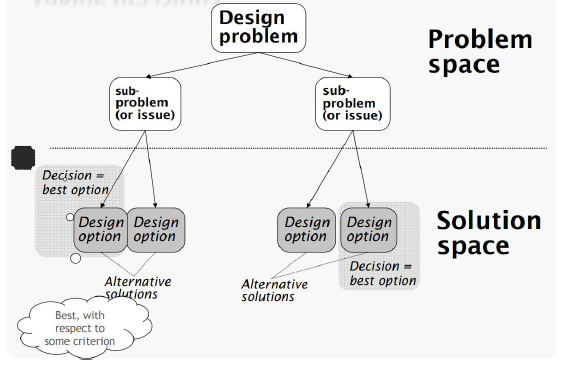
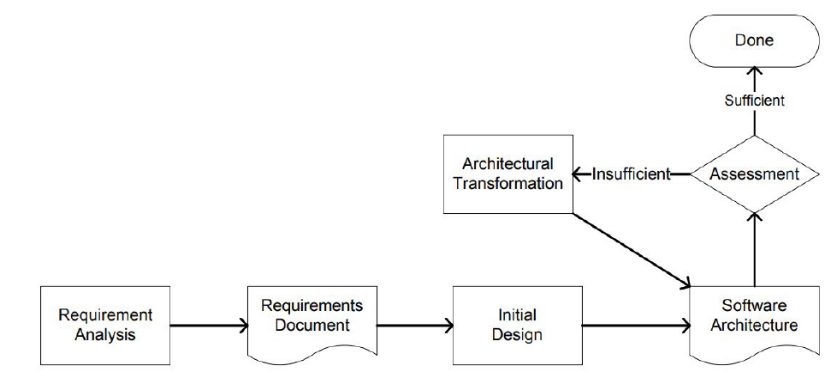
***Introduction to Software Engineering: Final Exam Review***

***Software Architecture Patterns:***

* Architectural design is a creative process so it differs depending on the type of system being developed.
* One or more QA or NFR and a set of design options that we think could satisfy system quality attributes
* **Architectural decision 🡪** selecting one of the available design options
* Validation is very difficult but very important
* The system architect or the architecture team of the organization is responsible for making these decisions.

***Making Design Decisions:***

* **Current knowledge**: system requirements, the design created so far
* **Previous knowledge:** available technology, what has worked previously, software design principles and best practices
* **Prediction of the future:** requirements changes

***Architecture Reuse:***

* System in the same domain often have similar architectures that reflect domain concepts
* Application product lines are built around a core architecture with variants that satisfy customer reqs.
* The architecture of a system may be design around 1+ architectural patterns or styles.

***Architectural Patterns (Styles):***

* A set of principles/design decisions that has been useful in one practical context (e.g. problem) and will probably be useful in others.
* Help define the basic characteristics and behavior of an application.
* Describe a re-usable solution to frequently recurring problems.
* Applications lacking a good architecture are generally tightly coupled, expensive to change.
* Patterns are a means of representing, sharing and reusing knowledge.
* An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
* Patterns should include information about when they are and when they are not useful.
* Patterns may be represented using tabular and graphical descriptions.

***What is a good architecture?***

* We can determine the arch characteristics of the application without fully understanding the inner-workings of every component in the system
* Can answer: does it scale, what are the performance characteristics of the application, how easily does the app respond to change

***Contents:***

1. **Name:** A meaningful and memorable way to refer to the pattern, typically a single word or short phrase.
2. **Requirement:** a single-sentence statement that presents the fundamental requirement addressed by the pattern in the form of a question.
3. **Problem:** A description of the problem indicating the intent in applying the pattern. Part of the problem description includes common circumstances that can lead to the problem (also known as “forces”).
4. **Solution**: A description, using text and/or graphics, of how to achieve the intended goals and objectives.
5. **Rationale**: An explanation/justification of the pattern, or of individual components within it. Indicating how the pattern works. How it resolves the forces to achieve the desired goals and objectives.
6. **Application**: This part is dedicated to describing how the pattern can be applied. It can include guidelines, implementation details, and sometimes even a suggested process.

***Common Patterns (Layered):***

* The most common architecture pattern is the **layered architecture** pattern.
* The components within the layered architecture pattern are organized into horizontal layers.
* Each layer performing a specific role within the application. Most layered architectures consist of four standard layers: presentation, business, persistence, and database.
* The layered architecture focuses on separation of concerns among layers.
* Each of the layers in the architecture is marked as being closed.
* **Closed Layer:** as a request moves from layer to layer, it must go through the layer right below it to get to the next layer below that one.

**Layer Isolation:**

* Each layer has little or no knowledge of the inner workings of other layers in the architecture.
* Changes made in one layer of the architecture generally don’t impact or affect components in other layers.
* Allow us to construct loosely coupled software application
* **Opened Layer**: a request can bypass a layer (open layer) and go directly to the layer below it.

**Quality Attribute:**

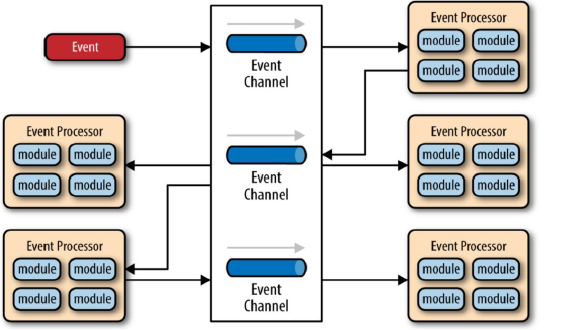
* Sinkhole antipattern: requests flow through multiple layers of the architecture as simple pass-through processing with little or no logic performed within each layer.
* Layered architecture tends to lend itself toward monolithic architecture
* Think about software quality attributes (e.g. performance, testability, modifiability, scalability, etc.)

**Common Usages:**

* The layered architecture is very common in in desktop applications, mobile applications, and e-commerce web applications.
* Different layers could be physically deployed on different machines or containers.
* It is straightforward to implement and test this pattern. However, the scalability and the performance of this pattern become an issue when the application size increase.

***Event-Bus Pattern:***

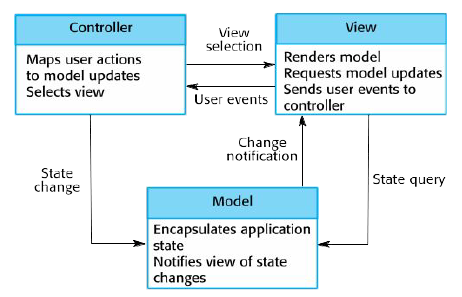
* Also known as publisher-subscriber pattern, that is commonly used to implement event-based application.
* This is a messaging patterns, commonly used when your system consists of several components that needs to communicate using effectively.
* publisher (message source), subscriber (message listener), event bus, and event channel
* **Complex pattern** to implement, mainly because its asynchronous distributed nature. It Inherits most of the complexities exist in distributed architecture.
* Enable building application with high performance and scalability.
* Very common in distributed application.



***Microkernel Architecture:***

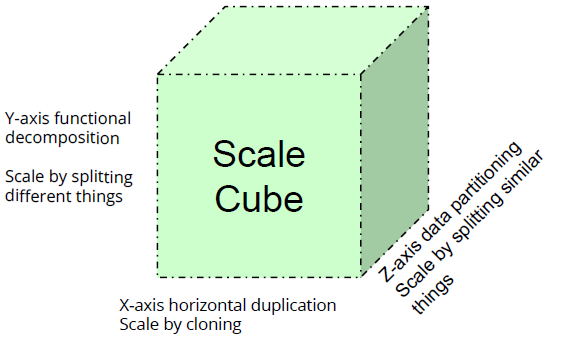
* Application logic is divided between independent plug-in modules and the basic core system.
* The core system contains only the minimal functionality required to make the system operational.
* The plug-in modules are independent components that contain specialized processing, additional features, and custom code that is meant to enhance or extend the core system.
* This architecture provides excellent support for evolutionary design and incremental development.
* It can be **embedded or used** as part of another architecture pattern.
* It enables developing highly customized application that target large market with common needs and custom specific requirements.
* System uses microkernel architecture has a **high performance**, **easy to test**, but **difficult to develop and scale.**
* It is easy with this pattern to fail in enforcing the **separation of concerns** principle

***Model View Control (MVC):***

* Separates presentation and interaction from the system data.
* The system is structured into three logical components that interact with each other.
* The **Model component** manages the system data and associated operations on that data.
* The **View component** defines and manages how the data is presented to the user.
* The **Controller** component handles the inputs from the user.
* Allows the data to change independently of its representation and vice versa.
* Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
* Can involve additional code and **code complexity** when the data model and interactions are simple.
* Very common pattern is **web application development**
* Layered architecture has a linear topology, MVC is triangular (see diagram)

***Microservices Architecture (Netflix, Ebay, Amazon):***

* Design your system as a set of standalone independent services with focused concern.
* Functionally decompose the application into a set of collaborating services.
* Each service implements a set of narrowly, related functions.
* Services communicate using either synchronous protocols such as HTTP/REST or asynch protocols (AMQP).
* Services are developed and deployed independently of one another.
* Each service has its own database in order to be decoupled from other services

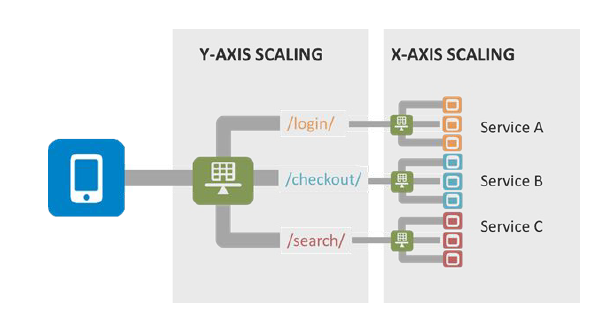
***Microservices Architecture Pattern:***

***Scalability:***

* Is the ability to continue to function well when workload increased by repeatedly applying a cost-effective strategy for extending a system’s capacity.
* Is the ability to continue to function with acceptable performance when the workload has been significantly increased.
* Is the ability not only to function well in the rescaled situation, but to actually take full advantage of it (upward scalability vs downward).

**X-Axis (Cloning, replication):**

* scaling an application by running clones behind a load balancer is known as X-axis scaling
* If there are N copies, then each copy handles 1/N of the load. This is a simple, commonly used approach of scaling an application. (Client 🡪 Load Balancer 🡪 Application Servers)
* Each copy potentially accesses all of the data; caches require more memory to be effective.
* It does not tackle the problems of increasing development and application complexity.

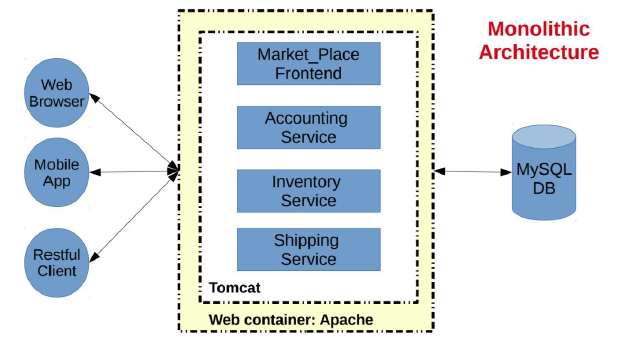
**Y-Axis (Functional Decomposition):**

* Splits the application into multiple, different services
* Each service is responsible for one or more closely related functions
* Services are small, autonomous, and independent
* Two options **verb-based** decomposition and **noun-based** decomposition
* Client 🡪 load balancer 🡪 login, checkout, search, service (A, B, C)

**Z-Axis:**

* Scaling an application by running clones on different servers/machines.
* Each server runs an identical copy of the code.
* Each server is responsible for only a subset of the data.
* The system is responsible for routing each request to the appropriate server.
* Routing criteria could be based on application specific or data-model specific
* Z-axis splits are commonly used to scale databases.
* improves fault isolation since a failure only makes part of the data inaccessible.
* improves cache utilization and reduces memory usage and I/O traffic.
* improves transaction scalability since requests are typically distributed across multiple servers.
* doesn’t solve the problems of increasing development and application complexity.
* increased application complexity, implement a partitioning scheme, which can be tricky especially if we ever need to repartition the data

***Monolithic Architecture:***



* **Simple to develop:** one technology stack, one framework
* **Simple to deploy:** one executable, rails application with single directory, Python or nodeJS single directory
* **Simple to scale (x-axis):** you can scale the application by running multiple copies behind a load balancer
* **Large Code Base:** modularity breaks down over time
* **Difficult continuous deployment, overloaded web container, difficult scaling.**
* Requires a long-term commitment to a technology stack

***How to Model & Integrate Services:***

* What makes a good service: small (2 weeks), louse coupling (changes shouldn’t require changes), high cohesion
* All the communication between the services themselves are via network calls.
* Smart endpoints and dumb pipes.
* Make your service APIs simple for consumer.
* Hide internal implementation detail.
* Keep your APIs technology agnostic.
* Use asynchronous calls if possible and avoid blocking your services.

***Inter-Process Communication:***

* **One-to-one:** each client request is processed by exactly one service instance. Request/Response 🡪 notification 🡪 Request/Async response
* **One-to-many:** multiple service instances process each request. Publish/subscribe 🡪 Publish/async response
* **Synchronous:** the client expects a timely response from the service and block while it waits.
* **Asynchronous:** the client doesn’t block while waiting for a response, and the response, if any, isn’t necessarily sent immediately.

***How to Deploy a Service?***

**Single service per instance host:**

* Deploy each single service instance on its own host
* Services instances are isolated from one another
* There is no possibility of conflicting resource requirements or dependency versions
* A service instance can only consume at most the resources of a single host
* It's straightforward to monitor, manage and redeploy each service instance
* Less efficient resource utilization compared to multiple services per host

**Multiple service instances per host pattern:**

* Run multiple instances of different services on a host (Physical or Virtual machine).
* Efficient resource utilization.
* Running multiple instances of different services on the same host raise the: Risk of **conflicting resource requirements** and **dependency versions**
* Difficult to limit the resources consumed by a service instance

**Service instance per container:**

* Package the service as a (Docker) container image and deploy each service instance as a container.
* Container encapsulates the details of the technology used to build the service.
* A container imposes limits on the CPU and memory consumed by a service instance.
* Containers are extremely fast to build and start.
* Example: Docker, Kubernetes, Marathon/Mesos Amazon EC2 Container Service

Accessing service? 🡪 client makes request to API Gateway who calls service. Service managed by framework (Django)

***API Gateway:***

* A server that is the single-entry point into the system.
* Encapsulates the internal system architecture and provides an API that is tailored to each client class.
* Have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.
* **Examples:** Netflix API Gateway, Netflix/Hystrix, Amazon API Gateway, Azure API Gateway
* The API Gateway can also provide each client with a custom API.
* Exposes a coarse-grained API for mobile clients.
* **Highly Available** component that must be developed, deployed and managed.
* Could become a development bottleneck, Could become a single point of failure
* Services typically need to call one another. In modern microservice-based application typically runs in a virtualized or containerized environment where the number of instances of a service and their locations changes dynamically.

***How to Discover Service (Client-Side Discovery):***

* The client is responsible for determining the network locations of available service instances and load balancing requests across them.
* The client queries a service registry, which is a database of available service instances.
* The client uses a load-balancing algorithm to select one of the available service instances and makes a request.
* **Netflix OSS** provides a great example of the client-side discovery pattern.
* **Netflix Eureka** is a service registry. It provides a REST API for managing service-instance registration and for querying available instances.
* **Netflix Ribbon** is an IPC client that works with Eureka to load balance requests across the available service instances.

***How to Discover Service (Server-Side Discovery):***

* The client makes a request to a service via a load balancer.
* The load balancer queries the service registry and routes each request to an available service instance.
* As with client-side discovery, service instances are registered and deregistered with the service registry.

***Service Registry:***

* A database containing the network locations of service instances.
* A service registry consists of a cluster of servers that use a replication protocol to maintain consistency.
* A service registry needs to be highly available and up to date. Clients can cache network locations obtained from the service registry.

***Service Registration Options:***

* Service instances must be registered with the service registry on startup and unregistered on shutdown
* Service instances that crash must be unregistered from the service registry
* Service instances that are running but incapable of handling requests must be unregistered

***Self-Registration:***

* A service instance is responsible for registering and deregistering itself with the service registry
* Service instance sends heartbeat requests to prevent its registration from expiring.
* It is relatively simple and doesn’t require any other system components.
* implement the registration code in each programming language and framework used by your services.
* Example: Netflix OSS Eureka client.

***Third-Party Registration:***

* Another system component known as the service registrar handles the registration.
* The service registrar tracks changes to the set of running instances by either polling the deployment environment or subscribing to events.
* When it notices a newly available service instance it registers the instance with the service registry.
* The service registrar also deregisters terminated service instances.
* Services are decoupled from the service registry.
* Unless it’s built into the deployment environment, it is yet another highly available system component that you need to set up and manage.

***Software Design Patterns:***

* A pattern is a recurring solution to a recurring problem.
* Design patterns in software engineering refer to reusable solutions for commonly occurring problems in software design and development.
* Design patterns are used to represent the solutions used by developers
* The design pattern describes the solution to the problem and the consequences of using the pattern.
* It is important that the design pattern be abstract enough to be reusable and not limited to a specific programming language.

**Structure:** Pattern Name, Intent, Applicability, Participants and consequences

**Benefits:**

1. **Patterns** provide the developer with a selection of tried and tested solutions for the specified problems.
2. All design patterns are language **neutral**.
3. Patterns help to achieve communication and maintain good documentation.
4. It includes a record of accomplishment to reduce any technical risk to the project.
5. Design patterns are highly flexible to use and easy to understand.

***Design Patterns vs. Architectural Patterns:***

* **Architectural Patterns (styles):** describe a solution for a software architecture requirement(s).
* **An Architectural Style** is the application design at the highest level of abstraction
* **An Architectural Pattern** is a way to implement an Architectural Style that focuses on the application design at the highest level of abstraction
* **Design Patterns** differ from Architectural Patterns in their scope, they are more localized, they have less impact on the code base (the general structure of the software system)
* **Design patterns** impact a specific section of the code base.

***Classifying Design Patterns:***

* **Creational**: concerned with the way of creating and initializing objects
* **Structural**: concerned with how classes and objects can be composed to/from larger structures
* **Behavioural**: concerned with the interaction and communication between objects and object responsibility

***Creational Patterns:***

* Useful when a design decision must be made during the object creation process.
* We use creational patterns when a simple object creation statement is not enough to satisfy our system requirements.

1. Factory Pattern
2. Singleton Pattern
3. Prototype Pattern
4. Builder Pattern
5. Object Pool Pattern
6. Abstract Factory Pattern

**Factory Pattern:**

* Define an interface or abstract class for creating an object but let the subclasses decide class to instantiate.
* The subclasses are responsible for creating the instance of the class.
* Allows the sub-classes to choose the type of objects to create.
* Help the developer to create a loose coupling code.
* We use Factory Pattern when:
  + Class doesn't know what sub-classes will be required to create
  + Class wants that its sub-classes specify the objects to be created.
  + The parent classes choose the creation of objects to its sub-classes.

**Singleton Pattern:**

* Only a single instance is reused again and again.
* Ensure that only a single instance of a given class X should be created, and all other classes interact with a single object of class X.
* Singleton pattern is mostly used in multi-threaded and database applications.
* It is used in logging, caching, thread pools, configuration settings, etc.
* **Early Instantiation:** the creation of instance at load time.
* **Lazy Instantiation:** the creation of instance when required.

**Creating a Singleton Pattern:**

1. **Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.
2. **Private constructor**: It will prevent to instantiate the Singleton class from outside the class.
3. **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

**Prototype Design Pattern:**

* Prototype Pattern assume that cloning of an existing object instead of creating a new one is more desired and can also be customized as per the requirement.
* This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.
* This pattern helps reduce the cost of creating a new object.

***Structural Patterns:***

* Focus on the classes structure and how classes are grouped to form a bigger structure.
* Helps you to decide how the classes inherit from each other and how they are composed of different classes.
* Simplify and reduce the relations between classes to **increase cohesion** and **decrease coupling**

Adapter Pattern, Bridge Pattern, Composite Pattern, Decorator Pattern, Facade Pattern, Flyweight Pattern, Proxy Pattern

**Adapter Pattern:**

* Converts the interface of a class into another interface that a client wants
* The adapter patterns enable classes with incompatible interfaces to work together
* The class that joins the unrelated interfaces is called an Adapter.
* It allows two or more previously incompatible objects to interact.
* It allows reusability of existing functionality. To implement:

1. **Target Interface**: This is the desired interface class which will be used by the clients.
2. **Adapter class:** This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.
3. **Adaptee class:** This is the class which is used by the Adapter class to reuse the existing functionality and modifies them for the desired use.
4. **Client**

**Decorator Pattern:**

* Attach flexible additional responsibilities (functionalities) to an object dynamically.
* Uses composition instead of inheritance to extend the functionality of an object at runtime.
* Decorators provide a flexible alternative to subclassing for extending functionality.

***Behavioral Design Patterns:***

* Concerned with the interaction and communication between classes (objects).
* Describe common communication methods between a group of objects in your application.
* Mainly focus on reducing coupling between objects

Chain of Responsibility Pattern, Command Pattern, Interpreter Pattern, Iterator Pattern, Mediator Pattern, Memento Pattern, Observer Pattern, State Pattern, Strategy Pattern, Template Pattern, Visitor Pattern, Null Object

**Chain of Responsibility:**

* In this pattern, an object (sender) sends a message that can be handled by a group of different objects (instances of different classes).
* Chain the receiving objects and pass the request along with the chain until an object handles it.
* Use to construct a pipeline of objects to process a message sent by one object

**Command Pattern:**

* Encapsulate a request under an object as a command and pass it to invoker object
* Issue requests to objects without knowing anything about the operation being requested or the receiver of the request.
* It separates the object that invokes the operation from the object that operates. Key elements:

1. **Command:** This is an interface for executing an operation.
2. **Concrete Command:** This class extends the Command interface and implements the execute method. This class creates a binding between the action and the receiver.
3. **Client:** This class creates the Concrete Command class and associates it with the receiver.
4. **Invoker:** This class asks the command to carry out the request.
5. **Receiver:** This class knows to perform the operation.

***Software Testing and Evaluation:***

* Testing is the process of uncovering evidence of flaws and fixing these flaws in software systems.
* Flaws may result from various reasons such as mistakes, misunderstandings, and omissions occurring during any phase of software development.
* Testing allows mitigating software risks. No single test can cover all errors. Use combos
* Testing remains one of the most costly and challenging aspects of the software development process.

**Defect testing:** guided by the objective of uncovering latent defects in the program, before delivering it; involves exercising the program in order to trigger some incorrect behavior, exposing some defect.

**Validation testing:** establish that the system behaves according to its specification; requires testing the system or correct behavior by exercising some acceptance testing criteria.

***Common Testing Objectives:***

1. **Find defects;** Maximize bug count
2. **Block premature product releases;** Help managers make ship / no-ship decisions
3. **Assess/Assure quality;** Conform to regulations
4. **Minimize technical support costs;** Minimize safety-related lawsuit risk
5. Assess **conformance** to specification/ Verify **correctness** of the product
6. Find safe scenarios for use of the product (find ways to get it to work, despite the bugs)

***Testing: White or Black:***

* **Black-Box Testing:** Consists of checking the system without having or using any knowledge of the internal logic in developing the test cases. Mostly accomplished using validation techniques. Checks that the software system is built according to the requirements.
* **White-Box Testing:** Use knowledge of the internal logic to develop test cases. Checks the structure of the software artifacts (e.g., code, design etc.). Checking that the system works according to the organization's standards and processes. (code review, code inspection, design review)
* **Functional testing:** unit, integration, system, sanity, smoke, interface, regression, beat/acceptance
* **Non-Functional testing:** performance, load, stress, volume, security, compactivity, install, recovery, reliability, usability, compliance, localization

***Functional Testing:***

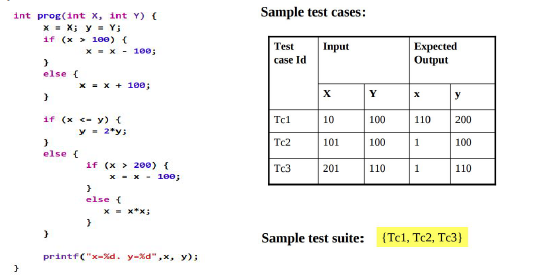
* **Unit Testing:** individual software component or module is termed as Unit Testing. It is typically done by the programmer and not by testers, as it requires a detailed knowledge of the internal program design and code.
* **Integration:** Testing the interactions between a collection of related functions or components. Typically done by developer or tester (today DevOps engineers)
* **System:** the whole system after all the components or subsystems are combined into the final product. It is a Black-box type testing that is based on overall requirement specifications and covers all the combined parts of a system.
* **Sanity:** Is done to determine if a new software version is performing well enough to accept it for a major testing effort or not. If an application is crashing for the initial use, then the system is not stable enough for further testing.
* **Smoke:** The software testing team validates the build and ensures that no major issue exists. The testing team ensures that build is stable and a detailed level of testing is carried out further.
* **Interface:** Testing the communication between the software system and other external systems that our system interact with. Verifies whether the communications between two different software systems are done correctly.
* **Regression:** Is the process of testing changes occurred in the system (new version) to make sure that the system still works with the new changes. Testing an application for the modification in any module or functionality is termed as Regression Testing.
* **Beta/UAT:** A formal type of software testing which is carried out by the customer. It is performed in the real Environment before releasing the product to the market for the actual end users.

***Non-Functional:***

* **Performance:** This term is often used interchangeably with ‘stress’ and ‘load’ testing. Performance testing is done to check whether the system meets the performance requirements. Different performance and load tools are used to do this testing.
* **Load:** To check how much of load or maximum workload a system can handle without any performance degradation. Load Testing Tools such as JMeter (Java) or Locust (Python)
* **Stress:** This testing is done when a system is stressed beyond its specifications to check how and when it fails. Crucial for mission critical systems.
* **Usability:** Test how user friendly the system is. Covers help documents, error messages.
* **Security:** performed by special teams (red/blue). Check if the system can be broken into, reduces system availability, reduces system trustability, WB/BB, expensive and ignored.

***Test Concepts and Tasks***

* **Test case:** specification of inputs and outputs required to reveal defects. Consists of the pretest state and environment, the test inputs and expected outputs and state Expected outputs may include data, messages or exceptions generated by the application under testing.
* **Test oracle:** a technique or mechanism used to produce expected results under specific conditions. It helps to determine whether software executed correctly for a test case.
* **Test strategy:** an algorithm and/or a collection of heuristics used to identify interesting test cases for an application under testing.
* **Test suite**: a collection of test cases, typically related by a testing goal or implementation dependency.
* **Test run:** the execution with actual results of a test suite(s). Actual results are compared against expected one, in order to decide the outcome: pass or no pass.



* **Test driver:** a class or utility program that applies test cases to an implementation under testing
* **Stub:** a partial, temporary implementation of a component, that may serve as a placeholder for an incomplete component or implement testing support code.
* **Test script:** a program written in a procedural script language (usually interpreted) that executes a test suite(s).
* **Test harness:** a system of test drivers and other tools supporting test execution.

***Key Tasks in Software Testing:***

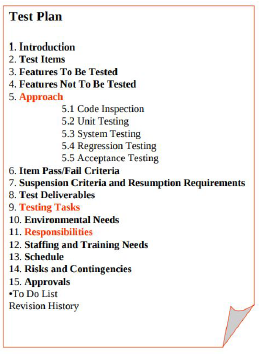
1. **Design test cases:** define and generate test cases using selected test strategies.
2. **Prepare test data**: instantiate the test cases by generating test points.
3. **Execute test cases/suites:** applies the test points to the IUT.
4. **Compare results:** compare the actual outputs with expected ones, and report corresponding test results.

***Lifecycle Testing:***

* Traditionally testing occurs at the latter phases of the software development cycle.
* However, restricting testing to a single phase creates the potential that errors with significant and costly consequences may occur.
* **50%** of the defects could be detected when testing is conducted before the implementation phase.
* **80%** could be detected when testing is conducted after implementation.
* It is at least 10 times more expensive to fix a defect after implementation than before, and 100 times more expensive during maintenance.
* Test as early as possible and **integrate testing in the entire development lifecycle**

1. **Requirements Testing:** ensure that the requirements are recorded or documented properly, and address user and business concerns.
2. **Design Testing:** ensure that the design is complete, accurate and matches with the requirements.
3. **Implementation Testing:** ensure that the design specification has been correctly implemented.
4. **Maintenance Testing:** involve deployment testing at the installation, and regression testing during operation

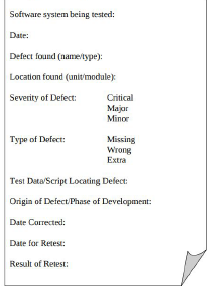
***Test Planning:***

* Test planning starts at the beginning of the development process; it should represent the first task in any testing process.
* The test plan defines the road map to be followed during the testing process.
* Provide background information on the system under testing,
* Specify the test objectives and Identify and describe the test and risks factors involved.
* Provide a description of the components and functions to be tested, and the nature of the tests to be conducted, Identify the potential members of the test team and their responsibilities.

***Common Tasks in Test Planning:***

1. **Identify risks factors**, which will guide the testing efforts
2. Identify test phases.
3. Map risks factors to test phases
4. Specify for each test phase, the tasks to be done and the workers assigned
5. Identify **test resources,** environments, and tools.
6. Determine preliminary test schedule and estimate initial test costs

***Test Roles and Responsibilities:***

* **Test Manager (QA Manager**): is tasked with the overall responsibility for the test effort's success. He is the the primary person in charge of advocating and assessing product quality.
* **Test Analyst** is responsible for initially identifying and defining the required tests, and subsequently evaluating the results of the test effort.
* **Test Designer** is responsible for defining the test approach and ensuring its successful implementation.
* **Tester** is responsible for the core activities of the test effort, which involves conducting the necessary tests and logging the outcomes of that testing.

***Defect Reporting:***

* A defect report is a tool that you use to convince someone to allocate time and energy to fix a bug
* **Change request:** any report of an incident, defect or potential enhancement, that is intended as a request for a change to the software under development
* **Defect report:** a change request reporting a (suspected) defect or error in the product
* **Bug:** some aspect of the product under test, that in the eyes of a stakeholder, unnecessarily reduces its value; possibly a suspected defect

***Test Design:***

***Functional and Systematic Testing:***

* Software tests are supposed to find defects (bugs, incomplete features, missing features).
* A good test is focused on finding defects, A successful test finds new/different classes of defects
* Designing and executing a good test is not a trivial task as it might appear.

***Good Testing:***

* **Test Planning and Analysis:** selecting the right test type, design enough and necessary test cases, create a reasonable and well-defined test plan.
* **Follow good testing practice:** start testing ASAP, design a test from a user perspective and not from a developer perspective, use independent testing staff not the people who designed or built the software.
* In Agile teams testers and developers are interchangeable!!!

***Fundamental Testing Types:***

1. **Unit Test:** for every new class, modules and components.
2. **Integration tests:** test the interaction between your system components
3. **System Test:** complete system testing at least one time or for major releases (e.g. migrated the system from on technology stack to another)
4. **Regression Tests:** for any changes of added feature
5. **Alpha Test:** internal test by the software house or company.
6. **Beta Test:** invite friends and family or a selected group of end users
7. **Acceptance Test:** on official delivery by end users

***Key Challenges in Software Testing:***

1. How do we decide that our **test is complete?**
2. What is good **enough testing?**
3. How do we measure the **quality of our testing?**
4. What is the **necessary testing?**
5. What is **sufficient testing?**

***Test Case Explosion:***

* A test case explosion is when the configuration settings and user actions and data entered etc. makes it impossible to test everything.
* The number of test cases required to test every single possibility individually is massively greater than what could really be tested.
* Test cases explosion is a practical example of a combinatorial explosion.
* Combinatorial is a fundamental problem in computing. It is a problem that the number of combinations that one has to examine grows exponentially.

***Test Cases / User Stories:***

* In general, every user story in our system could results in many test cases to test different scenarios.
* Test scenarios are sequences of test cases which represent a typical user story in the system
* Test scenarios should cover both successful scenarios and failure scenarios

***Good Number of Test Cases:***

* We need to have a precise balance between test quality and software delivery.
* We cannot keep testing forever; the software should be released eventually.
* A Good number of test cases will be sufficient to cover all known scenarios.
* A Good number of test cases will be sufficient to cover all key components, frequently occurred user stories.
* A Good number of test cases will be necessary to execute, no need to run duplicate or equal test cases.

***Test Cases Design Techniques and Discovery:***

1. **Domain Test Model:** This technique focuses on applying equivalence partition analysis and boundary value testing (black box testing)
2. **Code Coverage:** Uses the code execution percentage as a guide to create test case (white box testing)
3. **UML-Based Testing:** Uses UML models such as state diagrams and activity diagrams to generate/design test cases

**DTM (Creating Test Cases):**

* **Equivalence Partition Analysis:** Also known as equivalence classes analysis. It helps to select a reasonable number of test cases to run.
* **Boundary Value Testing:** helps to choose values for test cases.

***Equivalence Partitions Analysis:***

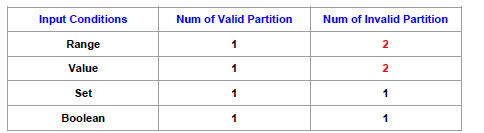
* Focus on reducing the number of test cases to a necessary minimum by defining equivalent partitions
* Only one test case of each class is needed to evaluate the behavior of the program for that partition/class of possible inputs
* To use more or even all test cases of one partition we will not find any new faults in the program.
* The values within one partition are considered to be "equivalent".
* To select the sufficient test cases (e.g valid and invalid) to cover all possible behavior
* **Pros:** find highest probability errors with small test sets, intuitively clear
* **Cons:** cannot discover errors that are not boundaries or obvious, set of possible values is unknown, selection of partition has often no necessary relationship to the discover of bugs **(quality of the test data).**

**How to ID EP’s:**

* Equivalence partitions are usually derived from the system specifications, user stories, and system behavior.
* You should consider both input equivalence partitions and output equivalence partitions
* An input has values which are valid and other values which are invalid, information about valid and invalid input should be used to identify the equivalence partitions.

**EP EX:**

* If you had defined a function which has to pass the parameter of type int to represent a "month" of a date. What are the valid and invalid values?
* The valid range for the month is 1 to 12, standing for January to December.
* This valid range is called a **valid partition/class.**
* The valid partition is 1 to 12, standing for January to December.
* There are two partitions of invalid ranges.
* The first invalid partition would be <= 0 and the second invalid partition would be >= 13.



***Equivalence Partition for a Set Input:***

* If each item in the set will be treated the same way, then identify one valid class for values in the set and one invalid class representing values outside the set.
* If each case will be treated differently, then identify one valid equivalence class for each element and one invalid equivalence class for values outside the set.

***Equivalence Partition for a Value/Range Input:***

* Apply boundary value analysis to selecting values for test data for each equivalence partition.
* Boundary value analysis means selecting values at the boundaries and somewhere in the middle of each partition
* If these data points produce correct results, it is safe to assume that other values will also be correctly processed

***Boundary Value Analysis:***

* Used to select values for each equivalence partition.
* Using the domain of all possible inputs and outputs
* Divide the domain into valid and invalid classes
* For each boundary we construct test cases with these inputs

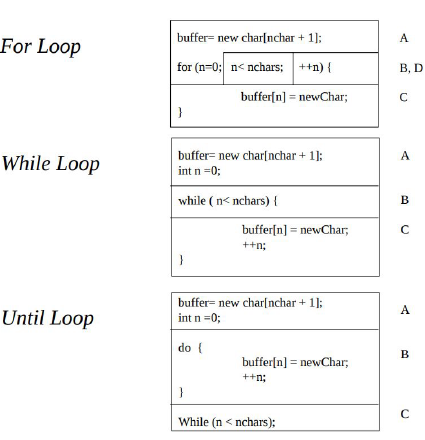
***Code Coverage:***

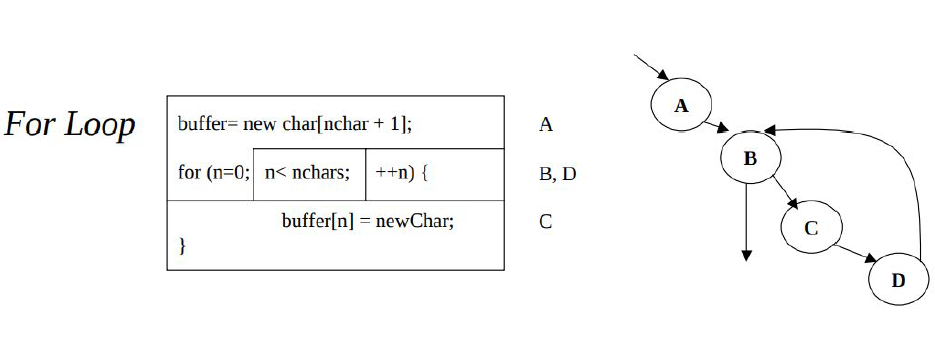
* Test coverage attempts to address questions about when to stop testing, or the amount of testing that is enough for a given program?
* Ideal testing is to explore exhaustively the entire test domain, which in general is impossible.
* In practice, some code may never be executed due to the possibility of missing test cases.
* We cannot understand how effective test suites are until one finds what code is or is not executed.

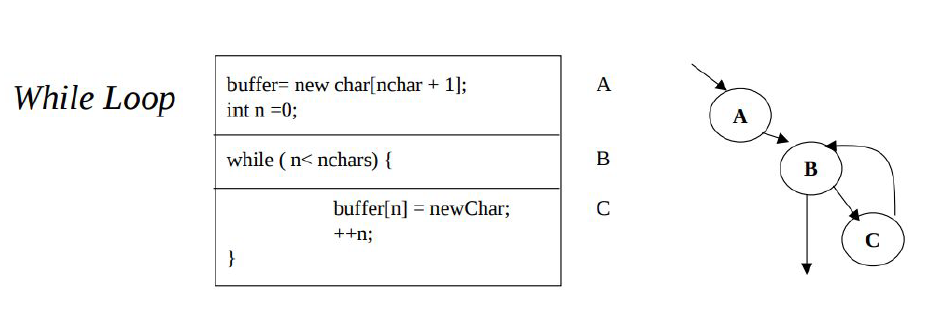
***Code Coverage Models:***

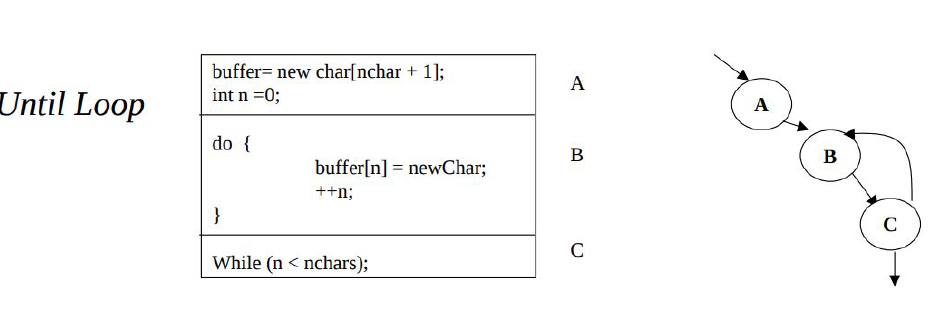
* A code coverage model calls out the parts of an implementation that must be exercised to satisfy an implementation-based test model.
* Coverage, as a metric, is the percentage of these parts exercised by a test suite.
* Hundreds of coverage models have been published and used since the late 1960s.
* Nearly all support implementation-based testing (Unit Testing).
* As such, most coverage models rely on control flow graphs, which give an abstract representation of the code.

***Control Flow Graph:***

* Control flow graph focus on analyzing code segments
* A code segment consists of one or several contiguous statements with no conditionally executed statements.
* That means once a segment is entered, all the statements involved will execute.
* The last statement in the segment must be another predicate, a method exit, a loop control, a break, or a goto.
* The last part of a segment includes the predicate or exit expression that selects another segment but does not include any of subsequent segment’s code
* A control flow graph (CFG) is a directed graph describes code segments and their sequencing in a program.
* A node corresponds to a code segment; nodes are labeled using letters or numbers.
* An edge corresponds to a conditional transfer of control between code segments; edges are represented as arrows.
* The entry point of a method is represented by the entry node which is a node with no inbound edges.
* The exit point of a method is represented by the exit node, which is a node with no outbound edges.
* Split anywhere there is a decision or path

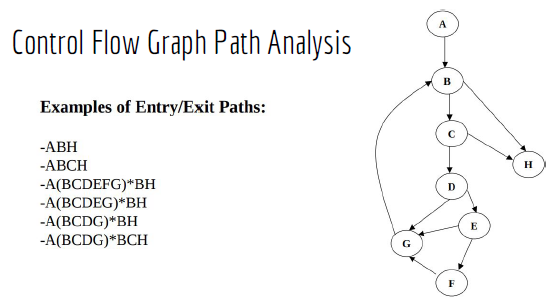






***Predicate:***

* A predicate expression contains one or many conditions that evaluate to true or false. One condition corresponds to each boolean operator in the predicate expression.
* Predicates are used in control statements: if, switch-case, do-while, while, do until, for, and so on.
* The evaluation of a predicate results in transfer of control to one or many code segments.
* A predicate with multiple conditions is called a compound predicate.

***Control Flow Graph Path Analysis:***

* A path corresponds to a sequence of segments connected by arrows.
* A path is denoted by the nodes that comprise the path.
* Loops are represented by segments within parentheses, followed by an asterisk to show that this group may iterate from zero to n times.
* An entry-exit path is a path starting with the entry node and ending with the exit node.
* Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths.

***Compound Predicates:***

* Compound predicate expressions are modeled separately by specifying a node for each individual predicate involved.
* All true-false combinations in a compound predicate should be analyzed, and the effects of short circuit boolean evaluation should be made explicit.
* Case and multiple-if statements are modeled by specifying a separate node for each predicate, each conditional action, and the default action.
* Switch statements are modeled by specifying a node for the switch expression, and a separate node for each action.

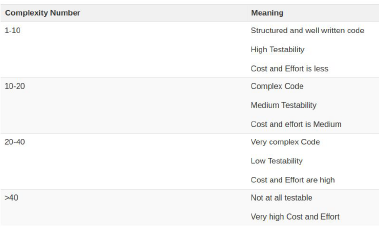
***Test Coverage Analysis:***

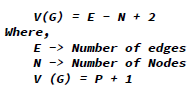
* Test coverage analysis uses some adequacy criteria to guide the testing process.
* This increases the confidence that an implementation has been thoroughly tested
* **It is recommended not** to use a code coverage model as a test model.
* Instead, established test strategies (e.g. equivalence, domain) should be used to devise test suites, while coverage are used at the same time to analyze generated test suites adequacy.
* Coverage reports can point out a grossly inadequate test suite
* Coverage reports can help to identify implementation constructs that may require implementation-based test design or the development of special stubs and drivers
* Common coverage criteria: statement, branch, condition/MC, basis-path, data flow

**Statement coverage:** achieved when all statements in a method have been executed at least once.

**Branch coverage:** Achieved when every path from a node is executed at least once by a test suite.

**Condition/Multiple Condition coverage:** requires that each condition be evaluated as true and false at least once.

***Cyclomatic Complexity:***

* Cyclomatic complexity is a software metric used to measure the complexity of a program.
* This metric measures independent paths through program source code.

***Test Coverage in Practice:***

* In practice, the three basic criteria most commonly used are statement coverage, branch coverage and condition coverage.
* It has been suggested that the combination of these three criteria can achieve **80-90 %** or more coverage in most cases.
* It is important to note that test coverage is not enough by itself
* 100% **test coverage cannot guarantee achieving error-free software.**
* However, the use of test coverage in combination with appropriate test strategies can mitigate the impact of uncovered code during software testing, and help the tester find some rational points at which to stop testing.

***Software Reliability:***

* **Reliability** is a measure of how closely a system matches its stated **specification**.
* **Reliability** is a measure of how well the users feel a system provides the required services.
* **Reliability** is the probability of **failure-free** operation of a system over a specified time within a **specified environmen**t for a specified purpose.
* In general study software reliability is difficult. However, it is important topic in developing dependable software systems.

***Dependable Software System:***

* Critical systems such as spacecraft, aircraft, nuclear power plant and pacemakers require a high level of dependability in their operation.
* Two different categories of techniques are used in the design and implementation of dependable software systems: fault avoidance and fault tolerance techniques.
* **Fault avoidance:** primary goal of any sound software engineering process.
* **Fault tolerance**: address the shortcoming of fault avoidance by mitigating the risk that there are some potential or hidden faults remaining in the software.

***Software Reliability:***

* Reliability is a popular aspect of software dependability, which relies on fault forecasting and fault removal.
* Fault forecasting consists of estimating the presence of faults and the occurrence and consequences of failures.
* Fault removal uses techniques such as testing or inspection to track and remove faults in software.
* It is difficult to define the term objectively.
* Difficult to measure user expectations
* Difficult to measure environmental factors.
* It’s not enough to consider simple failure rate: not all failures are created equal, some have more serious issues, might be able to recover from some failures reasonably

***Software Reliability Concepts:***

* Software reliability is the probability that the software system will function properly without failure over a certain time period.
* Reliability is one of the most important software quality attributes.
* It is an external quality attribute, which relates internally to the notion of program faults or defects.
* A failure corresponds to unexpected runtime behavior observed by a user of the software.
* A fault is a static software characteristic which causes a failure to occur.
* Not every fault causes a failure: code that is “mostly” correct, dead/infrequently-used code, faults that depend on a set of circumstances to occur

***Software Reliability vs. Hardware Reliability:***

* Many of the concepts and models used in software reliability are derived from hardware reliability, which is an established field.
* Hardware reliability metrics are not always appropriate to measure software reliability but that is how they have evolved.
* Hardware reliability tends to be stable or constant over time, **software reliability has tendency to change during test periods;** this phenomenon is referred to as r**eliability growth.**
* The sources of failures are different**. Hardware faults arise mostly from wear and physical deterioration,** while software faults arise mostly from design issues.

***Improving Reliability:***

* The primary objective of reliability is to remove faults with the most serious consequences.
* The secondary objective of reliability is to remove faults that are encountered most often by users.
* 90-10 Rule: 90% of the time you are executing 10% of the code.
* One study showed that removing 60% of software “defects” led to a 3% reliability improvement.

***Measuring Reliability:***

* Reliability models express the probability of failure over a certain execution exposure variable or metric for the system.
* Examples of exposure metrics include time (of execution), number of executed test cases or runs, number of transactions etc.
* Time is the most common exposure metric used. Three kinds of times variables are commonly used: calendar time, clock time, and (CPU) execution time.
* **Execution time** corresponds to the effective time used by the processor to execute the program instructions.
* **Calendar time** is the regular time we use in our daily business; as such it is important for users and managers.
* **Clock time** corresponds to the elapsed time between the start and end of program execution.
* Clock time and execution time are equivalent when computer utilization is constant.

***Failure Behaviour:***

* Failure behavior directly depends on the environment and the number of faults present in the program during execution.
* Failure occurrences are expressed as random variables.
* Using time variable, failure occurrences may be expressed in four different ways:
  1. Time of failure
  2. Time interval between failures
  3. Cumulative failures occurred up to a specified time
  4. Failures occurred in a specified time interval.

***Reliability Metrics:***

* Mean value function: expresses the average cumulative failures at each point in time
* Failure intensity function: the number of failures/time unit; its computed as the derivative mean value function with respect to time

**Probability of Failure on Demand (POFOD):**

* Likelihood that system will fail when a request is made
* E.g., POFOD of 0.001 means that 1 in 1000 requests may result in failure.
* Any failure is important; doesn’t matter how many if > 0
* Relevant for safety-critical systems

**Rate of Occurrence of Failure (ROCOF):**

* Frequency of occurrence of failures
* E.g., ROCOF of 0.02 means 2 failures are likely in each 100 time units.
* Relevant for transaction processing systems.

**Mean Time to Failure (MTTF):**

* Measure of time between failures.
* Is the average time between consecutive system failures.
* E.g., MTTF of 500 means an average of 500 time units passes between failures.
* Relevant for systems with long transactions.

***Reliability Metrics:***

**Availability:**

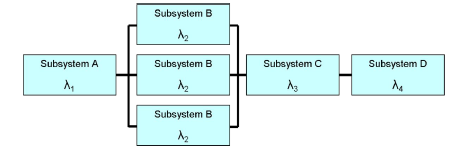
* Is the likelihood that the system will be working at a given time.
* Measure of how likely a system is available for use, taking in to account repairs and other down-time.
* E.g., Availability of 0.998 means that system is available 998 out of1000 time units.
* Relevant for continuously running systems.
* A = uptime / uptime + downtime
* Availability described in nines notation 3-nines means 99.9% (5 nines ambitious)

**Calculating availability:**

Availability is characterized by defining some basic concepts that describe quantitatively the operational state of the system. These include:

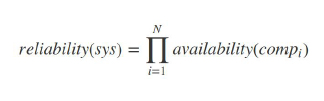
* **MTTF (Mean Time To Failure):** average time it takes for a system to fail
* **MTTR (Mean Time To Recover):** average time for the system to recover; correspond to the average time to repair the system.
* **MTBF (Mean Time Between Failure):** average time between consecutive system failures. MTBF = MTTF + MTTR

***Reliability Block Diagram***

* Reliability Block Diagram (RBD) is a graphical representation of how the components of a system are connected from reliability standpoint.
* The most common configurations of an RBD are the series and parallel configurations.
* In a **serial configuration**, all the elements must work for the system to work, the system fails if one of the components fails.
* In **parallel configuration,** the components are redundant and the system will still cease to work if all the parallel components fail.
* A system is usually composed of combinations of serial and parallel configurations.
* RBD analysis is used in practice to determine reliability, availability and down time of the system.

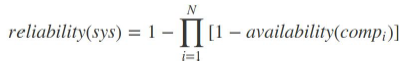
***Serial Structure:***

In this configuration, as the system will be considered working as far as all the components are working, the total reliability would be the multiplication of each component’s “independent” availability



A system is composed of 4 independent serially connected components A1 = 0.95 A2 = 0.87 A3 = 0.82 A4 = 0.73 **R(sys) = 0.95 × 0.87 × 0.82 × 0.73 = 0.4947**

***Parallel Structure:***

Sometimes, system designers would put identical (or even similar) components together, in a way that as far as one of the components are available, the system can survive. Component in this configuration are said to be made redundant

Example: A system is composed of 4 independent components connected in parallel A1 = 0.95 A2 = 0.87 A3 = 0.82 A4 = 0.73 **R(sys) = 1 – ((1 – 0.95) × (1 – 0.87) × (1 – 0.82) × (1 – 0.73)) = 0.9996**

***Hybrid Structure:***

Hybrid Configuration happens when the system consists of multiple components, from those some are serial and the others are parallel. In order to calculate the availability of such a system, one may calculate any consecutive serial/parallel components and replace them with blocks with new availability in order to be able to complete the calculation.

**What about partial operation or availability?**

* Consider a system with N components where the system is considered to be available when at least N-M components are available (i.e. no more than M components can fail)

***Failure Intensity and Density:***

* **Failure Intensity (Failure Rate):** the rate failures are happening, i.e., number of failures per natural or time unit.
* Failure Intensity is way of expressing system reliability. Example: 5 failures per hours, 2 failures per day, 10 failures per 10000 requests.
* Failure Intensity is suitable to express system reliability from the **end-user's perspective.**
* Failure Density is the rate of failure per software component or module
* Failure Density is better expressed as failures per LOC/ KLOC or any development code measuring unit (e.g Class).
* Failure Density is suitable to express the reliability of the system from the software developer’s perspective.

***Fault Tolerance:***

* A system is said to fail when it cannot meet its promises.
* A failure is brought about by the existence of errors in the system.
* The cause of a failure is called a fault.
* In general, fault avoidance, fault detection and fault tolerance are important concepts in developing dependable software systems.

***Dependable System:***

* **Availability:** A measurement of whether a system is ready to be used immediately. The system is up and running at any given moment
* **Reliability**: A measurement of whether a system can run continuously without failure. The system continues to function for a long period of time
* **Safety:** A measurement of how safe failures are system fails, nothing serious happens. For instance, a high degree of safety is required for systems controlling nuclear power plants
* **Maintainability**: A measurement of how easy it is to repair a system: A highly maintainable system may also show a high degree of availability.

***Types of Faults:***

* **Transient Fault:** appears once, then disappears
* **Intermittent Fault:** occurs, vanishes, reappears; but: follows no real pattern (the worst kind).
* **Permanent Fault**: once it occurs, only the replacement/repair of a faulty component will allow the system to recover to function normally.

***Types of Failures:***

1. **Crash Failure:** The system crashes (halt), but is working correctly until it crashes.
2. **Omission Failure:** The system fails to respond to incoming requests. This could be receive omission or send omission.
3. **Timing Failure:** The system response lies outside the specified time interval. (common in real-time applications)
4. **Response Failure:** The system response is incorrect either incorrect value or incorrect system state.
5. **Arbitrary Failure:** The system gives unpredictable response at arbitrary times.

***Fault Avoidance:***

* The basic idea is that if you are REALLY careful as you develop the software system, no faults will creep in. Simply avoid introducing bugs into the code.
* Use of information-hiding, strong typing, good engineering principles.
* Use of languages and tools that give good support to these ideas!
* Features that are likely not to be portable or robust, or make system evolution difficult.
* No cheats, hacks, or other risky behavior
* Uses formal specifications, logic, and related tools for formal verification.
* Avoid introducing faults in the first place by operating in a dedicated environment.
* The cost of this approach can be very high!
* Developers must be experienced and highly trained, not only in traditional software development techniques, but also in mathematics, logic, and special tools.
* Requires highly-trained and patient developers.
* Usually, the cost is so prohibitive that it is just not feasible.

***Fault Tolerance:***

* It is not enough for **reliable systems** to avoid faults, they must be able to tolerate faults.
* Faults occur for many reasons: Incorrect requirements, Incorrect implementation of requirements, Unforeseen situations.
* Roughly speaking, fault tolerance means able to continue operation despite software failure.

***Steps to Fault Tolerance:***

1. Detection of failure
2. Damage assessment: what has been infected, built-in redundancy to check for validity, values within legal range, invariants preserved?
3. Fault recovery/restore
4. Fault repair

***N-Version Programming (NVP):***

* From the specifications, 2 or more (commonly 3) versions of the program are independently developed, each by a group that doesn’t interact with the others.
* The implementations of these functionally equivalent programs use different algorithms and programming languages.
* The Airbus-320 uses NVP in its on-board software. Using three identical components that vote using majority rules.

***Forward Fault Recovery:***

* Try to fix what’s broken based on understanding of program and then carry on.
* E.g If you can figure out which value is obviously wrong, you may be able to derive its real value from the rest of the state.
* Sometimes, domain understanding is enough to help figure out what “real” values should be.
* Often requires redundancy in representation.

***Backward Fault Recovery:***

Restore state to a known safe state, for example restore file system from tape backup or database transaction from snapshot and checkpoint.

Some systems are transaction oriented:

* Changes are not committed until computation is complete.
* If an error is detected, changes are not applied.
* Requires period “snapshots” to be taken and maintained.
* Simple approach but may entail loss of data.

***On Failure Retry Pattern:***

* Enable an application to handle anticipated, temporary failures when it attempts to connect to a service or network resource by transparently retrying an operation that has previously failed in the expectation that the cause of the failure is transient.
* For examples, temporary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that arise when a service is busy.
* These faults are typically self-correcting, and if the action that triggered a fault is repeated after a suitable delay it is likely to be successful.

**Issues:**

* If the fault indicates that the failure is **not transient** or is unlikely to be successful if repeated (for example, an authentication failure caused by providing invalid credentials is unlikely to succeed no matter how many times it is attempted), **the application should abort the operation** and report a suitable exception.
* If the specific fault reported is unusual or rare, it may have been caused by freak circumstances such as a **network packet becoming corrupted** while it was being transmitted. In this case, the application could retry the failing request again immediately because the same failure is unlikely to be repeated and the request will probably be successful.
* If the fault is caused by one of the more **commonplace connectivity or busy failures**, the network or service may require a short period while the connectivity issues are rectified, or the backlog of work is cleared. The application **should wait for a suitable time** before retrying the request.
* The retry policy should be tuned to match the business requirements **of the application and the nature** of the failure.
* Ensure that all retry **code is fully tested against** a variety of failure conditions.
* **Implement retry logic only where** the full **context of a failing operation** is understood.

***Circuit Breaker Pattern:***

* **The Circuit Breaker** pattern can prevent an application trying to execute an operation that is **likely to fail.**
* This pattern can improve the stability and resiliency of an application.
* The Circuit Breaker pattern also enables an application to detect whether the fault has been resolved.
* The purpose of the Circuit Breaker pattern is different from that of the Retry Pattern.
* The Retry Pattern enables an application to retry an operation in the expectation that it will succeed.
* The Circuit Breaker pattern prevents an application from performing an operation that is likely to fail.
* **Combine** these two patterns by using the Retry pattern to invoke an operation through a circuit breaker.
* A circuit breaker acts as a **proxy (implemented as a state machine)** for operations that may fail.
* The proxy should monitor the number of recent failures that have occurred, and then use this information to decide whether to allow the operation to proceed, or simply return an exception immediately.

**Closed State:**

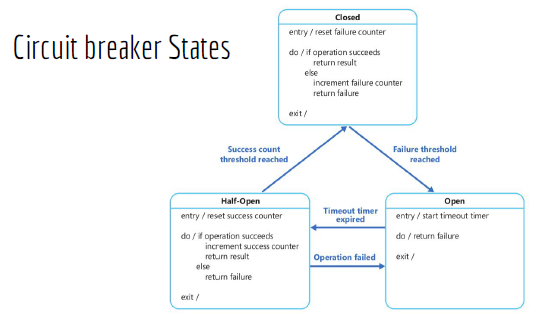
* The request from the application is routed through to the operation.
* The proxy maintains a count of the number of recent failures, and if the call to the operation is unsuccessful the proxy increments this count.
* If the number of recent failures exceeds a specified threshold within a given period, Open state.
* At this point, the proxy starts a timeout timer, and when this timer expires, placed into the **Half-Open state**

**Open State:**

* The request from the application fails immediately and an exception is returned to the application.
* Start a time out timer and when the timer reach zero, change the state into half-open.

**Half Open:**

* A limited number of requests from the application are allowed to pass through and invoke the operation.
* If these requests are successful, it is assumed that the fault that was previously causing the failure has been fixed and the circuit breaker switches to the Closed state (the failure counter is reset).
* If any request fails, the circuit breaker assumes that the fault is still present so it reverts back to the Open state and restarts the timeout timer to give the system a further period of time to recover from the failure

**Issues:**

* The failure counter used by the Closed state is time-based. It is automatically reset at periodic intervals. This helps to prevent the circuit breaker from entering the Open state if it experiences occasional failures.
* The failure threshold that trips the circuit breaker into the Open state is only reached when a specified number of failures have occurred during a specified interval.
* The success counter used by the Half-Open state records the number of successful attempts to invoke the operation.
* The circuit breaker reverts to the Closed state after a specified number of consecutive operation invocations have been successful