Mini-Project Database (MPDB)

COS 301 (Pty) Ltd.

COS 301
Mini-Project Playbook
2025

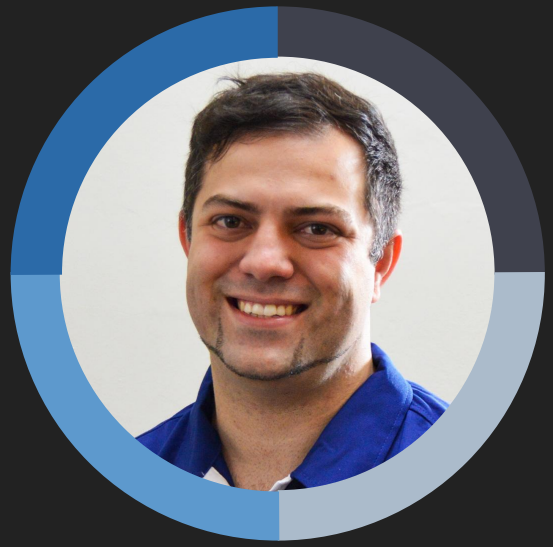
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Make today matter

www.up.ac.za

Full Stack Software Engineering

$whoami



Arné Schreuder

@arneschreuder

Phd Topic: Biologically Plausible Machine Learning

Lecturer - UP
Senior Software Engineer AI/ML
EPI-USE Global Services

Specialising in production applications of
Software Engineering**, **Data Science** and **Artificial Intelligence

Introduction

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Make today matter

www.up.ac.za

Introduction

Use this document as a tl;dr for Software Engineering during the Mini-Project. This playbook is your go-to guide for the Mini-Project.
Firstly, we present an overview of the Mini-Project specification details as follows:

1. Background & Motivation
2. System Requirements
3. Architectural Requirements
4. Design Requirements
5. Delivery Requirements
6. Constraints
7. Design
8. Project Plan
9. Client Details

Introduction (cont.)

Then, to assist you during the Mini-Project. The playbook consists of the following cheat-sheet/summarised topics:

- Software Engineering Processes
- Documentation
- Roles
- Architectures
- Interfaces
- Design Patterns
- Quality Assurance
- Testing
- Git
- Deployment
- DevOps
- Project Management
- Demos

Important Dates

1. **17 Feb 2025** - Mini-Project - Lecture 1 (Mini-Project Discussion)
2. **18 Feb 2025** - Mini-Project - Lecture 2 (Overview of Cheatsheet for Mini-Project) + *Sprint 1 Starts*
3. **14 Mar 2025** - Mini-Project - Demo 1 (Only project managers/leads) + *Sprint 2 Starts*
4. **25 Apr 2025** - Mini-Project - Demo 2 (Full team)

Mini-Project Details

Introduction

The **Mini-Project Database (MPDB)** is an educational initiative designed for **COS 301 final-year computer science students at the University of Pretoria**. This project will serve as the Mini-Project for the module in 2025.

You have **10 weeks** to **design and implement a functional NoSQL database system** from **scratch**. MPDB is a **feature-rich, database** that will challenge you to apply full stack software engineering.

The Mini-Project is structured into **two sprints**:
- **Sprint 1 (4 weeks):** Develop a Beta version with core functionality.
- **Sprint 2 (6 weeks):** Improve and finalize the system for production.

Your team will consist of **~12 members.** If you follow the project guidelines carefully, you will gain **valuable hands-on experience in full-stack development, security, testing, and CI/CD.**

The deliverable is:
A **deployed*, working, functional NoSQL database SYSTEM** that consists of a daemon (1), cli (2), an API (3) and a UI (4).
* Deployment in this case means an installable/runnable build produced by CI/CD.

Background & Motivation

Databases are the backbone of modern applications—from social networks to banking systems, every major system relies on efficient data storage and retrieval.

Although databases are a fundamental part of software engineering, they too themselves are full-stack software engineering systems. Many students do not get hands-on experience designing one from scratch. This project aims to bridge that gap by **challenging students to build a functional NoSQL database system** with modern features such as:

- **Design patterns/Architectures**
- **Security and access control**
- **Real-time capabilities**
- **Multiple interfaces (CLI, REST API, Web UI, JS Library)**

In previous years, Mini-Projects have included webapps to cover core software engineering concepts. This year is no different. MPDB is a full-stack software engineering project that will give students a good idea of the entire software engineering approach.

System Requirements [Required]

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Make today matter

www.up.ac.za

System Stack [Required]

The **MPDB project** consists of several sub-systems, each to be delivered:

** To get a good idea of what we are building, have a look at MySQL (the cli client, the daemon, UI (mysqlworkbench)). We are replicating something like this for the mini-project but using NoSQL.*

- Core Engine/Daemon (**mpdbd**): This runs the database engine
- CLI/Terminal Client (**mpdb**): This runs the terminal client to interact with the daemon.
- RESTful API (**mpdb-api**): This provides a RESTful API to interact with the daemon.
- JavaScript Client Library (**mpdbjs**): This provides a javascript client library to interact with the API.
- UI (**mpdb-studio**): This provides a database management interface, to visually interact with the database, by making use of mpdbjs, mpdb-api and mpdbd.

Project Requirements [Required]

The **MPDB project** consists of several sub-systems:

- You have to use the Git project we provide (monorepo). Use this for:
  - Version control
  - Project Management
  - CI/CD
    - Lint
    - Build
    - Test
      - Unit Testing
      - Integration Testing
      - E2E Testing
    - Deploy/Deliver
- Documentation: Markdown + GitBook (or equivalent)/Github Pages/Google Docs
- Delivery:
  - Please package the entire system into a single runnable docker-compose file.

Core Requirements [Required]

Although every database system has unique features, the following core requirements MUST be implemented as part of MPDB:

Data Storage and Retrieval:

- MPDB is a NoSQL database. Make use of JSON-based data storage for flexible data structures.
- Support for **nested JSON objects and arrays.**
- CRUD operations, including **batch operations** for efficiency.
- Simple query language supporting:
  - Equality and inequality comparisons
  - Logical operators **(AND, OR, NOT)**
  - **Regular expression matching**
  - Sorting and pagination

Data Operations:

- In a NoSQL setting, there are Collections & Documents. Collections are groups of documents that all have the same “schema”.
- Provide Create, Read (One and Many), Update and Delete data operations.

Core Requirements [Required] (cont.)

User Interfaces

- **Command-Line Interface (CLI)** for database operations:
  - Provide all CRUD operations to daemon
- **Web-based UI (MPDB Studio)** for visual database operations:
  - **Landing**: Must have login screen for authentication.
  - **Dashboard:** Displaying key database statistics.
  - **Data:** To explore/view the data.
  - **Query:** To query the data.
  - **Users:** To manage users and permissions

Security

- The system must support authentication and permission-based authorisation.

Optional Requirements [Optional]

- Indexing for faster querying
- Encryption (for security)

Wow Factors [Required]

To enhance the functionality and user experience of MPDB, teams must implement at least one WOW Factor. The WOW Factor for MPDB is real-time/streaming capabilities. Say a user queries a certain view, if that view updates, so must the client.

It is recommended to follow an Event-Sourcing/Driven Architecture similar to CQRS to provide such a capability.

Architectural Requirements [Required]

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Make today matter

www.up.ac.za

Architecture Requirements [Required]

For the MPDB project, students must design and implement an architecture that supports scalability, modularity, and maintainability. The following architectural requirements must be met:

A Client-Server Architecture:

- The system must follow a client-server architecture, separating the mpdbd (core engine) from the mpdb-studio (UI) through 1. mpdbjs (JavaScript library) and 2. mpdb-api (RESTful API)
- Additional services such as event-driven notification systems or background processing are encouraged for advanced functionality.

[Recommended] Event-Driven Architecture:

- The system can leverage event sourcing for capturing all changes in the database.
- A publish-subscribe (pub/sub) model should be implemented for real-time updates.
- Support for event replay and system recovery could be built into the architecture.

One Design Pattern:

The students must explicitly make use of at least one design pattern and be able to motivate their choice. CQRS is a recommended choice.

Testing Strategy [Required]

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Make today matter

www.up.ac.za

Testing Strategy [Required]

The mpdb project will employ a comprehensive testing strategy to ensure the quality and reliability of the system. This multi-faceted approach will cover various aspects of testing, providing students with exposure to industry-standard testing practices.

Unit Testing:

- Test individual components and functions in isolation
- Use mocking and stubbing to isolate units from their dependencies
- Aim for high code coverage (target: 90% or higher)
- Implement property-based testing for complex algorithms

Integration Testing:

- Test interactions between different components of the system
- Verify API endpoints using tools like Postman/Newman
- Test database operations end-to-end
- Validate event propagation in the event-driven architecture

Testing Strategy [Required] (cont.)

Performance Testing:

- Benchmark read and write operations under various loads
- Test query performance with different data sizes and complexities
- Evaluate real-time capabilities under concurrent connections
- Identify and optimize performance bottlenecks

Security Testing:

- Perform penetration testing to identify vulnerabilities
- Test authentication and authorization mechanisms
- Validate data encryption and secure communication
- Conduct SQL injection and XSS attack simulations

Testing Strategy [Required] (cont.)

User Interface & Acceptance Testing:

- Involves testing the UI from a user's perspective
- Can be automated

Continuous Integration and Testing:

- Integrate automated tests into the CI/CD pipeline
- Use code coverage tools to track testing effectiveness over time

Delivery Requirements [Required]

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Make today matter

www.up.ac.za

Delivery Requirements [Required]

The mpdb project will produce the following deliverable software artifacts:

System:

- mpdbd: Database daemon
- mpdb: Command-line interface
- mpdbjs: JavaScript/TypeScript client library
- mpdb-api: A RESTful API for mpdb
- mpdb-studio: A UI for mpdb.

Documentation

- Requirements Specification
- Architectural/Design Specification
- API Specification (Daemon)
- User Manual/Help for user-facing software artifacts (i.e. CLI, Studio)
- User Guide for MPDB Studio
- Developer Guide MPDBJS

Testing Artifacts

- Unit tests
- Integration tests

Delivery Requirements [Required] (cont.)

Docker Containers

- Dockerfile for each component
- Docker Compose file for orchestrating the entire system
- README with instructions for building and running containers

CI/CD Configuration

- Built in Github Actions

Project Management Artifacts

- Project management board on Github

Demos

- There will be 2 demos
- Slides for final project presentation
- Recorded video
- Live demonstrations

Project Plan [Required]

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Make today matter

www.up.ac.za

Team Breakdown

Generally, software engineering teams consists of various role-players. Organise and delegate according to strengths. Members might fill multiple roles depending on the skillset of the group.

- Project Manager
- Business Analyst
- Architect
- Designer
- UI Engineer
- Integration Engineer
- Services Engineer
- Data Engineer
- Testing Engineer
- DevOps

Project Plan [Optional]

Sprint 1 Beta version (5 Weeks):

Focus on getting the project “in beta”. This means that you should have a generally runnable system that can still be incomplete, unstable and sub-optimal. However, a working concept, nonetheless.

Sprint 2 Production version (5 Weeks):

Focus in getting the project “in production”. This means you should be ready to put this into the hands of clients. It should be 100% complete, working, without bugs and missing features.

Github [Required]

The students are expected to use Github for:

- Version Control
- GitHub Actions (CI/CD):
  - Continuous Integration
  - Continuous Testing
  - Continuous Deployment
- Pull-Requests/Merges - Integrated with CI/CD
- Project Planning (Project Boards)
- Issue Tracking (Issues)

Students **must** have each of these elements in place. We use this to track progress, contributions and host various artifacts, such as links to documentation etc on the project. Github will be a key component during the entire year, not just the Mini-Project.

Client Details

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Make today matter

www.up.ac.za

Contact Details

The client for the Mini-Project will be the COS 301 Lecturers. Contact details can be found in the study guide.

Corporate Assets

- Please use your own team’s assets.
- Where applicable, use the University’s Corporate Assets.

Cheat-Sheets

* Some content generated using AI

Software Engineering Processes

You will come across the following Software Engineering Processes.

Waterfall:

The Waterfall process uses a linear approach which follows a sequence of steps to be completed in order. A true waterfall approach views every sequence as a distinct stage in the development process. Waterfall is very advantageous when pair coding and group oriented work is required.

Agile:

The Agile process focuses on the rapid delivery of functional components through continuous improvement and working on components separately to deliver greater value faster. Agile is advantageous in small teams that work separately and tends to bring a faster return on investment if applied correctly.

Scrum:

The scrum process main focus is getting work done. Scrum and Agile are not the same as Agile is more of a philosophy while scrum is more of a framework. Scrum is based on continuous learning and adjustment. Scrum is very advantages in teams that have the time to learn through the experience and making mistakes.

Software Engineering Processes (cont.)

Test-Driven Development (TDD):

Test-driven development driven practices on creating tests before writing the code. It is the process of thinking what tests will be required to show that the application does what we want it to do. In the process of writing these tests to show the product works, the design and functionality of the application is discovered.

Domain-Driven Development (DDD):

Domain-Driven Development centred around a sphere of knowledge of those that use it. Takes the approach of modeling systems by the domain of their applications.

Documentation

You will come across the following Software Engineering Documentation.

Requirement Specification:

Documents the iterative process of evaluating the client's needs. A software engineering methodology is used to elicit, plan, and design the solution based on these requirements recorded here.

Design Specification:

Documents centered around the design of the system (think 214) UML diagrams etc.

Architectural Specification:

Documents focused on the design of the system architecture and the iterations of it.

Documentation (cont.)

Implementation Specification:

An Implementation Specification provides detailed guidance and instructions for developers on how to translate the design specifications into actual working code. It outlines the technologies, programming languages, libraries, frameworks, and methodologies to be used during the implementation phase of software development. This document typically includes code conventions, architectural guidelines, database schema, APIs, algorithms, and any other technical details necessary for developers to understand and implement the software system accurately.

Testing Specification:

A Testing Specification outlines the comprehensive plan and strategy for verifying and validating the functionality, performance, and reliability of a software system. It defines the testing objectives, scope, methodologies, test scenarios, test cases, test data, and success criteria for various types of testing such as unit testing, integration testing, system testing, acceptance testing, and performance testing.

Documentation (cont.)

Delivery Specification:

A Delivery Specification serves as a comprehensive guide for packaging, distributing, and deploying the software product to end-users or clients. It includes instructions on how to package the software components, configuration files, dependencies, installation procedures, system requirements, and any other necessary resources for deployment. This specification specifically focuses on production delivery.

User Manual:

A User Manual is a documentation artifact aimed at end-users or stakeholders who will interact with the software system. It provides detailed instructions, tutorials, and guidance on how to use the software effectively to accomplish specific tasks or achieve desired outcomes.

Roles

We have identified and implemented these roles that contribute to the Software Engineering Process. (This is not in the textbook and is something we implemented about 3 years ago)

Project Owner:

The client, or internally the person in charge of the project overall.

Project Manager:

The leader of the project. Organizes members, allocates tasks and ensures deadlines are met. Organizes help if team members are struggling. Client facing - communicates with the project owner(s). Can help contribute to documentation.

Business Analyst:

Documentation specialist, ensures members' document the parts of the system they are responsible for. Collects the documentation and ensures all documents are complete and up to date.

Roles (Cont.)

Designer:

Communicates with the architect and project manager the user interface and all components surrounding the frontend. The designer documents, signs off on the design, and any changes during the project. This can include wire frames etc.

Architect:

Communicates to the project engineer, designer, UI, and integration engineers the structure (architecture) of the application. They document and determine the most applicable design for the product, document it and sign off any changes during development. They can generate the initial structure and functionality of the application for the rest of the team to build upon.

UI Engineer (Front-end):

Implements and improves upon the design of the frontend with the designer and integration engineer. They document the code and communicate any changes to the designer to agree upon and document.

Integration Engineer (API/Services):

Develops the connection between the backend and frontend based on the architecture provided. This includes any functionality to make calls to the backend from the frontend and any code the UI engineer requires to make allow the frontend to work well. On the backend they develop any processing required to successfully transfer data to the frontend. This can include mapping the data on the backend to DTOs, integrity of data being transferred etc. Lastly they document these developments and communicate anything developed or changed to the architect.

Roles (Cont.)

Services Engineer (Backend):

Develops all functionality on the backend based on the architecture provided, documents these developments and provides the data required by the data and integration engineers. Implements the business logic of the system.

Data Engineer:

Designs and documents the data stores, models and DTOs of the system. This includes the database, integrations to the datastores, any mapping requires between them, and anything required by the service engineer to process and aggregate the data two and from the integration engineer.

Test Engineering:

Ensure all tests written by the other team members are not failing. Documenting the tests. Reporting any failing tests to the team member responsible and ensuring they are rectified.

DevOps:

Documenting, developing and maintaining the pipeline required to test, deploy and maintain the integrity of the code base.

Architectures

You will come across the following Architectures.

Monolithic (Standalone)

A monolithic architecture is an approach where all components of an application are tightly integrated and deployed as a single unit. In such systems, changes to one part of the application often require redeployment of the entire system.

Client-Server:

Client-server architecture is a computing model in which client devices request services or resources from servers, which provide responses accordingly.

Layered:

Layered architecture organises a software system into distinct horizontal layers, such as presentation, business logic, and database, ensuring separation of concerns and facilitating modular development.

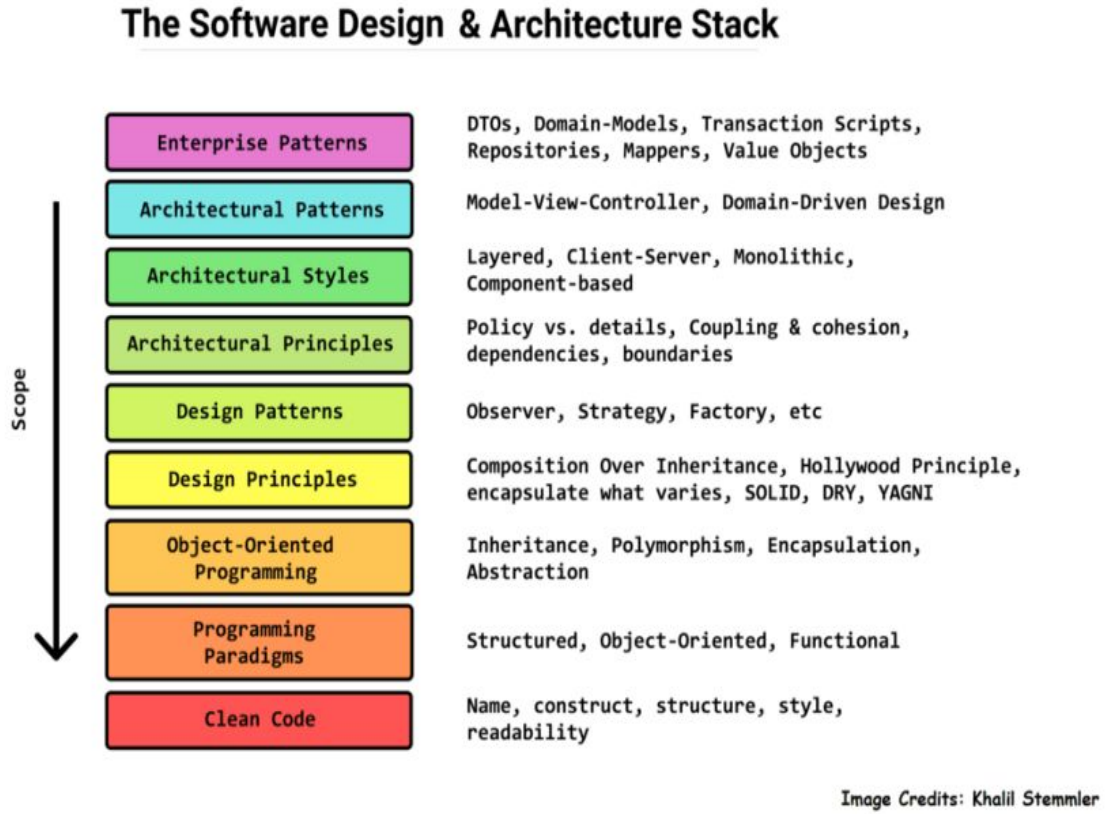


Image Credits: Khalil Stemmler

Architectures

N-Tier:

N-Tier architecture extends the layered architecture concept by introducing physical or logical tiers, enabling distributed deployment across multiple servers and communication between tiers.

Services Oriented:

Service-Oriented Architecture (SOA) is a software development approach where services, which are self-contained and loosely coupled, are provided to the other components by application components, through a communication protocol over a network

Micro-Services:

Microservices architecture is a design approach where an application is structured as a collection of loosely coupled services that are organised around business capabilities.

Interfaces

Services interact with each other over some communication protocol and media, using a specific application programming INTERFACE (API):

REST:

Most popular interface for Web Applications. REST is an architectural style for designing networked applications. It utilizes a set of architectural constraints and principles to define a stateless, client-server communication protocol. In a RESTful API, resources are represented as URIs (Uniform Resource Identifiers), and interactions with these resources are performed using standard HTTP methods (GET, POST, PUT, DELETE) along with the appropriate HTTP status codes. REST APIs typically communicate over the HTTP or HTTPS protocol and use standard data formats like JSON or XML for payload exchange. REST promotes scalability, simplicity, and interoperability between different systems, making it a popular choice for building web services.

SOAP:

SOAP is a protocol for exchanging structured information in the implementation of web services. It relies on XML (eXtensible Markup Language) as its message format and typically uses HTTP or SMTP (Simple Mail Transfer Protocol) as the transport protocol. SOAP APIs define a strict messaging format with standardized XML-based protocols for request and response envelopes, including headers for specifying message attributes like security, transaction management, and routing. SOAP APIs are known for their strict contract definition through WSDL (Web Services Description Language) files, providing strong typing and formalized error handling. SOAP is often used in enterprise-level applications where robustness, security, and reliability are critical requirements.

Interfaces (cont.)

GraphQL:

GraphQL is a query language and runtime for executing queries against data sources, originally developed by Facebook. Unlike traditional RESTful APIs where clients retrieve fixed data structures determined by the server, GraphQL enables clients to specify the exact data they need in their queries. This allows for more efficient data retrieval by minimizing over-fetching or under-fetching of data. GraphQL APIs are defined by a schema that specifies the types of data available and the operations that can be performed. Clients send queries to the server specifying the fields they want to retrieve, and the server responds with JSON payloads containing precisely the requested data. GraphQL promotes flexibility, efficiency, and improved client-server communication in modern web applications.

gRPC:

gRPC is a high-performance RPC (Remote Procedure Call) framework developed by Google. It uses HTTP/2 as its transport protocol and Protocol Buffers (protobuf) as its interface definition language (IDL). gRPC allows clients to call methods on a remote server as if they were local methods, abstracting away the complexities of network communication. gRPC APIs are defined using protobuf files, which specify the service interface and message types exchanged between the client and server. gRPC supports various programming languages and platforms, providing features like bi-directional streaming, authentication, and load balancing out-of-the-box. gRPC is well-suited for building efficient, low-latency, and highly scalable distributed systems, especially in microservices architectures.

Design Patterns

There are too many to list here, however, we include some of the most common here:

Model-View-Controller

The Model-View-Controller (MVC) architectural pattern separates an application into three interconnected components: the Model, responsible for managing data and business logic; the View, responsible for presenting data to the user and handling UI interactions; and the Controller, acting as an intermediary between the Model and View, processing user input events and updating the Model or View accordingly.

Model-View-ViewModel:

The Model-View-ViewModel (MVVM) architectural pattern is an evolution of the Model-View-Controller (MVC) pattern. The pattern facilitates separation of concerns in UI development by introducing the ViewModel as a mediator between the Model and the View. It promotes data binding, allowing automatic synchronisation between the View and ViewModel.

Facade:

The Facade design pattern is a structural pattern that provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use by wrapping a complex system with a simpler interface.

Proxy:

The Proxy design pattern is a structural pattern that provides a surrogate or placeholder for another object to control access to it. It allows you to create an intermediary that acts as an interface to another object, hiding the complexity of the underlying object or controlling access to it.

Strategy:

The Strategy design pattern is a behavioural pattern that enables an object to alter its behaviour dynamically at runtime. It defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern allows the algorithm to vary independently from clients that use it.

Design Patterns

Factory (AbstractFactory)

The Abstract Factory pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It allows a client to create objects without knowing their specific implementations, promoting loose coupling between client code and the concrete classes it uses.

Commander:

The Command design pattern is a behavioural pattern that encapsulates a request as an object, thereby allowing parameterization of clients with queues, requests, and operations.

Observer:

The Observer design pattern is a behavioural pattern that establishes a one-to-many dependency between objects, ensuring that when one object changes state, all its dependents are notified and updated automatically.

Singleton:

The Singleton design pattern is a creational pattern that ensures a class has only one instance and provides a global point of access to that instance.

CQRS:

CQRS (Command Query Responsibility Segregation) is an architectural pattern that segregates the responsibility for handling read and write operations on data, often employing separate models and mechanisms for each operation.

Quality Assurance

Encompasses a set of activities to ensure that the software under development or modification will meet functional and quality requirements.

Includes:

- Cost Analysis Verification (is it built to budget)
- Validation (is it the right product)
- Reviews (code reviews)
- Code Testing
- Non-functional requirements testing

Testing

You will come across a number of testing techniques. These are the most important:

Unit Testing

Testing the functions in a module / class / service all do what they are intended to do and otherwise fail predictably and reliably. **Importantly: INTEGRATIONS ARE MOCKED TO ISOLATE UNITS (Function A calling Function B? Function B is mocked).**

Integration Testing:

Testing the functionality and outcomes across services which work together. **Importantly: INTEGRATIONS ARE NOT MOCKED, AND FULL INTEGRATION FROM START TO END IS TESTED. (Function A calling Function B? Function B is actually called).**

End-To-End (e2e) Testing:

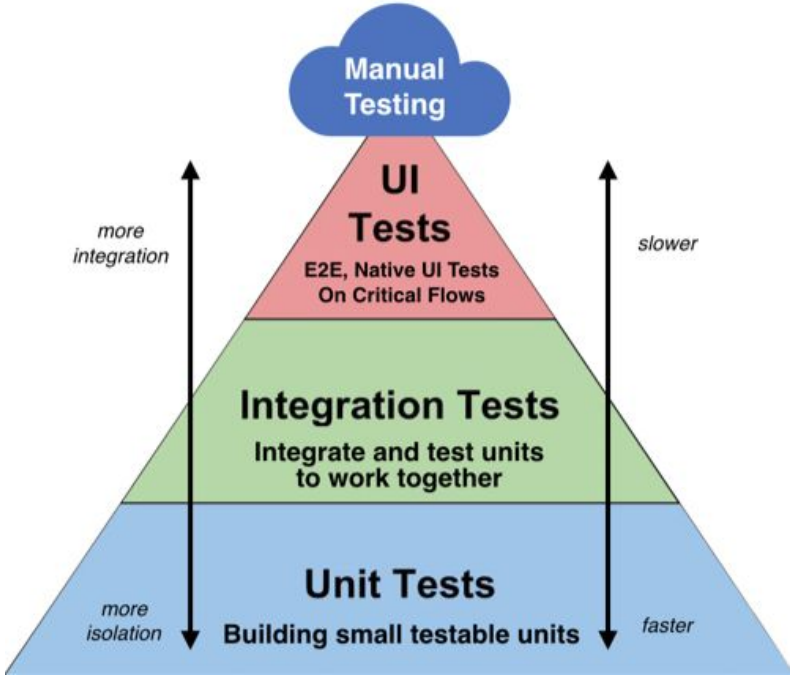
Subset of integration testing. Testing the process of an interaction from the frontend all the way to the backend (and possibly back).

Usability Testing:

UI / UX testing and how user-friendly the system is.

Non-Functional Testing:

Quality Assurance testing based on your quality requirements (security, portability, extensibility, performance etc.)



https://lawrey.medium.com/unit-tests-ui-tests-integration-tests-end-to-end-tests-c0d98e0218a6

Git

Git is a Software Version Control (SVC) tool used to manage the collaboration and integration of Software Source Artifacts in the context of multi-party development.

Track Files

Tracking files means to start monitoring them for changes. This is typically done by adding new files to the Git repository using the “**git add**” command. Once a file is tracked, Git knows to include it in future commits.

Stage Files:

Staging files refers to the process of preparing changes for a commit. After you have made changes to a file, you use the “**git add**” command to stage the changes. This adds the changes to the staging area, which is a temporary area where Git holds the changes that will go into your next commit.

Commit Files:

Committing files means to save your changes to the repository. After staging the changes with “git add”, you use the “**git commit**” command to create a new commit, which is a snapshot of your repository at a particular point in time. Each commit is given a unique ID and includes a message describing the changes made.

Push Files:

Pushing files involves sending your local repository's commits to a remote repository. This is done using the “**git push**” command, which uploads all local commits to the remote repository that your local repository is connected to. This allows you to share your changes with others and keeps the remote repository up-to-date with your local repository.

Pull Files:

Pulling files involves updating your local repository's commits with what is on a remote repository. This is done using the “**git pull**” command, which downloads all remote commits to the local repository.

Git (cont.)

Pull Requests:

A pull request is a method of submitting contributions to a project. It is a way to notify team members that you have completed a feature or fixed a bug and are ready for your code to be reviewed and merged into the main branch.

Branches:

Branches are a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug, you create a new branch that diverges from the main branch. This allows you to work on new features or bug fixes without affecting the main codebase.

Merge:

Merging is the process of combining the changes from one branch into another. This is typically done when you have completed work on a feature branch and want to integrate those changes into the main branch. The merge creates a new commit in the target branch that includes all the changes from the source branch.

Rebase:

Rebasing is the process of moving or combining a sequence of commits to a new base commit. It is often used to make a feature branch up to date with the latest code from the main branch, or to clean up a feature branch before merging. Rebasing rewrites the commit history to appear as if the changes were made directly on the base branch, which can result in a cleaner and more linear history.

Deployment

The students will have to show that they understand the deployment and delivery part of Software Engineering. For this consider:1

Environments:

Different environments of deployment serve different purposes. You will come across these environments:

- **Local/Source** - Local files
- **Development** - Development environment, meant to be used by developers, often breaks, point of CI.
- **Testing** - Testing environment, meant for either automated CI/CD testing or continuous testing by tester.
- **Production** - Final, client-and-user-facing environment.

DevOps

Automated development pipelines is a key tool in the arsenal of a Software Engineer. Students will be required to demonstrate their capabilities in this regard.

Continuous Integration (CI):

The process of continuously integrating different people's contributions to a source code repository in an organised manner. This process is usually highly automated with scripts/pipelines for:

- Linting
- Building
- Testing

This process is often kicked off as part of a Pull Request (PR) or Merge Request in a SVC Platform like Github. Here developers ensures that incoming code is correct, working and valid.

Continuous Deployment (CD):

Continuous deployment refers to the specific step added to the CI pipeline where artifacts are "deployed". This usually involves uploading build artifacts to some remote hosting server and triggering a redeployment. CD is just another step in the CI pipeline, specifically targeting regular deployments.

Project Management

In the context of “scrum” and agile:

Project Board:

Project boards like Github Projects, JIRA, Trello is often used to assist in project management. KANBAN flow is often used where “tickets” go from “to-do” phase to “doing” phase to “done” phase.

Milestones/Roadmap:

Milestones and roadmaps refer to key dates that connect some relevant deliverable of achievable aspect. These are often used to give customers/users a good idea of when to expect what.

Epics/Stories/Tasks:

Project management strategies like agile scrum provides various levels of milestone grouping or abstraction. Epics relate to larger, longer term goals. Stories relate to features, and modules and can consist of a number of tasks. Tasks relate to the immediate group of work to be done in a smaller time frame, either by an individual or a small group.

Demos

Demos are central to the success of the COS 301 module. Take this tips:

Be Prepared:

Being prepared not only helps avoid disasters, technical or non-technical, but it also shows confidence in what you are learning and applying. Remember, we already know what you are learning now, so let it come across, as you do as well.

Stick to the Scope/Instructions/Rubrik:

We give out demo instructions for a reason. Make sure to structure your demo around these instructions, because that is what we are marking for.

Presentation:

Please use clearly readable, simple, clean slides (like Canva or PPT slides). Avoid using videos to demo features (only have as a backup). Please do not add background music or distracting audio to your presentation)

Professionalism:

When demoing, imaging demoing to a real client or possible investor. Maintain the highest levels of professionalism in terms of your presentation, spelling, grammar, language and tone.

Demos (cont.)

Dry-Runs:

Take the time to dry-run on blackboard before the demos. Record these as a backup. Often thing go wrong on Blackboard. Make sure you can share, minimal people are talking and need to switch screens. Check audio, camera permissions etc.

Technicals:

Be sure of the technical requirements for the demo. Pay close attention to testing requirements, feature requirements, CI/CD requirements, delivery requirements.

Questions:

To avoid wasting time, differ questions to the end of your demo. This ensure that you finish your demo in time. Make sure to leave time for questions at the end.

THANK YOU
(and Good Luck)

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Make today matter

www.up.ac.za