. It focuses on improving the software development process and making it more efficient and more effective.
- It looks at:
  - How well are the processes documented?
  - How well do people follow these processes?
  - Does the organisation measure key quality indicators?
  - Does the organisation learn from its mistakes?
- Examples of QA standards and practices are:
  - ISO9001
  - TickIt
  - Capability Maturity Model (CMM)
  - Total Quality Management (TQM)
- **A History of Managing Quality for Industrial Engineering:**
- **Product Inspection** (1920s): Examine intermediate and final products and discard defective items.
- **Process Control** (1960s): Monitor defect rates to identify defective process elements & control the process.
- **Design Improvement** (1980s): Engineering the process and the product to minimize the potential for defects.
- **Total Quality Management (TQM):**
- **Total quality management** is the continual process of detecting and reducing or eliminating errors in manufacturing, streamlining supply chain management, improving the customer experience, and ensuring that employees are up to speed with training.
- Total quality management aims to hold all parties involved in the production process accountable for the overall quality of the final product or service.
- TQM uses statistical methods to analyze industrial production processes.
- The basic principle of TQM is counter-intuitive: In the event of a defect, don't adjust the controller or you'll make things worse. Instead, analyze the process and improve it.
- It was developed by William Deming.
- While TQM shares much in common with the Six Sigma improvement process, it is not the same as Six Sigma. TQM focuses on ensuring that internal guidelines and process standards reduce errors, while Six Sigma looks to reduce defects.

- **Six Sigma:**
- **Six Sigma** is a quality-control methodology developed in 1986 by Motorola.
- The method uses a data-driven review to limit mistakes or defects in a corporate or business process. The key ideas are to use statistics to measure defects and to design the process to reduce defects.
- Six Sigma emphasizes cycle-time improvement while at the same time reducing manufacturing defects to a level of no more than 3.4 occurrences per million units or events.
- Six Sigma points to the fact that, mathematically, it would take a six-standard-deviation event from the mean for an error to happen.
- **Quality Management for Software:**
- All defects are design errors, not manufacturing errors.
- Process improvement principles still apply to the design process.
- **Defect removal:**
- There are two ways to remove defects:
    1. Fix the defects in each product. (I.e patch the product)
    2. Fix the process that leads to defects. (I.e. prevent defects from occurring)
- The latter is cost effective as it affects all subsequent projects.
- **Defect prevention:**
- Programmers must evaluate their own errors.
- Feedback is essential for defect prevention.
- There is no single cure-all for defects. They must be eliminated one by one.
- Process improvement must be an integral part of the process.
- Process improvement takes time to learn.
- **Six Sigma Might not be Suitable for Software:**
- Software processes depend on human behaviour, meaning they are not predictable.
- Software characteristics are not ordinal:
    - We cannot measure the degree of conformance for software.
    - The mapping between software faults and failures is many-to-many.
    - Not all software anomalies are faults.
    - Not all failures result from the software itself.
    - We cannot accurately measure the number of faults in software.
- Typical defect rates:
    - NASA Space shuttle: 0.1 failures/KLOC
    - Best military systems: 5 faults/KLOC
    - Worst military systems: 55 faults/KLOC
    - Six Sigma would demand 0.0034 faults/KLOC
- **Process Modeling & Improvement:**
- **Process Description:** Understand and describe the current practices.
- **Process Definition:** Prescribe a process that reflects the organization's goals.
- **Process Customization:** Adapt the prescribed process model for each individual project.
- **Process Enactment:** Carry out the process. This means developing the software and collecting process data.
- **Process improvement:** Use lessons learned from each project to improve the prescriptive model. I.e. Analyze defects to eliminate causes.
- **Capability Maturity Model (CMM):**
- The **Capability Maturity Model** is a process improvement approach developed specially for software process improvement.
- CMM has 5 levels. An organization is certified at CMM level 1 to 5 based on the maturity of their quality assurance mechanisms.

- CMM Levels

Level	Characteristic	Key Challenges
1 (Initial)	In this stage the quality environment is unstable. Simply, no processes have been followed or documented. Ad hoc/Chaotic. No cost estimation, planning, management.	Project Management Project Planning Configuration Management Change Control Software Quality Assurance
2 (Repeatable)	Some processes are followed which are repeatable. This level ensures processes are followed at the project level. Processes are dependent on individuals.	Establish a process group Identify a process architecture Introduce SE methods and tools
3 (Defined)	A set of processes are defined and documented at the organizational level. Those defined processes are subject to some degree of improvement. A process is defined and institutionalized.	Process measurement Process analysis Quantitative Quality Plans
4 (Managed)	This level uses process metrics and effectively controls the processes that are followed. The process is measured.	Automatic collection of process data Use process data to analyze and modify the process
5 (Optimizing)	This level focuses on the continuous improvements of the processes through learning & innovation. Improvement and feedback are fed back into the process.	Identify process indicators Empower individuals

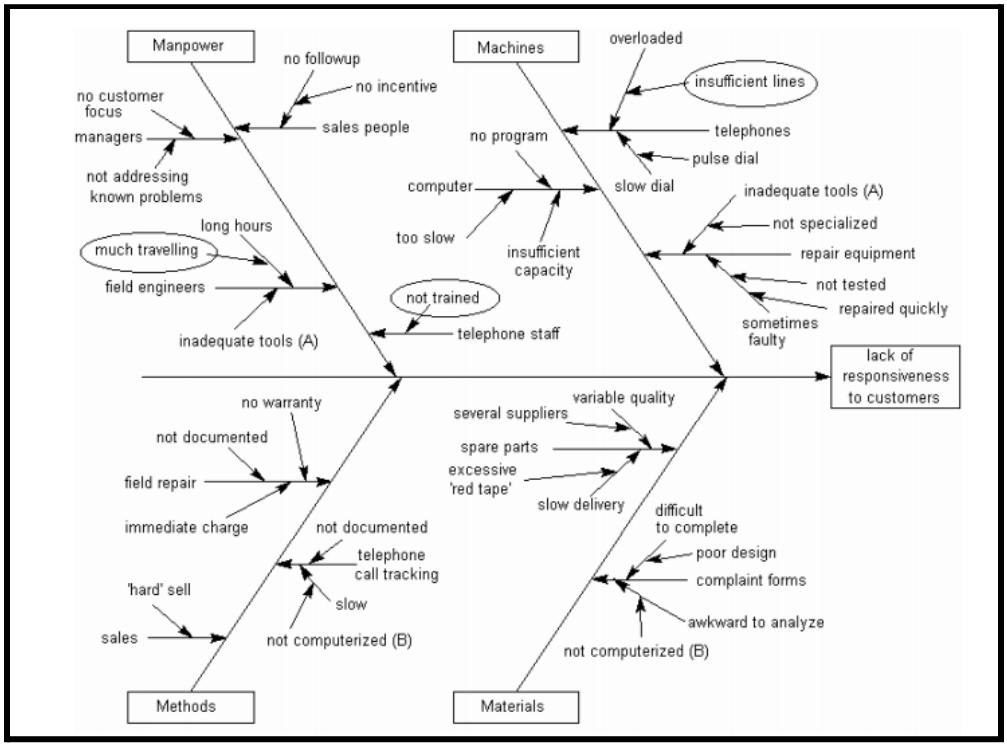
- **Arguments against QA:**

1. The costs may outweigh the benefits:
    - Costs: Increased documentation, more meetings, etc.
    - Benefits: Improved quality of the process outputs.
2. Reduced agility:
    - Documenting the processes makes them less flexible.
3. Reduced thinking:
    - Following the defined process gets in the way of thinking about the best way to do the job.
4. Barrier to Innovation:
    - New ideas have to be incorporated into the Quality Plan and get signed off.
5. Demotivation:
    - Extra bureaucracy makes people frustrated.

- **ISO 9000:**
- The ISO 9000 family of quality management systems is a set of standards that helps organizations ensure they meet customer and other stakeholder needs within statutory and regulatory requirements related to a product or service.
- ISO 9000 deals with the fundamentals of QMS, including the seven quality management principles that underlie the family of standards. It deals with the management systems

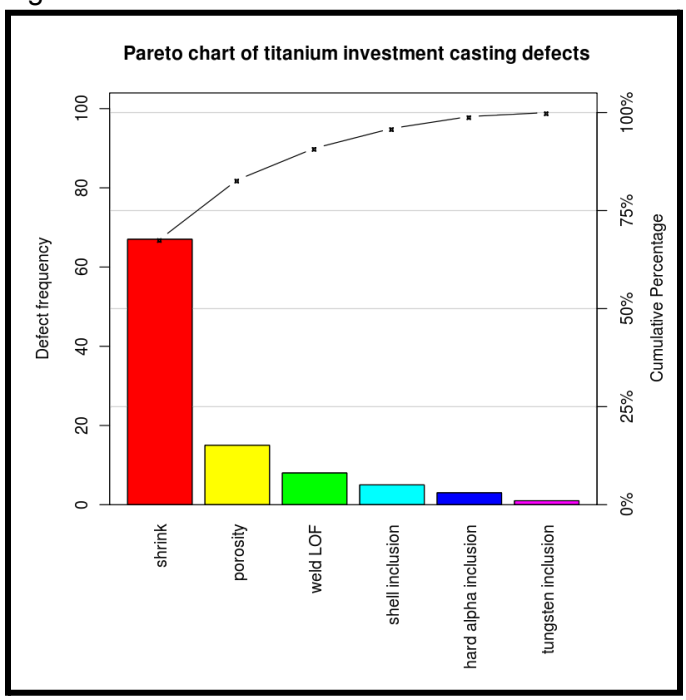
used by organizations to ensure quality in design, production, delivery, and support products.

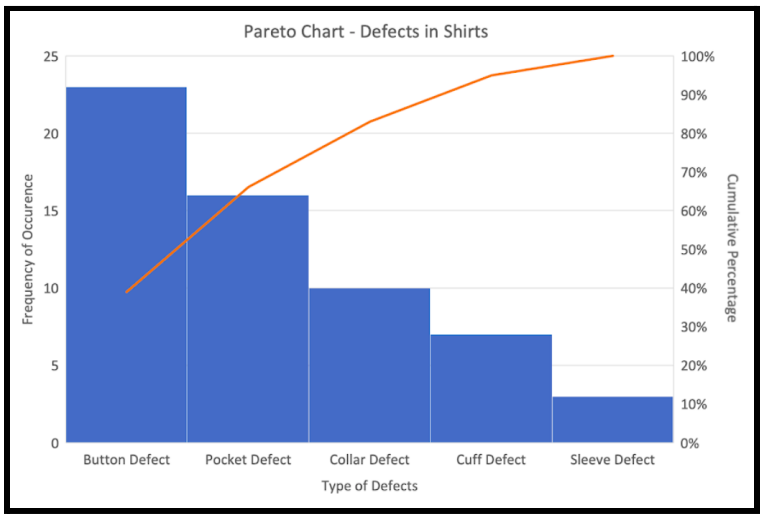
- ISO 9001 deals with the requirements that organizations wishing to meet the standard must fulfil. It includes several important changes for quality management systems, including modifications in terminology, the introduction of new context-based clauses, emphasis on management's role in quality, and a focus on risk-based approach.
- **Ishikawa (Fishbone) Diagram:**
- The fishbone diagram identifies many possible causes for an effect or problem.
- It can be used to structure a brainstorming session.
- It immediately sorts ideas into useful categories.
- We should use a fishbone diagram:
    - When identifying possible causes for a problem.
    - When a team's thinking tends to fall into ruts.
- E.g.



- **Pareto Chart:**
- A **Pareto chart** is a bar graph and a line graph. The lengths of the bars represent frequency or cost (time or money), and are arranged with longest bars on the left and the shortest to the right. In this way the chart visually depicts which situations are more significant. The line represents the cumulative percentage of defects.
- The left vertical axis is the frequency of occurrence. The right vertical axis is the cumulative percentage of the total number of occurrences.
- Pareto charts are useful to find the defects to prioritize in order to observe the greatest overall improvement.
- The Pareto chart is one of the seven basic tools of quality control.

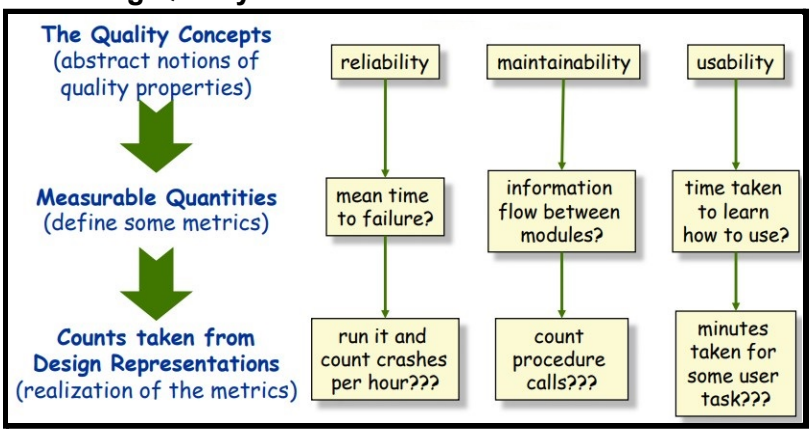
- E.g.



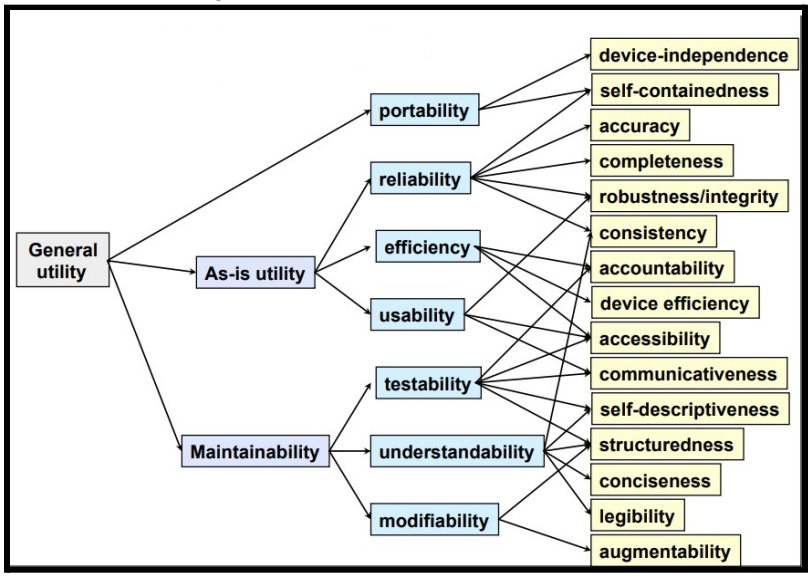


- **How to assess software quality:**
- Reliability:
- The designer must be able to predict how the system will behave:
    - Completeness - Does it do everything it is supposed to do?
    - Consistency - Does it always behave as expected?
    - Robustness - Does it behave well under abnormal conditions?
- Efficiency:
    - How efficient is the use of resources such as processor time, memory, network bandwidth?
    - This is less important than reliability in most cases.
- Maintainability:
    - How easy will it be to modify in the future?
- Usability:
    - How easy is it to use?

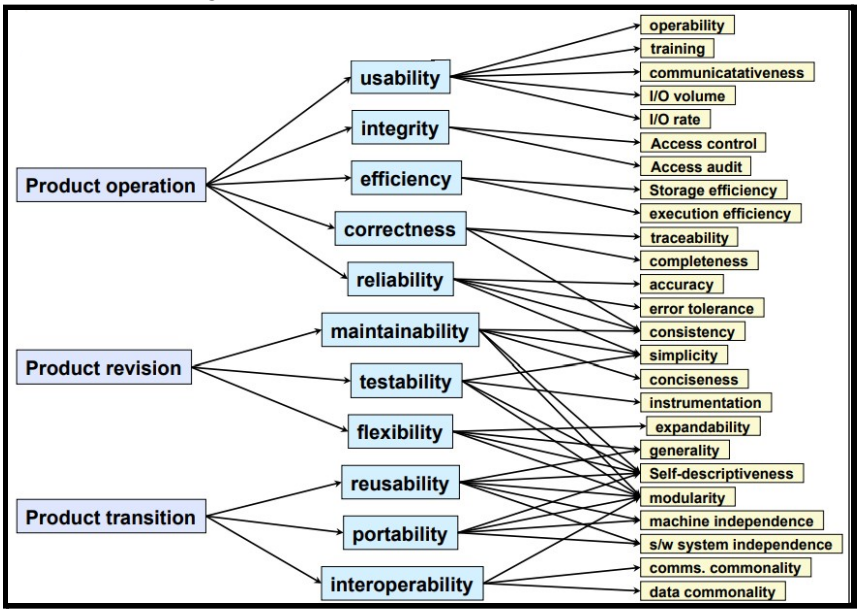
- **Measuring Quality:**



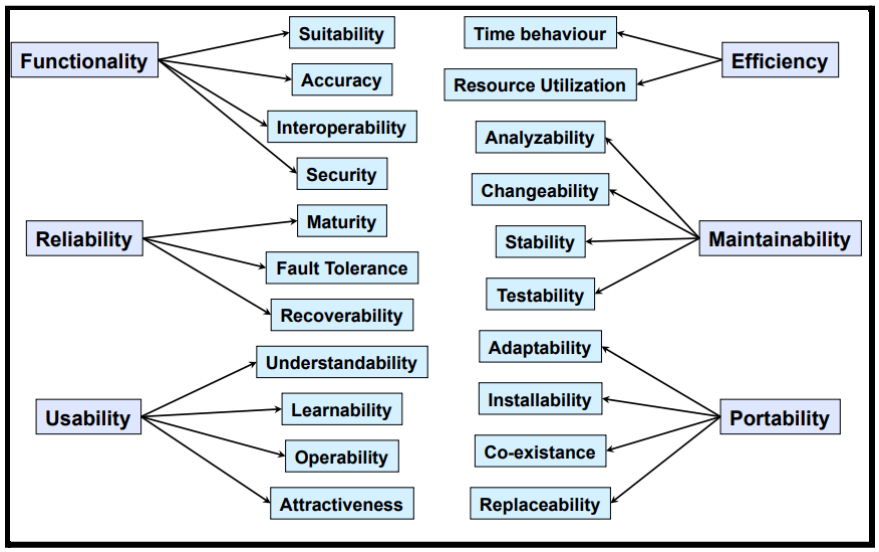
- **Boehm's Quality Map:**



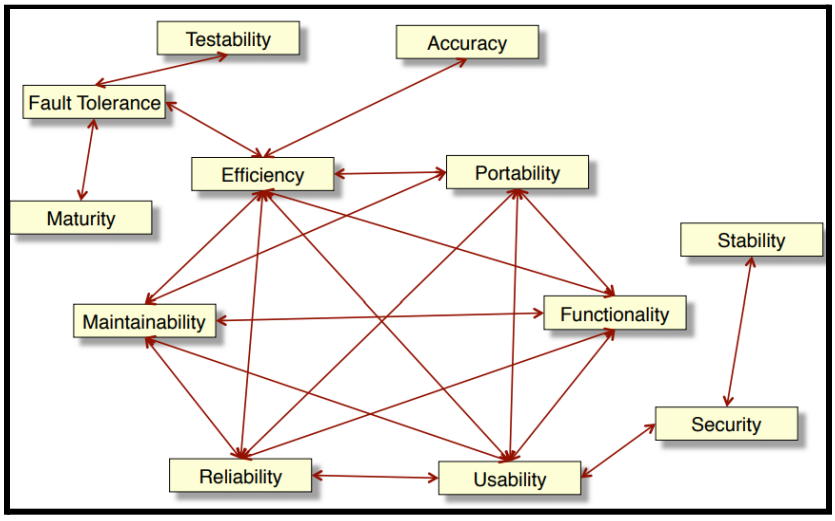
- **McCall's Quality Map:**



- **ISO/IEC 9126:**



- **Conflicts between Quality factors:**



We can summarize the above picture into the below one:

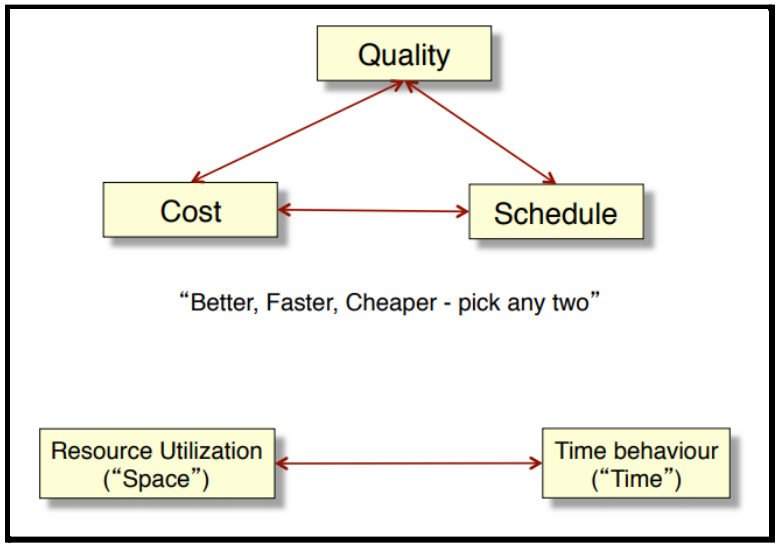


Table of Contents:

UML:

- **Introduction to UML:**
- **Unified Modeling Language (UML)** allows us to express the design of a program before writing any code.
- It is language-independent.
- It is an extremely expressive language.
- UML is a graphical language for visualizing, specifying, constructing, and documenting information about software-intensive systems.
- UML can be used to develop diagrams and provide programmers with ready-to-use, expressive modeling examples. Some UML tools can generate program language code from UML. UML can be used for modeling a system independent of a platform language.
- UML is a picture of an object oriented system. Programming languages are not abstract enough for object oriented design. UML is an open standard and lots of companies use it.
- Legal UML is both a descriptive language and a prescriptive language. It is a descriptive language because it has a rigid formal syntax, like programming languages, and it is a prescriptive language because it is shaped by usage and convention.
- It's okay to omit things from UML diagrams if they aren't needed by the team/supervisor/instructor.
- **History of UML:**
- In an effort to promote object oriented designs, three leading object oriented programming researchers joined forces to combine their languages. They were:
    1. Grady Booch (BOOCH)
    2. Jim Rumbaugh (OML: object modeling technique)
    3. Ivar Jacobsen (OOSE: object oriented software eng)
- They came up with an industry standard in the mid 1990's.
- UML was originally intended as a design notation and had no modelling associated with it.
- **UML diagrams can help engineering teams:**
- Bring new team members or developers switching teams up to speed quickly.
- Navigate source code.
- Plan out new features before any programming takes place.
- Communicate with technical and non-technical audiences more easily.
- **Uses of UML:**

1. It can be used as a sketch to communicate aspects of the system.
- **Forward design:** Doing UML before coding.
- **Backward design:** Doing UML after coding as documentation.
2. It can be used as a blueprint to show a complete design that needs to be implemented. This is sometimes done with CASE (Computer-Aided Software Engineering) tools. One of these tools is visual paradigm.
3. It can be used as a programming language.
- Some UML tools can generate program language code from UML.
4. As a sketch:
- Can be used to sketch a high level view of the system.
- **Forward engineering:** Describes the concepts we need to implement.
- **Reverse engineering:** Explains how parts of the code work.
5. As a blueprint:
- Should be complete and describes the system in detail.
- **Forward engineering:** Model as a detailed specification for the programmer.
- **Reverse engineering:** Model as a code browser.

- Tools provide both forward and reverse engineering to move back and forth between the program and the code.

6. As a programming language:

- UML diagrams can be automatically compiled into working code using sophisticated tools, such as Visual Paradigm.

- **Things to Model:**
- Structure of the code:
- Code dependencies.
- Components and couplings.
- Behaviour of the code:
- Execution traces.
- State machine models of complex objects.
- Function of the code:
- What function does it provide to the user?

Class Diagram:

- **Introduction to Class Diagrams:**
- A class describes a group of objects with:
- Similar attributes
- Common operations
- Common relationships with other objects
- Common meaning
- A **class diagram** describes the structure of an object oriented system by showing the classes in that system and the relationships between the classes. A class diagram also shows the constraints, and attributes of classes. It displays the system's classes, attributes, and methods. It is helpful in recognizing the relationship between different objects as well as classes.
  I.e.
  A UML class diagram is a picture of:
  - The classes in an object oriented system.
  - Their fields and methods.
  - Connections between the classes that interact or inherit from each other.
- Some things that are not represented in a UML class diagram are:
  - Details of how the classes interact with each other.
  - Algorithmic details, like how a particular behavior is implemented.
- **Note:** Coupling between classes must be kept low, while cohesion within a class must be kept high. Furthermore, we should respect the SOLID principles.
- UML class diagrams cans show:
  1. Division of responsibility
  2. Subclassing/Inheritance
  3. Visibility of objects and methods
  4. Aggregation/Composition
  5. Interfaces
  6. Dependencies
- **Notation:**
- Naming Convention:

1. **Class name**

- Use **<<interface>>** on top of interface names.
- To show that a class is abstract, either italicize the class name or put **<<abstract>>** on top of the abstract class name.

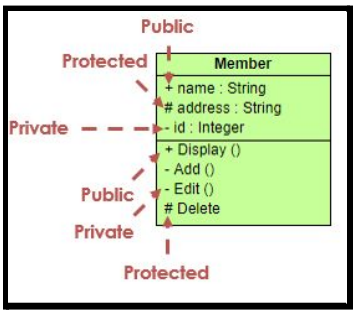
2. **Data members/Attributes**

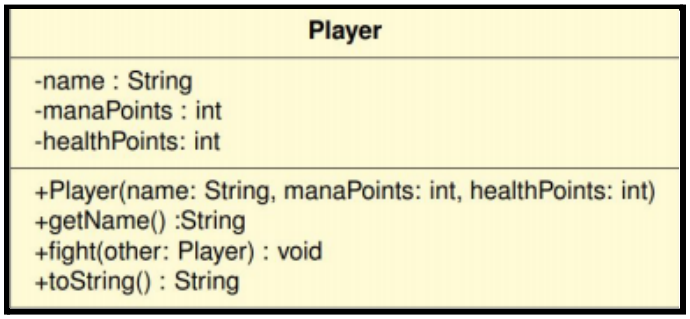
- The data members section of a class lists each of the class's data members on a separate line.
- Each line uses this format: **attributeName : type**
  E.g. **name : String**
- We must underline static attributes.

3. **Methods/Operations**

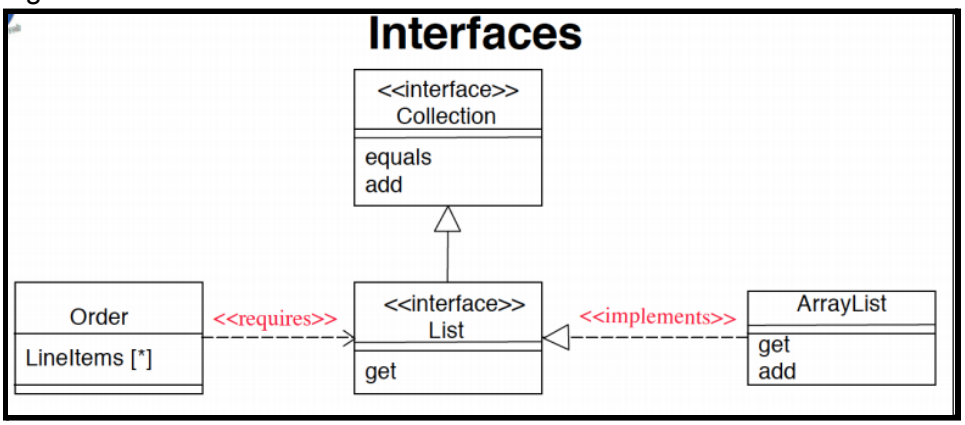
- The methods of a class are displayed in a list format, with each method on its own line.
- Each line uses this format:
  methodName(param1: type1, param2: type2, ...) : returnType
  E.g. **distance(p1: Point, p2: Point) : Double**
- We may omit setters and getters. However, don't omit any methods from an interface.
- Furthermore, do not include inherited methods.
- We must underline static methods.

- Visibility:
- − means that it is private.
- + means that it is public.
- # means that it is protected.
- ~ means that it is a package.
- / means that it is a **derived attribute**. A derived attribute is an attribute whose value is produced or computed from other information.
- **Note:** Everything except / is common for both methods and attributes.
- E.g.



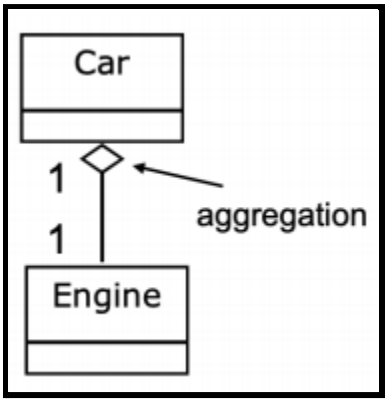


- Inheritance/Generalization and Realization Relationships:
- **Generalization/inheritance** is when a class extends another class while **realization** is when a class implements an interface.
- Generalization represents a “IS-A” relationship.
- Hierarchies are drawn top down with arrows pointing upward to the parent class. I.e. The parent class is above the child class and the arrow goes from the child class to the parent class.
- For a class, draw a solid line with a black arrow pointing to the parent class.
- For an abstract class, draw a solid line with a white arrow pointing to the parent abstract class.
- For an interface, draw a dashed line with a white arrow pointing to the interface.
- E.g.



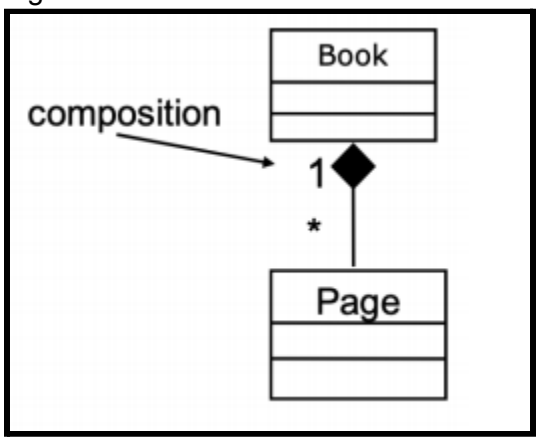
- Association:
- An **association** represents a relationship between two classes. It also defines the multiplicity between objects.
- Association can be represented by a line between the classes with an arrow indicating the navigation direction.
  Note: Sometimes, association can be represented just by a line between the classes. This means that information can flow in both directions.

- We need the following items to represent association between 2 classes:
    1. The multiplicity
    2. The name of the relationship
    3. The direction of the relationship
- Aggregation, composition and dependency are all types of association.
- Multiplicity:
- * means 0 or more.
- 1 means 1 exactly.
- 2..4 means 2 to 4, inclusive.
- 3..* means 3 or more.
- There are other relationships such as 1-to-1, 1-to-many, many-to-1 and many-to-many.
- Aggregation:
- A special type of association.
- Aggregation implies a relationship where the child class can exist independently of the parent class. This means that if you remove/delete the parent class, the child class still exists.
  I.e. Aggregation represents a “HAS-A” or “PART-OF” relationship.
  E.g. Say we have 2 classes, Teacher (the parent class) and Student (the child class). If we delete the Teacher class, the Student class still exists.
- Aggregation is symbolized by an arrow with a clear white diamond arrowhead pointing to the parent class.
- E.g.

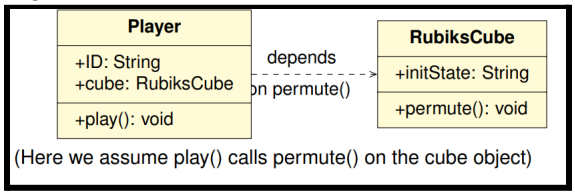


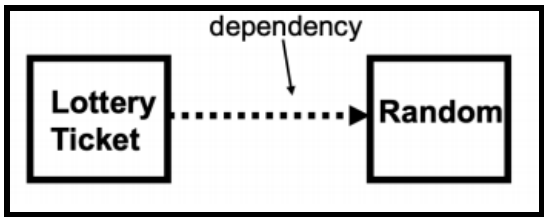
- Aggregation is considered as a weak type of association.
- Composition:
- A special type of association. Composition is considered as a strong type of association.
- It is a stronger version of aggregation where if you delete the parent class, then all the child classes are also deleted.
  I.e. Composition represents a “ENTIRELY MADE OF” relationship.
  E.g. Say we have 2 classes, House (the parent class) and Room (the child class). If we delete the House class, the Room class is also deleted.
- Composition is symbolized by an arrow with a black diamond arrowhead pointing to the parent class.

- E.g.

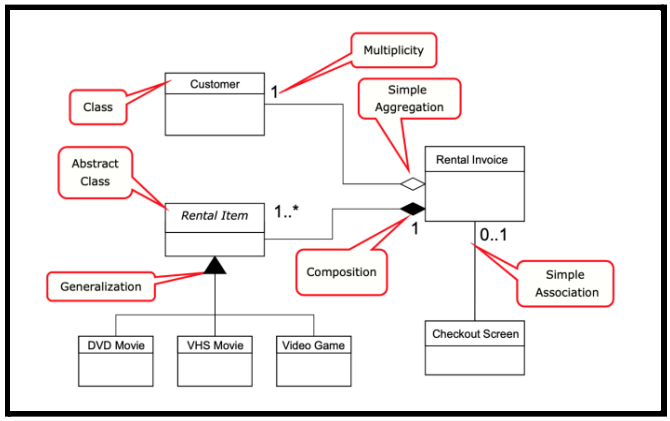


- Dependency:
- Is a special type of association.
- Dependency indicates a “uses” relationship between two classes. If a change in structure or behaviour of one class affects another class, then there is a dependency between those two classes.
- Dependency is represented by a dotted arrow where the arrowhead points to the independent element.
- E.g.





- Examples of UML class diagrams:



- How to draw class diagrams:
    1. Identify the objects in the problem and create classes for each of them
    2. Add attributes
    3. Add operations
    4. Connect classes with relationships
    5. Specify the multiplicities for association connections.

- **Summary:**
- Naming Convention:

1. **Class name**

- Use **<<interface>>** on top of interface names.
- To show that a class is abstract, either italicize the class name or put **<<abstract>>** on top of the abstract class name.

2. **Data members/Attributes**

- The data members section of a class lists each of the class's data members on a separate line.
- Each line uses this format: **attributeName : type**
  E.g. **name : String**
- We must underline static attributes.

3. **Methods/Operations**

- The methods of a class are displayed in a list format, with each method on its own line.
- Each line uses this format:
  methodName(param1: type1, param2: type2, ...) : returnType
  E.g. **distance(p1: Point, p2: Point) : Double**
- We may omit setters and getters. However, don't omit any methods from an interface.
- Furthermore, do not include inherited methods.
- We must underline static methods.
- Visibility:
- − means that it is private.
- + means that it is public.
- # means that it is protected.
- ~ means that it is a package.
- / means that it is a **derived attribute**. A derived attribute is an attribute whose value is produced or computed from other information.
- **Note:** Everything except / is common for both methods and attributes.
- Multiplicity:
- * means 0 or more.
- 1 means 1 exactly.
- 2..4 means 2 to 4, inclusive.
- 3..* means 3 or more.
- There are other relationships such as 1-to-1, 1-to-many, many-to-1 and many-to-many.

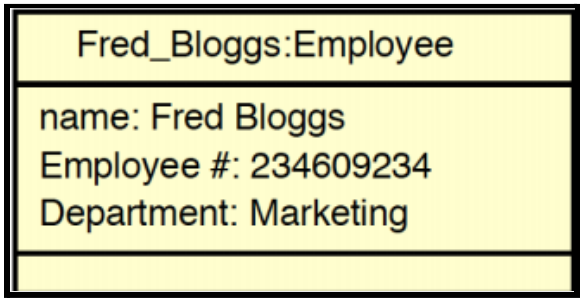
- Others:

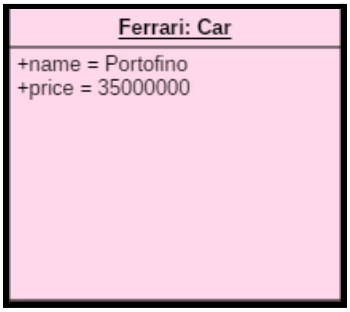
Item	Explanation	Depiction
Generalization/inheritance	When a class extends another class. Generalization represents a “IS-A” relationship.	For a class, draw a solid line with a black arrow pointing to the parent class. For an abstract class, draw a solid line with a white arrow pointing to the parent abstract class. For an interface, draw a dashed line with a white arrow pointing to the interface.
Realization	When a class implements an interface	
Association	Represents a relationship between two classes. It also defines the multiplicity between objects.	A line between the classes with an arrow indicating the navigation direction. **Note:** Sometimes, association can be represented just by a line between the classes. This means that information can flow in both directions.
Aggregation	A special type of association. It is a weak type of association. Aggregation implies a relationship where the child class can exist independently of the parent class. This means that if you remove/delete the parent class, the child class still exists. I.e. Aggregation represents a “HAS-A” or “PART-OF” relationship.	An arrow with a clear white diamond arrowhead pointing to the parent class.

Composition	A special type of association. Composition is considered as a strong type of association. It is a stronger version of aggregation where if you delete the parent class, then all the child classes are also deleted. I.e. Composition represents a "ENTIRELY MADE OF" relationship.	An arrow with a black diamond arrowhead pointing to the parent class.
Dependency:	Is a special type of association. Dependency indicates a "uses" relationship between two classes. If a change in structure or behaviour of one class affects another class, then there is a dependency between those two classes.	A dotted arrow where the arrowhead points to the independent element.

Object Diagram:

- Object diagrams look very similar to class diagrams.
- **Naming Convention:**
  Object name: Type
  Attribute: Value (Sometimes, it's Attribute = Value)
  E.g.



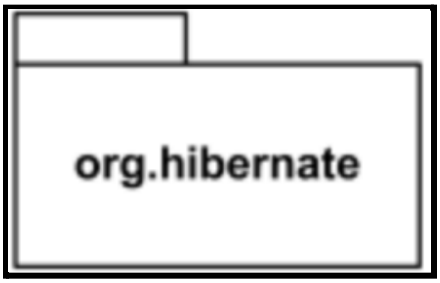


- **Note:** 2 different objects may have identical attribute values.
- **Purpose:**
- It is used to describe the static aspect of a system.
- It is used to represent an instance of a class.
- It can be used to perform forward and reverse engineering on systems.
- It is used to understand the behavior of an object.
- It can be used to explore the relations of an object and can be used to analyze other connecting objects.

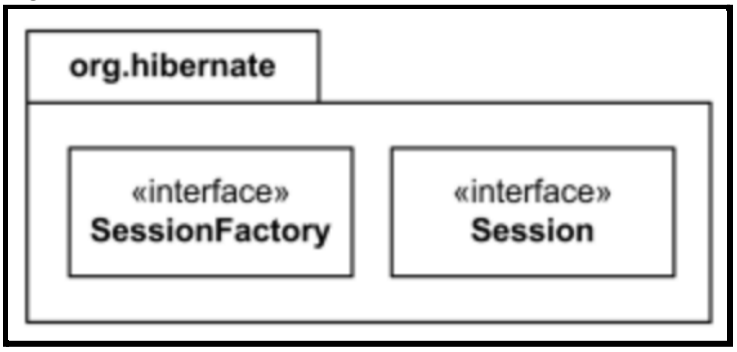
UML Packages:

- **Introduction:**
- A **package** is a namespace used to group together elements that are semantically related and might change together. It is a general purpose mechanism to organize elements into groups to provide a better structure for a system model.
- **UML package diagrams** are structural diagrams used to show the organization and arrangement of various model elements in the form of packages. A package is a grouping of related UML elements, such as diagrams, documents, classes, or even other packages. Each element is nested within the package, which is depicted as a file folder, and then is arranged hierarchically within the diagram. Package diagrams are most commonly used to provide a visual organization of the layered architecture within any UML classifier, such as a software system.
- Package diagrams are UML structure diagrams which show packages and dependencies between the packages.
  Note: Structure diagrams do not utilize time related concepts and do not show the details of dynamic behavior.
- If a package is removed from a model, so are all the elements owned by the package.
- A package could also be a member of other packages.
- A package in the UML helps:
- To group elements.
- To provide a namespace for the grouped elements.
- Provide a hierarchical organization of packages.
- Benefits of UML package diagrams:
- They provide a clear view of the hierarchical structure of the various UML elements within a given system.
- These diagrams can simplify complex class diagrams into well-ordered visuals.
- They offer valuable high-level visibility into large-scale projects and systems.
- Package diagrams can be used to visually clarify a wide variety of projects and systems.
- These visuals can be easily updated as systems and projects evolve.
- **Terminology:**
- **Package:** A namespace used to group together logically related elements within a system. Each element contained within the package should be a packageable element and have a unique name.
- **Packageable element:** A named element, possibly owned directly by a package. These can include events, components, use cases, and packages themselves. Packageable elements can also be rendered as a rectangle within a package, labeled with the appropriate name.
- **Dependencies:** A visual representation of how one element or set of elements depends on or influences another. Dependencies are divided into two groups: access and import dependencies.
- **Access dependency:** Indicates that one package requires assistance from the functions of another package.
  I.e. One package requires help from functions of another package. (Making an API call for example)
- **Import dependency:** Indicates that functionality has been imported from one package to another.
  I.e. One package imports the functionality of another package. (Importing a package)

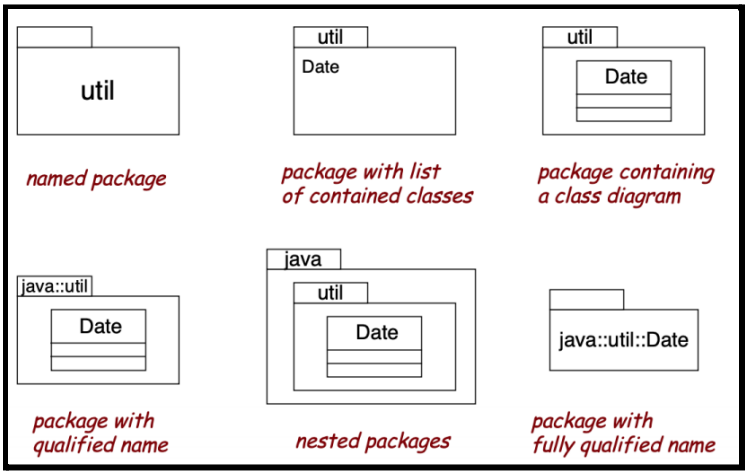
- **Notation:**
- A package is rendered as a rectangle with a small tab attached to the left side of the top of the rectangle. If the members of the package are not shown inside the package rectangle, then the name of the package should be placed inside.
  E.g.



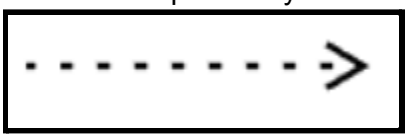
- The members/elements of the package may be shown within the boundaries of the package. If the names of the members of the package are shown, then the name of the package should be placed on the tab.
  E.g.



  Here, Package org.hibernate contains SessionFactory and Session.
- More examples:



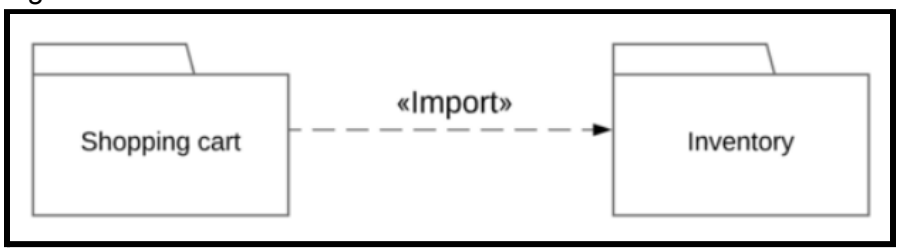
- To show a dependency between 2 packages, you draw a dotted arrow,



  , between the 2 packages such that the arrow is pointing to the independent package.

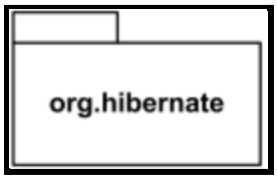
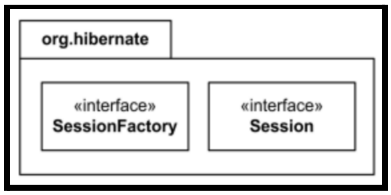
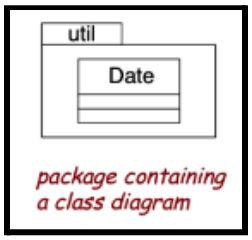
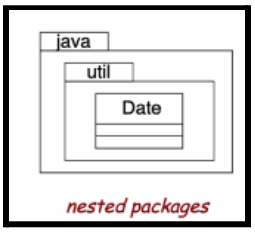
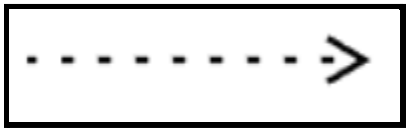
- To show an access dependency, write <<Access>> on the dotted arrow.
  E.g.

- To show an import dependency, write <<Import>> on the dotted arrow.
  E.g.



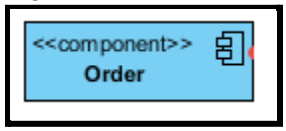
- **Criteria for Decomposing a System into Packages:**
- Different owners - who is responsible for working on which diagrams?
- Different applications - each problem has its own obvious partitions.
- Clusters of classes with strong cohesion - E.g. course, course description, instructor, student, etc.
- Or: Use an architectural pattern to help find a suitable decomposition such as the MVC Framework.
- **Other Guidelines for Packages:**
- Gather model elements with strong cohesion in one package.
- Keep model elements with low coupling in different packages.
- Minimize relationships, especially associations, between model elements in different packages.
- Namespace implication: An element imported into a package does not know how it is used in the imported package.
- We want to avoid dependency cycles.

- **Summary:**

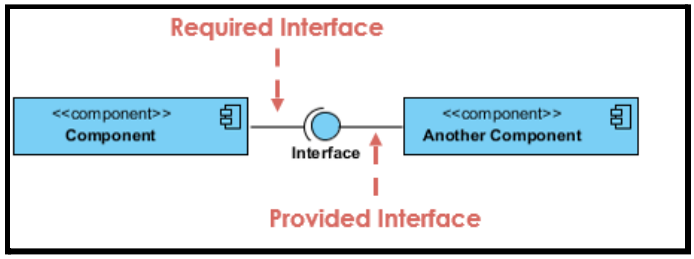
Item & Example	Description	Depiction
Package  	A namespace used to group together logically related elements within a system. Each element contained within the package should be a packageable element and have a unique name.	A rectangle with a small tab attached to the left side of the top of the rectangle. If the members of the package are not shown inside the package rectangle, then the name of the package should be placed inside.
Packageable element  	A named element, possibly owned directly by a package. These can include events, components, use cases, and packages themselves.	Packageable elements can also be rendered as a rectangle within a package, labeled with the appropriate name.
Dependency 	A visual representation of how one element or set of elements depends on or influences another. Dependencies are divided into two groups: access and import dependencies.	To show a dependency between 2 packages, you draw a dotted arrow, between the 2 packages such that the arrow is pointing to the independent package.
Access dependency 	Indicates that one package requires assistance from the functions of another package.	To show an access dependency, write <<Access>> on the dotted arrow.
Import dependency 	Indicates that functionality has been imported from one package to another.	To show an import dependency, write <<Import>> on the dotted arrow.

Component Diagrams:

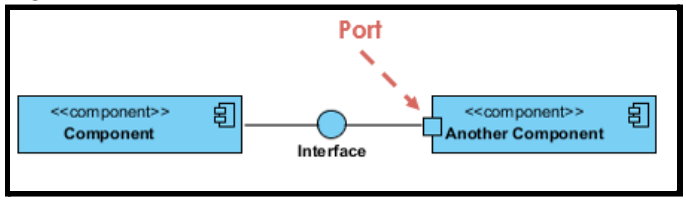
- **Introduction:**
- **Component diagrams** are used in modeling the physical aspects of object-oriented systems that are used for visualizing, specifying, and documenting component-based systems and also for constructing executable systems through forward and reverse engineering.
- A component diagram breaks down the actual system under development into various high levels of functionality.
- Each component is responsible for one clear aim within the entire system and only interacts with other essential elements on a need-to-know basis.
- Component diagrams can help your team:
    - Imagine the system's physical structure.
    - Pay attention to the system's components and how they relate.
    - Emphasize the service behavior as it relates to the interface.
- A component diagram gives a bird's-eye view of your software system. Understanding the exact service behavior that each piece of your software provides will make you a better developer. Component diagrams can describe software systems that are implemented in any programming language or style.
- **Notation:**
- **Component:** A rectangle with the component's name, stereotype text, and icon. A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.
  E.g.



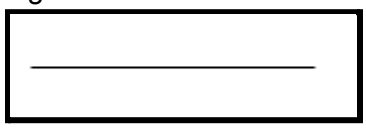
- **Interface:** There are 2 types of interfaces, **provided interface** and **required interface**.
  Provided interface: A complete circle with a line connecting to a component. Provided interfaces provide items to components.
  Required Interface: A half circle with a line connecting to a component. Required interfaces are used to provide required information to a provided interface.
  E.g.



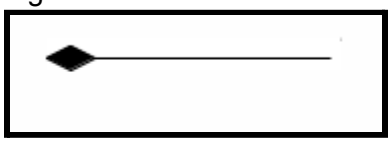
- **Port:** A square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component. Ports are used to hook up other elements in a component diagram.
  E.g.



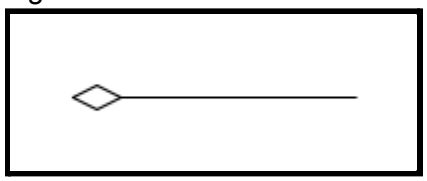
- **Association:** An association specifies a relationship that can occur between two instances. You represent an association using a straight line connecting 2 components.
  E.g.



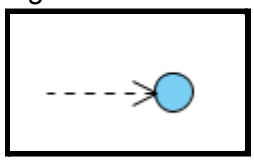
- **Composition:** Composition is a stronger form of aggregation that requires a part instance to be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Composition is a type of association. You can represent a composition using an arrow where the arrowhead is filled in and points to the parent class.
  E.g.



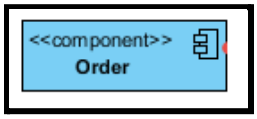
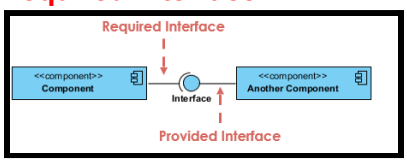
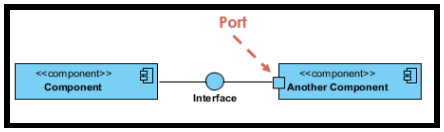
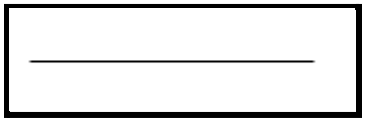
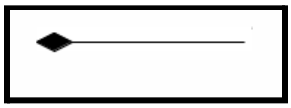
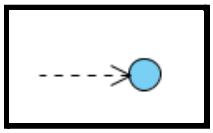
- **Aggregation:** Aggregation implies a relationship where the child class can exist independently of the parent class. This means that if you remove/delete the parent class, the child class still exists. It is a special type of association and a weak form of association. You can represent an aggregation using an arrow where the arrowhead is not filled in and points to the parent class.
  E.g.



- **Dependency:** A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. It is denoted as a dotted arrow with a circle at the tip of the arrow.
  E.g.



- **Summary:**

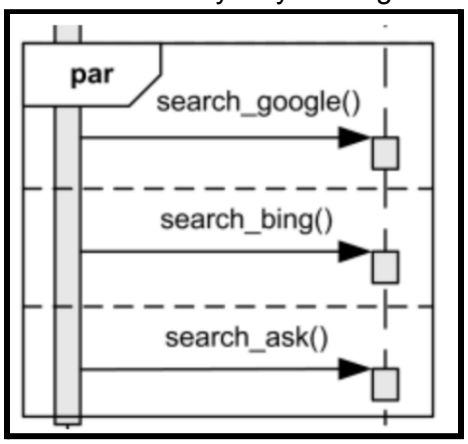
Item & Example	Description	Depiction
Component 	A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.	A rectangle with the component's name, stereotype text, and icon.
Provided interface **Required Interface** 	Provided interfaces provide items to components. Required interfaces are used to provide required information to a provided interface.	A complete circle with a line connecting to a component. A half circle with a line connecting to a component.
Port 	A port is often used to help expose required and provided interfaces of a component. Ports are used to hook up other elements in a component diagram.	A square along the edge of the system or a component.
Association 	An association specifies a relationship that can occur between two instances.	A straight line connecting 2 components.
Aggregation 	Aggregation implies a relationship where the child class can exist independently of the parent class. It is a weak form of association.	An arrow where the arrowhead is not filled in and points to the parent class.
Composition 	Composition is a stronger form of aggregation that requires a part instance to be included in at most one composite at a time. Composition is a type of association.	An arrow where the arrowhead is filled in and points to the parent class.
Dependency 	A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation.	It is denoted as a dotted arrow with a circle at the tip of the arrow.

Interaction Diagrams:

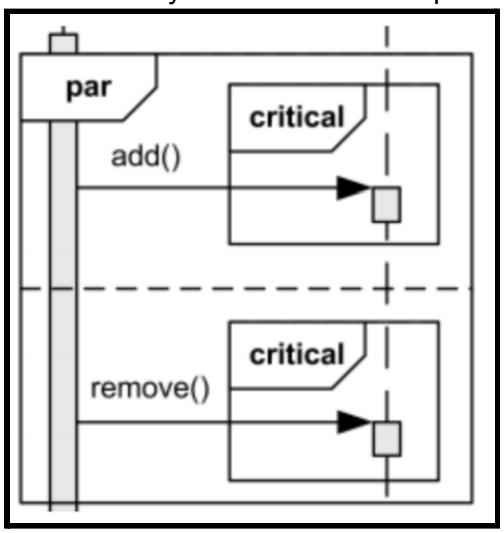
- **Interaction diagrams** describe how a group of objects collaborate in some behavior. They commonly contain objects, links and messages.
- Objects communicate with each other through function/method calls called **messages**.
- An interaction is a set of messages exchanged among a set of objects in order to accomplish a specific goal.
- Interaction diagrams:
  - Are used to model the dynamic aspects of a system.
  - Aid the developer in visualizing the system as it is running.
  - Are storyboards of selected sequences of message traffic between objects.
- After class diagrams, interaction diagrams are possibly the most widely used UML diagrams.
- A **lifeline** represents a single participant in an interaction. It describes how an instance of a specific classifier participates in the interaction. A lifeline represents a role that an instance of the classifier may play in the interaction.
- A **message** is the vehicle by which communication between objects is achieved. A function/method call is the most common type of message. The return of data as a result of a function call is also considered a message.
- A message may result in a change of state for the receiver of the message.
- The receipt of a message is considered an instance of an event.
- Interactions model the dynamic aspects of a system by showing the message traffic between a group of objects. Showing the time-ordering of the message traffic is a central ingredient of interactions.
- Graphically, a message is represented as a directed line that is labeled.
- The **sequence diagram** is the most commonly used UML interaction diagram. Typically a sequence diagram captures the behavior of a group of objects in a single scenario.
- Interaction Frame Operators:

Operator	**Name**	**Meaning**
Opt	Option	An operand is executed if the condition is true. (E.g. If-else)
Alt	Alternative	The operand, whose condition is true, is executed. (E.g. Switch)
Loop	Loop	It is used to loop an instruction for a specified period.
Break	Break	It breaks the loop if a condition is true or false, and the next instruction is executed.
Ref	Reference	It is used to refer to another interaction.
Par	Parallel	All operands are executed in parallel.
Region	Critical Region	Only 1 thread can execute this frame at a time.
Neg	Negative	Frame shows an invalid interaction.
Sd	Sequence Diagram	(Optional) Used to surround the whole diagram.

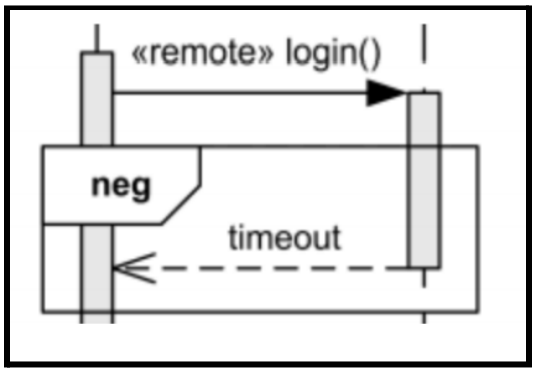
- **Parallel Example:** The interaction operator par defines potentially parallel execution of behaviors of the operands of the combined fragment. Different operands can be interleaved in any way as long as the ordering imposed by each operand is preserved.



- **Region Example:** The interaction operator region defines that the combined fragment represents a critical region. A critical region is a region with traces that cannot be interleaved by other occurrence specifications on the lifelines covered by the region.

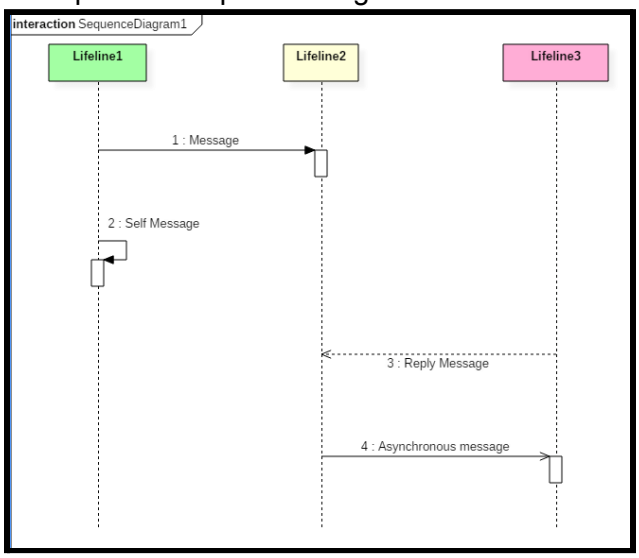


- **Negative Example:** The interaction operator neg describes a combined fragment of traces that are defined to be negative (invalid). Negative traces are the traces which occur when the system has failed. All interaction fragments that are different from the negative are considered positive, meaning that they describe traces that are valid and should be possible.

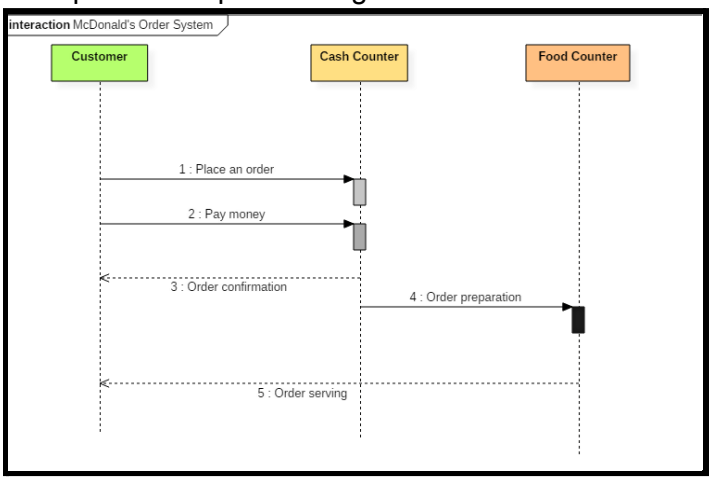


Sequence Diagrams:

- **Introduction:**
- A **sequence diagram** depicts interactions between objects in a sequential order. The purpose of a sequence diagram in UML is to visualize the sequence of a message flow in the system. The sequence diagram shows the interaction between two lifelines as a time-ordered sequence of events.
- A sequence diagram shows an implementation of a scenario in the system. Lifelines in the system take part during the execution of a system.
- In a sequence diagram, a lifeline is represented by a vertical bar.
- A message flow between two or more objects is represented using a vertical dotted line which extends across the bottom of the page.
- Sequence diagrams are built around an X-Y axis.
- Objects are aligned at the top of the diagram, parallel to the X axis.
- Messages travel parallel to the X axis.
- Time passes from top to bottom along the Y axis.
- Sequence diagrams most commonly show relative timings, not absolute timings.
- Links between objects are implied by the existence of a message.
- Example of a sequence diagram:

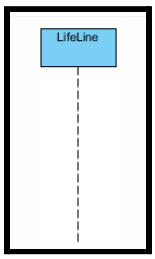
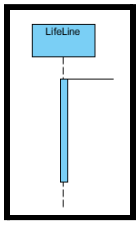
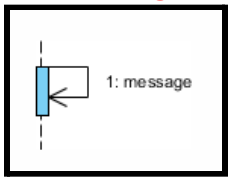


- Example of a sequence diagram:



- **Benefits of a sequence diagram:**
- Sequence diagrams are used to explore any real application of a system.
- Sequence diagrams are used to represent the message flow from one object to another.
- Sequence diagrams are easy to maintain and generate.
- Sequence diagrams can be easily updated according to the changes within a system.
- Sequence diagrams allow both reverse and forward engineering.
- **Drawbacks of a sequence diagram:**
- Sequence diagrams can become complex when too many lifelines are involved in the system.
- If the order of message sequence is changed, then incorrect results are produced.
- Each sequence needs to be represented using different message notation, which can be a little complex.
- The type of message decides the type of sequence inside the diagram.
- **When to use sequence diagrams:**
    1. Comparing Design Options:
        - Shows how objects collaborate to carry out a task.
        - Graphical form shows alternative behaviours.
    2. Assessing Bottlenecks
    3. Explaining Design Patterns:
        - Enhances structural models.
        - Good for documenting behaviour of design features.
    4. Elaborating Use Cases:
        - Shows how the user expects to interact with the system.
        - Shows how the user interface operates.
- **Modelling Control Flow By Time:**
- Determine what scenarios need to be modeled.
- Identify the objects that play a role in the scenario.
- Lay the objects out in a sequence diagram left to right, with the most important objects on the left.
  Most important in this context means objects that are the principle initiators of events.
- Draw in the message arrows, top to bottom.
  Adorn the message as needed with detailed timing information.
- **Style Guide for Sequence Diagrams:**
    1. Spatial Layout:
        - Strive for left-to-right ordering of messages.
        - Put proactive actors on the left.
        - Put reactive actors on the right.
    2. Readability:
        - Keep diagrams simple.
        - Don’t show obvious return values.
        - Don’t show object destruction.
    3. Usage:
        - Focus on critical interactions only.
    4. Consistency:
        - Class names must be consistent with class diagram.
        - Message routes must be consistent with navigable class associations.

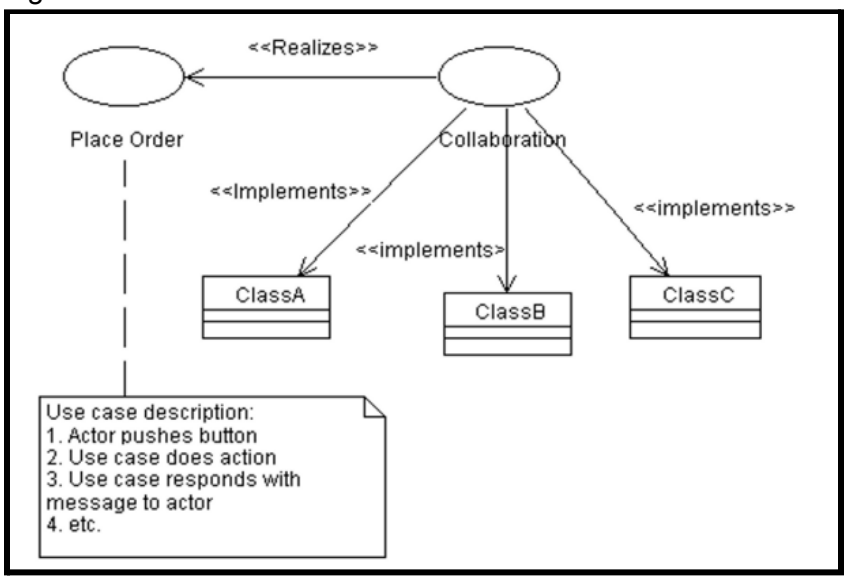
- **Summary:**

Item & Example	Description	Depiction
Lifeline 	A lifeline represents an individual participant in the interaction. Lifelines represent the passage of time as it extends downward. The dashed vertical line shows the sequential events that occur to an object during the charted process.	A labeled rectangle shape with a dotted line extending from its bottom.
Activation box 	Represents the time needed for an object to complete a task. The longer the task will take, the longer the activation box becomes.	A thin rectangle on a lifeline. The top and the bottom of the rectangle are aligned with the initiation and the completion time respectively.
Message 	A message defines a particular communication between lifelines.	A solid line with a solid arrowhead.
Asynchronous Message 	Asynchronous messages don't require a response before the sender continues. Only the call should be included in the diagram.	A solid line with a lined arrowhead.
Reply Message 	A return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.	A dashed line with a lined arrowhead.
Self Message 	A self message is a kind of message that represents the invocation of a message of the same lifeline.	A solid line with a lined arrowhead pointing to the same lifeline that it originated from.

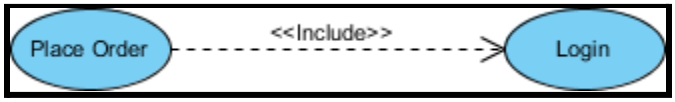
Use Case Diagrams:

- **Introduction:**
- A **use case diagram** is the primary form of system/software requirements for a new software program.
- Use cases specify the expected behavior (what), and not the exact method of making it happen (how).
- A key concept of use case modeling is that it helps us design a system from the end user's perspective. It is an effective technique for communicating a system's behavior in the user's terms.
- Use case diagrams are used to gather the requirements of a system including internal and external influences.
- A use case:
    - Specifies the behavior of a system or some subset of a system.
    - Is a set of scenarios tied together by a common user goal.
    - Does not indicate how the specified behavior is implemented, only what the behavior is.
    - Performs a service for some user of the system, called an **actor**.
- A use case represents a functional requirement of the system. A requirement:
    - Is a design feature, property, or behavior of a system.
    - States what needs to be done, but not how it is to be done.
    - Is a contract between the customer and the developer.
    - Can be expressed in various forms, including use cases.
- In brief, the purposes of use case diagrams are as follows:
    - Used to gather the requirements of a system.
    - Used to get an outside view of a system.
    - Identify the external and internal factors influencing the system.
    - Show the interaction among the requirements of the actors.
- An actor:
    - Is a role that the user plays with respect to the system. The user does not have to be human.
    - Is associated with one or more use cases.
    - Is most typically represented as a stick figure of a person labeled with its role name. Note that the role names should be nouns.
    - May exist in a generalization relationship with other actors in the same way as classes may maintain a generalization relationship with other classes.
- **Note:** Use cases diagrams do not show the order in which the steps are performed to achieve the goals of each use case. It only shows the relationship between actors, systems and use cases.
- Use cases are a technique for capturing the functional requirements of a system. Use cases work by describing the typical interactions between the users of a system and the system itself, providing a narrative of how the system is used.
- Use case development process:
    1. Develop multiple scenarios.
    2. Distill the scenarios into one or more use cases where each use case represents a functional requirement.
    3. Establish associations between the use cases and actors.
- A use case is graphically represented as an oval with the name of its functionality written inside. The functionality is always expressed as a verb or a verb phrase.
- A use case may exist in relationships with other use cases much in the same way as classes maintain relationships with other classes.

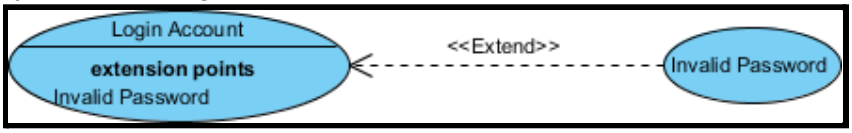
- As stated earlier, a use case by itself does not describe the flow of events needed to carry out the use case. The flow of events can be described using informal text, pseudocode, or activity diagrams.
  I.e. You can attach a note to a use case to show the flow of the event. Be sure to address exception handling when describing the flow of events.
  E.g.



- **Relationships Between Use Cases:**
- A use case may have a relationship with other use cases.
- Generalization between use cases is used to extend the behavior of a parent use case.
- An **<<include>> relationship** between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base.
  Note: Sometimes **<<uses>>** is used instead of **<<include>>**.
  When a use case is depicted as using the functionality of another use case, the relationship between the use cases is named as an **<<include>>** or **<<uses>>** relationship.

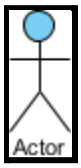


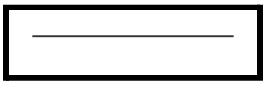
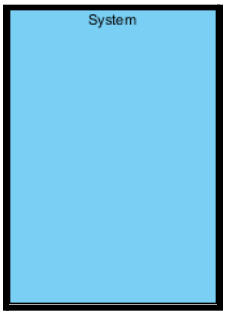
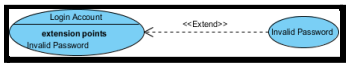
- An **<<extend>> relationship** between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case.



- Extended behavior is optional behavior, while included behavior is required behavior.
  I.e. Extended means “may use” while include/uses means “will use”.
- Extend occurs when one use case adds a behaviour to a base use case while include occurs when one use case invokes another.
- **Actor Classes:**
- Identify classes of actors.
- Actors inherit use cases from the class.

- **Describing Use Cases:**
- For each use case, a flow of events document, written from the actor's point of view, describes what the system must provide to the actor when the use case is executed.
- Typical contents:
  - How the use case starts and ends.
  - Normal flow of events.
  - Alternate flow of events.
  - Exceptional flow of events.
- Documentation style:
  - Activity Diagrams - Good for business process.
  - Collaboration Diagrams - Good for high level design.
  - Sequence Diagrams - Good for detailed design.
- **Finding Use Cases:**
- Noun phrases may be domain classes.
- Verb phrases may be operations and associations.
- Possessive phrases may indicate attributes.
- **For each actor, ask the following questions:**
  1. What functions does the actor require from the system?
  2. What does the actor need to do?
  3. Does the actor need to read, create, destroy, modify or store information in the system?
  4. Does the actor have to be notified about events in the system?
  5. Does the actor need to notify the system about something?
  6. What do these events require in terms of system functionality?
  7. Could the actor's daily work be simplified or made more efficient through new functions provided by the system?
- **Summary:**

Item & Example	Description	Depiction
Actors 	Is a role that the user plays with respect to the system. The user does not have to be human.	A stick figure.
Use Case 	Represents a functional requirement of the system. It specifies the behavior of a system or some subset of a system. It is a set of scenarios tied together by a common user goal. It does not indicate how the specified behavior is implemented, only what the behavior is.	An oval

Association 	Shows which actors use which use cases.	A line connecting an actor to a use case.
System Boundary Box 	The system boundary is potentially the entire system as defined in the requirements document. It is a box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system. For large and complex systems, each module may be the system boundary.	A blue box containing all the relevant use cases.
<<include>>/<<uses>> relationship 	An <<include>>/<<uses>> relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base.	A dotted line with a lined arrowhead originating from a use case pointing to the used use case. It has <<includes>> or <<uses>> written on the arrow.
<<extend>> relationship 	An <<extend>> relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case.	A dotted line with a lined arrowhead originating from a use case pointing to the used use case. It has <<extends>> written on the arrow.