SOEN 6471
ADVANCED SOFTWARE ARCHITECTURES
SUMMER 2024

Deliverable 3
ECLIPSE

Declaration

We, the members of the team, have read and understood the Fairness Protocol and the Communal Work Protocol, and agree to abide by the policies therein, without any exception, under any circumstances, whatsoever.

Team A
Duy Than Phan
Dhruvi Patel
Neha Sanjay Deshmukh
Mutasimur Tasin

Table of Contents

Problem 6

There are many patterns, style principles and tactics present in Eclipse's software architecture. There are also undesirable traits as well. The following subsections list theseout.

(A)Desirable Traits

1. Layered architecture:

Eclipse is structured in a layered manner, facilitating modularity and extensibility, where different components and services are organized into layers, each responsible for specific functionalities. For example, there could be layers for user interface, language support, project management, and debugging, allowing for separation of concerns and promoting maintainability.

2. Separation of Concerns (SoC) Principle:

Eclipse emphasizes the separation of concerns, dividing the software system into distinct modules or plugins. Each module focuses on a specific functionality, promoting modularity, maintainability, and code reusability.

3. Dependency Injection (DI) and Inversion of Control (IoC):

These tactics facilitate loose coupling, testability, and flexibility in managing component dependencies. With DI, an external entity (like a framework or custom code) injects the required dependencies into your components.

4. Event-Driven Architecture:

Eclipse follows an event-driven architecture, where various components and plugins can subscribe to and emit events. This enables loose coupling between components, facilitating extensibility and modularity.

5. Acyclic Dependency Principle (ADP):

Eclipse IDE maintains a clean and modular architecture, facilitating easier maintenance, extension, and overall better software engineering practices. Using interfaces and abstractions allows for loose coupling between components. This means components depend on functionalities defined in interfaces, not on specific implementations.

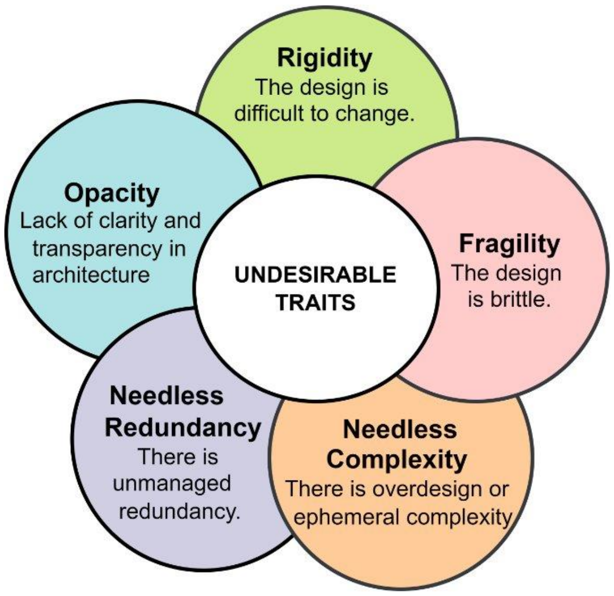
6. Stable Dependency Principle (SDP):

By adhering to the Stable Dependency Principle, Eclipse IDE achieves a robust and maintainable system architecture, where dependencies are clear, predictable, and easier to manage.

7. Open-Closed Principle (OCP):

Eclipse IDE allows for a highly extensible and flexible development environment, where new features and customizations can be added seamlessly, maintaining the integrity and stability of the core system.

(B) Undesirable Traits


Rigidity
The design is difficult to change.
Opacity
Lack of clarity and transparency in architecture
UNDESIRABLE TRAITS
Fragility
The design is brittle.
Needless Redundancy
There is unmanaged redundancy.
Needless Complexity
There is overdesign or ephemeral complexity

A trait, pattern, or part that is deemed harmful or unwanted for the overall quality and efficacy of the architecture can be referred to as undesirable in a software architecture. These undesirables can make the software architecture unstable, unsupportive, and/or ugly.

Undesirables may appear as architectural odours, anti-patterns, or design defects, among other manifestations. In the context of the Eclipse software architecture, we refer to odours as potential design flaws or areas for development. These odours can be used to spot architectural flaws that might impair the Eclipse system's ability to be scaled,maintained, or otherwise be of high quality.

Here are some common smells in Eclipse software architecture:

1. **Rigidity**:

- Rigidity is a smell that suggests the architecture is inflexible and resistant to change.
- Eclipse relies heavily on a vast ecosystem of plugins. If the interface or behavior of the core system changes, many plugins might need updates, which increases the effort and coordination required for any change.

2. **Fragility**

- Fragility refers to the tendency of the system to break in unexpected ways when changes are made. In Eclipse, this smell can arise when modifying one part of the system inadvertently affects unrelated components due to hidden dependencies or improper isolation.

3. **Needless complexity**

- Needless complexity refers to architectural designs that are overly complicated, introducing unnecessary intricacies and making the system harder to understand and maintain.

- As a long-standing and continuously evolving project, Eclipse has accumulated legacy code and technical debt, which can add to the complexity.

4. **Needless redundancy**

- Needless redundancy occurs when multiple components or functionalities perform similar or identical tasks, resulting in duplicated effort and increased maintenance overhead.

- Over time, the codebase of Eclipse can accumulate redundant code, especially as new features are added, and older ones are deprecated or forgotten.

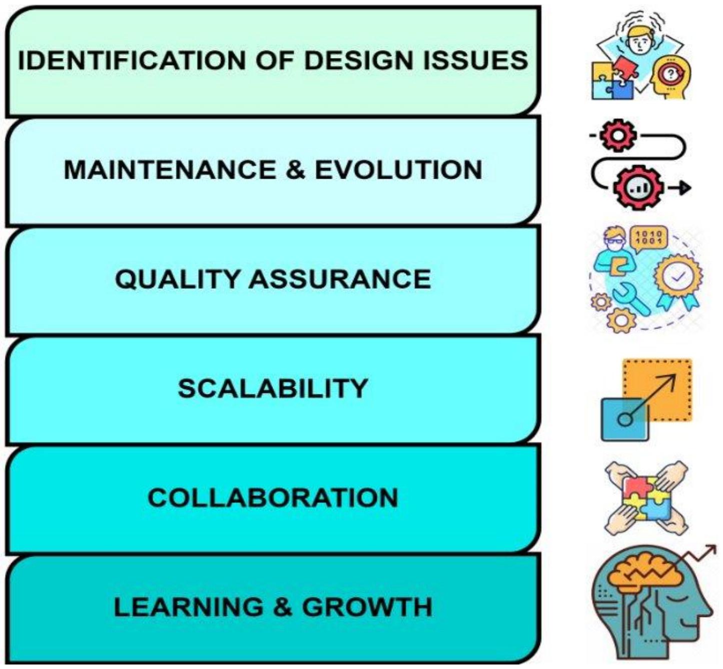
5. **Opacity**

- Opacity refers to a lack of clarity and transparency in the architecture, making it difficult to understand the system's behaviour and dependencies.

- The interactions between different components and plugins can sometimes lead to hidden dependencies and side effects that are not immediately apparent.

Uses of Undesirable Traits of Software Architecture

There are some uses of knowing the undesirables in a Software Architecture as well. Let’s have a look at those.



1. **Identification of Design Issues**

- It helps developers and architects identify potential design issues or weaknesses in the architecture. By recognizing these smells, they can proactively address them, improving the overall quality of the software system.

2. **Maintenance and Evolution**

- Smells often indicate areas of the architecture that are difficult to maintain or modify. By being aware of these smells, developers can prioritize refactoring efforts, making the codebase more maintainable, extensible, and adaptable to future changes.

3. **Quality Assurance**

- When conducting code reviews or quality assurance activities, knowledge of architectural smells enables developers to identify and provide feedback on potential issues early in the development process. It promotes better code quality and adherence to architectural best practices.

4. **Scalability**

- Smells related to performance bottlenecks or inefficiencies in the architecture can guide developers in optimizing critical paths and ensuring the system meets performance requirements. Understanding these smells helps in identifying and addressing potential scalability issues as well.

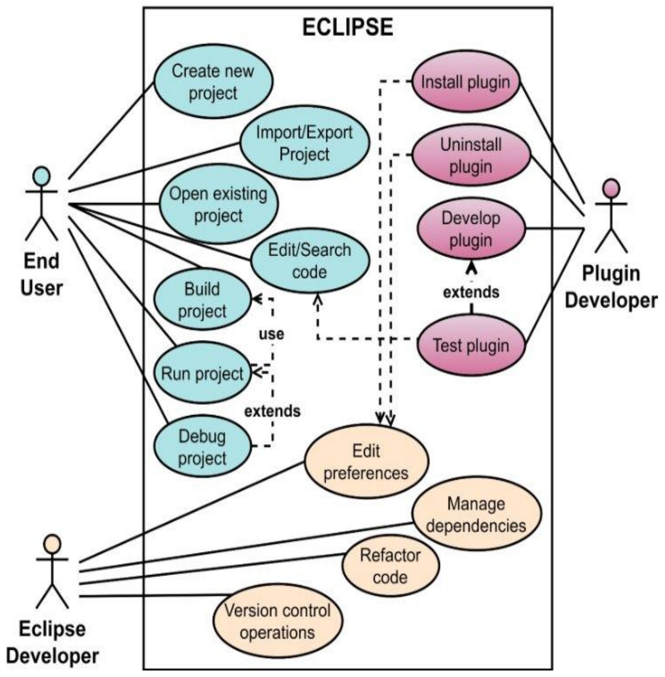
5. **Collaboration**

- It enhances collaboration among developers and architects. By having a shared understanding of common smells, team members can effectively discuss architectural decisions, identify areas for improvement, and communicate the rationale behind refactoring efforts.

6. **Learning and Growth**

- Understanding smells in the architecture of Eclipse provides a valuable learning opportunity for developers. By analysing and addressing these smells, developers gain experience in architectural design principles, software engineering best practices, and the nuances of building complex software systems.

Problem 7



Actor 1: End User

Use Case 1 : Developing a Java Application

- **Setting Up the IDE:** The end user downloads and installs Eclipse IDE on their computer.
- **Creating a New Project:** The end user creates a new Java project in Eclipse.
- **Writing Code:** The end user writes Java code in the editor, utilizing features like syntax highlighting, code completion, and real-time error detection.
- **Running and Debugging:** The end user compiles and runs their Java application directly within the IDE. They also use Eclipse's debugging tools to set breakpoints, step through code, and inspect variables.
- **Using Version Control:** The end user integrates their project with a version control system (e.g., Git) using Eclipse's built-in tools for committing, pushing, and pulling code changes.
- **Utilizing Plugins:** The end user installs additional plugins for functionality like enhanced code editing, project management, and other development tools.

Actor 2: Eclipse IDE Developer

Use Case 2 : Enhancing Core Features of Eclipse

- **Developing Core Features:** The Eclipse IDE developer works on the core components of Eclipse, such as the Java Development Tools (JDT) or the Platform UI.
- **Maintaining the Codebase:** The developer checks out the source code of Eclipse from the repository, makes necessary enhancements or fixes bugs, and commits the changes back to the repository.
- **Building and Testing:** The developer builds the Eclipse IDE from source and runs extensive tests to ensure stability and performance. They write unit tests and integration tests to cover new and existing functionality.
- **Handling Contributions:** The developer reviews and integrates contributions from other developers, ensuring they meet the project's coding standards and quality requirements.
- **Releasing Updates:** The developer participates in the release process of new Eclipse versions, packaging the IDE, and ensuring that all features are properly documented and tested.

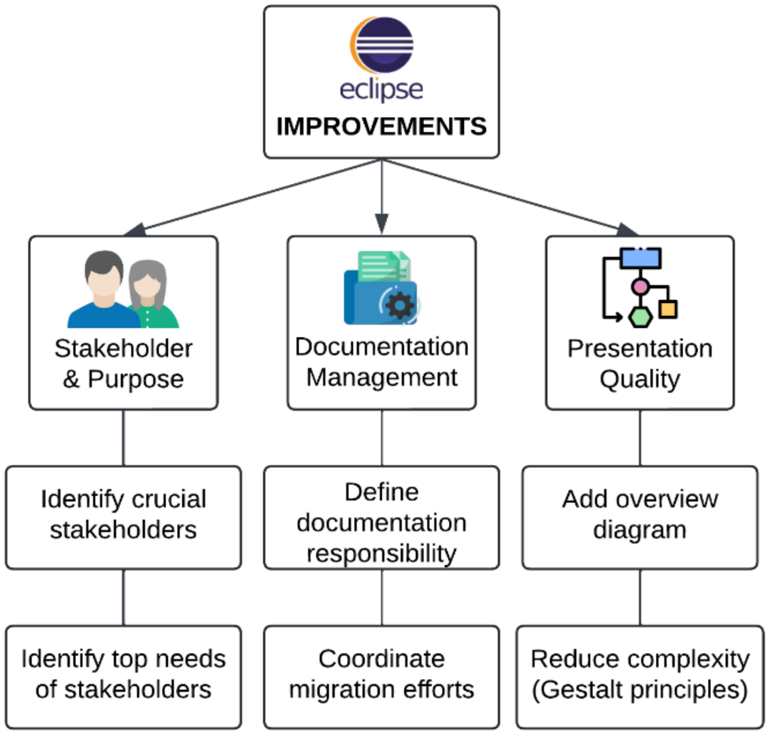
Actor 3: Plugin Developer

Use Case 3 : Creating a New Plugin for Eclipse

- **Setting Up Development Environment:** The plugin developer sets up Eclipse IDE for Eclipse Plugin Development Environment (PDE), including necessary dependencies.
- **Creating a Plugin Project:** The developer creates a new plugin project using the PDE tools provided by Eclipse.
- **Developing Plugin Features:** The developer writes code to implement the desired functionality of the plugin, such as adding new views, editors, or project types to the IDE.
- **Testing the Plugin:** The developer uses Eclipse's runtime workbench to test the plugin in a sandbox environment, ensuring that it integrates well with existing features of the IDE.
- **Packaging and Distribution:** Once the plugin is developed and tested, the developer packages it into a deployable format and distributes it through Eclipse Marketplace or other distribution channels.
- **Providing Documentation:** The developer creates comprehensive documentation to help end users understand how to install and use the plugin.

Problem 8

Future Improvements In Eclipse Architecture


eclipse
IMPROVEMENTS
Stakeholder & Purpose
Documentation Management
Presentation Quality
Identify crucial stakeholders
Define documentation responsibility
Add overview diagram
Identify top needs of stakeholders
Coordinate migration efforts
Reduce complexity (Gestalt principles)

Using the Question Framework for Architectural Description Quality Evaluation, we have arrived at three possible improvement suggestions:

1. **Stakeholder and Purpose Orientation**

- We suggest that Eclipse needs to identify the stakeholders and their needs better. This is because of two reasons.

- Although Eclipse is popular, more and more developers have moved on to other better IDEs such as IntelliJ, and Eclipse is no longer the most popular IDE for Java development. The core reason for this is the needs of stakeholders are not being met. For example, many developers claim Eclipse to be hard to use and has an unfriendly user interface. So, these needs should be identified and addressed.

- Although Eclipse has done a decent job in presenting the documentation for both contributors and end-users like developers, we found that there is little to no public documentation for other stakeholders such as investors, external organizations. This is crucial because Eclipse Foundation, the one behind Eclipse IDE, operates based on the financial contribution of participant companies. Presenting documentation or white papers for such target demographics would allow more organizations to join and the long-term longevity of Eclipse ecosystems.

2. **Documentation Management**.

- We suggest that Eclipse should define the documentation responsibility clearly and have people who are willing to coordinate the documentation migration efforts among different teams.
- Since Eclipse is made of dozens of submodules, we found each submodule corresponding to a repository. Although each repository created a general guideline for contribution, the responsibilities of documentation maintainers are unclear and not made transparent. We also found that Eclipse is in the process of migrating the documentation to new platforms, but the migration is being performed by each team of repositories individually very slowly.
- Having clear responsibilities and better coordination would reduce time spending on updating the documentation and allow the contributors and end-users to have an easier time for understanding each concrete part of the Eclipse platform.

3. **Presentation and Visualization Quality**

- We suggest that Eclipse lower the complexity of the architecture description. We found many components of Eclipse don't have diagrams to illustrate their structure and relationship at a high level. Adding diagrams can help readers track all the elements and their relationship by minimizing the maximum number of interactions between elements at once. It can be applied by following some Gestalt principles, such as those for proximity, common fate, closure and similarity, increasing the readability and presentation value.

Problem 9

Retrospective Lessons Learned from D1 and D2

We have learned 3 lessons:

1. Have concrete references to back up for each piece of information in the presentation.

2. Ensure every presentation has a completed agenda, with an introduction and a conclusion.
3. Frequent communication and self-discipline are of high importance to the success of the project.

Conclusion

During this project, we have investigated different architectural aspects of Eclipse IDE. We know that its architecture is modular and extensible, allowing advanced use cases from different stakeholders to be satisfied. That said, more work can be applied to improve the architecture and make the Eclipse IDE more developer friendly.

References

[1] “Wiki shutdown plan · Wiki · Eclipse Foundation / Help Desk · GitLab,” GitLab. Accessed: Jun. 09, 2024. [Online]. Available: https://gitlab.eclipse.org/eclipsefdn/helpdesk/-/wikis/Wiki-shutdown-plan

[2] Wermelinger, M., Yu, Y., Lozano, A. et al. Assessing architectural evolution: a case study. Empir Software Eng 16, 623–666 (2011). https://doi.org/10.1007/s10664-011-9164-x

[3] P. Kamthan, THE ‘UNDESIRABLES’ OF SOFTWARE ARCHITECTURE

[4] D. Todorovic, “Gestalt principles,” Scholarpedia, vol. 3, no. 12, p. 5345, 2008, doi: 10.4249/scholarpedia.5345.

[5] “IBM Host On-Demand 15.0.0.” Accessed: Jun. 01, 2024. [Online]. Available: https://www.ibm.com/docs/en/host-on-demand/15.0?topic=support-creating-host-demand-plug-ins.