FOSSEE Winter Internship Report

On

GUI Modernization, Unit Testing Framework, MagicMock and OSI Standardization

Submitted by

Agam Arora

3rd Year B.Tech Student, Department of Computer Science and Enginnering

Vellore Institute of Technology

Bhopal

Under the Guidance of

Prof. Siddhartha Ghosh

Department of Civil Engineering

Indian Institute of Technology Bombay

Mentors:

Ajmal Babu M S

Parth Karia

Ajinkya Dahale

February 5, 2026

Acknowledgments

- Start with a general statement of thanks. Express your overall gratitude to everyone who supported you during your project or research.
- Project staff at the Osdag team, Ajmal Babu M. S., Ajinkya Dahale, and Parth Karia,
- Osdag Principal Investigator (PI) Prof. Siddhartha Ghosh, Department of Civil Engineering at IIT Bombay
- FOSSEE PI Prof. Kannan M. Moudgalya, FOSSEE Project Investigator, Department of Chemical Engineering, IIT Bombay
- FOSSEE Managers Usha Viswanathan and Vineeta Parmar and their entire team
- Acknowledge the support from the National Mission on Education through Information and Communication Technology (ICT), Ministry of Education (MoE), Government of India, for their role in facilitating this project
- Acknowledge your colleagues who worked with you during your internship or project.
- If appropriate, thank your college, department, head, and principal for their support during your studies.

Contents

Chapter 1

Introduction

1.1 National Mission in Education through ICT

The National Mission on Education through ICT (NMEICT) is a scheme under the Department of Higher Education, Ministry of Education, Government of India. It aims to leverage the potential of ICT to enhance teaching and learning in Higher Education Institutions in an anytime-anywhere mode.

The mission aligns with the three cardinal principles of the Education Policy—**access, equity, and quality**—by:

- Providing connectivity and affordable access devices for learners and institutions.
- Generating high-quality e-content free of cost.

NMEICT seeks to bridge the digital divide by empowering learners and teachers in urban and rural areas, fostering inclusivity in the knowledge economy. Key focus areas include:

- Development of e-learning pedagogies and virtual laboratories.
- Online testing, certification, and mentorship through accessible platforms like EduSAT and DTH.
- Training and empowering teachers to adopt ICT-based teaching methods.

For further details, visit the official website: www.nmeict.ac.in.

1.1.1 ICT Initiatives of MoE

The Ministry of Education (MoE) has launched several ICT initiatives aimed at students, researchers, and institutions. The table below summarizes the key details:

No.	Resource	For Students/Researchers	For Institutions
Audio-Video e-content			
1	SWAYAM	Earn credit via online courses	Develop and host courses; accept credits
2	SWAYAMPRABHA	Access 24x7 TV programs	Enable SWAYAMPRABHA viewing facilities
Digital Content Access			
3	National Digital Library	Access e-content in multiple disciplines	List e-content; form NDL Clubs
4	e-PG Pathshala	Access free books and e-content	Host e-books
5	Shodhganga	Access Indian research theses	List institutional theses
6	e-ShodhSindhu	Access full-text e-resources	Access e-resources for institutions
Hands-on Learning			
7	e-Yantra	Hands-on embedded systems training	Create e-Yantra labs with IIT Bombay
8	FOSSEE	Volunteer for open-source software	Run labs with open-source software
9	Spoken Tutorial	Learn IT skills via tutorials	Provide self-learning IT content
10	Virtual Labs	Perform online experiments	Develop curriculum-based experiments
E-Governance			
11	SAMARTH ERP	Manage student lifecycle digitally	Enable institutional e-governance
Tracking and Research Tools			
12	VIDWAN	Register and access experts	Monitor faculty research outcomes
13	Shodh Shuddhi	Ensure plagiarism-free work	Improve research quality and reputation
14	Academic Bank of Credits	Store and transfer credits	Facilitate credit redemption

Table 1.1: Summary of ICT Initiatives by the Ministry of Education

1.2 FOSSEE Project

The FOSSEE (Free/Libre and Open Source Software for Education) project promotes the use of FLOSS tools in academia and research. It is part of the National Mission on Education through Information and Communication Technology (NMEICT), Ministry of Education (MoE), Government of India.

1.2.1 Projects and Activities

The FOSSEE Project supports the use of various FLOSS tools to enhance education and research. Key activities include:

- **Textbook Companion**: Porting solved examples from textbooks using FLOSS.
- **Lab Migration**: Facilitating the migration of proprietary labs to FLOSS alternatives.
- **Niche Software Activities**: Specialized activities to promote niche software tools.
- **Forums**: Providing a collaborative space for users.
- **Workshops and Conferences**: Organizing events to train and inform users.

1.2.2 Fellowships

FOSSEE offers various internship and fellowship opportunities for students:

- Winter Internship
- Summer Fellowship
- Semester-Long Internship

Students from any degree and academic stage can apply for these internships. Selection is based on the completion of screening tasks involving programming, scientific computing, or data collection that benefit the FLOSS community. These tasks are designed to be completed within a week.

For more details, visit the official FOSSEE website.

Figure 1.1: FOSSEE Projects and Activities

1.3 Osdag Software

Osdag (Open steel design and graphics) is a cross-platform, free/libre and open-source software designed for the detailing and design of steel structures based on the Indian Standard IS 800:2007. It allows users to design steel connections, members, and systems through an interactive graphical user interface (GUI) and provides 3D visualizations of designed components. The software enables easy export of CAD models to drafting tools for construction/fabrication drawings, with optimized designs following industry best practices [1, 2, 3]. Built on Python and several Python-based FLOSS tools (e.g., PyQt and PythonOCC), Osdag is licensed under the GNU Lesser General Public License (LGPL) Version 3.

1.3.1 Osdag GUI

The Osdag GUI is designed to be user-friendly and interactive. It consists of

- **Input Dock**: Collects and validates user inputs.
- **Output Dock**: Displays design results after validation.
- **CAD Window**: Displays the 3D CAD model, where users can pan, zoom, and rotate the design.
- **Message Log**: Shows errors, warnings, and suggestions based on design checks.

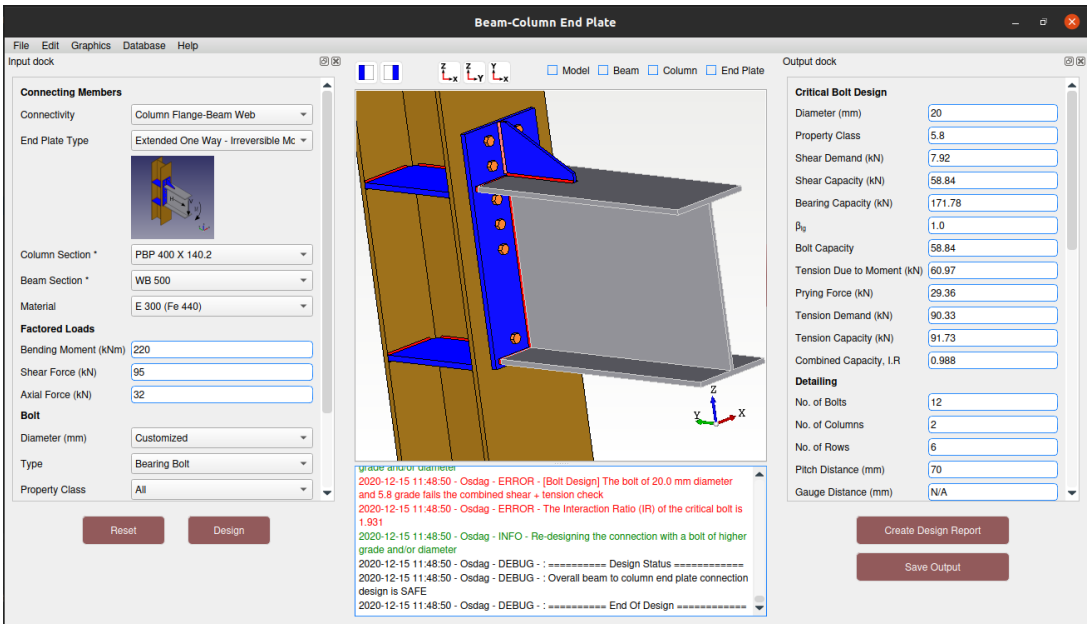

Figure 1.2: Osdag GUI

1.3.2 Features

- **CAD Model**: The 3D CAD model is color-coded and can be saved in multiple formats such as IGS, STL, and STEP.
- **Design Preferences**: Customizes the design process, with advanced users able to set preferences for bolts, welds, and detailing.
- **Design Report**: Creates a detailed report in PDF format, summarizing all checks, calculations, and design details, including any discrepancies.

For more details, visit the official Osdag website.

Chapter 2

Screening Task

2.1 Problem Statement

I attempted to develop a Python-based tool for structural steel design calculations in accordance with **IS 800:2007**, with a primary focus on bolted connections and tension member design. The intended deliverable combined a modular calculation engine, a lightweight SQLite datastore, and a PyQt5-based graphical user interface (GUI), with provisions for exporting results into structured reports.

While several key components were successfully initiated and partially implemented, the complete end-to-end workflow could not be fully realised within the screening duration. This chapter therefore documents what was developed, what functioned as intended, the gaps identified during implementation, and the proposed way forward.

2.2 Approach and Implementation

2.2.1 Task 1: Core Calculation Engine

The calculation engine was structured into small, testable units addressing butt-joint bolted connections and tension members. The objective was to compute plate requirements, bolt capacities (shear, bearing, and combined effects), and applicable reduction factors related to slip, bearing, and shear, while maintaining code clarity and long-term maintainability.

Intermediate abstractions were intentionally aligned with domain terminology—such

as net area, bolt bearing capacity, and slip reduction factors—to facilitate straightforward verification against the relevant clauses of **IS 800:2007**.

2.2.2 Task 2: Database and Configuration

A lightweight SQLite schema was drafted to store steel grades (yield and ultimate strengths), bolt series (diameters and strengths), and detailing parameters such as edge distances and pitch limits, along with configurable design constants.

The design intent was for the calculation engine to retrieve authoritative values directly from this datastore, thereby minimising hard-coded values and improving traceability and auditability of design decisions.

2.2.3 Task 3: GUI (PyQt5)

A prototype PyQt5-based graphical user interface was developed using a tabbed layout to logically group inputs related to material and member properties, loading and design factors, and connection geometry.

The UI scaffold incorporated guarded input fields with type and range validation, context-sensitive help text, and a reserved region for real-time design feedback. The layout followed a progressive disclosure philosophy, enabling novice users to proceed step-by-step without being overwhelmed by excessive inputs.

2.2.4 Task 4: Advanced Features (Planned and Partial)

Preliminary outlines for optimisation routines were created, targeting automated bolt selection and cover-plate sizing. A structured results panel was also planned to display calculated capacities, utilisation ratios, and compliance notes linked explicitly to relevant **IS 800:2007** clauses.

Longer-term extensions, such as CAD model generation and automated report templating, were conceptually planned but remained at a stub or placeholder stage during the screening task.

2.2.5 Task 5: Code Quality and Documentation

The codebase followed a layered architecture, clearly separating the user interface, calculation logic, and data access components. Consistent naming conventions were adopted, along with docstrings for public functions and pytest-ready test skeletons.

Input validation and exception-handling mechanisms were drafted to present user-friendly messages within the GUI while preserving detailed stack traces in log files for debugging and future development.

2.3 Outcomes

- A workable calculation skeleton was developed for key checks in bolted butt joints and tension members, with clearly identified placeholders where verification and validation are pending.
- A minimal SQLite dataset and corresponding access layer were implemented to centralise material and fastener properties.
- The PyQt5 user interface scaffold compiles and renders correctly, featuring validated input fields and a reserved area for design outputs.
- A fully verified, end-to-end design workflow could not be completed within the allotted screening period.

2.4 Notes on Code and Standards

All code identifiers and literals are presented using monospaced typewriter font, for example: `bolt shear capacity`, `IS 800:2007`, `SQLite`, and `PyQt5`. Indian English spellings are used consistently throughout the document (for example, optimisation, behaviour, and modelling), except where external API identifiers or library conventions require otherwise.

Chapter 3

System Modernization: PySide6 Migration

3.1 Task Overview

The primary objective of the Winter 2025 internship (commencing December 1, 2025) was to modernize the Osdag codebase, with a major focus on migrating the entire Graphical User Interface (GUI) framework from the dual-licensed **PyQt5** to the LGPL-compliant **PySide6 (Qt for Python)**.

This migration was not merely a technical upgrade but a strategic requirement for the Osdag ecosystem. PyQt5 is distributed under the GPL v3 license, which imposes restrictive conditions on downstream usage and integration. PySide6, being the official Qt binding maintained by The Qt Company, allows LGPL-based distribution, enabling Osdag to be linked and distributed more freely. Ensuring licensing compliance and long-term sustainability of the project was therefore a primary objective of the FOSSEE team.

3.2 Technical Strategy

The Osdag codebase consists of over 6,000 files, making a direct or naive migration approach infeasible. The migration strategy was therefore divided into clearly defined phases to ensure correctness, traceability, and minimal regression.

3.2.1 Dependency Mapping

Regex-based search tools such as `grep` and `find` were used to identify every instance of `PyQt5` imports across the repository. This process resulted in the identification of more than 75 core source files that directly depended on PyQt5 and required systematic modification.

3.2.2 API Translation and Refactoring

A structured reference mapping was created to translate PyQt5-specific constructs—such as signals, slots, decorators, and enums—into their PySide6 equivalents. This ensured consistency across modules and reduced the risk of subtle runtime incompatibilities during refactoring.

3.2.3 Resource Re-compilation Strategy

Osdag relies heavily on Qt resource files (`.qrc`) for icons and graphical assets. A plan was established to regenerate these resources using the PySide6-compatible compiler `pyside6-rcc`, replacing the legacy `pyrcc5` workflow.

3.3 Implementation Details

3.3.1 GUI Migration and Source Refactoring

More than 75 source files were successfully migrated, covering core UI infrastructure (such as `osdagMainPage.py`) as well as individual connection modules including Fin Plate and End Plate connections.

All import statements were updated from PyQt5 to their corresponding PySide6 modules. Custom signal definitions were refactored by replacing `pyqtSignal` with `Signal`, and slot decorators were updated from `@pyqtSlot` to `@Slot`. Each migrated signal was manually verified to ensure correct data emission and reception.

3.3.2 Enum Access Refactoring

One of the most time-consuming aspects of the migration involved refactoring enum access patterns. PyQt5 permitted flat enum access (for example, `Qt.AlignLeft`), whereas PySide6 enforces scoped enum usage (for example, `Qt.AlignmentFlag.AlignLeft`).

Hundreds of enum references—including alignment flags, window hints, and keyboard modifiers—were audited and updated across the codebase. This process resolved multiple runtime `AttributeError` issues that emerged during early testing.

3.3.3 Qt Resource Compilation

The legacy resource compiler `pyrcc5` was replaced with the PySide6-compatible command:

```
pyside6-rcc icons.qrc -o icons_rc.py
```

Because the generated resource module differed internally from its PyQt5 counterpart, all dependent GUI files were updated to import icons from the new resource module. This ensured that visual elements such as toolbar icons and buttons retained their intended appearance after migration.

3.4 Challenges and Solutions

3.4.1 OCC Visualisation Crashes and Dark Screen Issues

After migration, the Open CASCADE (OCC) 3D viewer failed to initialise on certain Windows systems, particularly those with high-DPI displays or incompatible GPU drivers. This resulted in application crashes or a dark blue screen where the 3D model was expected.

To improve robustness, the OCC initialisation logic was wrapped in a comprehensive exception-handling block. When OCC failed to initialise, the application now displays a user-friendly message indicating that the 3D preview is unavailable, while allowing users to continue with calculations and report generation.

3.4.2 White Screen Issue in Column End Plate Module

A persistent white screen issue was encountered in the `ColumnEndPlate` module, where the interface froze without logging an error. Execution trace analysis revealed that the failure occurred during the initial `paintEvent`, caused by access to an uninitialised attribute (`flange_bolt_spacing`).

This issue was resolved by explicitly initialising the variable at the beginning of the class constructor. The fix ensured that all required attributes existed in a safe default state before any rendering events were triggered, effectively eliminating the race condition introduced by PySide6's faster event handling.

3.5 UI/UX Enhancements and Bug Fixes

Several additional UI and usability issues were addressed during the migration:

- Resolved invisible dock headings caused by stylesheet incompatibilities across themes.
- Fixed output dock blanking issues encountered during screen recording and when loading specific `.osi` files.
- Improved responsiveness and layout stability of refactored UI components.

3.6 3D Visualisation and CAD Improvements

Work was undertaken to stabilise and improve the 3D rendering pipeline:

- Fixed blank and blue-screen rendering issues in Fin Plate and End Plate modules.
- Integrated availability checks for the Open CASCADE backend to provide meaningful user feedback.
- Restored CAD interaction features such as hover detection for bolts and plates in Beam-to-Beam End Plate connections.

3.7 Test Suite Refactoring and Automation

3.7.1 Database Abstraction and Test Isolation

To decouple tests from machine-specific database configurations, a mocking layer was implemented using `unittest.mock`. Mock database connections and cursors were defined in `conftest.py`, allowing engineering calculations to execute without accessing physical SQLite files.

Virtual schema definitions were embedded into the mock layer to simulate standard steel sections and fastener data, enabling realistic and repeatable test execution.

3.7.2 Implementation of the OSI Standard

Legacy test inputs were migrated to the `.osi` format, a structured JSON-based standard for representing structural input data. Test runners were enhanced to automatically validate execution flags and result dictionaries returned by the `osdag.cli.run_module` interface.

3.7.3 Regression Testing and Expected Failures

Complex engineering edge cases that require further refinement were documented using `@pytest.mark.xfail`. This allowed continuous integration pipelines to proceed while explicitly tracking known limitations for future resolution.

Chapter 4

Testing Infrastructure: CLI & Mocking Strategy

4.1 The Testing Problem

Historically, testing in Osdag was a predominantly manual and GUI-driven process. Developers were required to launch the full application, wait for the graphical interface to initialise, navigate through multiple layers of menus to reach a specific connection module, manually enter dozens of design parameters, and then visually verify the numerical results displayed in the output dock against reference spreadsheets. This manual testing approach introduced several critical limitations. First, it was extremely time-consuming, with a single test case often taking up to five minutes to complete. Second, it was fragile in nature—minor variations such as window resizing, display scaling, or differences in local database paths could cause inconsistent behaviour. Most importantly, this workflow was fundamentally un-automatable. GUI-based testing cannot be reliably executed in a CI/CD environment such as GitHub Actions, making it impossible to run regression tests automatically on every code change. As a result, defects and regressions were frequently introduced and only detected much later in the development cycle.

4.2 Implemented Solution: Headless Testing using `osdag.cli`

To overcome these limitations, the testing strategy was redesigned around the concept of **headless testing**. This was achieved by utilising and extending the Osdag Command Line Interface (CLI) module, `osdag.cli`, which allows the core engineering calculation engine to be executed independently of the graphical user interface.

The workflow implemented through the CLI follows a clear and reproducible pattern:

1. **Input**: A test case supplies a standardised `.osi` (Open Structural Input) file containing all user inputs in a structured key–value format.
2. **Processing**: The CLI parses and validates the input file, then instantiates the relevant backend module (for example, Fin Plate or End Plate connections) in a headless mode. A lightweight dummy object is created to mimic the presence of the UI without rendering any graphical elements.
3. **Output**: The function returns a pure Python dictionary containing the complete design state, including success flags, governing capacities, and intermediate calculation results.

This approach fundamentally transformed the testing paradigm. Instead of checking whether a GUI label displayed the correct text, tests could directly assert numerical correctness, for example verifying that a computed bolt capacity matched an expected value. The result was a testing framework that is fast, precise, repeatable, and fully compatible with automation pipelines.

4.3 Advanced Mocking Strategy using `MagicMock`

A major challenge in unit testing Osdag is its dependency on external systems. The application interacts extensively with PySide6 GUI widgets, which require a display server, and with SQLite databases, which introduce file I/O and environment-specific dependencies. Running such components directly during tests is both slow and unreliable.

To isolate the engineering logic from these external dependencies, the `unittest.mock` framework—specifically `MagicMock`—was used extensively. Mocking enables the creation of lightweight stand-ins for real objects, allowing the code to execute as if the dependencies were present, without actually invoking them.

Within `conftest.py`, GUI-related components such as `QMessageBox` were patched using `MagicMock`. For example, when the application attempts to call `QMessageBox.warning()` due to invalid user input, the mocked object simply records the call and allows execution to continue. This makes it possible to test whether the warning logic was triggered, without requiring an actual GUI environment or causing test failures due to missing display contexts.

4.4 The Mock Database Layer

One of the most significant enhancements to the testing infrastructure was the introduction of a fully mocked database layer. The production Osdag application relies on a large SQLite database (over 500 MB) to retrieve section properties, material strengths, and detailing parameters. Including this database in automated tests is impractical and severely slows down execution.

4.4.1 The `MockCursor` and `MockConnection` Design

To address this, a custom `MockCursor` class was implemented in `tests/conftest.py`. This class replicates the behaviour of a standard `sqlite3` cursor but operates entirely in memory using predefined datasets.

When an engineering module executes a query such as:

```
SELECT * FROM Columns WHERE Designation = 'ISMB 400'
```

the query string is intercepted by the `MockCursor`. The query is parsed to determine the requested entity type (beams, columns, angles, or bolts), and a deterministic, hard-coded response is returned that mirrors the structure of the real database.

This approach ensures that:

- Tests are completely independent of local database files.

- Every test executes against identical data on all systems.
- Execution time is reduced from minutes to milliseconds.

A corresponding `MockConnection` class was used in combination with `MagicMock` to globally redirect all database connections during testing.

4.5 CLI Automation and the OSI Workflow

The `.osi` file format plays a central role in enabling headless testing. It acts as a bridge between the graphical interface and the backend calculation engine by standardising how input data is represented.

Each `.osi` file defines:

- The target design module (e.g., Fin Plate, Cleat Angle),
- Member configuration and connectivity,
- Applied loads,
- Connection geometry and detailing parameters.

By storing these files in a dedicated test fixtures directory, Osdag gains a permanent, reproducible record of test cases. The same inputs can be reused for regression testing or shared with external developers to reproduce reported issues.

4.6 Developer Manual Contributions

To ensure long-term maintainability, all testing-related workflows were documented in the Osdag Developer Manual. This documentation serves as a reference for future interns and contributors and explains how to extend the mocking framework, add new virtual database schemas, create `.osi` test fixtures, and use the CLI for rapid verification.

Additional guidance was also provided on PySide6-specific development practices, including correct signal–slot usage and updated dialog execution patterns, to prevent common runtime errors introduced during the framework migration.

4.7 Resulting Impact

The testing infrastructure introduced during this phase represents a significant shift in how Osdag is validated and maintained. By transitioning from manual, GUI-driven verification to a fully automated, headless testing framework, the development process became substantially more reliable, scalable, and transparent.

The use of the CLI-based execution model ensured that engineering logic could be tested independently of the graphical interface. This separation reduced false negatives caused by UI-related issues and enabled developers to focus on verifying numerical correctness and governing design conditions. As a result, regression testing became faster and more precise, allowing changes in the codebase to be validated within seconds rather than minutes.

The introduction of a comprehensive mocking strategy further strengthened test reliability. By eliminating dependencies on external systems such as GUI event loops and disk-based SQLite databases, tests became fully environment-independent. This ensured consistent behaviour across different operating systems and development setups, significantly reducing the time spent debugging environment-specific failures.

From a project-level perspective, the new testing framework lays a strong foundation for continuous integration and long-term maintainability. Automated tests can now be executed as part of CI/CD pipelines, enabling early detection of regressions and improving overall code confidence. Additionally, the documented workflows and reusable test fixtures lower the onboarding barrier for future contributors, allowing them to add features or refactor existing modules with greater assurance of correctness.

Overall, this work establishes a sustainable testing culture within the Osdag codebase, transforming testing from a manual afterthought into an integral part of the development lifecycle.

Chapter 5

OSI Standardization & Data Migration

5.1 The Shift to OSI Format

At the start of the internship, I observed a critical mismatch between how Osdag was being tested and how it was actually used in practice. A small number of existing developer tests relied on ad-hoc input formats, primarily plain text (`.txt`) files with loosely defined, comma-separated values. In contrast, real users of Osdag interact with the software through the graphical interface and save their work using the native `.osi` format.

This discrepancy posed a serious risk. It meant that the test suite exercised a code path that was different from the one used in production. In practical terms, the application's file loader could fail for real users while automated tests continued to pass, giving a false sense of correctness. Such divergence undermines confidence in testing and defeats the purpose of regression verification.

To address this issue, a deliberate architectural shift was introduced. On **January 5, 2026**, I committed a major change titled *"changing input file extension to osi"*, marking the transition to a unified input standard. This change was not limited to renaming files; it represented a formal standardization of the data contract between the Osdag application and its automated test suite.

The `.osi` (Osdag Input) format is the native, structured format used internally by Osdag's save and load functionality. By standardising all tests to consume `.osi` files, the test framework now uses the exact same parsing logic (`load_design_inputs`) as the

production GUI. This ensured that any failure in file handling, schema evolution, or input validation would be immediately detected during automated testing.

An additional advantage of this approach is improved maintainability. When a test fails, developers can directly open the corresponding `.osi` file in the Osdag GUI to visually inspect and debug the input parameters. This dramatically reduces debugging time and eliminates the need to interpret opaque, custom-formatted text files.

5.2 Implementation of Module-Level Tests using OSI

Once the `.osi` format was adopted as the standard test input mechanism, it became possible to design structured, readable, and extensible test suites for individual connection modules. The file `test_fin_plate.py` represents the first complete implementation of this new workflow and serves as a reference pattern for future tests.

Each test follows a clear and consistent structure. Input data is stored in a dedicated `.osi` file (for example, `FinPlateTest1.osi`), which contains all relevant design parameters such as member sections, bolt properties, loads, and geometric constraints. The test script uses standard Python file handling to pass the input file path directly to the CLI interface.

The validation logic adheres to the widely accepted **Arrange–Act–Assert** testing pattern. Expected values, such as shear capacity or governing bolt strength, are defined during the arrange phase. The CLI execution (`osdag.cli.run_module`) performs the full design computation in headless mode during the act phase. Finally, the numerical outputs returned in the result dictionary are compared against the expected values using precise assertions.

This structure makes each test case self-documenting: the `.osi` file defines the engineering scenario, while the test script clearly defines the expected software behaviour for that scenario.

5.3 Validation Coverage and Results

Using the OSI-based testing framework, a total of twelve distinct test cases were implemented and verified across three major Osdag modules, providing broad functional

coverage of the application.

For the **Fin Plate Connection** module, tests were written to verify key limit states and checks, including bolt shear capacity, applicable reduction factors, and Von Mises stress criteria. The numerical results produced by Osdag were mapped directly against benchmark Excel spreadsheets provided by senior structural engineers, ensuring alignment with established manual calculations.

In the **Cleat Angle Connection** module, the focus was on geometric feasibility and detailing logic. The tests verify that the software correctly evaluates whether a selected angle section can physically fit within the beam web. This addresses a common class of user errors and ensures that invalid configurations are reliably rejected by the design engine.

For the **Tension Member** module, more complex limit states were validated, including net section rupture (T_{dn}) and block shear failure (T_{db}). These checks involve intricate logic related to the number of shear planes, rupture paths, and effective areas. The test cases confirmed that the implementation matches manual calculations and behaves consistently across different configurations.

All of these tests were integrated into the central repository and committed on **December 30, 2025** under the addition of the `tests` directory. Together, they form a stable baseline for future development, refactoring, and regression testing within the Osdag codebase.

5.4 Impact of OSI Standardization

The adoption of the OSI format as the single source of truth for both user interaction and automated testing significantly improved the reliability and credibility of the test suite. By eliminating parallel input formats, the risk of silent divergence between tested behaviour and real-world usage was removed.

More importantly, this standardization enables long-term scalability. New modules can now be tested by simply adding `.osi` fixtures and corresponding assertions, without introducing new parsing logic. This work establishes a strong foundation for consistent data handling, easier debugging, and more confident evolution of the Osdag platform.

Chapter 6

Conclusions

6.1 Tasks Accomplished

During the Winter Internship conducted from **December 1, 2025** to **January 31, 2026**, I successfully completed all major objectives defined by the FOSSEE team. The work focused on modernising the Osdag software stack, stabilising the graphical user interface, and establishing a robust and scalable testing infrastructure.

A primary accomplishment was the complete migration of the Osdag graphical user interface from the legacy **PyQt5** framework to **PySide6 (Qt for Python)**. This effort involved systematic refactoring across more than fifty core source files, replacing deprecated imports, updating event-handling mechanisms, and resolving framework-specific incompatibilities. During this process, I identified and fixed deep-seated threading and initialisation issues that caused application crashes and "white screen" failures during splash screen rendering and main window loading. Signal–slot connections were rewritten to strictly adhere to PySide6 type-checking requirements, ensuring reliable behaviour of interactive components.

In parallel, I engineered an advanced automated testing environment using **pytest**. This included the design and implementation of a custom database mocking layer using `MockCursor` and `MockConnection`, combined with extensive use of `MagicMock` to isolate application logic from GUI and disk-based dependencies. This infrastructure enables fast, deterministic, and database-free testing—capabilities that were previously unavailable in the project.

Additionally, I standardised all developer-side test inputs by migrating legacy input

formats to the native `.osi` (Open Structural Input) format. This alignment ensured that automated tests exercise the same data-loading pipeline as the production GUI and CLI, reducing the risk of silent regressions caused by divergent input paths. Known engineering edge cases were documented using expected-failure (XFAIL) markers to clearly distinguish between genuine regressions and acknowledged design limitations.

6.2 Skills Developed

This internship served as an intensive real-world learning experience in maintaining and extending a large open-source engineering software system.

6.2.1 Technical Skills

I gained specialised expertise in desktop application development using **PySide6**, including a deep understanding of the Qt event loop, widget hierarchies, and the Model–View–Controller (MVC) pattern. I learned to diagnose and resolve complex UI issues such as frozen interfaces and rendering deadlocks.

In the domain of software testing, I moved beyond basic unit testing to master advanced techniques involving mocking and dependency injection. I developed proficiency in `unittest.mock`, particularly the use of `MagicMock` and patching strategies to inject fake dependencies into a running system. I also strengthened my skills in **pytest**, including the effective use of fixtures, parametrisation, and markers to build maintainable regression suites.

Working within a refactored legacy codebase improved my ability to read, understand, and safely modify existing infrastructure. I adopted a test-first mindset when making changes to critical components such as database connectors and input loaders, ensuring behavioural stability throughout the refactoring process.

6.2.2 Professional Skills

I assumed clear ownership of the PySide6 migration effort, making architectural decisions related to shared modules such as resource handling and common utilities. This experience strengthened my ability to evaluate trade-offs and implement solutions with long-term maintainability in mind.

Contributing to the Osdag Developer Manual enhanced my technical documentation skills. I learned to convert complex internal behaviours into clear, actionable guidance for future contributors, recognising that maintainable software depends as much on documentation as on code quality.

6.3 Overall Impact

The contributions made during this internship act as a bridge between the legacy Osdag system and its future evolution. Migrating to PySide6 ensures long-term compatibility with modern Python versions and operating systems, while also resolving licensing constraints associated with the previous framework.

The introduction of a mock-based, headless testing framework has fundamentally changed the development workflow. Verification time has been reduced from hours of manual effort to seconds of automated execution, enabling rapid iteration with high confidence. New contributors can now run tests without complex environment setup, significantly improving developer onboarding and productivity.

Overall, the work completed during this internship has resulted in a more stable, maintainable, and future-ready Osdag codebase. The shift from reactive bug fixing to proactive infrastructure design has given me a strong appreciation for software architecture and tool development. This experience has solidified my interest in pursuing a career in software engineering, with a particular focus on building reliable, developer-centric engineering tools.

Chapter A

Appendix

A.1 Work Reports

Internship Work Report

Name:	**Agam Arora**
Project:	**OSDAG**
Internship:	**FOSSEE Winter Internship 2025**

DATE	DAY	TASK	Hours Worked
01-Dec-2025	Monday	Internship onboarding; complete walkthrough of Osdag repository and understanding overall project structure	4
02-Dec-2025	Tuesday	Studied existing PyQt5-based GUI architecture and identified major UI entry points	4
03-Dec-2025	Wednesday	Analysed PyQt5 dependencies across 6000+ files and documented migration impact	4
04-Dec-2025	Thursday	Set up Conda environment; installed PySide6 and verified compatibility	4
05-Dec-2025	Friday	Mapped PyQt5 imports and prepared migration plan for core UI files	4
08-Dec-2025	Monday	Began PyQt5 to PySide6 migration; updated basic imports and resolved syntax errors	4
09-Dec-2025	Tuesday	Regenerated resource files using pyside6-rcc and validated icon loading	4
10-Dec-2025	Wednesday	Migrated osdagMainPage.py to PySide6 and fixed runtime import issues	4
11-Dec-2025	Thursday	Debugged signal–slot failures caused by strict PySide6 type checking	4
12-Dec-2025	Friday	Tested migrated UI components and stabilised event loop behaviour	4
15-Dec-2025	Monday	Investigated OCC 3D viewer crash (“Dark Blue Screen”) on Windows systems	4
16-Dec-2025	Tuesday	Implemented fallback mechanism for OCC visualisation failures	4

Internship Work Report

Name:	**Agam Arora**
Project:	**OSDAG**
Internship:	**FOSSEE Winter Internship 2025**

DATE	DAY	TASK	Hours Worked
17-Dec-2025	Wednesday	Debugged “White Screen” crash in Column End Plate module	4
18-Dec-2025	Thursday	Fixed variable initialisation order causing rendering deadlock	4
19-Dec-2025	Friday	Verified stability of GUI after crash fixes	4
22-Dec-2025	Monday	Designed folder structure for automated testing (tests and fixtures)	4
23-Dec-2025	Tuesday	Designed architecture for database-free testing using mocking	4
24-Dec-2025	Wednesday	Implemented initial MockCursor class in conftest.py	4
26-Dec-2025	Friday	Prototyped CLI-based execution using osdag.cli	4
29-Dec-2025	Monday	Integrated pytest framework with Osdag backend	4
30-Dec-2025	Tuesday	Commit: Added tests folder and stabilised test execution pipeline	4
31-Dec-2025	Wednesday	Implemented MagicMock to isolate GUI and database dependencies	4
05-Jan-2026	Monday	Commit: Standardised test inputs to OSI format	4
06-Jan-2026	Tuesday	Converted legacy text inputs into YAML-based OSI files	4
07-Jan-2026	Wednesday	Updated CLI parser to correctly load and validate OSI files	4
08-Jan-2026	Thursday	Verified OSI-based test execution across multiple modules	4

Internship Work Report

Name:	**Agam Arora**
Project:	**OSDAG**
Internship:	**FOSSEE Winter Internship 2025**

DATE	DAY	TASK	Hours Worked
12-Jan-2026	Monday	Integrated CLI execution with pytest harness	4
13-Jan-2026	Tuesday	Developed test cases for Cleat Angle module using OSI fixtures	4
14-Jan-2026	Wednesday	Refactored test imports and improved separation of concerns	4
19-Jan-2026	Monday	Validated Fin Plate module against benchmark Excel sheets	4
20-Jan-2026	Tuesday	Validated Tension Member module including rupture and block shear checks	4
21-Jan-2026	Wednesday	Executed full regression test suite and analysed results	4
22-Jan-2026	Thursday	Achieved stable test pass across all tested configurations	4
26-Jan-2026	Monday	Documented testing workflow for Osdag Developer Wiki	4
27-Jan-2026	Tuesday	Authored and refined internship report chapters	4
28-Jan-2026	Wednesday	Final documentation review and formatting cleanup	4
29-Jan-2026	Thursday	Internal review with mentors and applied feedback	4
30-Jan-2026	Friday	Final submission preparation and verification	4

Bibliography

[1] Siddhartha Ghosh, Danish Ansari, Ajmal Babu Mahasrankintakam, Dharma Teja Nuli, Reshma Konjari, M. Swathi, and Subhrajit Dutta. Osdag: A Software for Structural Steel Design Using IS 800:2007. In Sondipon Adhikari, Anjan Dutta, and Satyabrata Choudhury, editors, *Advances in Structural Technologies*, volume 81 of *Lecture Notes in Civil Engineering*, pages 219–231, Singapore, 2021. Springer Singapore.

[2] FOSSEE Project. FOSSEE News - January 2018, vol 1 issue 3. Accessed: 2024-12-05.

[3] FOSSEE Project. Osdag website. Accessed: 2024-12-05.