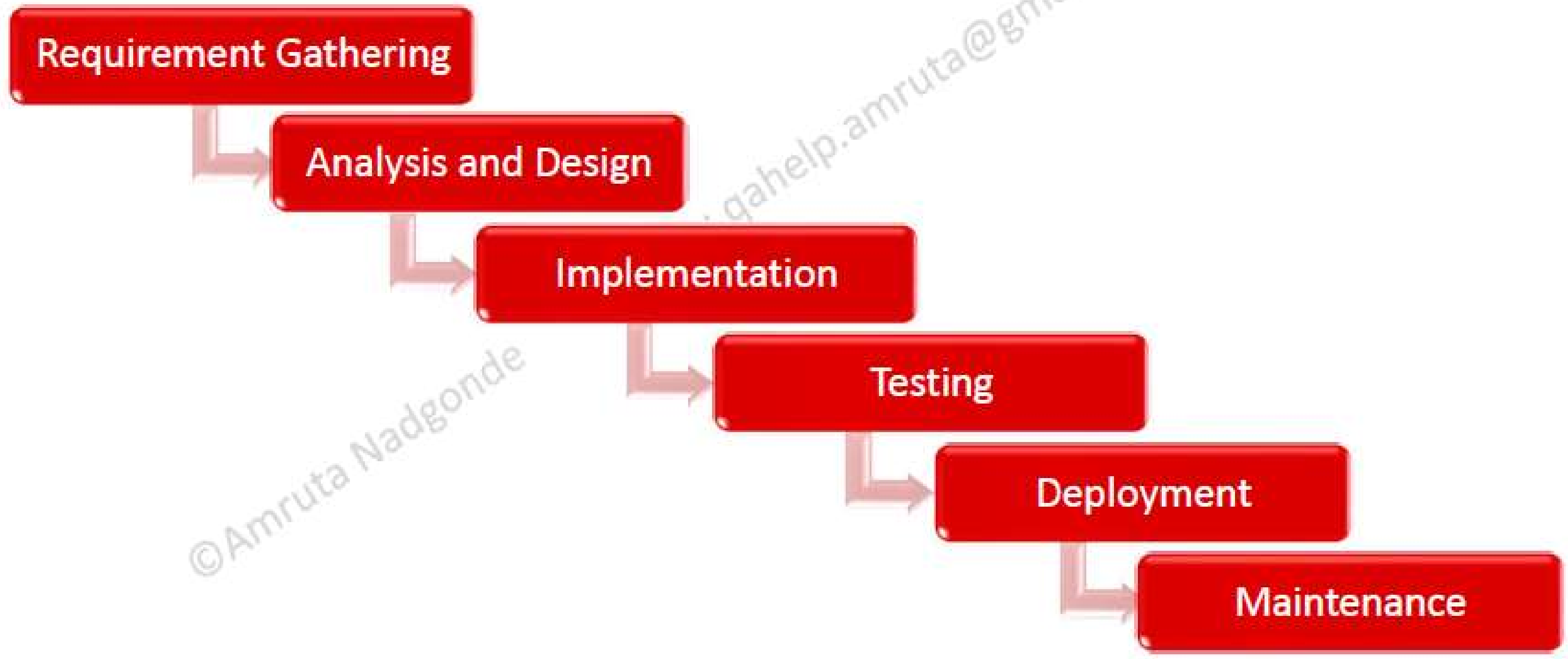
Software Engineering

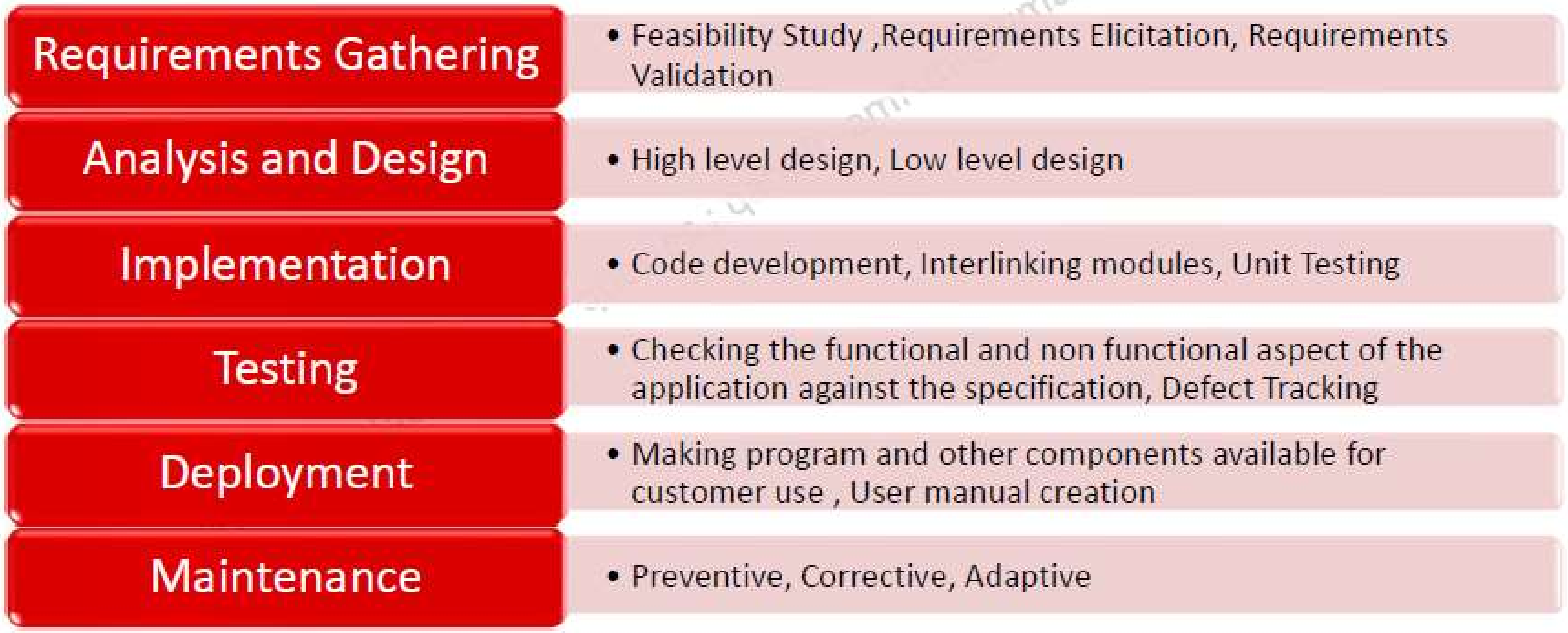
Software Engineering

- An engineering-based approach to software development
- Deals with the design, development, testing, and maintenance of software applications
- Applies engineering principles and knowledge of programming languages to build software solutions for end users.
- The main goal of Software Engineering is to develop software applications for improving quality, budget, and time efficiency
- Ensures that the software to be built is consistent, correct, also in budget, on time, and within the required requirements

Software Development Life Cycle



Software Development Life Cycle



Requirements Gathering	• Feasibility Study ,Requirements Elicitation, Requirements Validation
Analysis and Design	• High level design, Low level design
Implementation	• Code development, Interlinking modules, Unit Testing
Testing	• Checking the functional and non functional aspect of the application against the specification, Defect Tracking
Deployment	• Making program and other components available for customer use , User manual creation
Maintenance	• Preventive, Corrective, Adaptive

Software Process Model

- Software processes are the activities for designing, implementing, and testing a software system
- A software process model is an abstract representation of the development process.
- Defines the stages and order in which different activities /phases are carried out
- A software model will define :
  - The tasks to be performed
  - The input and output of each task
  - The pre and post-conditions for each task
  - The flow and sequence of each task
- The goal is to provide control and coordination over the tasks performed to achieve end product and objectives efficiently

Factors in choosing Software Process

- Nature of Project Requirements
- Project Size
- Project complexity
- Customer involvement
- Familiarity with the technology
- Project recourses

Software Process Models

- Waterfall
- V model
- Spiral
- Incremental
- Agile

When to Choose What

Model	Cost	Development Speed	Delivery Frequency	Project Complexity
Waterfall	High	Low	Once	Medium
Spiral	Moderate	Medium	Monthly	Medium
V Model	High	Low	Once	Medium-High
Incremental	Low-Moderate	High	Weekly-Monthly	Medium-High
Scrum	Low	High	Weekly-Monthly	Medium-High
Xtrem Programming	Low	High	Weekly-Monthly	Medium

Importance of Software Engineering

- Reduce Complexity
- Handling Big Projects
- Minimizing software cost
- Decrease in Time
- Developing reliable software

Requirements Engineering

Requirements

- Requirements are the things your systems must do to work correctly
- Initial requirements come from the customer or gathered from the end user
- Programmers need to check if customer hasn’t thought about something
- A use case describes what a system does to accomplish a particular customer goal
- Has a single goal but can have multiple paths to achieve the goal
- A good use case should capture all real world problems your software may encounter

Requirements

Business	User	Software
• Outlines measurable goals for business • Defines ‘Why’ • Matches project goals to stakeholder’s goals • BRD – with req, changes /updates	• Reflects specific user needs/ expectations • Defines ‘Who’ • Highlights how user will interact • Create URD	• Identify features, functions or non functional req • Defines ‘How’ • Describes s/w as functional modules • SRS, Use cases and FRS

Characteristics of Good Requirements

- Clear and understandable : should be written plain language, free of domain-specific terms and jargon. Clear and concise statements make requirements documents easy to evaluate for subsequent characteristics
- Correct and complete : The requirement documents should accurately detail all requirements
- Consistent, not redundant : Consistent requirements have no conflicts, such as differences in terminology. Requirements should not be duplicate
- Unambiguous. No software requirement can leave room for interpretation. Collaboration and peer reviews help ensure unambiguous requirements documentation.
- Traceable : Requirements must be traced. Links must be establish between requirement, finished code and tests.

Requirements Engineering

➢ Process of identifying, eliciting, analyzing, specifying, validating, and managing requirements

Requirement Elicitation	• Gain knowledge about project domain and requirements • Interviews, brainstorming, task analysis, prototyping
Requirement Analysis	• Identify high level goals and objectives of the system • Identify constraints and limitations of the system
Requirement Specification	• Process of documenting the requirements in a clear, consistent, and unambiguous manner • Models used – ER , DFD , FDD
Requirement Validation	• Process of checking requirements are complete, consistent, and accurate. • Checks if requirements are testable
Requirement Maintenance	• The process of managing the requirements throughout the software development life cycle • Takes care of changing requirements

Requirements Analysis and Designing

- Major tasks include:
  - Understanding business requirements
  - Decomposition and analysis of requirements
  - Categorization of requirements
  - Modelling of requirements
- The objective is to understand the requirements from the key stakeholders and convert them into business requirements so that development team understands them

Requirements Analysis Process

- Identifying key stakeholders and end users

  Stakeholders make the most important decisions about the projects, where as end user’s inputs and feedbacks will majorly impact the success of the project

- Capturing the requirements

  Some common techniques include one-on-one interviews, group interviews, Use cases to get projects view from end users perspective and building prototypes

- Categorizing the requirements

  Requirements can be categorized as functional, technical, transitional and operational

- Interpreting and recording the requirements

  Defining Requirements Precisely, Prioritizing the Requirements, Carry Out an Impact Analysis and Resolve conflicts

Requirements Modeling Techniques

- Business Process Modeling Notation (BPMN)

  A graphical technique of representing your business process using various graphs, such as State diagrams, Use case diagrams, Class diagrams etc

- Gantt Charts

  Used for representing the project tasks along with their timelines. Help in keeping track of timelines

- Data Flow Diagrams

  Used for highlighting how data processing is done by a system with regard to inputs and outputs

- Gap Analysis

  Helps business analysts in determining the present state and the target state for a product.

Documenting Your Requirements

- Good requirements ensure your system works like your customers expect.
- Make sure your requirements cover all the steps in the use cases for your system.
- Use your use cases to find out about things your customers forgot to tell you.
- Your use cases will reveal any incomplete or missing requirements that you might have to add to your system.
- With a good use case textual analysis is a quick and easy way to figure out the classes in your system
- Your requirements will always change (and grow) over time.

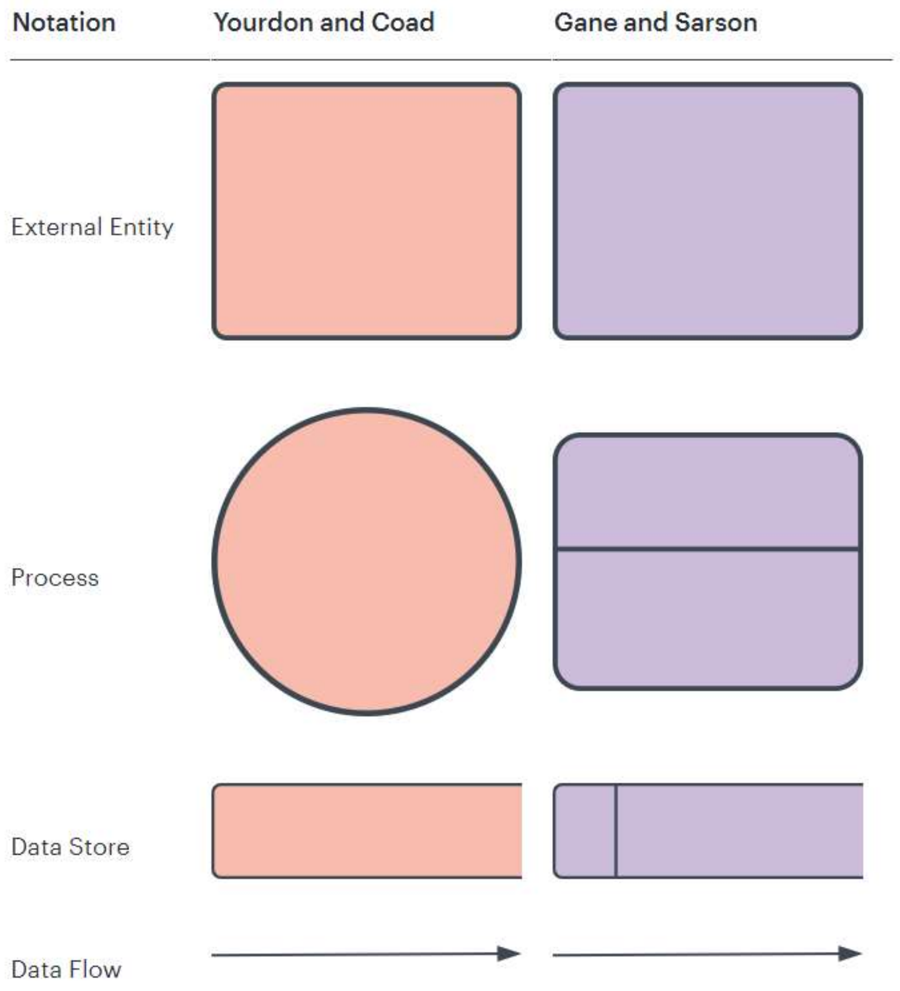
Design Models

Data Flow Diagram

- A Data flow diagram (DFD) maps out the flow of information for any process or system
- It uses different symbols to represent data inputs, outputs, storage points and the routes between each destination
- They can be used to analyze an existing system or model a new one
- Help in visually representing the systems which are hard to describe through words and work for both technical and nontechnical audiences

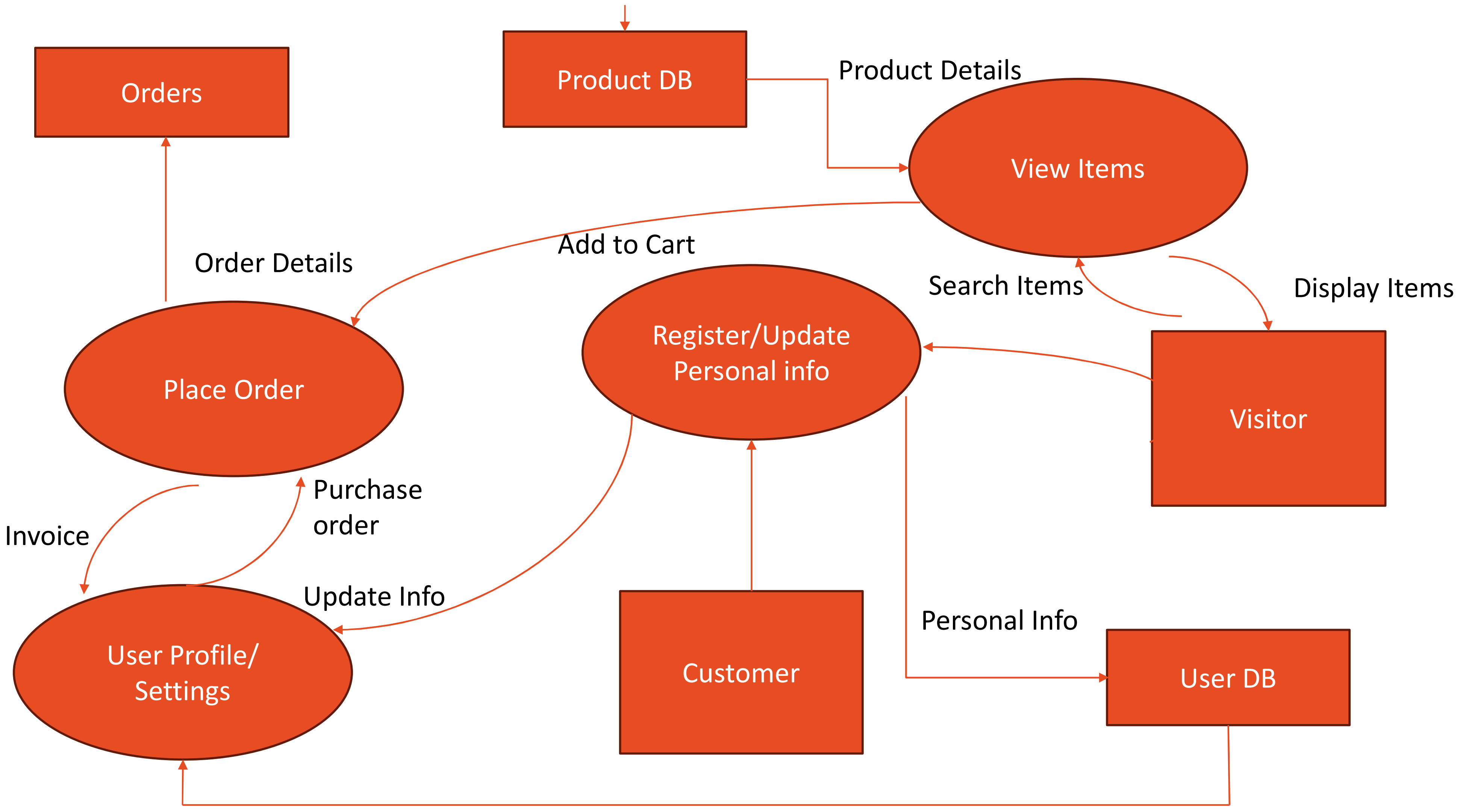
Components of DFD

External Entity	• An outside system that sends or receives data • Sources/Destinations of info entering or leaving the system
Process	• Any process that changes the data, producing an output • Perform computations/sort data/direct the data flow based on business rules
Data Store	• Files or repositories that hold information for later use, such as a database table or a membership form
Data Flow	• The route that data takes between the external entities, processes and data stores • Represents the interface between the other components


Notation
Yourdon and Coad
Gane and Sarson
External Entity
Process
Data Store
Data Flow

DFD Rules and Levels

- Each process should have at least one input and an output.
- Each data store should have at least one data flow in and one data flow out.
- Data stored in a system must go through a process.
- All processes in a DFD go to another process or a data store
- Level 0 [Context Diagram] - a basic overview of the whole system or process being analyzed or modeled
- Level 1 - provides a more detailed breakout of pieces of the Context Level Diagram
- Level 2 – provides specific details of the process


Orders
Product DB
Product Details
View Items
Add to Cart
Order Details
Search Items
Display Items
Register/Update Personal info
Place Order
Visitor
Purchase order
Invoice
Update Info
User Profile/ Settings
Customer
Personal Info
User DB

UML

- A standard visual modeling language intended to be used for:
  - Modeling business and similar processes
  - Analysis, design, and implementation of software-based systems
- Used to describe, specify, design, and document existing or new business processes, structure and behavior of artifacts of software systems
- Contains graphical elements (symbols) - UML nodes connected with edges (paths or flows)
- Use Case diagrams - describe system functionality from the point of view of the user.
- Class Diagrams - describe the structure of the system in terms of objects, attributes, associations, and operations

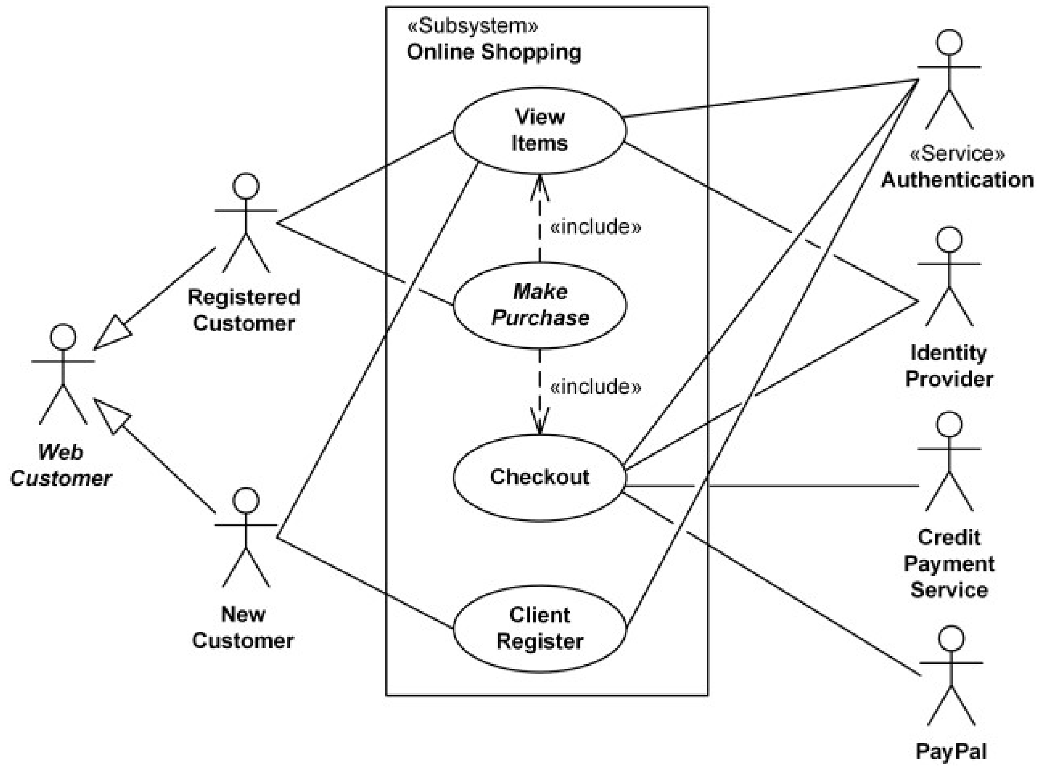
Use Case Diagram

- Demonstrates the different ways that a user might interact with a system
- Summarize the details of your system's users (also known as actors) and their interactions with the system
- Use case diagram depicts a high-level overview of the relationship between use cases, actors, and systems
- Common components include :
- Actors: The users that interact with a system (person/organization/external system)
- System: Anything your are developing (system/app/scenario)
- Goals : The end result of most use cases.

Symbols and Notations

- Use cases: Horizontally shaped ovals that represent the different uses that a user might have.
- Actors: Stick figures that represent the people actually employing the use cases.
- Associations: A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.
- Include Relationship : relationship between base use case and include use case. User doesn't interact with include use case. Include use case executes every time base use case executes
- Extend Relationship : relationship between base use case and extend use case. Extend use case may not always execute if base use case is executed
- System boundary boxes: A box that sets a system scope to use cases. Anything outside the vox doesn't happen in the use case

UML


«Subsystem»
Online Shopping
View Items
«include»
Make Purchase
«include»
Checkout
Client Register
Registered Customer
Web Customer
New Customer
«Service» Authentication
Identity Provider
Credit Payment Service
PayPal

Class Diagram

- Class diagrams are the building blocks of UML
- Popular among software engineers to document software architecture
- The various components of a class diagram represent the actual classes to be programmed, the main objects, or the interactions between classes and objects.
- The class shape itself consists of a rectangle with three rows – top row contains name of the class, middle shows attributes and the bottom one is for the methods/behaviors
- Classes and subclasses are grouped together to show the static relationship between each object

Basic Components

Member access modifiers

- Public (+)
- Private (-)
- Protected (#)
- Package (~)
- Derived (/)
- Static (underlined)

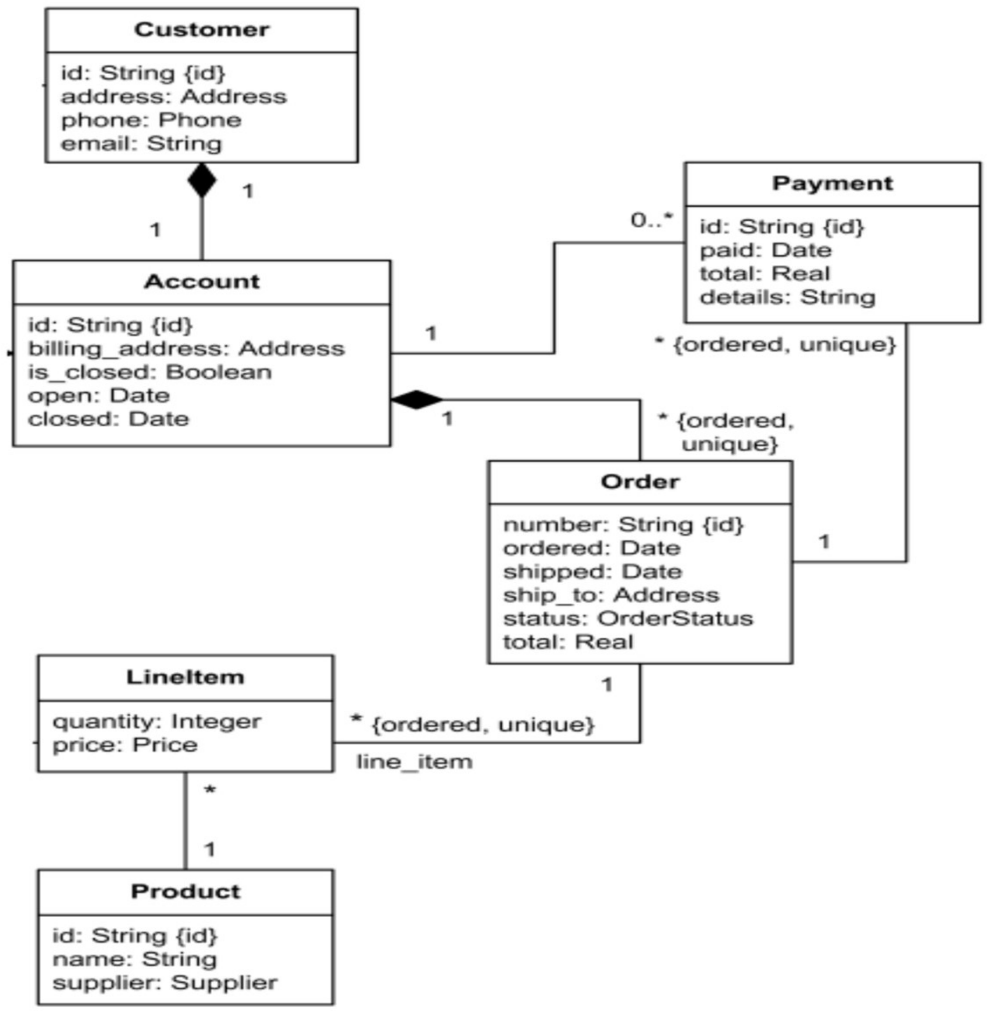
Interactions

Refers to the various relationships and links that can exist in class and object diagrams

- Inheritance : also known as generalization symbolized with a straight connected line with a closed arrowhead pointing towards the superclass.
- Bidirectional association: The default relationship between two classes. Both classes are aware of each other and their relationship with the other. represented by a straight line between two classes.
- Aggregation : Shows the association between two objects, where contained object can exist/used by other classes than the associated one. symbolized with a straight connected line with a diamond pointing towards the containing class.
- Composition : Shows the association between two objects, where if containing object is deleted the contained object doesn’t have any meaning. symbolized with a straight connected line with a solid diamond pointing towards the containing class.

Benefits

- Illustrate data models for information systems, no matter how simple or complex.
- Better understand the general overview of the schematics of an application.
- Visually express any specific needs of a system and disseminate that information throughout the business.
- Create detailed charts that highlight any specific code needed to be programmed and implemented to the described structure.
- Provide an implementation-independent description of types used in a system that are later passed between its components.


Customer
id: String {id}
address: Address
phone: Phone
email: String
1
1
Account
id: String {id}
billing_address: Address
is_closed: Boolean
open: Date
closed: Date
1
0..*
Payment
id: String {id}
paid: Date
total: Real
details: String
* {ordered, unique}
1
* {ordered, unique}
Order
number: String {id}
ordered: Date
shipped: Date
ship_to: Address
status: OrderStatus
total: Real
1
1
LineItem
quantity: Integer
price: Price
* {ordered, unique}
line_item
*
1
Product
id: String {id}
name: String
supplier: Supplier

Object Oriented Analysis & Design

How to write a good software always???

Your Answer?

- Great software does what the customer wants it to , so even if the customer finds out a new way of using it , it doesn't break or gives unexpected result
- Great software is code that is object oriented. So there's not a bunch of duplicate code, and each object pretty much controls its own behavior. It's also easy to extend because your design is really solid and flexible
- Great software is when you use tried-n-tested design patterns and principles. Your objects are loosely coupled, and your code is open for extension but closed for modification. Your code is also reusable so you don't have to rework on everything when only part of the system needs to be used

Good Design

Make sure software does what the customer wants it to do

Apply basic OO Principles to add flexibility

Make your design maintainable and reuseable

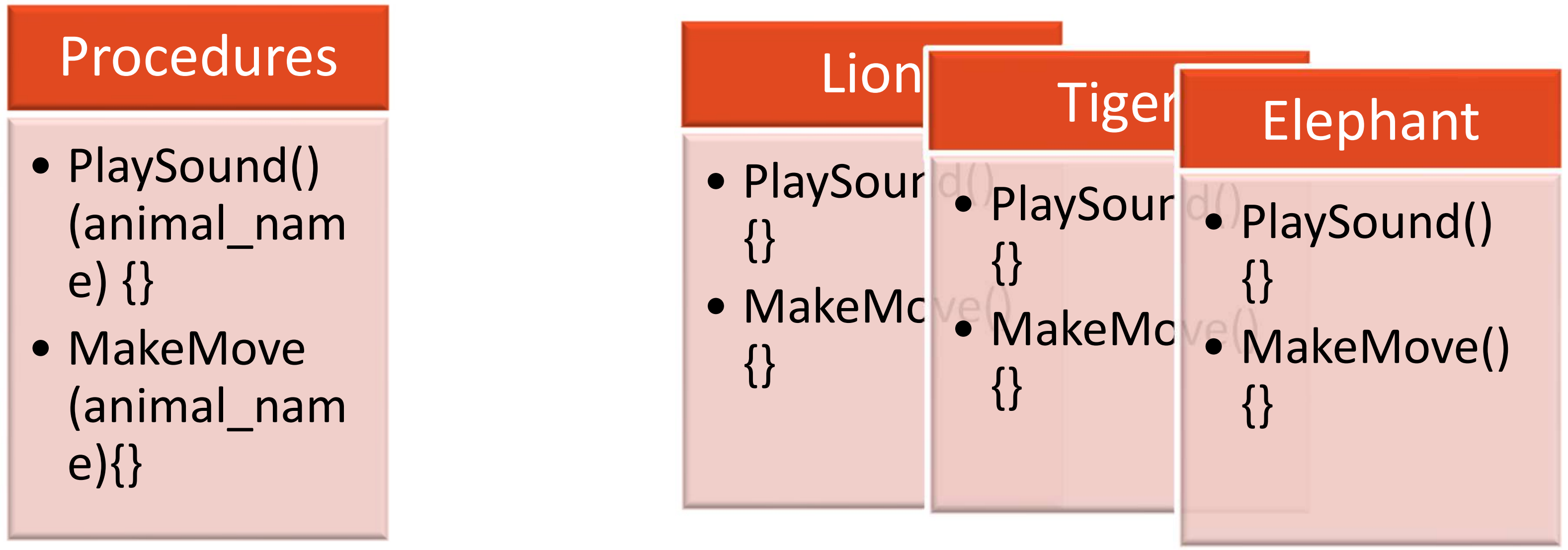
Sharpen Your Pencil...

A page will display pictures of land animals [Lion, Tiger, Elephant] When you click on the picture the sound of specific animal will be played and the animal will make a linear move on the screen

What are the things in the program have to do?

What are the things in this program?

Procedural Vs Object Oriented



Procedural Vs Object Oriented

- Requirements [specs] are ever changing

  - Add a new category of animals -> Sea animals
  - Sea animals will come splashing from the water

- What do you like about OOP?

Object Oriented Programming

- Helps in designing the applications in more natural way. Whenever specs change the objects can be modified easily
- New code can be added without disturbing the previously tested code
- Data and methods that operate on that data are together in one class
- Code can be reused in other applications

Object-Oriented Programming

An approach to problem-solving where all computations are carried out using objects

What is an Object?

- Object is the basic unit of object-oriented programming
- Component of program that performs certain actions and interact with other elements
- Has well defined structure (properties) and behavior (methods)

What is a Class?

- A class is a blueprint of an object.

Examples - C#, Java, Perl and Python.

What is an Object

A real world entity with well-defined structure and behavior.

Definition:

- An object represents an individual, identifiable item, unit or entity, either real or abstract, with well-defined role in the problem domain.

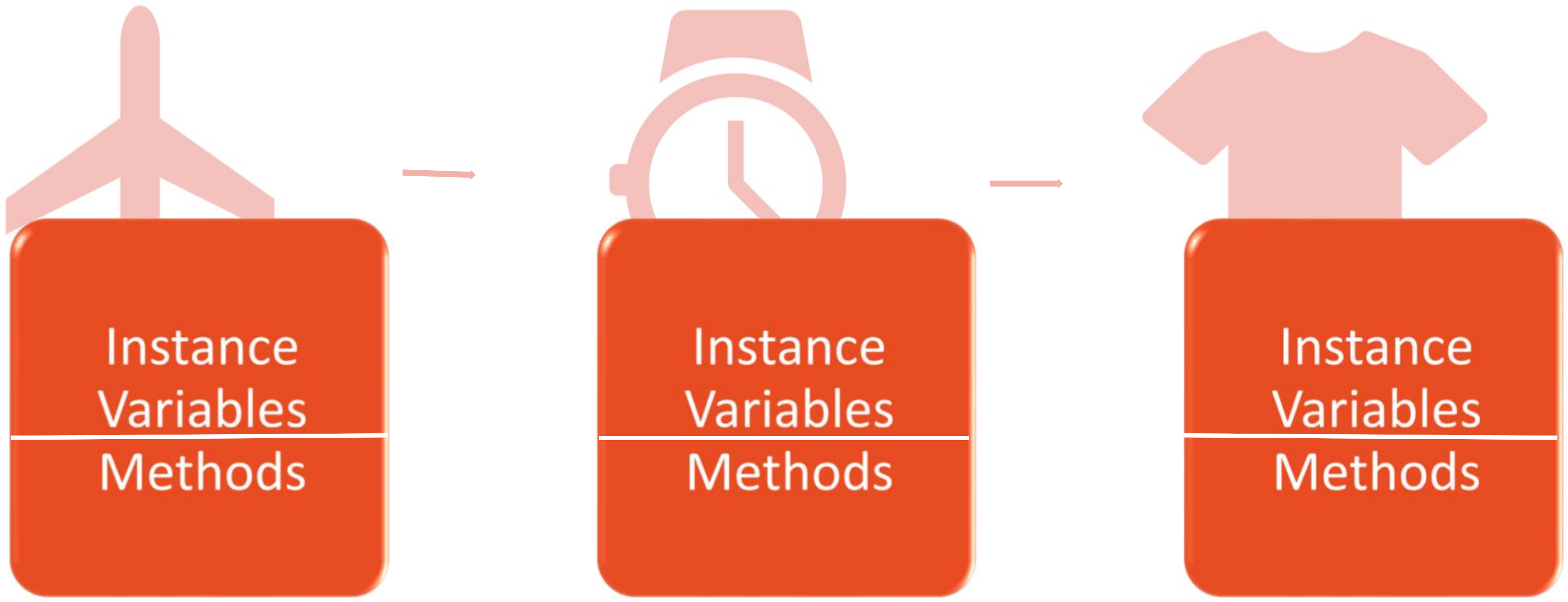
Characteristics:

- State
- Behavior
- Identity
- Responsibility

Examples

- Person, car, pen, sound, instrument, back account, student, signal, transaction, medical investigation etc.

Objects


Instance Variables
Methods
Instance Variables
Methods
Instance Variables
Methods

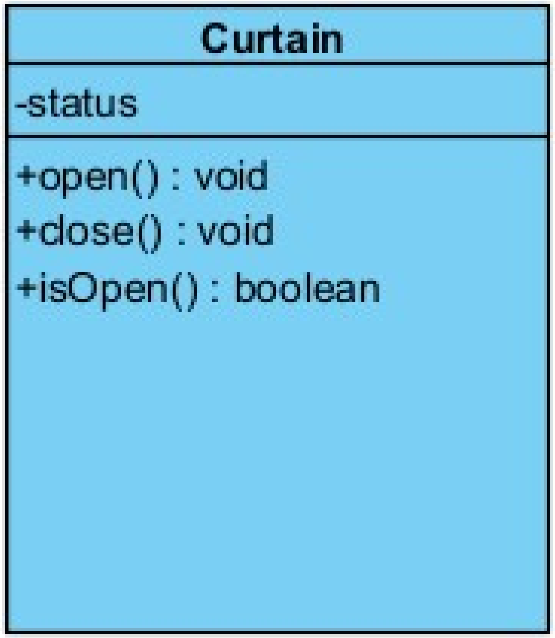
Guess the Objects

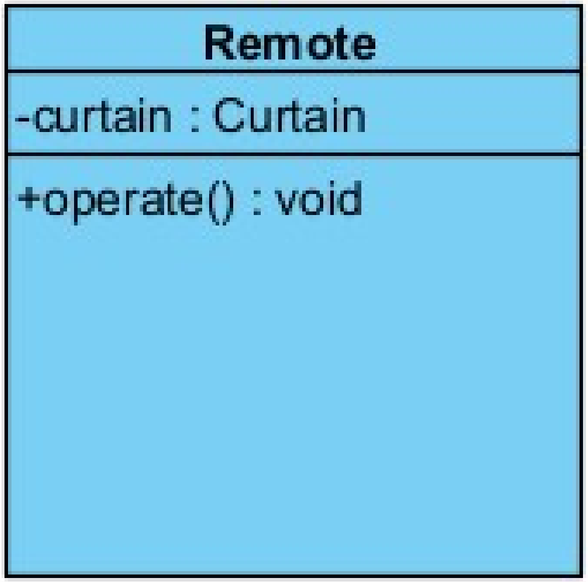
- ❑ Develop an automized curtain controller application
- ❑ Using a remote the user should be able open or shut the curtains

- ❑ Owner wants to open the window curtains
- ❑ Owner clicks on the ON/OFF button of the remote
- ❑ Curtains are opened
- ❑ Owner wants to close the curtains
- ❑ Owner presses the ON/OFF button of the remote again
- ❑ Curtains are closed

Developing a Class

- Figure out what a class is supposed to do
- List the instance variables and methods
- Write pseudo code for the methods
- Implement the class by replacing the pseudo code by real code
- Write the test code to check if methods are actually doing what they are supposed to do
- Debug and reimplement as needed





```
public static void main(String[] args) {
Curtain curtain = new Curtain(true);
Remote remote = new Remote(curtain);
remote.operate();
remote.operate();
}
```

Abstraction

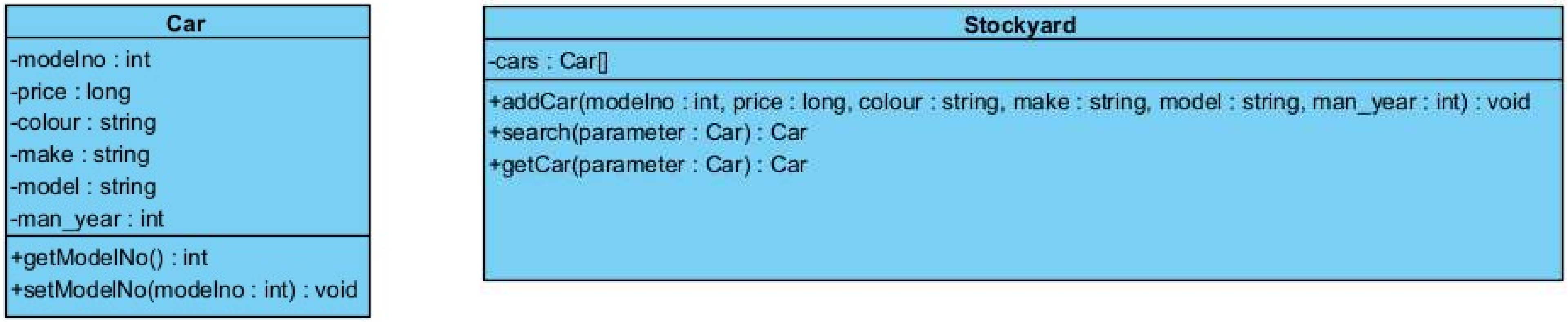
- Process of identifying key aspects of an entity and ignoring the rest.
- Process of extracting essential/relevant details of an entity from the real world.
- Refers to both attributes (data) as well as behavioral (functions) abstraction.

- Consider ‘person’ as on object; abstraction of person would be different in different programming paradigm.

Social Survey	Health Care	Employment
Name	Name	Name
Age	Age	Age
Birth date	Birth date	Birth date
Marital Status		
income group		
Address	Address	Address
Occupation	Occupation	Occupation
	Blood Group	
	Weight	
	Prev records	
		Qualification
		Department
		Experience

Encapsulation

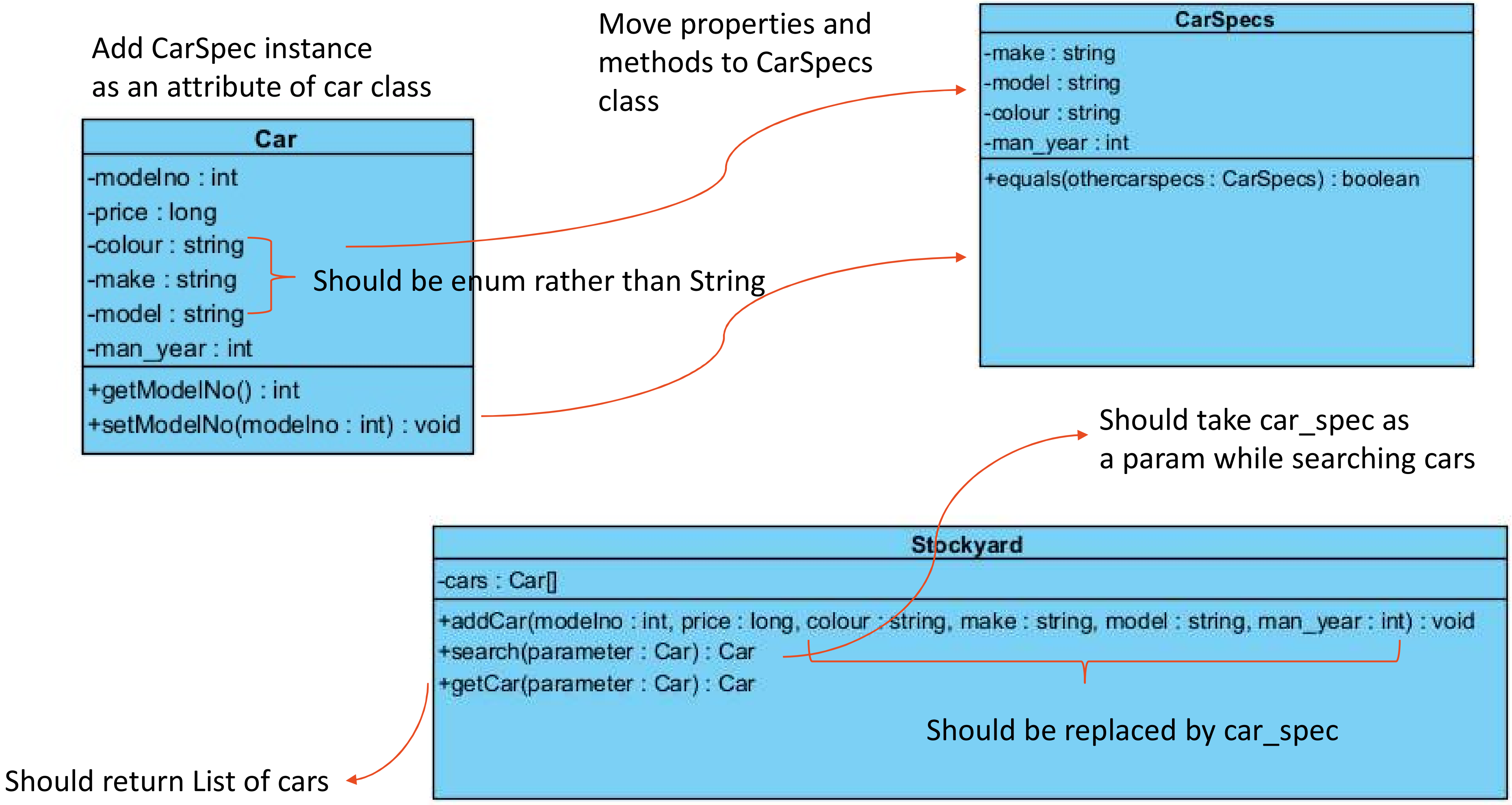
- The mechanism used to hide the data, internal structure and implementation details of an object.
- Internal data can be accessed only through a public interface – Methods
- Ensures security of Data
- Separates interface of abstraction from its implementation
- Even if implementation is changed user need not to worry as long as interface remains unchanged.


Car
-modelno : int
-price : long
-colour : string
-make : string
-model : string
-man_year : int
+getModelNo() : int
+setModelNo(modelno : int) : void
Stockyard
-cars : Car[]
+addCar(modelno : int, price : long, colour : string, make : string, model : string, man_year : int) : void
+search(parameter : Car) : Car
+getCar(parameter : Car) : Car

- ❑ The Search method will check if a cars with given colour, make, model and manufacturing year is available in the Stockyard and return the matching car
- ❑ While searching model_no and price should be ignored as they are unique for each car

Encapsulation

- As per object oriented principles there should not be duplicate code and each object should pretty much control its own behavior, and do only its own job.
- Ideally, each object should represent one concept
- Good design patterns require the objects to be loosely coupled
- Such objects help in creating extensible and reusable code, at the same time keeping an object closed for modification [hiding internal details]
- Applying basic OO principles is a start to writing good software
- It helps in creating maintainable and reusable software

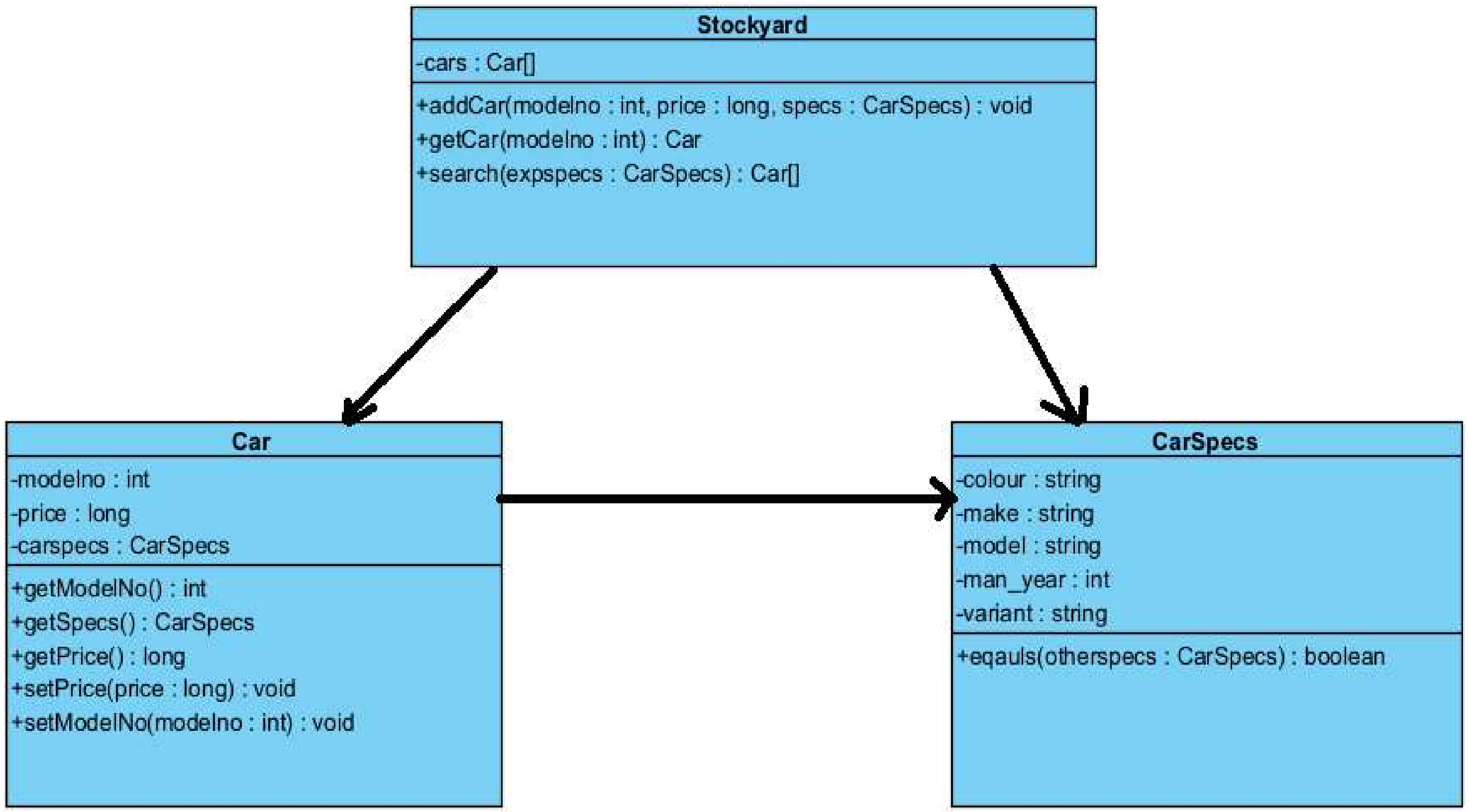

Add CarSpec instance as an attribute of car class
Move properties and methods to CarSpecs class
Car
-modelno : int
-price : long
-colour : string
-make : string
-model : string
-man_year : int
+getModelNo() : int
+setModelNo(modelno : int) : void
Should be enum rather than String
CarSpecs
-make : string
-model : string
-colour : string
-man_year : int
+equals(othercarspecs : CarSpecs) : boolean
Should take car_spec as a param while searching cars
Stockyard
-cars : Car[]
+addCar(modelno : int, price : long, colour : string, make : string, model : string, man_year : int) : void
+search(parameter : Car) : Car
+getCar(parameter : Car) : Car
Should be replaced by car_spec
Should return List of cars

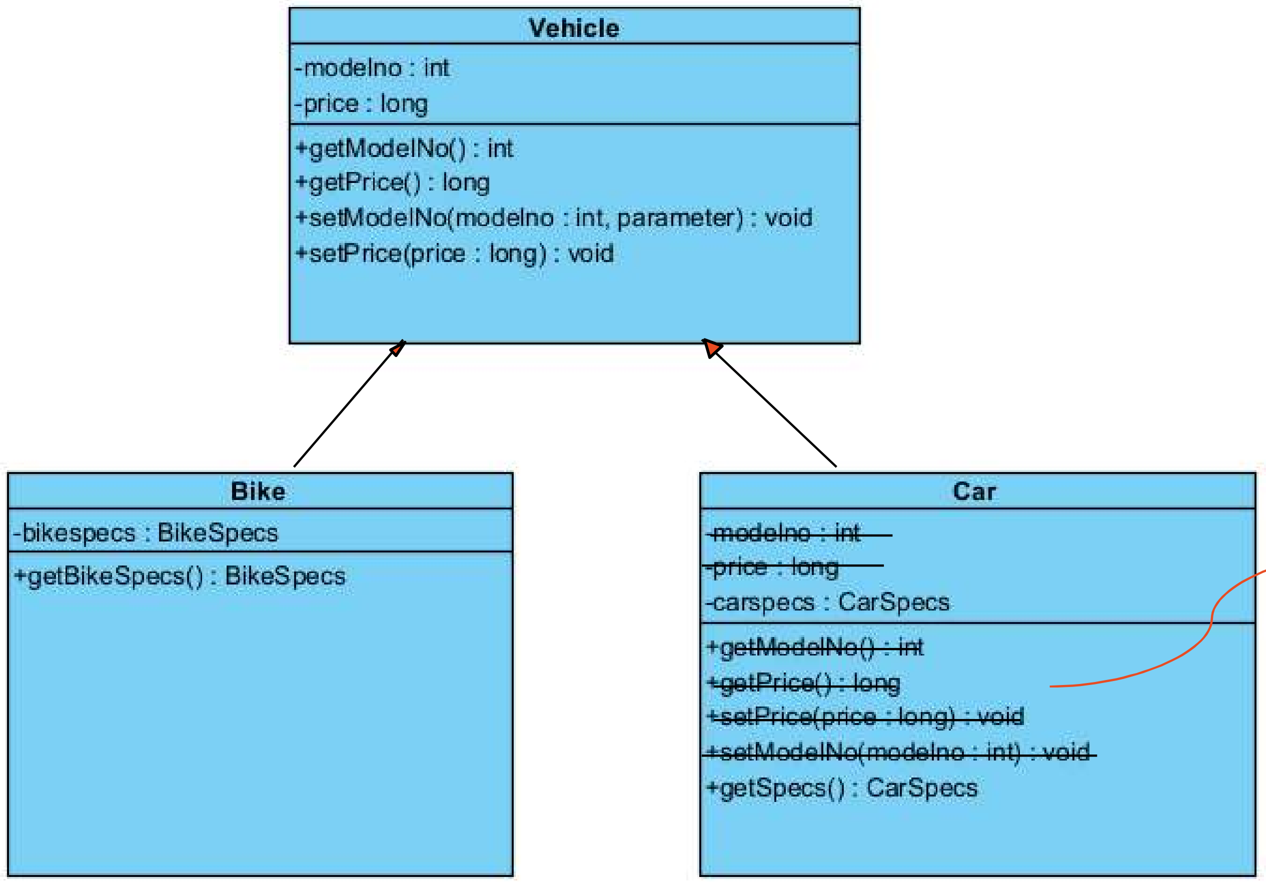
Encapsulation

- Encapsulation is used to protect information in one part of the application from other parts
- One form of encapsulation helps us in hiding the data
- Sometimes we need to separate out entire set of properties of an object [e.g. Car specs]
- By breaking an app [or class] in different parts we can modify one part without affecting the other
- Here we are encapsulating part of an Object which might vary and affect parts that do not vary
- Encapsulation hides the implementation of a class in such a way that it is easy to use and change

Association

- Association is when one object uses another object as a part of it
- Has-A type of relation
- Can be one-to-one, one-to-many, many-to-one or many-to-many
- Two types – aggregation and composition
- Aggregation – Type of association where the container and referenced object's lifecycle are not bound to each other [e.g. Trainer has a list of Subjects]
- Composition – Type of association where referenced object is depended on container object. The composed object can not exists when container is destroyed [e.g Car has CarSpecs]


Stockyard
-cars : Car[]
+addCar(modelno : int, price : long, specs : CarSpecs) : void
+getCar(modelno : int) : Car
+search(expspecs : CarSpecs) : Car[]
Car
-modelno : int
-price : long
-carspecs : CarSpecs
+getModelNo() : int
+getSpecs() : CarSpecs
+getPrice() : long
+setPrice(price : long) : void
+setModelNo(modelno : int) : void
CarSpecs
-colour : string
-make : string
-model : string
-man_year : int
-variant : string
+eqauls(otherspecs : CarSpecs) : boolean



These attributes and methods are moved to Vehicle class

Inheritance

- One of the powerful features of object oriented programming.
- It is an “is-a” kind of a relationship.
- To inherit means to receive properties of already existing class.
- All data members and methods are inherited except the private fields.
- Hierarchy of classes is created using inheritance; new class (derived) is derived from an existing(base) class.

Advantages:

- Reusability: Once a class written and tested, it can be used to create new classes.
- Extensibility : As new classes can be derived from existing class ;this provides the extensibility of adding and removing classes in a hierarchy as and when needed.

Inheritance

- Figure out the common, abstract characteristics that your objects have and place then in a common type called the superclass
- Objects in your application will extend this class
- A good level of abstraction is required to create the generic category – superclass
- The specific behavior should be implemented in the specific categories - subclass

Inheritance : Do’s and Don'ts

- Do use inheritance when one class is more specific type of a superclass
- Do use inheritance when you have a common behavior that should be shared among multiple classes of generic type
- Do not use inheritance just because you want to re-use code written in some class, it must not violet above two rules
- Do not use inheritance if Superclass and subclass do not pass the ‘Is-A’ test

What are the issues in the design

- The respective vehicles classes are almost empty [they do not have any distinct behavior]
- Every time a new category is added new Spec class will be needed
- The stockyard class is quite messy it needs search method for every type

Solution???

- Coding to an interface rather than to an implementation makes your code more extensible By
- Coding to an interface your code will work for all the sub classes , even ones that are not created yet

Polymorphism

- "Poly" means many and "morph" means form.
- Ability to make more than one form is called polymorphism.
- In programming terminology, one command may invoke different implementation for different related objects.
- Allows different objects to share the same external interface although the implementation may be different.
- Can be achieved in two ways – Compile time polymorphism and run time polymorphism.

Polymorphism

- Inheritance guarantees that subclasses will have all the methods defined by the superclass
- With the help of hierarchies we define common protocol for group of classes
- Using inheritance, superclass reference can refer to subclass object

```
Shape s = new Circle()
```

- This helps in designing code that is more flexible, extensible and cleaner
- One can define array of related object with superclass a reference type
- One can have polymorphic arguments and return types

What can be improved

- Discard all vehicle sub classes and add ‘vehicletype’ property in the Vehicle class
- VehicleType enum can be used to return the String value showing the vehicle type
- VehicleSpecs is used to represent common properties for vehicles and respective Spec classes are used to store additional properties
- Discard these classes and add a Map : properties in VehicleSpecs class to store all properties for all type of vehicles
- A method getProprty(String) can be added to the VehicleSpecs class to return value of a specific property
- When a vehicle specific property needs to be added just add it to the property Map.

To summarize....

- Requirements are ever changing, a good software is the one that is easy to change
- Always try to encapsulate what varies : make sure each class has only one reason to change, can be done by minimizing that number of things in that class that can cause change
- Whenever you find common behavior in two or more places, look to abstract that behavior into a class, and then reuse that behavior in the common classes
- Coding to an interface, rather than to an implementation, makes your software easier to extend.
- When you have a set of properties that vary across your objects, use a collection, like a Map, to store those properties dynamically
- A cohesive class does one thing really well and doesn't do or be something else
- Design is iterative... and you have to be willing to change your own designs

Modularity, Cohesion & Coupling

- A module simply means a software component created by dividing a large software
- Each module is designed to work independently
- When modules are combined , they work together as a single function
- Basic principle of modularity expects the components to be highly cohesive and loosely coupled
- Coupling - refers to the degree of interdependence between software modules. Loosely coupled modules will have less impact on each other
- Cohesion - refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose
- High cohesion means each class has a very specific set of closely related actions it performs.

Programming [Design] Principles

- Using proven OO design principles results in more maintainable, flexible, and extensible software.
- **Keep It Simple Stupid *[KISS]*** *:* Designs / systems should avoid complexity.
- Greatest level of user acceptance and interaction is guaranteed by simple interfaces
- Should be applied for interface design, product design, and software development
- **Don’t Repeat Yourself [DRY]** : Repeated code should be abstracted and placed in more general category – superclass
- This also means you are implementing one functionality/ requirement only once in your application
- Whether you’re writing requirements, developing use cases, or coding, you want to be sure that you don’t duplicate things in your system

SOLID Principles

- **Single Responsibility** : Every object in your system should have a single responsibility, and all the object's services should be focused on carrying out that single responsibility.
- Useful when something about that responsibility changes, you'll know exactly where to look to make those changes in your code
- If implemented correctly , each object will have only one reason to change
- A simple test A <<Class_Name>> <<does_something>> itself will help figure out of SRP is implemented correctly
- E.g. Car : start, stop, changeTires, wash, drive, checkOil
- Design classes by putting related functionalities together

SOLID Principles

- **Open-Closed** : Software entities (classes, modules, functions) should be **open for extension** but **closed for modification**
- Once code is written and tested for its correctness nobody should be able to modify it
- If any changes needed for such code, it should be easily extensible – subclassing your classing and overriding the method will do the job
- However classes and objects created through inheritance are tightly coupled because changing the parent or superclass in an inheritance relationship risks breaking your code
- Classes and objects created through composition are loosely coupled

SOLID Principles

- **Liskov Substitution Principle** : Any instance of a class should be able to be used in place of its parent
- Means if class B is a subtype of class A, then we should be able to replace object of A with B without breaking the behavior of the program
- A child class should extend the capabilities of a parent class and not narrow it down
- **Interface Segregation** : Clients should not be forced to depend on interfaces they don't use
- Interfaces should be designed in such a way that clients should not worry about implementing something they do not require
- Segment your interface to only add functionality that will make sense

SOLID Principles

- **Dependency inversion** : Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes rather than upon concrete functions and classes.
- When one component (class object) depends on another we don’t want to directly inject a component’s dependency into another. Instead we should use a level of abstraction between them
- A good rule to apply is that higher-level components should not depend on a lower level
- Lower-level components should depend on abstraction whenever possible. If we don’t follow this, the chance of creating tightly coupled modules is very high

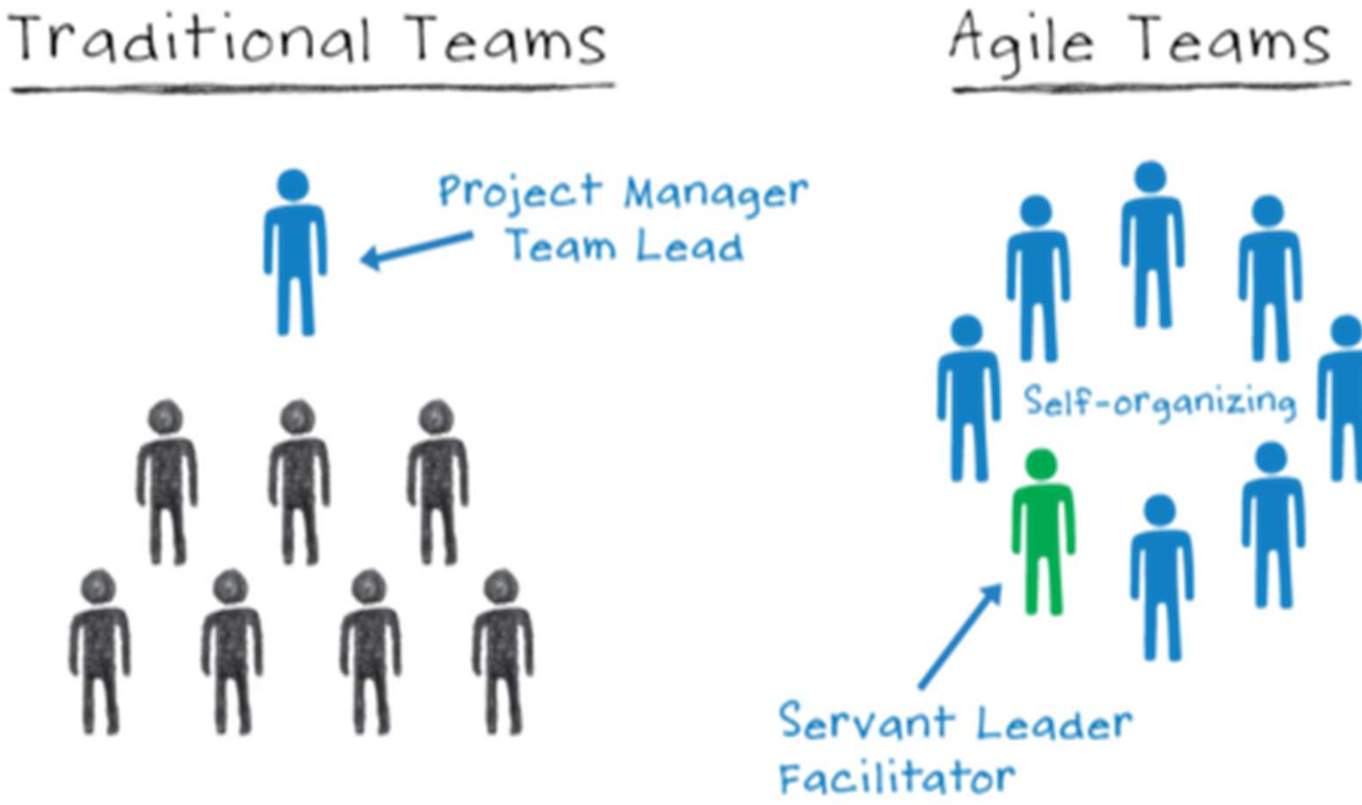
Programming Principles

- **Your Aren't Gonna Need It [YAGNI]** : Do not write something just because you will need it in the future.
- In order to apply DRY principle programmers end up writing most abstract and generic code. Apply abstraction only when you need it
- **Document your code** : Adding proper comments to your code is necessary to make it more readable
- **Refactor** : Never shy away from refactoring your code. Most good designs come from analysis of bad designs
- **Clean your code**: Don't add tons of logic in one line. Keep your code clear and concise.

Coding Best Practices

- Programmers employ numerous tactics to ensure readable and organized code. These include:
  - Using naming conventions for variables;
  - Placing whitespaces, indentations and tabs within code;
  - Adding comments throughout to aid in interpretation.

Agile & Scrum


Traditional Teams
Project Manager
Team Lead
Agile Teams
Self-organizing
Servant Leader
Facilitator

Introduction to Agile

- Agile practices involve discovering requirements and developing solutions through the collaborative effort of self-organizing and cross-functional teams and their customer(s)/end user(s).
- It is a approach that helps shift the focus from writing about requirements to talking about them.
- Agile methodologies emphasizes on small teams delivering small increments of working software with great frequency while working in close collaboration with the customer and adapting to changing requirements.

Agile Manifesto - Values

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

***Individuals and interactions** over processes and tools*

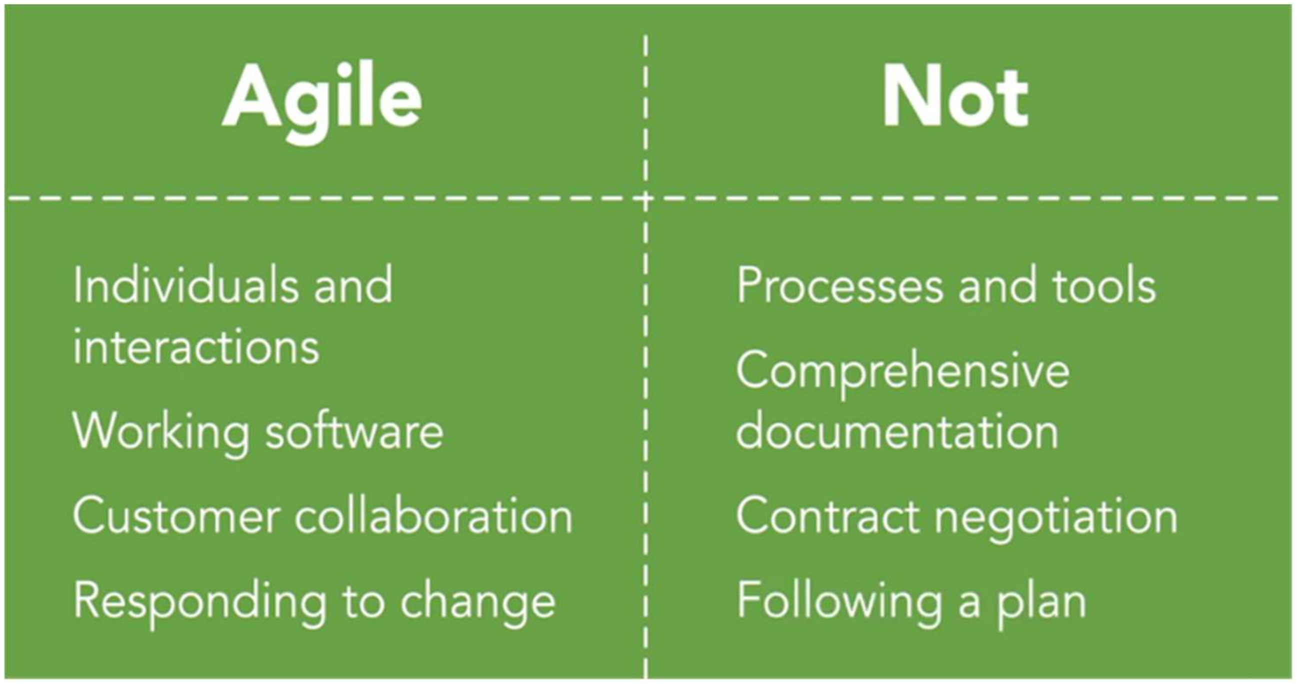
***Working software** over comprehensive documentation*

***Customer collaboration** over contract negotiation*

***Responding to change** over following a plan*

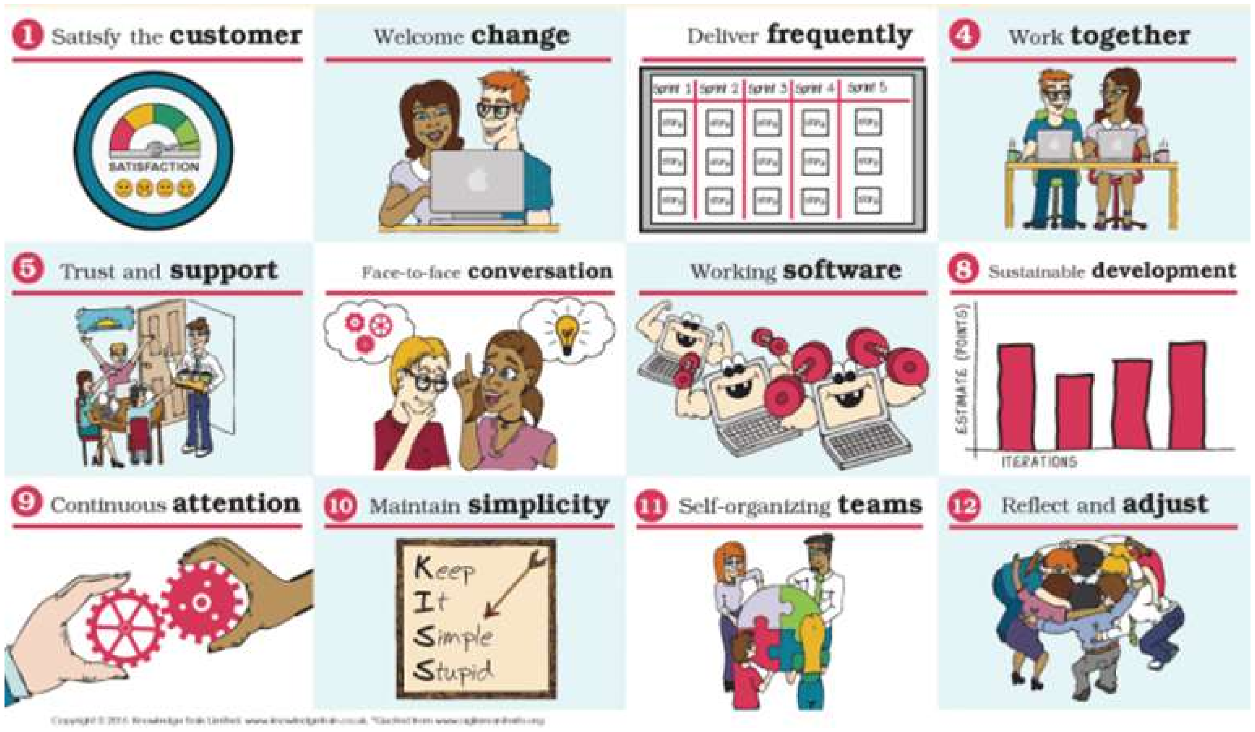
That is, while there is value in the items on the right, we value the items on the left more.

Agile Manifesto - Values



Agile	Not
Individuals and interactions	Processes and tools
Working software	Comprehensive documentation
Customer collaboration	Contract negotiation
Responding to change	Following a plan

Agile Manifesto - Principles



Agile Principles

- **Deliver Value Faster** – Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- **Welcome Change** – Welcome changes to requirements, even late in projects. Agile processes harness that change for the customer’s competitive advantage.
- **Deliver Working Software Frequently** – Working software should be delivered after a couple of weeks to a couple of months, with a preference to the shorter timescale.
- **Work Together Daily** – Business people and developers must work together daily throughout the project.

Agile Principles

- **Build Projects Around Motivated Individuals** – Give them the environment and support they need and trust them to get the job done.
- **Face-to-Face Conversations** – The most efficient and effective method of conveying information to and within a development team is with face-to-face conversation.
- **Working Software is Key** – Working software is the primary measure of progress.
- **Sustainable Development** – Agile processes promote sustainable development. The sponsors, project team members (developers), and users should be able to maintain a constant pace indefinitely.

Agile Principles

- **Attention to Technical Excellence** – Continuous attention to technical excellence and good design enhances agility.
- **Simplicity** – The art of maximizing the amount of work not done is essential.
- **Self-Organizing Teams** – The best architectures, requirements, and designs emerge from self-organizing teams.
- **Reflect and Adjust** – At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Benefits of Agile

- Easy to make changes in various development phases as against the waterfall model
- Value is delivered in smaller increments, unlike in waterfall where end product is delivered only at the end of the entire SDLC
- Feedbacks are periodic and timely
- Implementing changes are easier
- Boasts confidence and productivity of the team

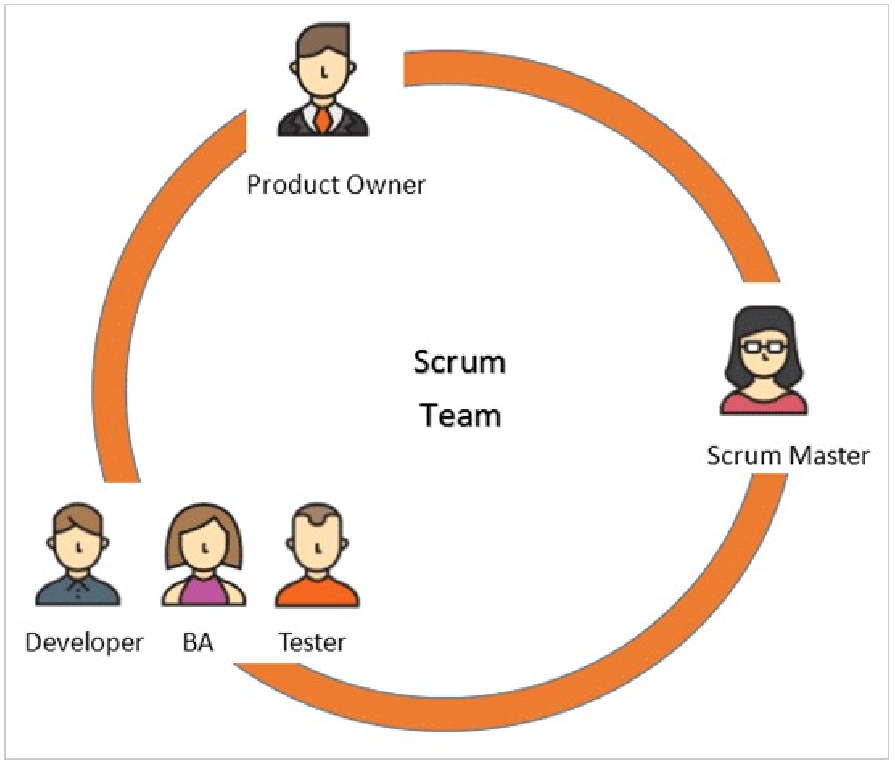
Challenges in Agile

- It’s a about entire Mindset change
- Difficult to implement : all in or not at all
- Lack of project documentation
- Roles are not clearly defined

Scrum

- Def : Scrum is a lightweight framework that helps people, teams and organizations generate value through adaptive solutions for complex problems
- Designed to solve complex problems that requires sprinting or iterating through a solution
- A framework that helps people and organizations discover what works best for them
- Inspection, Adaption and transparency are the three pillars of Scrum
- Sprint : Heart of Scrum , timeline decided by the team (1-4 weeks)
- Planning, Development, Testing and Review of a working unit of software is completed in each sprint

Scrum Roles



Product Owner

- Responsible for ensuring the team is delivering the most value
- A representative of the business, who tells the development team what is important to deliver
- Takes inputs from the customer and prioritizes the work items to deliver MVP (minimal viable product)
- Should have a vision for the value the scrum team is delivering to the customer

Responsibilities

- Managing the scrum backlog
- Release management
- Stakeholder management

Scrum Master

- Responsible for gluing everything together and ensuring that scrum is being done well.
- A servant leader who not only describes a supportive style of leadership but describes what they do on a day-to-day basis.
- Helps the Product owner better understand and communicate value, to manage the backlog
- Helps development team self-organize, focus on outcomes, get to a “done increment,” and manage blockers

Scrum Master Responsibilities

Scrum Master helps the development team by

- Coaching the team members in self-management and cross-functionality;
- Helping the Scrum Team focus on creating high-value Increments that meet the Definition of Done
- Causing the removal of impediments to the Scrum Team’s progress; and,
- Ensuring that all Scrum events take place and are positive, productive, and kept within the time box.

Scrum Master Responsibilities

Scrum Master helps the Product owner by

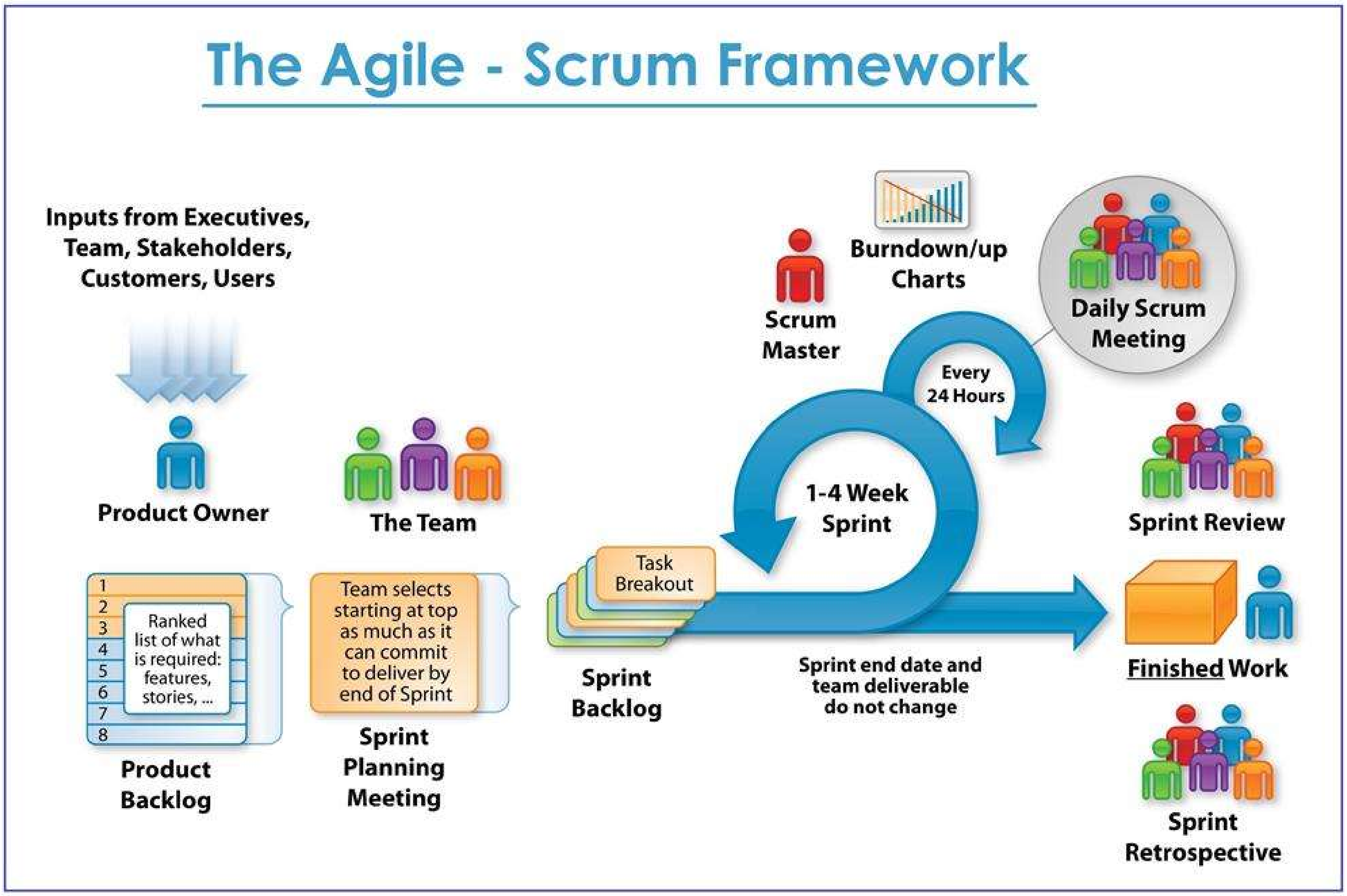
- Helping find techniques for effective Product Goal definition and Product Backlog management;
- Helping the Scrum Team understand the need for clear and concise Product Backlog items;
- Helping establish empirical product planning for a complex environment; and,
- Facilitating stakeholder collaboration as requested or needed

The Development Team

- Comprised of designers, writers, programmers, QAs and BAs
- The ‘developer’ role in Scrum means a team member who has the right skills, as part of the team to do the work.
- Should be able to self-organize so they can make decisions to get work done
- To ensure transparency during the sprint they meet daily at the daily scrum.

Responsibilities

- Creating a plan for the Sprint, the Sprint Backlog;
- Instilling quality by adhering to a Definition of Done;
- Adapting their plan each day toward the Sprint Goal; and,
- Holding each other accountable as professionals


The Agile - Scrum Framework
Inputs from Executives, Team, Stakeholders, Customers, Users
Product Owner
The Team
1
2
3
4
5
6
7
8
Ranked list of what is required: features, stories, ...
Product Backlog
Team selects starting at top as much as it can commit to deliver by end of Sprint
Sprint Planning Meeting
Task Breakout
Sprint Backlog
Scrum Master
Burndown/up Charts
Daily Scrum Meeting
Every 24 Hours
1-4 Week Sprint
Sprint end date and team deliverable do not change
Sprint Review
Finished Work
Sprint Retrospective

Sprint Planning

- The sprint is initiated with Sprint planning meeting
- The Product Owner, the development team and the Scrum Master determine which are the most important Product Backlog Items for the Sprint.
- Addresses following topics
  - ✓ Why is this sprint valuable
  - ✓ What can be done in this sprint
  - ✓ How will the chosen work get done

Daily Scrum/Stand-up

- Conducted everyday during the sprint, preferably at same time and place
- Lasts a maximum of 15 minutes , geared by the scrum master
- The scrum values of Inspect and Adapt are practiced by everyone answering the following questions
  - ✓ What have I achieved since the last Daily Scrum?
  - ✓ What do you think I will work on in the next Daily Scrum?
  - ✓ What exactly is stopping me from making progress?
- Improves communications, identifies impediments, promotes quick decision-making and consequently eliminates the need for other meetings

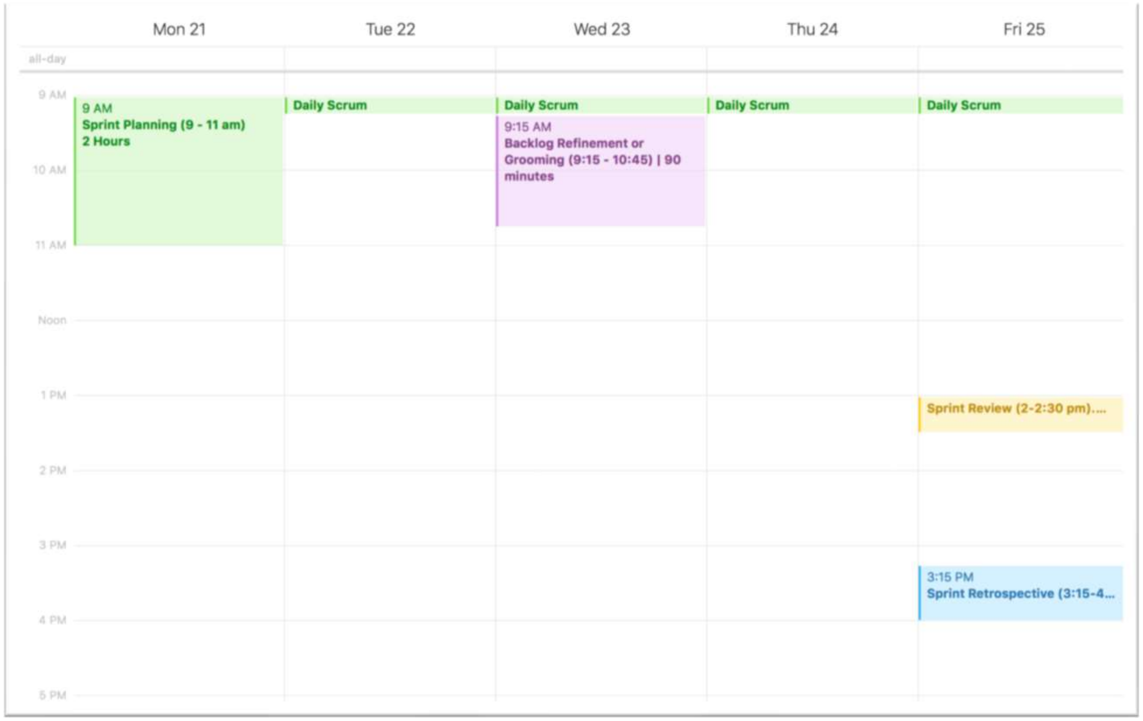
Sprint Review

- Communication between the Scrum Team, stakeholders, sponsors, customers and interested members of other teams
- The focus is on checking the features that have just been completed with regard to the overall development effort.
- The people who are not in the Scrum team can get information about the status of the product and influence the next steps.
- The members of the Scrum Team have the opportunity to develop a better understanding of the business through constant communication.
- Sprint Review is a planned opportunity to review and adapt a product

Sprint Retrospective

- Sprint Retrospective is an opportunity to review and adapt the process.
- The development team, the Scrum Master and the Product Owner come together and discuss what works and what doesn’t in practice with Scrum
- The focus is on continuous improvement of the process
- The idea is, Scrum Team should identify a meaningful set of improvement actions that they will then implement in the next Sprint.

Sample Calendar – 1 Week



9 AM
Sprint Planning (9 - 11 am)
2 Hours

Daily Scrum

Daily Scrum

9:15 AM
Backlog Refinement or Grooming (9:15 - 10:45) | 90 minutes

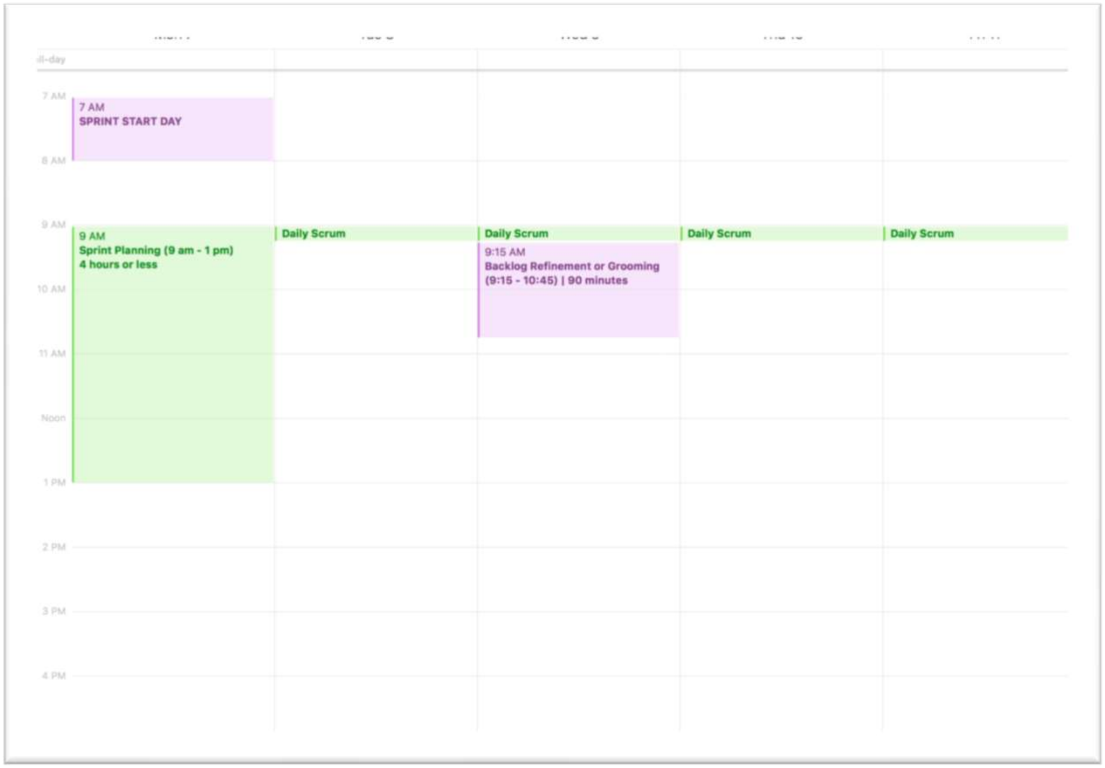
Daily Scrum

Daily Scrum

Sprint Review (2-2:30 pm)....

3:15 PM
Sprint Retrospective (3:15-4...

Sample Calendar – 2 Weeks



7 AM
SPRINT START DAY

9 AM
Sprint Planning (9 am - 1 pm)
4 hours or less

Daily Scrum

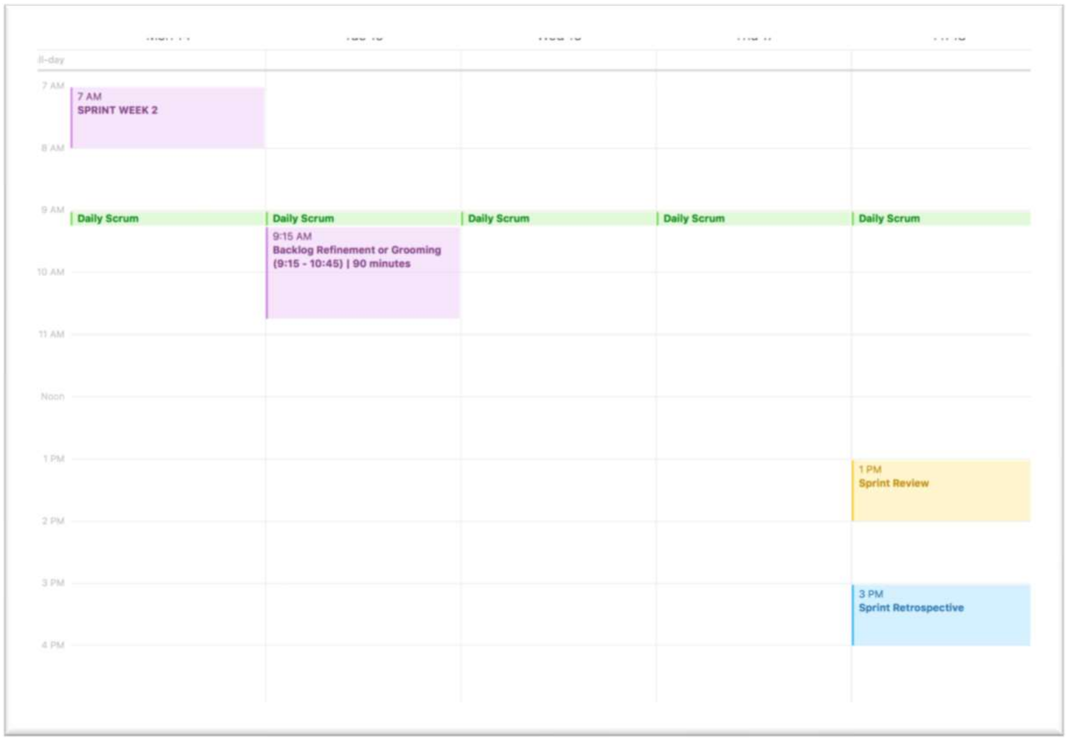
Daily Scrum

9:15 AM
Backlog Refinement or Grooming
(9:15 - 10:45) | 90 minutes

Daily Scrum

Daily Scrum

Sample Calendar – 2 Weeks



User Stories

- Simple description of product feature that is written from end users point of view.
- Captures 3 important items
  - ✓ Who
  - ✓ What
  - ✓ Why
- Is not a replacement of requirement document, it is a lightweight conversation about what user wants in the application

User Stories – 3 C’s

- Capture the essence of any user story
- Stand for
  - ✓ Card
  - ✓ Conversation
  - ✓ Confirmation
- The product owner writes down the user story on a ‘card’ after discussion with the customer
- The team and the product owner have a ‘Conversation’ about who, what and why
- The development team then ‘Confirms’ with the product owner that what they have understood is correct

Acceptance Criteria

- Simple notes or conditions added to the user story
- Tells what the user story must do to satisfy the needs of a customer
- These are defined by the Product owner
- Enriches the user story by making it testable and ensures that the story is ready for demo to the customer
- Product owner writes the initial acceptance criteria which is later confirmed by taking to the team
  - ✓ Should be testable with pass/fail results
  - ✓ Should be clear and concise
  - ✓ Should be created with shared understanding

Product Backlog

- Is the list of requirements requested by the customer and prioritized by the Product Owner
- Team pulls the user stories from this list before each sprint to create a sprint backlog
- The product owner changes and re-prioritizes the backlog throughout the project development process as needed.
  - ✓ Communicates the priority of PBIs
  - ✓ Allows longer term planning
  - ✓ Ensures all customer needs are noted down

Sprint Backlog

- Is a list of Product Backlog Items (PBIs) that the team selects to complete during a Scrum sprint
- These PBIs are further broken down in terms of tasks and assigned an estimate in work hours
  - ✓ Ensures that the highest-priority PBIs are completed first.
  - ✓ Breaks down work into manageable components
  - ✓ Allows the team to determine the amount of PBIs they can accomplish during the sprint.

Definition of Done

- An agreement document the team creates that provides everyone a shared understanding of what work needs to be completed as part of the Increment.
- Product backlog must meet the Definition of done before release
- Conformed by the team of developers
- The definition of done not only means the feature is developed, but also tested with no pending bugs and is deployable
- Guiding document for team members
- Ensures transparency by being public and visible

Software Testing

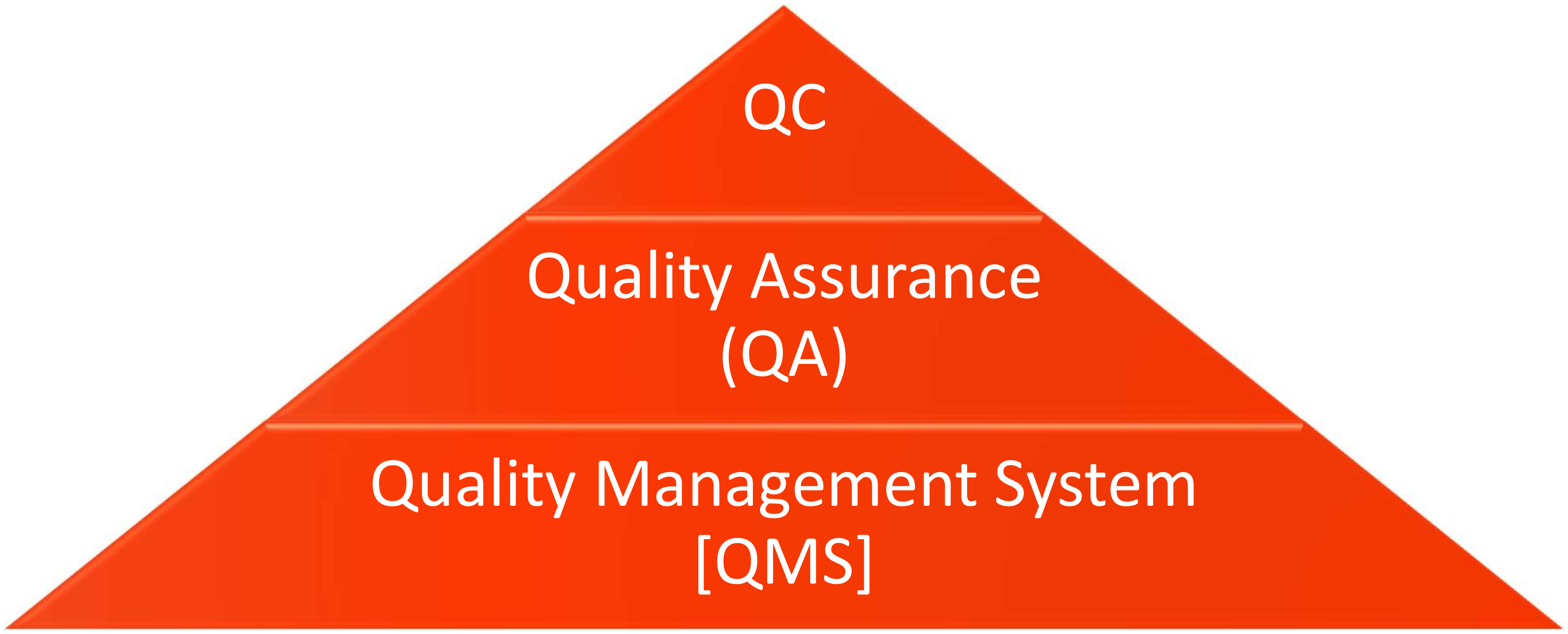
Why Software Testing

- Vast array of programming languages, operating systems and hardware platforms
- Software reaching to wide spectrum of people and fulfilling almost every need
- Almost 90% of the devices society relies on are software driven
- Software written today touches millions of lives and help them do their work efficiently and effectively
- Can be reason for their frustration or dissatisfaction.
- Could result in loss of work or loss of business

Objectives of Testing

- To prevent defects in various work items – Requirements, Design, Code , User Stories etc. using early testing
- To verify all requirements have been fulfilled
- To check if an object under test behaves as per the stakeholder /user's expectation
- To build confidence in the level of quality of the object under test
- To detect defects or failures in developed software
- To provide information regarding the quality of software to the concerned stakeholder and inform them about risk of releasing the software
- To comply with contractual, legal, or regulatory requirements or standards

Quality Assurance and Testing



Quality Assurance and Testing

- Quality Assurance and Testing are related terms, but they are not the same
- Quality Management System includes defining and implementing the quality objectives and processes to achieve quality in the organization
- Quality management includes both quality assurance and quality control
- Quality assurance is typically focused on adherence to proper processes to ensure quality of developed product
- Quality control involves various activities, including test activities, that support the achievement of appropriate levels of quality

Error, Defects and Failures

- Error is any human mistake that leads to faulty product [incorrect code]
- Errors introduce defects (bugs) in the software [Taxes not calculated correctly]
- One defect can lead to another one in related work items [faulty requirements - > faulty design]
- When defective code is executed we get failures
- Some failures are never identified, if date/preconditions are not exercised
- Root cause analysis is a technique of finding out the initial cause for failure in the application
- Root cause analysis can help in identifying the weaker areas in development process and limit the defects by improving those

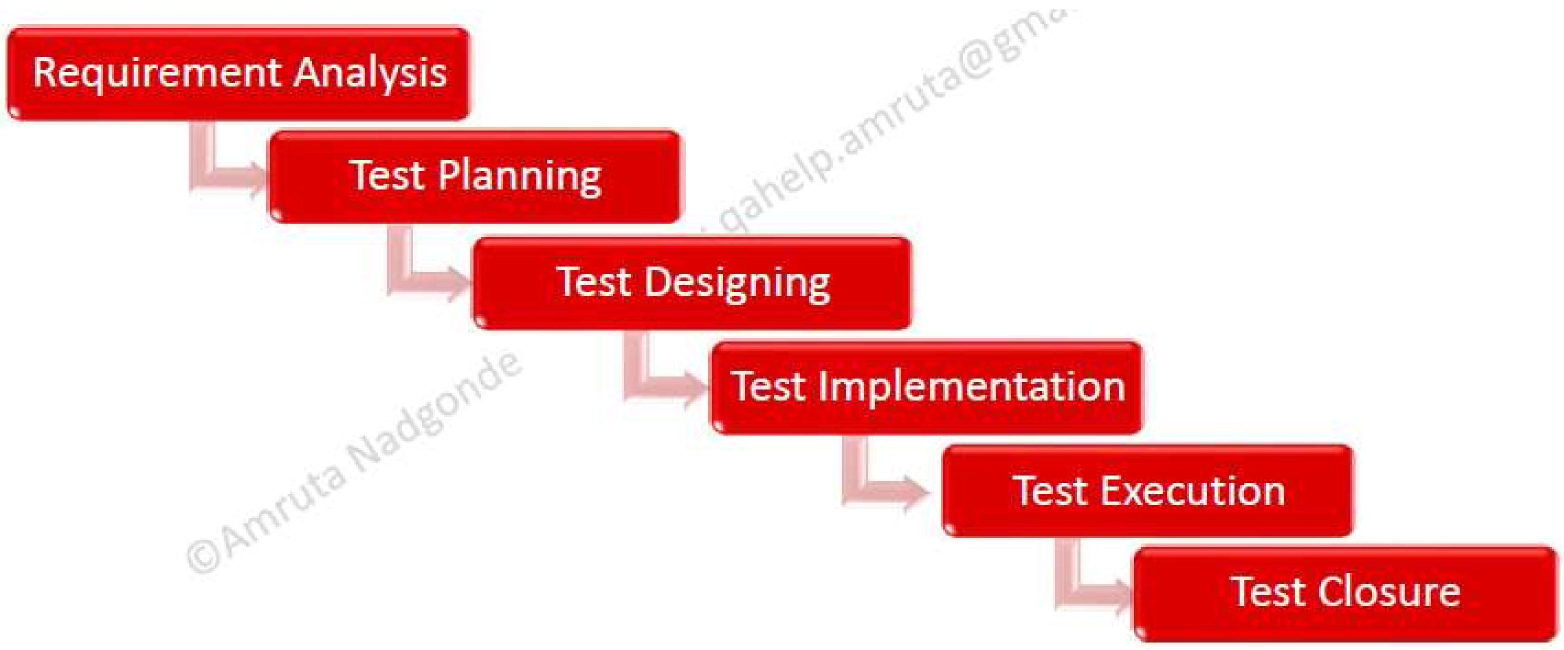
Causes of Errors

- Strict Deadlines
- Error made by Humans
- Miscommunicated or mis-interpreted requirements/design
- Complexity of code/design/architecture
- New /unfamiliar technologies
- Mis-interpretation of inter/intra system interfaces
- Not unexpected results are defects, false positives may occur due to errors in the way tests were executed, or due to defects in the test data, the test environment, or other testware.
- False negatives are tests that do not detect defects that they should have detected, false positives are reported as defects, but aren't actually defects.

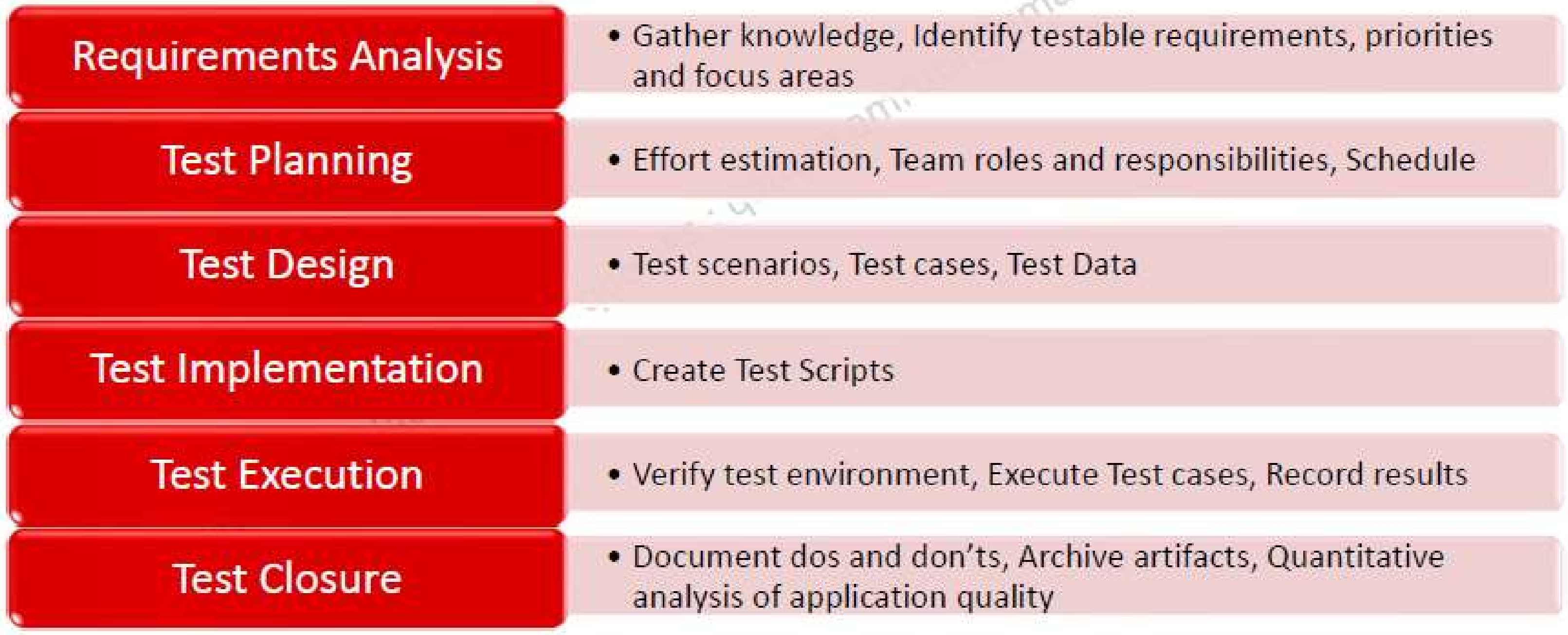
Software Testing Principles

1. Testing shows the presence of defects
2. Exhaustive testing is not possible
3. Early testing
4. Defect clustering
5. Pesticide paradox
6. Testing is context-dependent
7. Absence of errors fallacy

Software Testing Life Cycle



Software Testing Life Cycle



Requirements Analysis	• Gather knowledge, Identify testable requirements, priorities and focus areas
Test Planning	• Effort estimation, Team roles and responsibilities, Schedule
Test Design	• Test scenarios, Test cases, Test Data
Test Implementation	• Create Test Scripts
Test Execution	• Verify test environment, Execute Test cases, Record results
Test Closure	• Document dos and don'ts, Archive artifacts, Quantitative analysis of application quality

Test Process

- The process is defined by set of activities that should be carried out for achieving the test objectives
- There is no universal process that may prove effective for every kind of project/organization
- The test activities depend on factors like
  - SDLC Framework or model used
  - Environment or Product Domain
  - Testing types / levels considered
  - Product / project risks
  - Business constraints like budget, schedule, complexity
  - Organizational processes or policies
- Irrespective of which type/level of testing is chosen, it is desirable that it should have measurable coverage

Test Activities and Tasks

- Various activities and tasks conducted as part of the testing process include

  - Test planning
  - Test analysis
  - Test design
  - Test implementation
  - Test execution
  - Test completion

Test Work Products

Planning	• Test Plans
Analysis	• Test Scenarios • Test Conditions
Design	• Test Cases • Test Data
Implementation	• Test Scripts • Test Suits
Execution	• Test Pass\Fail Reports • Defect Reports
Closure	• Summary Reports • Finalized Test ware

Test Planning

- Serves as the guild line throughout the testing process, which defines Test objectives and overall approach for testing
- Test management is important considering the thousands of tests, resources and activities conducted during testing
- A clear and concise goal is very important for a testing project (since definition of testing may vary and could falsely depict the goal)
- Resource management and resource efficiency is another task which can be handled by effective planning
- Test plan also highlights the schedule and budget

Test Analysis

- Test basis is analyzed to identify the testable features and define test conditions. It defines "what to test"
- Depending on level of testing test basis may differ, usually contains analysis of:
  - Requirement specifications [business/functional] , Use cases, User stories or similar work products that define the functional and non function specifications of a component/ system under test
  - Design specifications, control or data flow diagrams, entity relationship diagrams, interface specification that define structure of the component / system under test
  - Work product that specify the implementation of the component itself – code, database metadata and queries, interfaces
- It is also important to analyze the risk analysis reports which may consider functional, non-functional, and structural aspects of the component or system

Test Analysis

- The test basis should be evaluated with an intension to find defects related to
  - Ambiguities
  - Omissions
  - Inconsistencies
  - Inaccuracies
  - Contradictions
- Evaluation is also necessary for
  - Identifying all testable features
  - Defining and prioritizing test conditions for each feature
  - Capturing bi-directional traceability between each element of the test basis and the associated test conditions

Test Design

- While test analysis defines what to test, test design specifies how to test
- During test designing, the test conditions are elaborated to create the test cases / test scenarios (in agile)
- Test designing including following main activities
  - Creating and prioritizing test cases
  - Identifying and designing required test data
  - Designing the test environment and identifying necessary tools
  - Creating Requirement traceability matrix

Test implementation

- While test design determines how to test, test implementation check “do we have everything in place to start testing”
- Typical activities include:
  - Developing and prioritizing test procedures
  - Creating automated scripts
  - Creating and arranging test suits for execution
  - Building test environment
  - Loading test data in the test environment
- Test design and test implementation tasks are often combined

Test Execution

- During test execution, test suites are run as per the test execution schedule and actual behavior of the system is recorded
- Typical activities include
  - Executing selected tests either manually or by using test execution tools
  - Comparing actual results with expected results
  - Analyzing the defects are their causes
  - Reporting the defects
  - Logging the outcome of test execution (e.g., pass, fail, blocked)
  - Repeating the execution as per the regression plans or as required after defect fix.

Test Closure

- Test closure activities are conduced at a milestone in project like release / increment / completion of test level
- Idea is to capture and collect data from completed test activities to consolidate experience, test ware, and any other relevant information
- Typical activities include
  - Checking which planned activities were completed as per the plan
  - Creating Test summary report
  - Ensure all the reported defects are either closed / deferred / de-scoped
  - Handing over the test ware to maintenance team
  - Analyzing lessons learned from the completed test activities to determine changes needed for future iterations, releases, and projects or process improvements

Software Development and Testing

- In every SDLC model , testing is characterized by:
  - For every development activity, there should be a corresponding test activity
  - Each test level should have test objectives specific to that level
  - Test analysis and design for a given test level should begin during the corresponding development activity [V-model]
  - Testers should participate in discussions to define and refine requirements and design, and should be involved in reviewing work products like requirements, user stories , design etc
- No matter what type of SDLC model you choose testing should start at an early stage in SDLC

Testing throughout the SDLC

Test early , Test often

Software Development and Testing

- Software Development is the process of communicating information about eventual program and translating the information from one form to another
- Vast majority of errors can be attributed to this breakdown, translation and noise during the communication and transmission of information
- Some documents that need attention include
  - Requirements – specify why program is needed
  - Objectives – specify what program should do
  - External Specifications – define exact representation of programs to end user
  - Design documents – specify how program is constructed

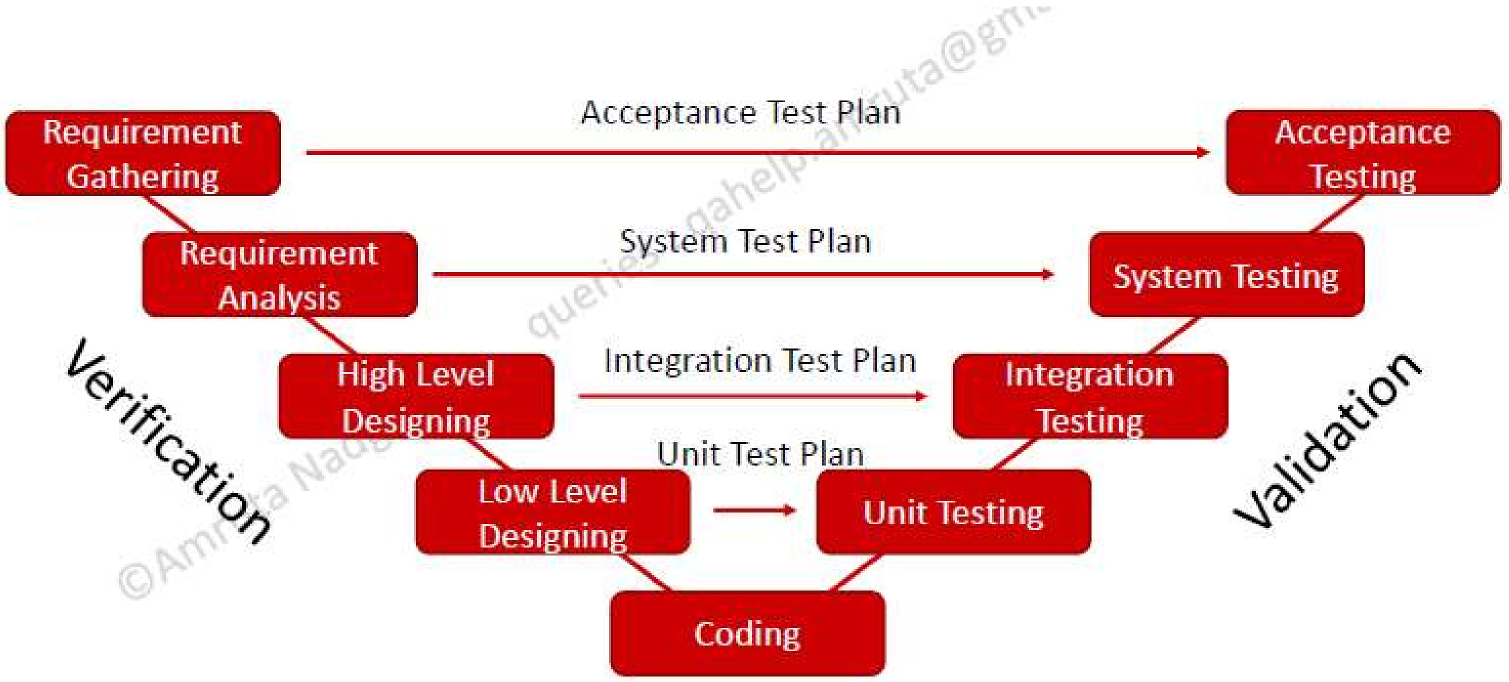
Software Testing Methodologies

- Defects exist in application even before it is coded
- Software testing should not only be focused around finding the defects in the developed application but also finding the defects in the process of development
- Finding defects as early as possible is the key
- Software testing efforts can broadly be divided into two categories
  1. Verification
  2. Validation

V Model

- Integrates the test process throughout the development process, implementing the principle of early testing
- Deliverables of each phase of the development process undergo verification
- The validation is conducted using different levels of testing
- Each development phase provides input to the respective test plan used in validation
- Supports the principle of “early testing” by including a test level for every phase of development
- The execution of tests associated with each test level proceeds sequentially, but in some cases overlapping occurs.

V Model


Requirement Gathering
Acceptance Test Plan
Acceptance Testing
Requirement Analysis
System Test Plan
System Testing
High Level Designing
Integration Test Plan
Integration Testing
Low Level Designing
Unit Test Plan
Unit Testing
Coding
Verification
Validation

Verification – Static Testing

- Static testing relies on the manual examination or tool driven automation evaluation of work products
- Focuses on finding defects as early as possible – right after the requirement gathering phase
- Reduces the cost of fixing a defect found in the later stages
- Localizes the defects
- Reduces the pressure of fixing a defects / probability of adding new defects while fixing
- something under pressure
- Slogan : Are we doing the job right?

Verification – Static Testing

- The work products that need to be verified include but not limited to
  - Specifications, including business requirements, functional requirements
  - Epics, user stories, and acceptance criteria
  - Architecture and design specifications
  - Code
  - Testware, including test plans, test cases, test procedures, and automated test scripts
  - User guides
  - Web pages
  - Contracts, project plans, schedules, and budget planning

Verification methods

- Verification methods such as inspection/walkthroughs/review work on early detection of errors
- Involve a team of people visually reading or inspecting a work artifact like code
- Since verification is done by other people than the author himself chances of finding errors are more than older desk-checking method
- Idea is to find the error but not correct them, errors are corrected later en masse
- Effective technique to find almost 30-70% of logic related error

Benefits of Static Testing

Some Defects found by Static Test

- Requirement defects (e.g., inconsistencies, ambiguities, contradictions, omissions, inaccuracies, and redundancies)
- Design defects (e.g., inefficient algorithms or database structures, high coupling, low cohesion)
- Coding defects (e.g., variables with undefined values, variables that are declared but never used, unreachable code, duplicate code)
- Deviations from standards (e.g., lack of adherence to coding standards)
- Incorrect interface specifications (e.g., different units of measurement used by the calling system than by the called system)
- Gaps or inaccuracies in test basis traceability or coverage (e.g., missing tests for an acceptance criterion)

Verification Techniques

- Verification methods such as inspection/walkthroughs/review work on early detection of errors
- Involve a team of people visually reading or inspecting a work artifact like code
- Since verification is done by other people than the author himself chances of finding errors are more than older desk-checking method
- The focus of verification depends on the agreed objectives of the technique used – finding defects/gaining understanding/educating participants/discussing or deciding by consensus]
- Idea is to find the error but not correct them, errors are corrected later en masse
- Effective technique to find almost 30-70% of logic related error

Review - Peer

- Main purpose: detecting potential defects
- Possible additional purposes: generating new ideas or solutions, quickly solving minor problems
- Not based on a formal (documented) process
- May not involve a review meeting
- May be performed by a colleague of the author (buddy check) or by more people
- Results may be documented
- Varies in usefulness depending on the reviewers
- Use of checklists is optional

Review - Technical

- Main purposes: gaining consensus, detecting potential defects
- Possible further purposes: evaluating quality and building confidence in the work product,
- generating new ideas /alternate implementations
- Reviewers should be technical peers of the author, and technical experts in the same or other
- disciplines
- Individual preparation before the review meeting is required
- Review meeting is optional, ideally led by a trained facilitator (typically not the author)
- Scribe is mandatory, ideally not the author
- Use of checklists is optional
- Potential defect logs and review reports are produced

Inspection

- Main purposes: detecting potential defects, evaluating quality and building confidence in the work product, preventing future similar defects through author learning and root cause analysis
- Possible further purposes: motivating and enabling authors to improve future work products and the software development process, achieving consensus
- Follows a defined process with formal documented output , based on rules and checklists
- Moderator leads the session, notes down all the errors and make sure that the errors are subsequently corrected
- The test specialist should be well versed with testing and should have enough knowledge about the artifact under test

Inspection

- Specified entry and exit criteria are used
- Scribe [recorder] is mandatory
- Review meeting is led by a trained facilitator (not the author)
- The author of the document (e.g. code) reads the document line by line and explains the logic
- The document is analyzed and errors are noted
- At the end of the session the author is given the list of the errors uncovered

Walkthrough

- Main purposes: find defects, improve the software product, consider alternative implementations, evaluate conformance to standards and specifications
- Possible additional purposes: exchanging ideas about techniques or style variations, training of participants, achieving consensus
- Less formal than inspection involving 3-5 people including a moderator, test specialist, author and others
- Follows procedure similar to that of inspection where author reads the artifact and others find the errors
- But here the tester comes prepared with set of tests to be executed on the code, mentally checking the output

Success factor of Verification

- Clear Objectives which are defined during planning and applied as measurable exit criteria
- Right type of verification technique used depending on the work product type/level and participants
- Check lists used are up to date and address the main risks
- Participants have adequate time to prepare
- Reviews are scheduled with adequate notice
- Reviews are integrated in the company's quality and/or test policies

Success factor of Verification

- The right people are involved to meet the review objectives [people with different skill sets/perspective]
- Participants dedicate adequate time and attention to detail
- Defects found are acknowledged, appreciated, and handled objectively
- The meeting is well-managed, so that participants consider it a valuable use of their time
- The overall atmosphere is positive and not used to critique the participants
- Adequate training is provided, especially for more formal review types such as inspections

Validation

- A disciplined approach to evaluate weather the final, developed application fulfills its specific intended purpose
- Validation is done towards the end of the development process where actual tests are executed on the application
- Also knows as dynamic testing as software is tested using computer based tests
- Conducted using various levels of testing namely – unit, integration, system and user acceptance.
- Shows presence of error but no location
- Slogan : Are we doing the right job?

Component / Unit Testing

- Module Testing /Unit testing is the process of testing individual subprograms, subroutines, classes or procedures in the program
- As long as each module works as defined the application as a whole has a better chance of working as intended
- Unit testing isolates the errors which are easy to debug
- Reduces the future cost due to early detection of errors
- Simplifies the next level of testing i.e. integration as individual modules to be integrated are already tested

Component / Unit Testing

- The purpose is to compare the function of the module to some functional or interface specification defining the module
- Module specifications and source code are used to design the test cases
- White box testing techniques like decision coverage and condition coverage are used as applying these techniques are easier at the module level than at the application level
- Module's logic is tested using white box testing technique which is supplemented by black box tests applied to the module's specification
- Usually conducted with the help of tools like Junit, NUnit

Component / Unit Testing

- Components /units are tested by the developer right after they are developed.
- Automated Tests prove useful in incremental models like Agile where it is important to check if existing functionality breaks after changes in code
- A technique like TDD can also be used where tests are identified first, followed by component development
- Typical defects found are:
  - Incorrect functionality (e.g., not as described in design specifications)
  - Data flow problems
  - Incorrect code and logic

Integration Testing

- Integration Testing is an incremental approach to combine the tested modules
- The focus on integration testing is on the interfaces between the components rather than the components themselves
- Purpose of integration testing is to confirm that the individually tested units can work together to deliver intended functionality
- Units are depended on each other as
  - One module is calling other
  - Transfer of data from one module to another
  - Logical dependency between the module (one module can be executed only after the previous one
  - completes)

Integration Testing

Component Integration Testing

- Focuses on integration and interactions between two units/components. Includes running on automated tests.
- System design , sequence diagrams and external interface specifications are used as the basis for test designing

System Integration Testing

- Focuses on integration and interactions between Systems/micro services/external web services.
- Can be done along with system testing/after system testing is done
- Since external interfaces are managed by provider organization, some challenges can be faced (e.g. Test blocking defects in APIs not getting fixed, arranging the Test environments etc)

Integration Testing

Typical defects found in component integration testing

- Incorrect data, missing data, or incorrect data encoding
- Incorrect sequencing of interface calls
- Interface mismatch
- Failures in communication between components
- Incorrect assumptions about the meaning, units, or boundaries of the data being passed between components

Integration Testing

Typical defects found in system integration testing

- Inconsistent message structures between systems
- Incorrect data, missing data, or incorrect data encoding
- Interface mismatch
- Failures in communication between systems
- Incorrect assumptions about the meaning, units, or boundaries of the data being passed between systems
- Failure to comply with mandatory security regulations

System Testing

- System testing can be viewed as testing of entire system, considering end-to-end tasks it performs
- Requirement documents are analyzed to understand the expected functional and non functional behavior of the system
- Typical Objectives include:
  - Validating that the system is complete and will work as expected
  - Building confidence in the quality of the system as a whole
  - Finding defects
  - Preventing defects from escaping to higher test levels or production

Acceptance Testing

- Acceptance testing is done to check the system as a whole
- The intension to evaluate if the system is ready for release and use by customer
- The acceptance criteria is derived from the user requirements specification, confirming that the application behaves as required by the user
- Done with data supplied by the end user
- Validates the functional and non functional needs of the end user

Alpha /Beta Testing

- Objective is to build confidence among potential or existing customers, that they can use the system under normal, everyday conditions and in the operational environment(s) to achieve their objectives with minimum difficulty, cost, and risk
- Another objective may be the detection of defects related to the conditions and environment(s) in which the system will be used
- Typical defects include:
  - System workflows do not meet business or user requirements
  - Business rules are not implemented correctly
  - System does not satisfy contractual or regulatory requirements
  - Non-functional failures such as security vulnerabilities, inadequate performance efficiency under high loads

Alpha Testing

- Tested at the developer’s site by a customer
- Developer observes and records errors and usage problems
- The tests are conducted in test environments which is under developers control
- The developer is available whenever needed

Beta Testing

- Performed after Alpha testing is done
- Done by the end user [beta users] at the customer’s site
- Done in pre-production environment –which simulates the live application environment
- Usually done by trusted partners or community users
- The developer is not present during beta testing

Functional Testing

- External Specifications is precise description of application behavior from end user’s point of view
- Function testing is conducted to verify the correctness and completeness of these external specifications
- Goal should be to find the discrepancies between the external specifications and actual behavior
- Specifications should be analyzed to derive the test cases Effective test data creation techniques like equivalence class partitioning, boundary value analysis, error guessing etc can be used
- Except for a very small application , function tests are executed as black box tests

Test Case Designing

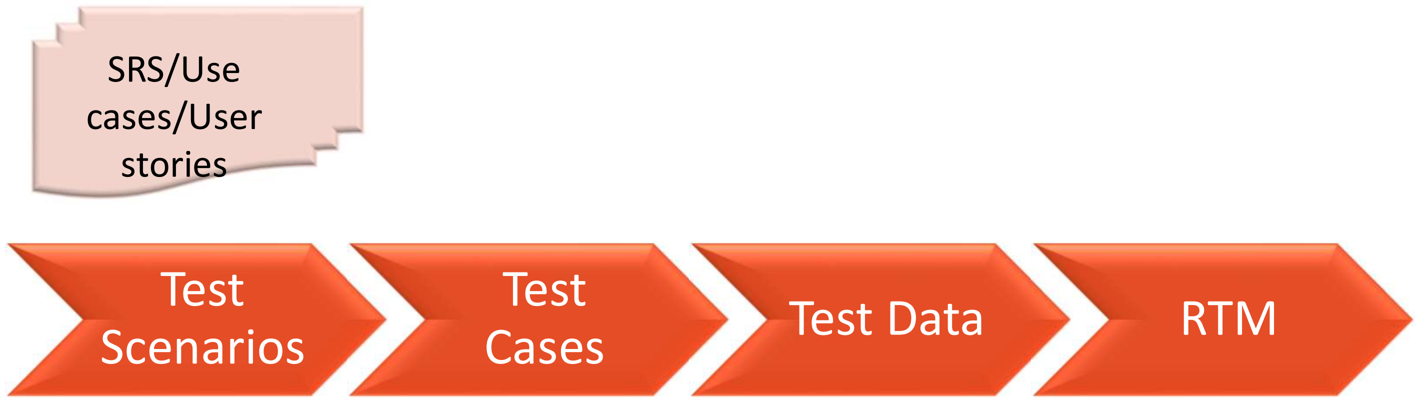
- Testing can not guarantee absence of all errors
- Complete / exhaustive testing is impossible considering all possible input /output data
- Design and creation of effective test cases is very important in program testing
- Time and cost constraints make it more challenging
- Goal should be to find out

What subset of all possible test cases has the

highest probability of detecting the most errors

Test Case Designing

- Starts after the Test plan is prepared which defines scope and objectives of testing
- Design activity consists of creation of Test scenarios, Test cases, Test data
- Testing team should gather enough knowledge of the application before design activity starts during the test analysis phase



Inputs for Test Designing

- Various artifacts can be available for test design activity depending on stage of development

Application under Development

- SRS
- FRS
- Use cases
- User Stories
- Proto-type s /Wireframes

Application in Use

- Application Demo
- Previous Test Artifacts
- Change Request Document

Test Case Designing

- Following questions can be important to ask while test design activity
  - Which components of application are under current scope
  - What are the features and functionalities of various modules and their interdependencies
  - What are the user roles and their privileges
  - How does data flow in the application
  - Which business rules and validations are implemented

Test Scenario Creation

Use Case :

- Describes interaction between the application functionality (process) and user of the application (Actor)
- Each process is one scenario (e.g. Banking system possible scenarios could be withdrawal, deposit etc.)

Functionality Breakdown :

- Includes identification of various features or functionalities provided by the application as mentioned in the SRS document
- Each functionality represents one scenario

Test Case

- Describes detailed procedure that helps to test a particular aspect or feature
- Specifies how to test a particular functionality
- Describes steps to be performed along with input data
- The detailed description of expected outcome is also written

Test Case Creation

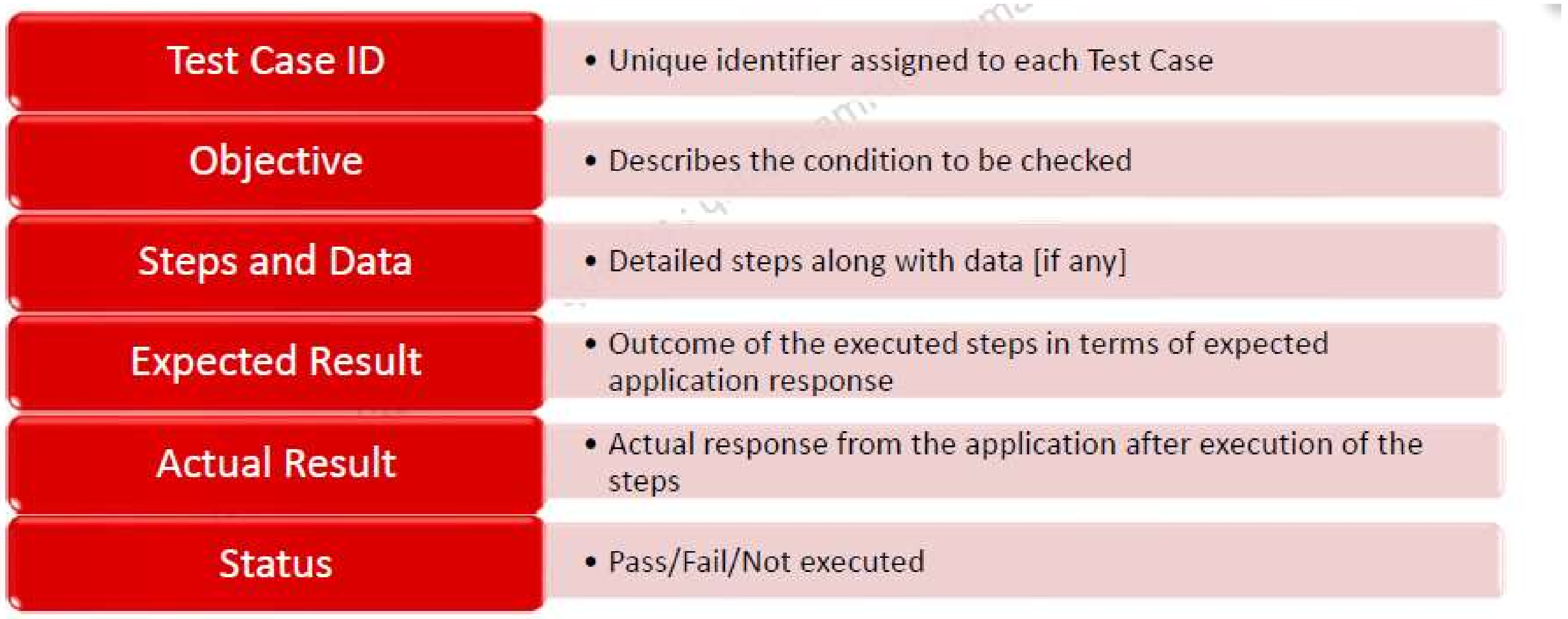
From Use Case

- Happy Path
- Error Paths
- Alternate Paths

From SRS / FRS

- Happy Path
- Error Paths
- Alternate Paths

Test Case Template



Test Case ID	• Unique identifier assigned to each Test Case
Objective	• Describes the condition to be checked
Steps and Data	• Detailed steps along with data [if any]
Expected Result	• Outcome of the executed steps in terms of expected application response
Actual Result	• Actual response from the application after execution of the steps
Status	• Pass/Fail/Not executed

White Box Testing

- Strategy to test applications by examining the internal structure
- Also known as logic driven testing, emphasizes on finding faults in the logic of the program
- Test data is devised from the logic of the program
- Methods used are path coverage, decision coverage and condition coverage
- Just like exhaustive input/output in black box, exhaustive path testing is difficult and impractical
- Cannot prove that the program behaves as per the specifications

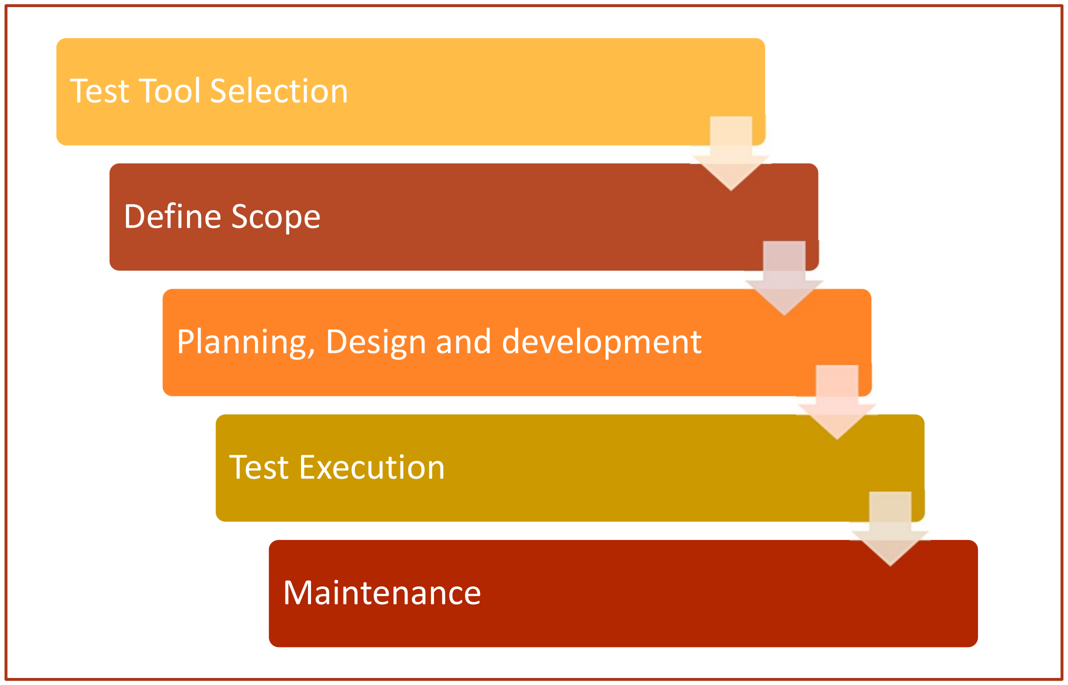
White Box Testing

- Concerned with the degree to which test cases exercise or cover the logic of the program
- Statement Coverage - % of the number of statements executed by the tests divided by the total number of executable statements
- Branch/Condition Coverage – % of the true/false branch outcomes executed by the tests divided by the total number of branch outcomes
- Decision Coverage - % of the number of decision outcomes executed by the tests divided by the total number of decision outcomes
- Statement coverage is not good enough to find all logical mistakes made in the code. A stronger logic coverage criteria is decision coverage or branch coverage

Non Functional Testing

- System Testing
- User Experience Testing
- Performance Testing
- Installation Testing
- Compatibility Testing
- Configuration Testing
- Security Testing

Introduction to Automation



Why Automation Testing

- Manual testing of all work flows, all fields, all negative scenarios is time and cost consuming
- It is difficult to test for multi lingual sites manually
- Automation does not require Human intervention. You can run automated test unattended (overnight)
- Automation increases speed of test execution
- Automation helps increase Test Coverage
- Manual Testing can become boring and hence error prone.
- Useful while testing complex and business critical test scenarios.

Selecting Tests to Automate

Tests to be automated:

- High Risk - Business Critical test cases
- Test cases that are executed repeatedly
- Test Cases that are very tedious or difficult to perform manually
- Test Cases which are time consuming.

Tests not to be automated:

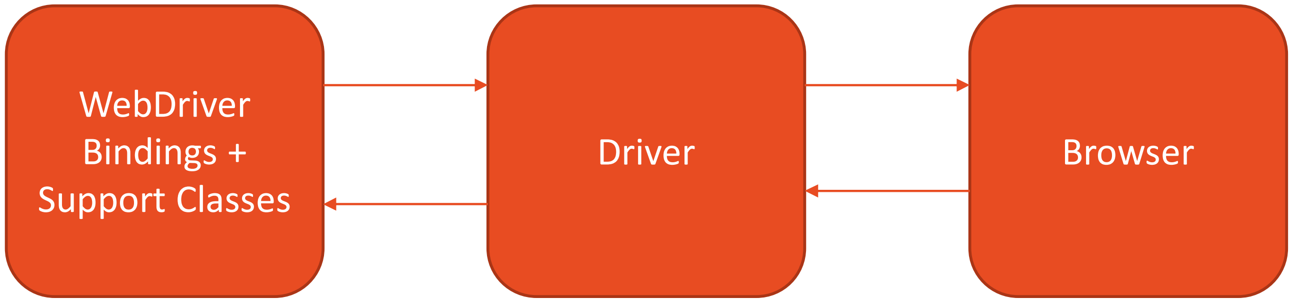
- Test Cases that are newly designed and not executed manually at least once
- Test Cases for which the requirements are changing frequently
- Test cases which are executed on ad-hoc basis.

Selenium Terminology

- **Selenium WebDriver:** A browser automation framework that accepts commands and sends them to a browser.
- Instead of injecting a JavaScript code into the browser to simulate user actions, it uses the browser's native support for automation. Uses Object oriented API
- **Selenium Server** allows using Selenium - WebDriver on a remote machine.
- **Selenium IDE**: is a Firefox add-on that records user activity and creates a test case based on it. It can also play the tests back and save them as a program in different languages.
- **Selenium Grid:** allows you run the tests on different machines against different browsers in parallel; in other words it enables distributed test execution.

WebDriver API

- An API and protocol that defines language neutral interface for controlling browsers
- Provides a set of interfaces to discover and manipulate DOM elements in web documents and to control the behavior of browsers
- Refers to both the language bindings and the implementations of the individual browser controlling code



Browser Driver

- Selenium provides drivers specific to each Browser
- Browser driver interacts with the respective browser by establishing a secure connection
- For every selenium command an HTTP request is generated and delivered to the browser driver through HTTP server
- The steps (like click, find text) are executed on the browser and response is sent back to the browser driver
- Driver then sends it back to the script
- A bridge between WebDriver wire protocol and Browser API

WebDriver Manager

- As discussed in earlier slide, compatible driver version needs to be installed for every browser version
- From selenium 4.6 there is not need to explicitly download the browser drivers
- WebDriver Manager helps in setting up the drivers required by Selenium WebDriver
- Add following dependency in your project to work with WebDriver Manager

```
<dependency>
    <groupId>io.github.bonigarcia</groupId>
    <artifactId>webdrivermanager</artifactId>
    <version>x.x.x</version>
  </dependency>
```

WebDriver Interface

- The WebDriver interface represents an idealized web browser
- The methods in this interface fall into three categories:

1. Control of the browser itself
2. Selection of WebElements

Locating Elements

- The success of automated GUI (Graphical User Interface) tests depends on identifying and locating GUI elements and then performing operations and verifications on these elements.
- Highly depends on the test tool's ability to recognize various GUI elements effectively.
- Selenium's feature-rich API provides multiple locator strategies such as Name, ID, CSS selectors, XPath, and so on to locate elements.
- When we automate using Selenium, We need to identify information such as attribute values and elements structure for locating elements and perform user actions using Selenium WebDriver API.

Finding Elements on page

- Selenium WebDriver interface provides methods for locating elements from underlying web applications

```
findElement()
WebElement we = driver.findElement(By.<locator>);
```

- Finds the first WebElement using the given locator.
- Returns a matching row, or try again repeatedly until the configured timeout is reached.
- Throws : NoSuchElementException - If no matching elements are found

Finding Elements on page

findElements()

List<WebElement> we = driver.findElements(By.<*locator*>));

- Finds all elements within the current page using the given mechanism.
- Returns as soon as there are more than 0 items in the found collection, or will return an empty list if the timeout is reached.
- Note : : findElement should not be used to look for non-present elements, use findElements(By<locator>) and assert zero length response instead.

By Class

- It is the Mechanism used to locate elements within a document.
- All subclasses rely on the basic finding mechanisms provided through static methods of this class

```
public WebElement findElement(By by)

WebElement input = driver.findElement(By.id("lst-ib"))
```

- Finds the first WebElement using the given method.

Selenium Locators

Locator	Description
ID	Identifies a web element using html attribute ‘id’
WebElement we = driver.findElement(By.id(“lst-ib”);	
Name	Identifies a web element using html attribute ‘name
WebElement we = driver.findElement(By.name(“btnK”);	
ClassName	Identifies a web element using html attribute ‘class name
WebElement we = driver.findElement(By.classname(“gsfi”);	

Selenium Locators

linkText	Used for identifying web links using exact ‘linktext’ visible on the page
WebElement we =driver.findElement(By.linkText("Gmail"))	
PartialLinkText	Used for identifying web links using partial ‘linktext’ visible on the page
WebElement we = driver.findElement(By.partialLinkText("ma"))	
TagName	Identifies one/all web element/s having same html ‘tag’
List<WebElement> lw = driver.findElements(By.tagName("a"));	

Locating Elements using CSS Selector

- A style sheet language a for specifying how documents written in markup language such as HTML or XML are presented to users.
- Major browsers implement CSS parsing engines for formatting or styling the pages using CSS syntax.
- Selectors (pattern matching rules) determine which style should be applied to elements in the DOM.
- If all conditions in the pattern are true for a certain element, the selector matches the element and the browser applies the defined style in CSS syntax.
- Selenium WebDriver uses same principles of CSS selectors to locate elements in DOM.

Lets look at how we can locate various elements on Login form using CSS Selectors:

Consider following piece of HTML code for Login form:

```
<form name="loginForm">
<label for="username">UserName: </label>
 <input type="text"  id="username" /><br/>
<label for="password">Password: </label>
<input   type="password" id="password" /><br/>
<input name="login" type="submit" value="Login" />
 </form>
```

- **Absolute Path:**
  - Refers to the very specific location of the element considering its complete hierarchy in the DOM.

```
WebElement userName = driver.findElement(By.cssSelector("html body div div form input"));
                                        or
WebElement userName = driver.findElement(By.cssSelector("html > body > div > div > form > input"));
```

- **Relative Path:**
  - With relative path, we can locate an element directly irrespective of its location in the DOM.

```
WebElement userName = driver.findElement(By.cssSelector("input"));
```

- **Using Class Selector:**

Done by specifying the type of HTML tag, then adding a dot followed by the value of the class attribute in the following way:

```
WebElement userName = driver.findElement(By. cssSelector(“”input.className"));
```

- **Using ID Selector:**

Done by specifying the type of HTML tag, then adding a hash followed by the value of the ID attribute in the following way:

```
WebElement userName =     driver.findElement(By.cssSelector("input#username"));
```

➢**Using Attributes Selector**

Apart from the class and id attributes, CSS selectors also enable the location of elements using other attributes of the element.

```
WebElement userName = driver.findElement(By. cssSelector("input[name=‘Login’]']"));
```

➢Multiple attributes can be used to locate an element where one attribute may not be sufficient :

```
WebElement previousButton = driver.findElement(By.cssSelector("input[t
ype='submit'][value='Login']"));
```

Performing partial match on attribute values

- CSS Selectors also provide a way to locate elements matching partial attribute values.
- Useful for testing applications where attributes values are dynamically assigned and change every time a page is requested.

Syntax	Example
^= (Starts with)	input[id ^=‘userN']
$= (Ends with)	input[id$=‘rName']
*= (Contains)	Input[id *='erNam']

Locating Elements using XPath

- The XML path language, is a query language for selecting nodes from an XML document.
- Tells how to traverse a particular element using HTML node hierarchy.
- Selenium WebDriver supports XPath for locating elements using XPath expressions or queries.
- Using the xpath() method of the By class we can locate elements using XPath syntax.
- Using XPath, a test can locate elements in multiple ways based on the structure of the document, attribute values, text contents and so on.

Lets look at how we can locate various elements on Login form using XPath:

Consider following piece of HTML code for Login form:

```
<form name="loginForm">
<label for="username">UserName: </label>
 <input type="text"  id="username" /><br/>
<label for="password">Password: </label>
<input   type="password" id="password" /><br/>
<input name="login" type="submit" value="Login" />
 </form>
```

- **Absolute Path:**

- Refers to the very specific location of the element, considering its complete hierarchy in the DOM. It always starts with html.

```
WebElement userName = driver.findElement(By.xpath(“ html/body/div/div/form/input”) )
```

- **Relative Path:**

- With relative path, we can locate an element directly irrespective of its location in the DOM.

```
WebElement userName = driver.findElement(By.xpath("//input"));
```

- Returns the first <input> element in DOM

- **Using Index:**
  - Used when there are multiple elements with same <html tag>. we can also locate the element by using its index in DOM

```
WebElement userName = driver.findElement(By.xpath("//input[2]"));
```

- **Using Attribute Values:**
  - We can also locate elements using their attribute values in Xpath in following manner:

```
WebElement userName =     driver.findElement(By.xpath("//input[@id='username']"));

WebElement loginButton =  driver.findElement(By.xpath
("//input[@type='submit'][@value='Login']"));
```

Performing partial match on attribute values

- XPath also provides a way to locate elements matching partial attribute values using XPath functions.
- Useful for testing applications where attributes values are dynamically assigned and change every time a page is requested.

Syntax	**Example**
startswith()	input[starts-with(@id,'ctrl')]
endswith()	input[ends-with(@id,'userName')]
contains()	Input[contains(@ id,'userName')]

Chapter 5

WORKING WITH WEB ELEMENTS

Web Element Methods

- **Input box**: Represents an Edit box/Text box where user can input data.
- Identified by <Input> tag and <type> attribute as ‘text’

Web Element	Method	Description
Input Box	sendKeys("Text")	Enters "Text" in a Input box

Web Element Methods

- **Button:** Represents a web button that is clickable.
- Identified by <button> or <input> tag and attribute <type> as ‘submit’

Web Element	Method	Description
Button	click()	Clicks on a button

Web Element Methods

- **Check box:** Represents a web checkbox.
- Identified by <input> tag and <type> attribute as ‘checkbox’

Web Element	**Method**	**Description**
CheckBox	isDisplayed()	Checks if checkbox is displayed on page
	isEnabled()	Checks if checkbox is enabled
	isSelected()	Checks if checkbox is selected
	click()	Checks/Unchecks a checkbox based on previous condition

- **Check box:** Represents a web checkbox.
- Identified by <input> tag and <type> attribute as ‘checkbox’

Web Element	Method	Description
CheckBox	isDisplayed()	Checks if checkbox is displayed on page
	isEnabled()	Checks if checkbox is enabled
	isSelected()	Checks if checkbox is selected
	click()	Checks/Unchecks a checkbox based on previous condition

- **Radio Button:** Represents a web Radio button.
- Identified by <input> tag and <type> attribute as ‘radio’

Web Element	**Method**	**Description**
Radio Button	isDisplayed()	Checks if Radio button is displayed on page
	isEnabled()	Checks if Radio button is enabled
	isSelected()	Checks if Radio button is selected
	click()	clicks on a Radio button

- **Dropdown:** Represents a dropdown (list).
- Uses class ‘Select’ of *‘org.openqa.selenium.support.ui.Select’* package. The dropdown must be specified using tag <select>.

Web Element	Method	Description
Drop down Class used : Select Select dd = new Select(driver.findElement(By.id("lang-chooser")));	getOptions()	Returns all elements in a dropdown in a WebElement list
	selectByValue()	selects an element based on its value in 'value' attribute
	selectByIndex()	selects an element based on its index
	selectByVisibleText()	selects an element based on its visible text
	isMultiple()	Returns true if the dropdown is multiselect.
	getFirstSelectedOption().getText()	Returns the text of first selected item
	deselectAll()	Deselects all items in a multiselect list box

- **Multiselect:** Represents multiselect List, where user can select multiple items at a time.
- Uses class ‘Select’ of ‘*org.openqa.selenium.support.ui.Select*’ package.

Web Element	Method	Description
MutliSelect Class Used : Select Select dd = **new** Select(driver.findElement(By.*name*("cars")));	selectByValue()	selects an element based on its value ein 'value' attribute
	selectByIndex()	selects an element based on its index
	selectByVisibleText()	selects an element based on itsvisible text
	isMultiple()	Retruns true if the dropdown is multiselect.
	deselectAll()	Deselects all items in a multiselect list box
	getAllSelectedOptions()	Retruns an array of all selected items

- **Web Table**: Represents a web table defined by <thead> and <tbody> tages.
- Table columns and rows are returned in collection as <webElement> list
- <table><thead><tr><th> gives Table header
- <table><tbody><tr> gives table rows
- <table><tbody><tr>>td[i] gives data in i-th row

Web Element	**Method**	**Description**
Tables	getText()	Retruns text of a perticular cell
	size()	Generalized method of List collection returns no of columns or rows based on the collection

- **Menu:** Represents a menu item.

Web Element	Method	Description
Menu	click()	Clicks on a particular menu item

- **Dynamic Menus:** Dynamic menus are the ones, submenus for whom appear on hovering mouse over them.
- ‘Actions’ class is used to perform ‘mouse hover’ action on dynamic menus

```
Actions act = new Actions(driver);
WebElement menuFoundation =
driver.findElement(By.xpath("//a[@id='menu640']/span"));
act.moveToElement(menuFoundation).perform();
```

Dynamic Menus

Web Element	Method	Description
Dynamic menus	act.moveToElement(Menu_name).perform()	Hovers mouse over the dynamic menu item to display its submenus
	getText()	Returns text of a menu item in a collection/singleton
	size()	Prints no of submenus under a particular menu item where submenus are captured as a Webelement collection
	click()	Clciks on a particular menu item.

Checking Element's Text

Method	Description
getText()	Returns value of the innerText attribute of the element.

- Used to verify that elements are displaying correct values or text on the page.

```
WebElement message = driver.findElement(By.id("message"));
String messageText = message.getText();
assertEquals("Click on me and my color will change",
messageText);
```

- If the element has child elements, the value of the innerText attribute of child elements will also be returned along with parent element.

Checking Element’s attribute value

- Attributes are set to control the behavior or style of elements when they are displayed in browser.

Method	Description
getAttribute()	Returns the value of the attribute back to the test.

- Used to verify that element attributes are set correctly.

```
WebElement message = driver.findElement(By.id("message"));
assertEquals("justify",message.getAttribute("align"));
```