Java Delivery Playbook Document

Table of Contents

1. Purpose of this document

This playbook is a live document that evolves as the team continues to embark of new projects and learn through its implementation. This provides necessary guidance and reference materials to understand the expectations and deliver the software solution to the customers.

In scope:

Describe the approach to overall delivery of a typical Java project for a subset of the organization(wave) that is embarking upon larger transformation program.

Provide guidelines/guardrails on activities to be performed as part of overall Delivery.

- Possible approach to various phase of implementation
- Guidelines and roadmaps to be followed for a selected SDLC cycle.
- Phase planning, tool sets to be used.
- Methodology, Best Practices, Case studies, etc.

2. Delivery Methodology - Selection of Project Life Cycle:

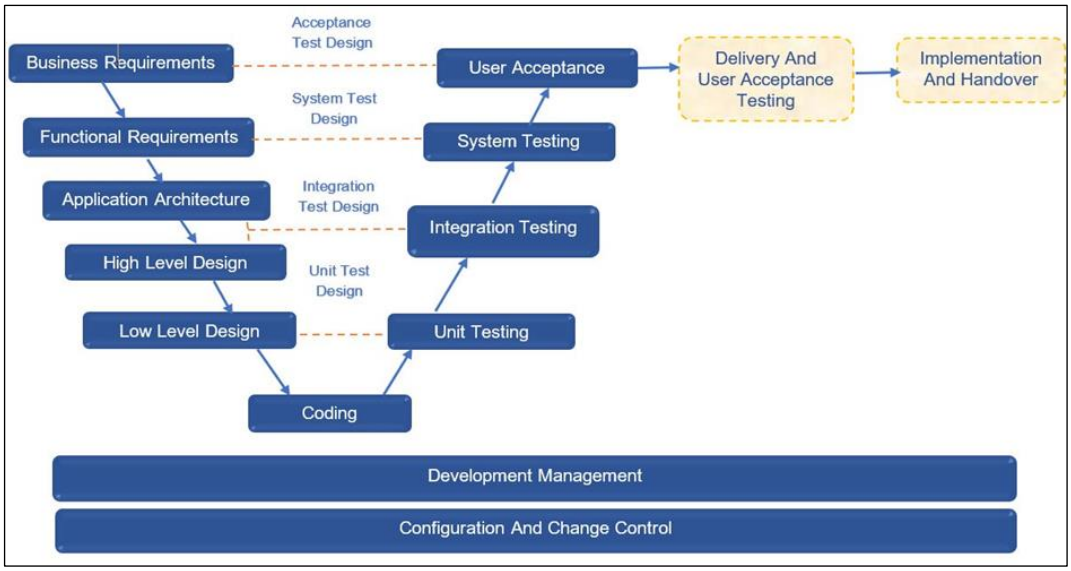
This section provides guidance on various methodologies applicable for Waterfall, Agile ways of execution of the engagements. Methodology selection depends upon various factors such as the nature of engagement, Timelines for delivery, clarity of the requirements, etc.

2.1. Linear Development Lifecycle (LDM)

Overview:

The Linear Development Method is the Capgemini recommended lifecycle method for development engagements executed in waterfall/v-model. LDM is designed to support application development where there are clearly stated business requirements.
Linear Development Method is structured as a progressive, deliverables-driven life cycle that is divided into successive phases. In this method, both development team and test teams work in parallel and final product is produced at the end.

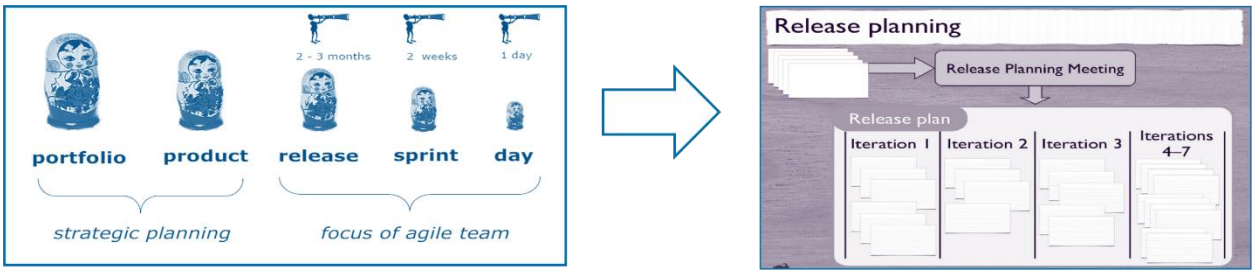
LDM Lifecycle phases:



2.2. Agile Delivery Lifecycle – Cap Gemini Agile Framework (CAF)

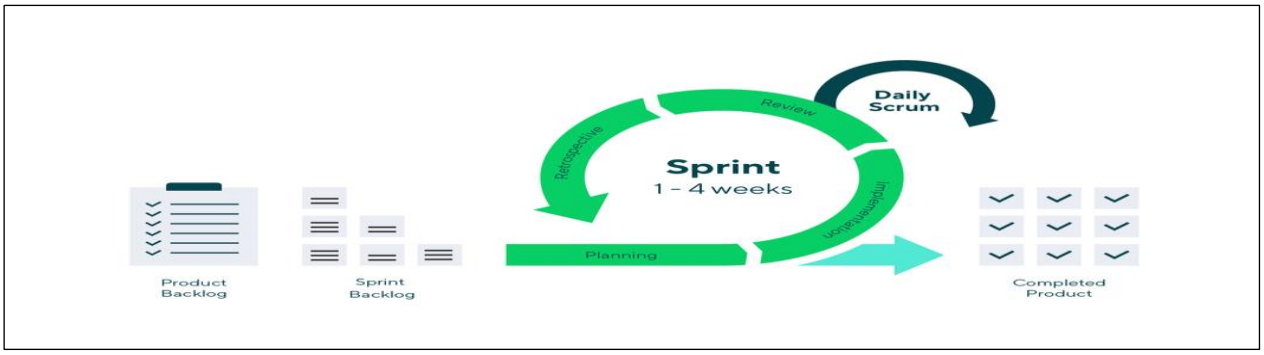
Overview:

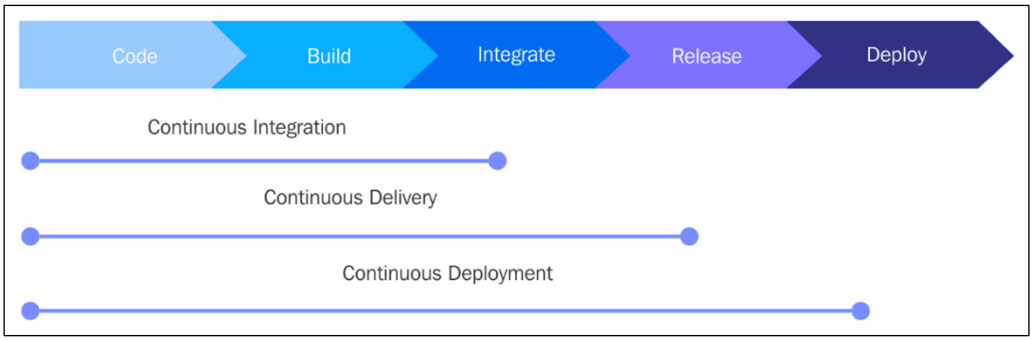
Agile planning covers three different levels—the release, the iteration, and the current day—each plan can be made with a different level of precision thus enabling a wholistic planning view



Refer - https://capgemini.sharepoint.com/sites/SSO10/SitePages/Agile-CAF-Lifecycle.aspx

Standard Lifecycle phases





Entry criteria, Task, Verification, Exit criteria (ETVX) Approach:

Phases	Entry Criteria	DoR	Tasks	Verification	Exit Criteria	DoD
Business Requirements	• Proposal • MSA, SOW or LOI	Signed off Contract/agreement	• Capture And Analyse Business Requirements • Provide High level estimates for budgeting. • Develop Test Strategy • Create Acceptance Test Plan and Test Cases	Review and sign off of Business Requirements, test strategy	• Baselined Business Requirement Document • Baselined Test Strategy • Traceability Matrix updated with Business Requirements and User Acceptance Testcases • User Acceptance Test Plan and Test Cases	Project Scope Definition
Functional/Non-Functional Requirements	• Business Requirements • NFRs • Traceability Matrix • Test Strategy	Baselined Business Requirement Document	• Capture And Analyze Functional Requirements • Create System Test Plan • Create Performance Test Plan • Create System Test Cases • Create Test Data	Review and sign off of Functional/Non-Functional Requirements, test plan and test cases	• Baselined FSD • Baseline NFR specs with defined Acceptance criteria • Baselined System Test Plan and Test Cases	Signed of Functional Spec Document
Solution Architecture	• Signed off FSD • Signed of NFRs	Current state Architecture	• Understand Architecture Requirements • Create Architecture document	Review and sign off of Architecture document	• Baselined Architecture Document • Traceability Matrix updated with Architecture details	Future State System Architecture
High Level Design	• Requirement Documents • Architecture Document • Test strategy, system test plan and testcases	System architecture	• Create High Level Design Document • Create Integration Test Plan • Create Integration Test Cases	Review and sign off of HLD document, IT test plan and test cases	• Baselined High Level Design Document • Baselined Integration Test Plan and Test Cases • Traceability Matrix updated with High Level Design and Integration Testcases	Database design Functional Design Interface design UI design
Low Level Design	• High Level Design Document	Architecture diagrams, Component diagrams, interface definitions,	• Create Low Level Design • Create Unit Test Cases	Review and sign off of LLD document and UT cases	• Baselined LLD Document • Baselined Unit Test Cases • data flow diagrams	Class diagram Sequence Diagram Interaction diagram

Development/Unit Testing	• High Level Design Document • Architecture Document • Requirement Specification Document	Low Level Design Document Approved Estimations	• Perform Coding by following test driven development and coding standards. • Establish CI/CD pipeline. • Run code quality tools in the IDE • Fix the code quality issues, Check-in the code daily into the config repository. • Perform Code Review by following pull approach and fix code review comments • Perform Unit Testing and fix the defects	• Code quality checks using tools • Manual Code reviews and signoff • Code coverage checks using tools	• Baselined Source Code • Traceability Matrix updated with Source Code details. • Code quality targets passed. • Manual Code reviews and signoff • Code coverage targets passed	Successful Build with agreed %code coverages Deployable created
Integration Testing	• Integration Testcases • Integrated Source Code	Deployed on SIT Test environment	Perform Integration Testing and fix the defects	Quality gate checks using tools integrated into CI/CD pipeline	Quality gates passed	Signed of SIT test report
System Testing	• System Testcases • Requirement Specifications • Build • System Test Environment	Deployed on Regression Test environment	• Set Up System Test Environment • Perform System Testing and fix the defects	• Test coverage • Test execution results • Release/product delivery audits	• System Test Results Passed • User documentation	Signed of Regression test report
User Acceptance Testing	• Acceptance Test Plan/Cases • Business Requirement Document • UAT Test Environment		Provide Support for Acceptance test by customer and fix the defects	UAT coverage Test execution results	• Acceptance Test Results Passed	UAT Signoff from Customer and Go-live approval
Implementation	• Release Package • Implementation Plan • Release Notes • UAT Signoff from Customer		• Prepare For Implementation • Deploy The code in production. • Conduct User Training • Cut over and go live	Review and Sign off on successful deployment	Successful code deployment in Production	Release notes
Warranty and Handover	• Training Materials • All deliverables from above steps		• Provide Warranty Support • Handover To Support	Review and sign off of Handover document	Handover to support team successfully completed	Project closure document

2.3. API & Microservices

Overview:

This section will provide a various steps and considerations for **Accelerated API & Microservices Development.**

https://capgemini.sharepoint.com/:w:/r/sites/CloudMicroservices/_layouts/15/Doc.aspx?sourcedoc=%7B8E1003B9-9615-4F21-B418-DDCB1194009A%7D&file=API%20%26%20MIcroservices%20-%20Work%20items-V2.docx&action=default&mobileredirect=true&wdsle=0

3. Estimation Model and Sizing:

Estimation is used as an input to:

- Pricing
- Planning/Scheduling
- Staffing

This section provides an overview on Sizing and reference to recommended standard Estimation models which are used by FS SBU projects.

3.1. Sizing:

1. Sizing an application is a crucial step for any estimation & measurement activity in the industrialization process:
   - Estimating effort for building or maintaining applications
   - Project progress tracking
   - Productivity measurement
2. Types of sizing
   - Measure/estimate the physical size of the solution (e.g. counting/estimating # of lines of code)
   - Determine the size of the requirements.

3.2. Reference

Waterfall / V-Model:

https://capgemini.sharepoint.com/sites/PEG69/SitePages/SDI-Model.aspx

Agile Delivery:

- CapGemini e-GREAT estimating models at deal stage

https://capgemini.sharepoint.com/sites/PEG69/SitePages/Agile-Estimations.aspx

- User Story points, T'shirt sizing & Other Relative Sizing-based approaches at execution stage

https://capgemini.sharepoint.com/:p:/r/sites/SSO10/_layouts/15/Doc.aspx?sourcedoc=%7BD772C67D-FD3D-43AF-85B7-60CA2D2FD7B0%7D&file=Agile%20CoP%20%20Session-Agile%20Estimation%20%26%20Planning.pptx&action=edit&mobileredirect=true

4. Metrics Landscape

Category	LDM Lifecycle		Agile Delivery Lifecycle	
	Development & Enhancements	Maintenance	Development /Enhancement	Devops

Quality	Code Coverage%	**Code Coverage %**	Code Coverage %	% Availability
	Testcase Execution Coverage% (SIT)	Testcase Execution Coverage% (SIT)	Testcase Execution Coverage% (SIT)	% Deployment Success Rate
	Defect Density SIT	Defect Density SIT	Production Defects (Sev1 &Sev2)	
	Production Defects (Sev1 &Sev2)	Production Defects (Sev1 &Sev2)	Defect leakage % (UAT) (Sev1 &Sev2)	
	Code Smell	Defect leakage % (UAT) (Sev1 &Sev2)	Code Smell	
	Security **Vulnerability**	**Security Vulnerability**	**S**ecurity **Vulnerability**	
	Technical Debt	Technical Debt		
Schedule	Schedule Variance %	Response Time SLA Compliance %	Release Burndown	Deployment Time
	SLA/KPI Met%	SLA/KPI Met%	Story Point Delivery % (Predictability of velocity)	
		Resolution Time SLA Compliance %	SLA/KPI Met%	SLA/KPI Met%
Efficiency	Productivity	Productivity (size: Lines of code)	Product Backlog Readiness%	Deployment Frequency
	Test Case Automation % (SIT)	Regression Testing Automation %	Velocity	
		Monthly Throughput	Productivity	
			Test Case Automation % (SIT)	
			Regression Testing Automation %	
Cost	Effort Variance	Effort Variance	Sprint Stability %	**Time to Market**
	% Capacity Utilization		Sprint Spillover %	

			Sprint Non-productive time	

Terminologies:

- **Defect Density SIT/QA:** Number of defects detected in System Integration Testing OR QA Environment / Size of work tested. [Size can be measured in Test case points OR Test cases OR Story points OR Function points OR LOC OR Volume of data migrated in MB for files OR # of rows for database etc]
- **Code Review Coverage%:** Size of code reviewed /Size of code checked-in*100
- [Size can be measured in Story points OR Function points OR LOC etc]
- **Code Coverage %:** Amount of code covered during unit testing by Static Analyzers
- **Review Effectiveness % :** Total review defects/ (Review defects +Testing Defects)*100
- **Testcase Execution Coverage**%: (Number of Testcases executed in test environment/Total no. of testcases estimated to be executed) *100
- **Code Smell:** No of Code Smells (maintainability issues in code) from Static Analyzers like SONAR, CAST etc

5. Best Practices and Lessons Learnt

5.1. Development:

Along with industry standard tooling, there are some recommended platforms that support the development and CICD journey for engineers.

Getting Started:

1. Download Spring or Quarkus from the official websites.
   https://start.spring.io/
   https://code.quarkus.io/
2. Establish the coding conventions in the team.
3. Align the architecture of the new service with an architecture pattern

Development Starter Kit

Please refer -
https://capgemini.sharepoint.com/:w:/r/sites/CloudMicroservices/_layouts/15/Doc.aspx?sourcedoc=%7B2AF22C6E-F689-44BC-8190-5AF9C199CA9D%7D&file=Workstation_Environment_Setup_Instructions_V1.0.docx&action=default&mobileredirect=true&wdsle=0

Additional Development Tooling:

Include tools that contribute in static analysis and code security scanning aspects:

- Nexus – Distribute packaged and containerized apps like Docker, Helm, Yum and APT
- SonarQube – Continuous inspection of code code quality to detect bugs and code smells.

- NexusIQ – Scan software libraries(3rd party, open source and custom) to create a detailed inventory of the components that comprise your applications. It constantly monitors inventoried components for new risks.

Coding Conventions:

Coding best practices in Java are often ignored by developers due to lack of knowledge or time to implement them. The best practices in this article are intended for both junior and senior Java developers. These best practices will help improve performance, reduce security risks, converse heap memory and simplify troubleshooting.

Use Correct Naming Conventions

A well-formed name not only aids in reading the code but also conveys a great deal about the code's intent. Here is an example.

```
public class User {
        private String UserName;
        public void SetUserName( String userName) {
                this.userName = username;
        }
}
```

As you can see from the example:-

- Class or interface names should be nouns because they represent real-world objects
- Method names should be verbs because they are always a part of the class and represent an action in general
- Variable names should be short and meaningful and follow Camel case format

Logging

- Avoid excessive logging
- Log the info that may be useful in troubleshooting
- Select the log levels carefully, as we may need to enable log levels in production for troubleshooting

Defensive Coding and resiliency

Always check for NULL value.

- Failure to check for null value will cause the system to throw NullPointerException

***Catch Exception and Log*:**

- If you are expecting exception thrown by the method call, always put try-catch block especially when the call occurred within a loop
- Exception handling should be specific and error message should be meaningful

In an effort to continuously push developers toward "a standard way to develop", we recommend to refer –
https://google.github.io/styleguide/javaguide.html
https://www.oracle.com/technetwork/java/codeconventions-150003.pdf
https://devonfw.com/docs/java/current/cross_cutting/coding_conventions.html

In this the below topics are covered in the coding standards

- File Names

- File Organization
- Indentation
- Comments
- Declarations
- Statements
- White Space
- Naming Conventions for class, method, variables used by the Java program

Naming Conventions:

We recommend using the following naming conventions for services, object, functions, programs, tables, files etc as they are means by which we identify things
Refer- https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html
The naming standards are focussed on following areas:

- XML namespaces
- Java packages
- Class & Interface Names
- Method Names
- Variable Names & Assignments
- Constants

Java Application Logging Framework

Logging equips the developer with detailed context for application failures. To align with the global /
Refer: https://logging.apache.org/log4j/2.x/manual/index.html

Exception Handling

To preserve the readability, maintainability and extensibility of our code, it's a good idea to create our own subtree of application-specific exceptions that extend the base exception class. Hence it is always recommended to have your own Application Error hierarchy.
Refer:
https://www.javaguides.net/2018/06/guide-to-java-exception-handling-best-practices.html

5.2. Unit Testing:

Junit Unit Testing Framework:

Unit tests should be done to ensure the functionality and behaviour of individual software component prior to any system level testing. Junit, an open source, light-weighted framework for Java application unit testing. Refer – https://junit.org

- It allows developer to write test code faster while increasing quality.
- It is elegantly simple and inexpensive.
- It checks their own results and provides immediate feedback.
- It can be composed into a hierarchy of tests suites.

5.3. Continuous Integration

Software development

- Code development:
- Code development would be done via IDE IntelliJ, Eclipse.
- It can be Test driven development to resolve some coding bugs.

Build process – Used for building Java projects.

Gradle :

- https://spring.io.guides/gs/gradle/
- IntelliJ set up for Gradle https://www.jetbrains/help/idea/gradle.html
- Sample gradle configuration

Maven:

- https://sping.io.guide/gs/maven/
- IntelliJ set up for Maven https://www.jetbrains.com/help/idea/maven.html
- Sample pom configuration https://maven.apache.org/guides/introduction/introduction-to-the-pom.html
- Git and GitHub- These guides cover new repo creation, branch permissions and creation and other features.
- Jenkins
- Jenkins Pipeline – Jenkins pipeline is a suite of plugins that supports implementing and integrating continuous delivery pipelines into Jenkins.

Recommended Build tools

Maven (POM):

- A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project
- Minimal POM: The minimum requirement for a POM are the following:
    - project root
    - modelVersion - should be set to 4.0.0
    - groupId - the id of the project's group.
    - artifactId - the id of the artifact (project)
    - version - the version of the artifact under the specified group

Sample pom

```
1. <project>
2. <modelVersion>4.0.0</modelVersion>
3.
4. <groupId>com.mycompany.app</groupId>
5. <artifactId>my-app</artifactId>
6. <version>1</version>
7. </project>
```

Refer - https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

Gradle:

- Like we have pom.xml with Maven, we have build.gradle with Gradle.
- A Gradle project will look something like:

```
project
├── gradle
│   ├── libs.versions.toml
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── settings.gradle(.kts)
├── subproject-a
│   ├── build.gradle(.kts)
│   └── src
└── subproject-b
    ├── build.gradle(.kts)
    └── src
```

Refer - https://docs.gradle.org/current/userguide/getting_started_eng.html

5.4. Patterns:

Validate Inputs and handle errors:

You can validate backend inputs at two levels:

1. **API gateway** — the validation of inputs at the API gateway level via policies; mainly generic validation, i.e. schema, format.
2. **Microservice** — the microservice level validations involve the checking of the existence of entities, etc. There are libraries, e.g. Joi validator, that you can leverage on for ease of input validation depending on your development stack.

Handle error using:

You can consider using two methods to handle errors to mitigate the downstream impact

1. **Circuit breaker —** prevents repeated invocation of services that are likely to fail. There are available libraries and examples for NodeJS, Springcloud, etc. which you can implement the pattern quickly.
2. **Handle exceptions with error codes** — returns a response without crashing the service with a business or HTTP error code to facilitate troubleshooting.

Unit Testing:

Source Code:

It's a good idea to keep the test classes separate from the main source code. So, they are developed, executed and maintained separately from the production code.

Also, it avoids any possibility of running test code in the production environment.
We can follow the steps of the build tools such as Maven and Gradle that look for src/main/test directory for test implementations.

Package Naming Conventions:

We should create a similar package structure in the src/main/test directory for test classes, this way improving the readability and maintainability of the test code**.**

Test Case Naming Conventions:

The test names should be insightful, and users should understand the behaviour and expectation of the test by just glancing at the name itself.
It's often helpful to name the test cases in given_when_then to elaborate on the purpose of a unit test
We should also describe code blocks in the *Given*, *When* and *Then* format. In addition, it helps to differentiate the test into three parts: input, action and output

Expected Vs Actual:

A test case should have an assertion between expected and actual values.
To corroborate the idea of the expected vs actual values, we can look at the definition of the *assertEquals* method of JUnit's *Assert* class:
public static void assertEquals(Object expected, Object actual)
It's suggested to prefix the variable names with the actual and expected keyword to improve the readability of the test code.

Appropriate Annotations:

Always use proper assertions to verify the expected vs. actual results. We should use various methods available in the *Assert* class of JUnit or similar frameworks such as AssertJ.
For eg: Assert.assertEquals, assertNotEquals, assertNotNull, assertTrue

Specific Unit Tests:

Instead of adding multiple assertions to the same unit test, we should create separate test cases. Always write a unit test to test a single specific scenario

5.5. Anti-Patterns:

God Class:

Avoid creating bulky classes having large blocks of functionality holding too much data and has too many methods. This breaks the principle cohesion. Having non-cohesive functionality also means this class becomes critical; making even a small change to it has a system-wide impact in terms of testing efforts

Feature Envy:

In object-oriented design, generally, data should be packaged together with the processes that use that data. Failure to keep data and functionality together is a violation of encapsulation. The envied class can tend to act like a Data Class simply providing access to its data. The envied class can tend to act like a Data Class simply providing access to its data.

Intensive Coupling:

When methods of a class excessively access operations of one or a few other classes, it is said that the class is intensively coupled with those one or few provider classes. Understanding the relation between the client method & the classes providing services becomes difficult.

Brain Method:

Brain method is a method that tends to centralize the functionality of its owner class (very much in a way a God Class tends to centralize a system's intelligence). It tends to be long as it carries out most of the tasks of the class that are supposed to be distributed among several methods. It has too many conditional branches and deep nesting. This impacts understandability and reusability and necessitates excessive testing.

Dispersed Coupling:

When methods of a class access (depend on) many operations across an excessive number of classes, it is said that the class is dispersedly coupled with those provider classes. A change in a dispersively coupled method leads to changes in all the dependent (coupled) classes.

Superficial Test Coverage:

- Only Happy Path tests
- Only easy Tests

This can impact the overall test % coverage.

Catching Unexpected Exceptions:

Unless the test case is checking that an exception should be thrown, there is no reason to catch an exception.

6. Testing and Validation

6.1. Functional Testing

System Testing

Functional testing aims to figure out whether given functionality works as specified. System testing aims to figure out whether the whole system fulfills the requirements given to it.

System Integration Testing

System integration testing helps to ensure that all the components of a system work together by ensuring that interactions between different systems work as expected.

Regression Testing

Software testing technique that re-runs functional and non-functional tests to ensure that a software application works as intended after any code changes, updates, revisions, improvements, or optimizations.

User Acceptance Testing

User acceptance testing (UAT), also called application testing or end-user testing, is a phase of software development in which the software is tested in the real world by its intended audience. It is typically completed after unit testing, quality assurance, system testing and integration testing.

6.2. Non-Functional Testing

Performance Testing

Performance Testing is a type of software testing that ensures software applications perform properly under their expected workload.

Security Testing

Security Testing uncovers vulnerabilities in the system and determines that the data and resources of the system protected against unauthorized access, data breaches, and other security-related issues.

6.3. Other Aspects of Testing Strategy

Test Production Scenarios:

Write tests considering real scenarios in mind, it helps to make unit tests more relatable

Mock External Services:

We should mock the external services and merely test the logic and execution of our code for varying scenarios. We can use various frameworks such as Mockito, EasyMock and JMockit for mocking external services

Avoid Code Redundancy:

Create more and more helper functions to generate the commonly used objects and mock the data or external services for similar unit tests. This enhances the readability and maintainability of the test code.

Annotations:

We should leverage annotations to prepare the system for tests by creating data, arranging objects and dropping all of it after every test to keep test cases isolated from each other. Often, testing frameworks provide annotations for various purposes, for example, performing setup, executing code before and tearing down after running a test. Various annotations such as JUnit's @Before, @BeforeClass and @After and from other test frameworks such as TestNG are at our disposal.

6.4. Test Coverage:

As a rule of thumb, we should try to cover 80% of the code by unit tests. Additionally, we can use tools such as JaCoCo and Cobertura along with Maven or Gradle to generate code coverage reports

Coverage Tools:

1. **Jacoco**: JaCoCo stands for Java Code Coverage. It is a free code coverage library for Java, which has been created by the EclEmma team. It creates code coverage reports and integrates well with IDEs like IntelliJ IDEA, Eclipse IDE, etc. JaCoCo also integrates with CI/CD tools like Jenkins, Circle CI, etc., and project management tools like SonarQube, etc.
   https://www.geeksforgeeks.org/how-to-generate-code-coverage-report-with-jacoco-in-java-application/

2. **Sonar:**

   SonarQube is the leading tool for continuously inspecting the Code Quality and Security of your codebases, and guiding development teams during Code Reviews
   https://docs.sonarsource.com/sonarqube/latest/try-out-sonarqube/

6.5. Contract driven testing:

Contract testing is about making sure your consumer team and provider team have a shared understanding of what the requests and responses will be in each possible scenarios

- Which consumer-side tests specifications should I write?
- For a given service, for a given end point, write a contract test for all methods/responses status combinations, unless your service explicitly returns 401 and 403, you don't need to test this
- Headers when they are specific to the service
- In return response, validate only that the information your consumer cares about was delivered in the format you are expecting

7. Monitoring:

This term is also referred as APM i.e. Application performance monitoring. Using application performance monitoring (APM) solutions, businesses can monitor whether their IT environment meets performance standards, identify bugs and potential issues, and provide flawless user experiences via close monitoring of IT resources. In short, end-to-end application performance monitoring works by:

- Observing whether your apps are behaving normally.
- If not, alerting to and collecting data on the source of the problem (be it the app, app dependencies, or supporting infrastructure)
- Analysing the data in the context of the impact on the business
- Adapting your application environment to fix similar problems before they impact the business.

Few of the most critical application performance monitoring metrics:

- CPU usage: At the server level, APM looks at CPU usage, memory demands, and disk read/write speeds to make sure usage doesn't affect app performance.
- Error rates: At the software level, APM tracks how often app performance degrades or fails. For example, when web requests end in an error or during memory-intensive processes like searching a database.

- Response times: Average Response Time is the metric that shows whether speed is affecting app performance.
- Number of instances: For elastic, cloud-based applications, you need to know how many server or app instances you have running at any one time. APM solutions that support autoscaling can then cost-effectively scale your app to meet user demand.
- Request rates: This metric measures how much traffic your application receives — any spikes, inactivity, or numbers of concurrent users.
- Application availability/uptime: This metric, which monitors whether your app is online and available, is the one most enterprises use to check compliance with SLAs.
- Garbage collection (GC): If your app is written Java or another programming language that uses GC, you'll be all too familiar with the problems that arise from its heavy use of memory. This is an often-hidden performance problem that's worth paying attention to.
- Commonly used APM tools: Splunk, New Relic, Elastic Search

8. Guidelines, Templates, & Checklists

LDM:

https://capgemini.sharepoint.com/sites/QAIMS/SEPG/QMS_Wrapper/SitePages/LDM%20Wrapper.aspx

Agile Delivery Lifecycle – Cap Gemini Agile Framework (CAF)

Capgemini Agile Framework (CAF) provides guidance and good practices for efficient execution of our agile engagements. It is lightweight, based on Scrum, extended with eXtreme Programming (XP) practices and complemented by Capgemini's best practices. It can be tailored based on the client needs. It covers activities to be performed for delivering an Agile engagement.

The Link below will provide you with:

- Agile terminologies
- Agile Ceremonies
- Artifacts
- Agile Roles
- Guidelines

https://deliver2.capgemini.com/components/CAF-V4.1/#Capgemini%20Agile%20Framework/guidances/supportingmaterials/CAF%20Overview_B368AD2D.html

https://capgemini.sharepoint.com/:p:/r/sites/SSO10/_layouts/15/Doc.aspx?sourcedoc=%7B5C24AD93-4ECB-47C6-ACE2-E4780370E132%7D&file=Capgemini%20Agile%20Framework.pptx&action=edit&mobileredirect=true

9. Tools

9.1. LDM DevOps Tools:

The table below depicts the recommended phase wise open source tools.

Phase	Tool	Reference
Project Management	• Open Workbench • Jira • Microsoft project	• https://java-source.net/open-source/project-management/open-workbench • https://www.atlassian.com/software/jira • https://www.microsoft.com/en-in/microsoft-365/project/project-management-software
Requirements Gathering	• Confluence • Sharepoint	• https://www.atlassian.com/software/confluence • https://www.microsoft.com/en-us/microsoft-365/sharepoint
Functional Specifications	• Confluence	
Architecture	• Enterprise Architect • Lean IX • IBM Rational Software Architect. • Erwin	• https://www.leanix.net/en/ • https://www.leanix.net/en/ • https://www.ibm.com/products/rational-software-architect-designer • https://www.erwin.com/software-demos-and-trials/
Design	• Draw.io • LucidChart	• https://draw.io • https://www.lucidchart.com/pages/?
Build	• Maven • Gradle	• https://maven.apache.org/download.cgi • https://gradle.org/install/
CI/CD pipeline	• Jenkins • Team city	• https://www.jenkins.io/download/ • https://www.jetbrains.com/teamcity/download/
Code repository	• Git • BitBucket	• https://github.com/ • https://bitbucket.org/
Development and Unit testing	• IntelliJ • Eclipse • Net Beans	• https://www.jetbrains.com/idea/ • https://eclipseide.org/ • https://www.oracle.com/in/tools/technologies/netbeans-ide.html
Code review	• Checkmarx • Sonar lint • Sonar Q	• https://checkmarx.com/ • https://www.sonarsource.com/products/sonarlint/ • https://www.sonarsource.com/products/sonarqube/

Unit Testing	• Junit • Mockito	• https://junit.org/junit5/ • https://site.mockito.org/
System Testing	• Apache JMeter • SOAPUI • Postman	• https://jmeter.apache.org/ • https://www.soapui.org/downloads/soapui/ • https://www.postman.com/api-platform/api-testing/
Regression Testing	• Selenium • Tosca	• https://www.selenium.dev/downloads/ • https://www.tricentis.com/software-testing-tool-trial-demo/tosca-trial

9.2. Agile Tools:

Project management	Code Development	Code Quality	Code Build/Packaging	Unit Testing	Functional Testing	Performance testing	Deployment / Monitoring
Confluence	IntelliJ Ecilpse	SonarQube	Maven	Junit 5 Mockito	TESTIM	NEOTYS	SYNK

9.3. Gen AI tools:

Tools Samples:

Project management	Code Development	Code Quality	Code Build/Packaging	Unit Testing	Functional Testing	Performance testing	Deployment/Monitoring
GitFluence	GITHUB COPILOT	TABNINE GITHUB	SNYK	TABNINE DIFFBLUE	TESTIM	NEOTYS	SYNK
Sidekick	Amazon CodeWhisperer	COPILOT MINTLIFY	GITHUB COPILOT	INTELLITEST	Functionize		KubeFlow
Kite	Codeium	AskCodi	Deep Code	GITHUB COPILOT	DiffBlue		DataDog
		CodiumAI	Codiga				

** Most of the tools are available at a cost. They either have a fixed or subscription costs associated with them

10. Case Studies:

https://capgemini.sharepoint.com/sites/fscloudkm/Lists/Client%20Case%20Studies/AllItems.aspx

11. Appendix A:

Java-Microservices

Architecture-Context

- https://microservices.io/patterns/microservices.html

Design Patterns

- https://microservices.io/patterns/index.html

Microservices security patterns

- https://tsh.io/blog/microservices-security-patterns/

Dev Ops

Git Hub Commands

- https://git-scm.com/docs/gittutorial

CI/CD Pipeline Set up for Java Apps

- https://buddy.works/docs/quickstart/java

Jenkins CI/CD pipeline setup

https://www.jenkins.io/doc/tutorials/#pipeline

Build a Java app with Maven

- https://www.jenkins.io/doc/tutorials/build-a-java-app-with-maven/

Java

Java 8 Features

- https://www.digitalocean.com/community/tutorials/java-8-features-with-examples

Java Design Solid Principles

- https://www.jrebel.com/blog/solid-principles-in-java

Java OOPS Design Patterns

- https://www.oodesign.com/

Code cleans up Tools for Java Apps

- https://snyk.io/learn/code-review/java-tools/

Mockito, Junit Test cases for Spring boot

- https://howtodoinjava.com/spring-boot2/testing/spring-boot-mockito-junit-example/