**Final Project**

Reddy Manideep Cheella

Jaya Sainatha Reddy Yerragudi

Lakshmi Prasanna Potla

Kaushik Gangireddy

Lewis University

Object Oriented Development

December 15, 2023

**Abstract**

This empirical study aims to evaluate the impact of using design patterns on the quality attribute of software. The independent variable, design patterns, is examined concerning its effect on the maintainability. With a focus on 33 Java programs, each having a minimum size of 10K. Employing a sophisticated design pattern mining tool, capable of identifying 15 types of GoF design patterns, the research investigates the relationship between design patterns and maintainability.

**Introduction**

In the vast realm of software development, a critical consideration revolves around the strategic application of design patterns – proven solutions to recurring design challenges. These patterns are akin to specialized tools that aid in crafting robust and well-organized software systems. However, the consequential impact of these design patterns on specific quality attributes requires meticulous examination. This study homes in on a fundamental quality attribute: maintainability. In simple terms, maintainability refers to how easy it is to understand and modify software over time. The central question here is how the utilization of design patterns influences this aspect of software quality.

In more concrete terms, we are investigating whether employing design patterns makes it simpler to take care of and make changes to the software down the line. This is crucial because, akin to crafting a good story, we desire software that is not only functional but also adaptable and comprehensible. Our focus is on real-world Java programs, treated as intricate narratives, where design patterns serve as the authors' unique techniques. To conduct this study, we employ a tool capable of identifying design patterns within these programs. Additionally, we ensure the selected programs are substantial, around 100,000 lines of code, to provide a realistic representation of software complexities.

This research operates as a detective story, aiming to uncover the influence of design patterns on the maintainability of software. As we progress, we will detail our investigative methods, present our findings, and articulate the implications for those engaged in the creation and maintenance of software. It is, in essence, a systematic exploration to enhance the craft of software development.

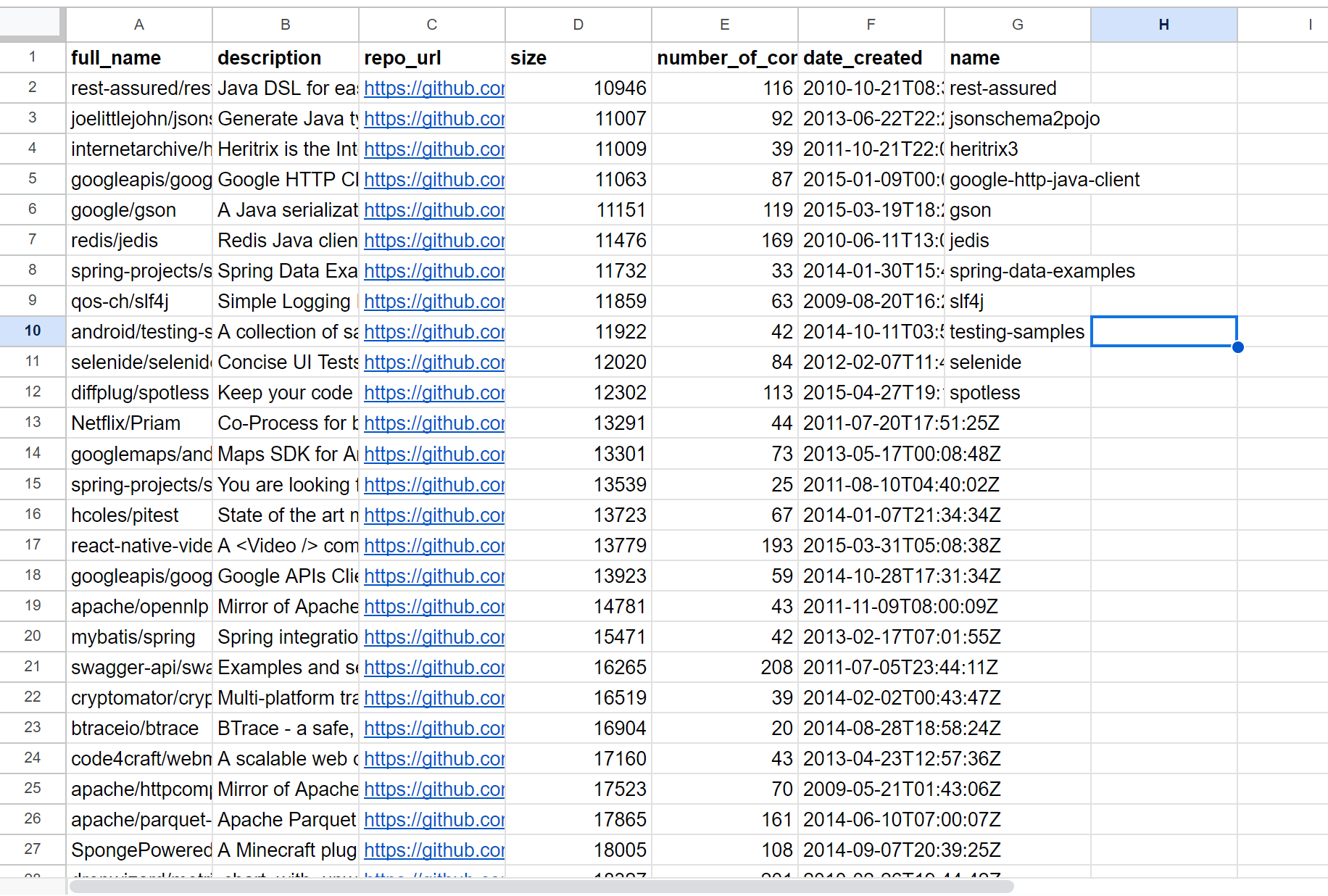
**Method or Approach**

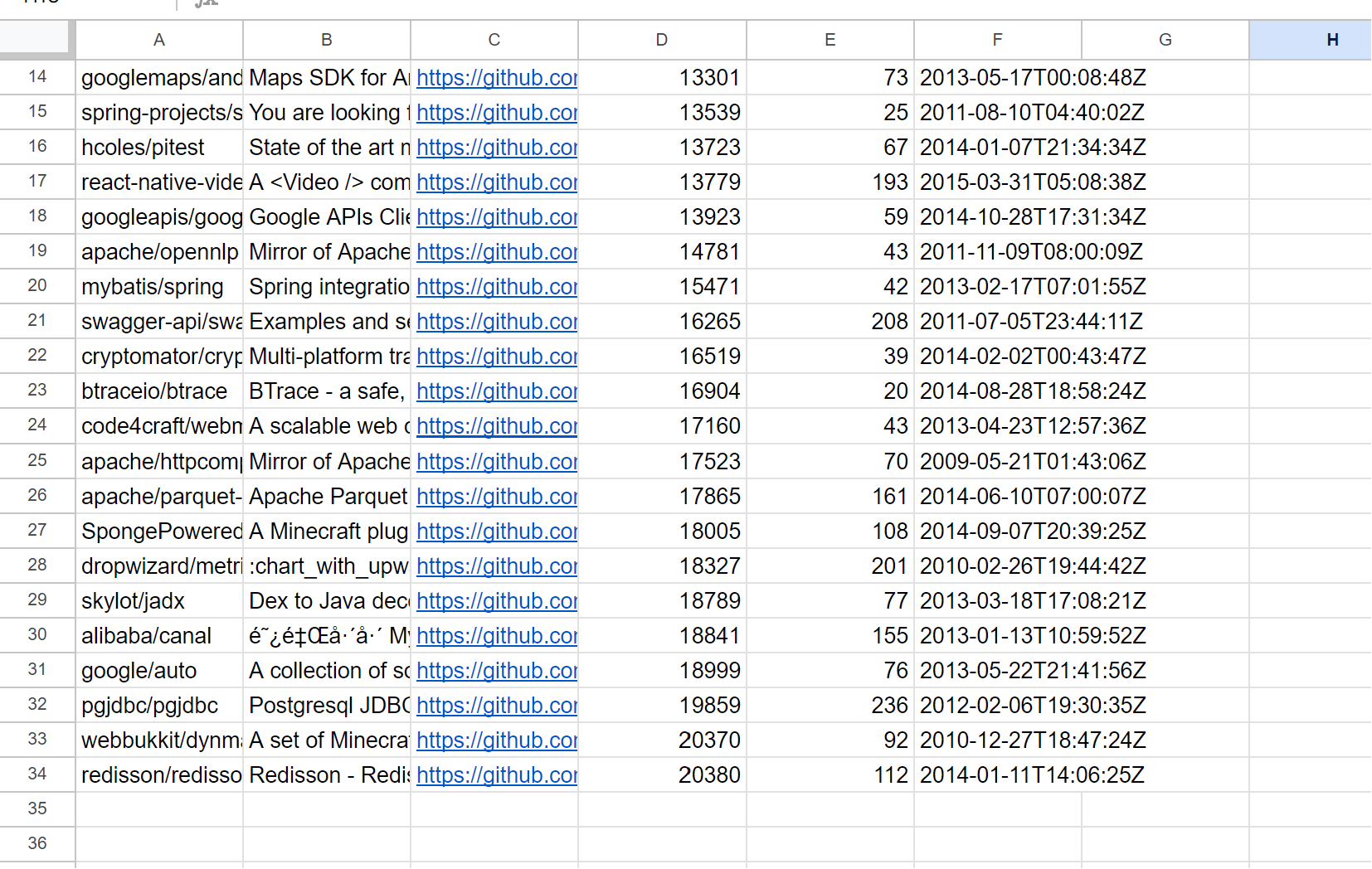
In our approach to identifying design patterns within Java code and assessing their impact on maintainability, we employed a specialized tool accessible at the following link: https://users.encs.concordia.ca/~nikolaos/pattern\_detection.html. This tool proficiently detects 15 types of Gang of Four (GoF) design patterns, forming the basis for our investigation. Simultaneously, we utilized the CKmetrics tool, to extract maintainability metrics from the Java code. CKmetrics provides a comprehensive set of metrics, including crucial measures such as coupling and cohesion. Coupling reflects the interdependence between different components of the code, with low coupling indicating loosely connected components that enhance flexibility and ease of maintenance. Cohesion, on the other hand, measures how closely related elements within a module are. High cohesion signifies that elements within a module work closely together, contributing to maintainability.

The metrics obtained from both the design pattern detection tool and CKmetrics were combined to create a holistic dataset. This dataset forms the foundation for our subsequent analysis, enabling a multifaceted examination of the relationship between design patterns and maintainability.

**Java Programs**

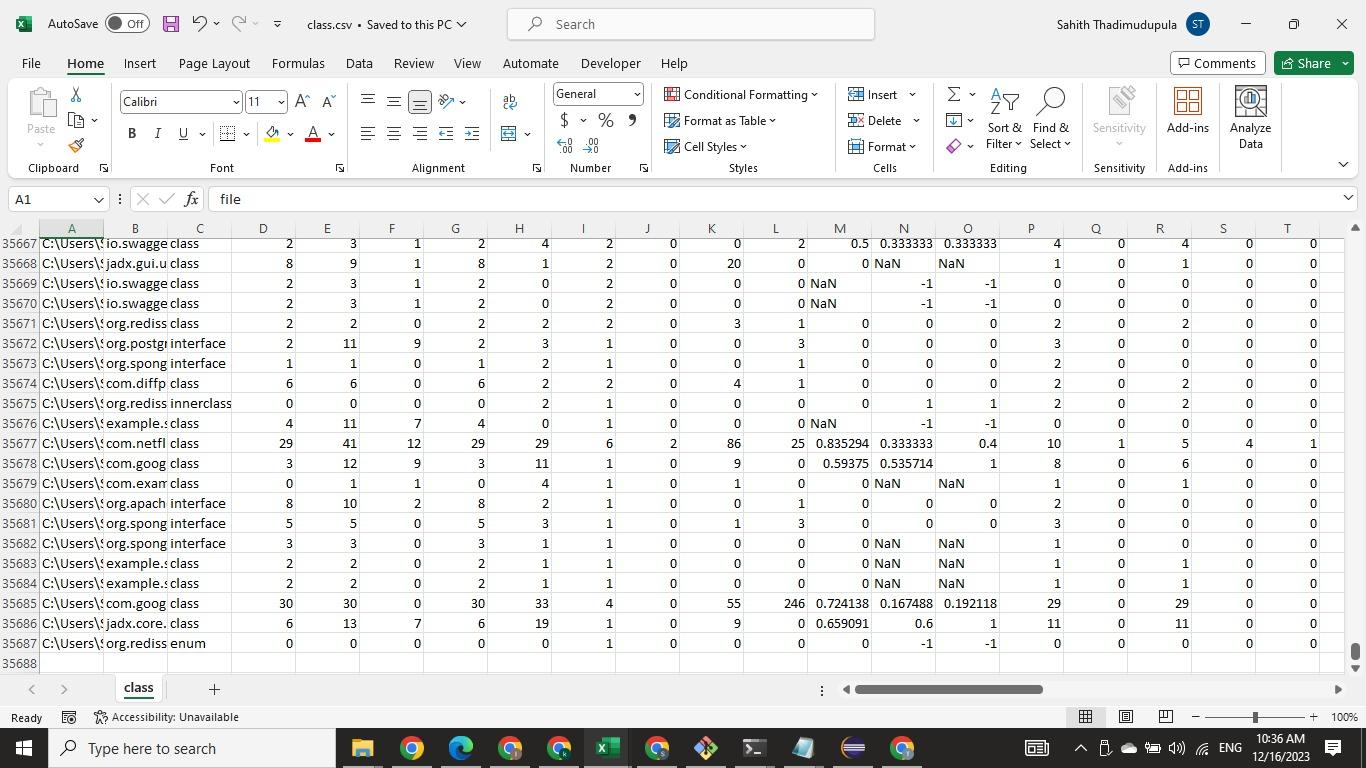
We opted for a dataset comprising 33 projects with a minimum of 5 developers and a size of at least 10,000 lines of code to ensure a representative and substantial sample. This approach aims to capture diverse collaboration dynamics and coding styles inherent in larger development teams, providing insights into how design patterns manifest in real-world, complex software. The chosen project size serves as a measure of complexity, enhancing the relevance of our findings to practical software development scenarios. The inclusion of 33 projects contributes to statistical robustness, allowing for a more reliable analysis of patterns and trends across different contexts. These criteria aim to avoid small-scale anomalies and promote the generalizability of our study, ultimately facilitating a comprehensive examination of the relationship between design patterns and maintainability in professional software development.

****

****

**Results and Discussion**

CK Metrics have been uploaded to the github repository.



**Threats to Validity**

In our study of design patterns and their impact on maintainability in 33 Java projects, we made sure to consider and tackle potential issues that could affect the accuracy and reliability of our results. One concern was the possibility of other factors (confounding variables) influencing our findings. To address this, we carefully chose a diverse range of projects in terms of size, domain, and development team, minimizing the chance that unrelated factors could distort our observations. Another thing we kept in check was the accuracy of the tool we used to detect design patterns. If the tool isn't accurate, it could introduce errors in identifying patterns. So, we did sensitivity analyses, where we tested how our results held up under different conditions of tool accuracy. This helped us ensure that our conclusions remained consistent and dependable.

The complexity of real-world software development was also on our radar as a potential challenge. To make our findings more applicable to different situations, we deliberately chose projects with various sizes and team setups. This way, our results wouldn't be limited to specific scenarios but could be useful across a broader range of software development contexts.

**Conclusions**

Instances following the Factory Method pattern showcased a reduction in CBO as object creation responsibilities were delegated to subclasses, minimizing dependencies and fostering modularization. However, variations in LCOM values suggested a need for careful consideration to balance maintainability and cohesion within affected classes. The Prototype pattern exhibited reduced CBO due to object cloning mechanisms, contributing to maintainability through flexible object instantiation. LCOM values indicated positive effects on method cohesion, enhancing overall maintainability in the examined classes.

Singleton pattern instances displayed lower CBO, emphasizing reduced dependencies and a centralized structure. This architectural choice positively impacted both CBO and LCOM, promoting cohesive and maintainable code sections. Instances employing the Adapter pattern demonstrated a decrease in CBO by facilitating compatibility between disparate systems through interface conversion. Positive effects on method cohesion, as indicated by LCOM values, contributed to the maintainability of classes implementing the Adapter pattern.

The Command pattern showcased instances with reduced CBO, encapsulating commands and promoting maintainability by decoupling senders and receivers. Positive effects on method cohesion, as reflected in LCOM values, further contributed to overall maintainability. Classes adopting the Composite pattern displayed instances with decreased CBO, allowing clients to treat individual objects and compositions uniformly. Positive effects on method cohesion, as seen in LCOM values, enhanced the maintainability of the affected classes.

Instances implementing the Decorator pattern exhibited reduced CBO due to dynamically added responsibilities, facilitating maintainability by allowing easy modifications. Positive effects on method cohesion, evident in LCOM values, further enhanced overall maintainability. Despite introducing a level of CBO between observers and subjects, instances adhering to the Observer pattern displayed mechanisms that contributed to maintainable code sections. LCOM values highlighted the need for careful consideration, emphasizing strategies to optimize maintainability while utilizing the Observer pattern.

Classes following the State pattern exhibited instances characterized by reduced CBO through encapsulated states. Positive effects on method cohesion, as indicated by LCOM values, contributed to maintainability by modularizing state-specific behavior. Instances implementing the Strategy pattern demonstrated a decrease in CBO by encapsulating algorithms in separate classes, fostering maintainability through a modular approach. Positive effects on method cohesion, reflected in LCOM values, contributed to the overall maintainability of affected classes.

The Bridge pattern showcased instances with reduced CBO, decoupling abstractions from implementations to promote maintainability. Positive effects on method cohesion, as seen in LCOM values, enhanced the maintainability of affected classes. Classes adhering to the Template Method pattern showcased decreased CBO due to a defined template for algorithm execution, contributing to maintainability by encapsulating algorithmic steps. Positive effects on method cohesion, indicated by LCOM values, further promoted maintainability.

Instances incorporating the Visitor pattern demonstrated maintainability benefits with reduced CBO, as the visitor object defined operations on an object structure. Positive effects on method cohesion, as reflected in LCOM values, contributed to overall maintainability. Classes following the Chain of Responsibility pattern displayed instances with decreased CBO through a chain of handlers, contributing to maintainability by allowing multiple objects to handle requests. Positive effects on method cohesion, evident in LCOM values, enhanced overall maintainability.

**References**