PIE: A Tool for Visualizing the Life Cycle of Design Patterns in Open Source Software Projects

Kashif Hussain
Ontario Tech University
Oshawa, ON, Canada
kashif.hussain1@ontariotechu.net

Christopher Collins
Ontario Tech University
Oshawa, ON, Canada
christopher.collins@ontariotechu.ca

Jeremy S. Bradbury
Ontario Tech University
Oshawa, ON, Canada
jeremy.bradbury@ontariotechu.ca

***Abstract*—Design patterns are employed in source code to solve commonly occurring programming tasks using understood best practices. Object-oriented design patterns usually span multiple classes and objects and play an integral role in the way object-oriented software is built. One challenge with using object-oriented design patterns is that over the life of a project, these patterns can undergo both planned and unplanned changes. Unplanned changes are often the result of bug fixes or code maintenance tasks that modify a design pattern as a side effect. Furthermore, these unplanned changes can result in increased brittleness of the code and can compromise the overall stability of the software. Over the lifetime of a software project, developers may only become aware of these unplanned changes when the code brittleness results in a software bug. To improve developers' understanding of object-oriented design pattern evolution, we introduce the design Pattern Instance Explorer (PIE) – an exploratory visualization tool that enable developers to visualize a git repository's object-oriented design patterns and their life cycles. In addition to discussing the PIE tool, we provide examples of how this tool can be used to identify and understand design pattern changes. Tool demonstration video: https://www.youtube.com/watch?v=Gkn_5q8_Awg**

***Index Terms*—design patterns, visualization, software maintenance.**

I. Introduction

In general, a pattern *"...describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"* [1]. Many popular software design patterns are for object-oriented software and are known as the Gang of Four (GoF) patterns [2]. This catalog of design patterns includes five creational patterns focused on objection instantiation, seven structural patterns focused on object and class composition, and eleven behavioral patterns focused on the object behaviour [2]. These design patterns can be especially useful to non-expert programmers by providing them with best design practices that address common software problems. Their use can improve non-functional software properties including code quality, performance and readability.

However, there is one notable drawback to the use of object-oriented design patterns that can negatively impact the maintenance of the software leading to brittle source code that is prone to breaking and leading to software bugs. This drawback is that object-oriented design patterns usually involve multiple classes and objects and are implemented across multiple source code files. This makes the documentation of object-oriented design patterns challenging as the majority of in-program comments are at a class or method/function level. Where to document software design patterns and how to ensure software design pattern documentation is provided to software maintainers are well researched topics [3], [4]. Previous studies have found that *"...maintainers achieved better comprehension of the source code when design pattern instances are provided as a complement to the source code"* [4]. Furthermore, it has been noted in the literature that *"...the benefits of design patterns are compromised because designers cannot communicate with each other in terms of the design patterns they used and their design decisions and trade-offs"* [5]. One method to provide design pattern instance information is through graphical language like UML although the regularity with which developers use UML documentation during software maintenance is unknown.

Over the lifetime of a software system, design patterns may be added, removed, replaced, or broken. Specifically, during the evolution of software, design patterns can change into different patterns, break or even turn into anti-patterns [6]. Design patterns that support critical feature implementation undergo regular maintenance due to their importance [7] which can create a vulnerability if the corresponding design patterns are not well documented and maintained. Thus, the health of design patterns is critical to maintaining software quality.

To better understand and potentially address the known challenges with design pattern maintenance, we have developed the Pattern Instance Explorer (PIE) as a tool to aide developers better understand the life cycle of design patterns in software. PIE is an exploratory visualization tool that enable developers to visualize a git repository's object-oriented design patterns and their life cycle at the commit level. Thus, easily identifying how design patterns morph and become both broken and repaired. Furthermore, by exploring the life cycle of design patterns, developers can better understand both their source code, documentation and development practices and use the knowledge to improve their practices.

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), grant #2018-06588.

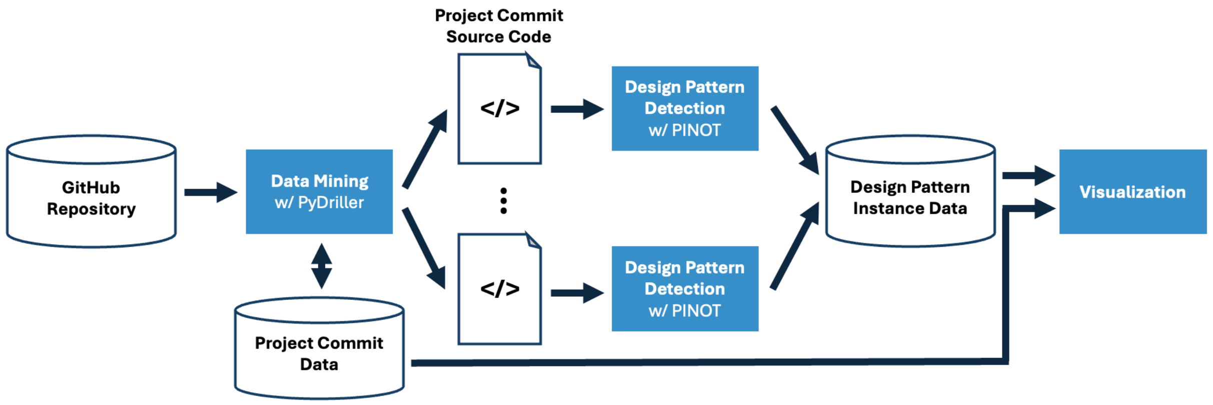

Project Commit Source Code
GitHub Repository
Data Mining w/ PyDriller
</>
Design Pattern Detection w/ PINOT
Design Pattern Instance Data
Visualization
Project Commit Data

Fig. 1. The PIE tool architecture

II. Background & Related Work

A. Design Pattern Detection

Design pattern detection is an active research area focused on the identifying and locating design patterns in software [8]. Strategies for detection can include structural, behavioural and static analysis using different source code representations like Abstract Syntax Trees (AST) and matrices [9]. Design pattern detection tools largely focus on detecting the object-oriented design patterns in the GoF catalog [2].

PIE utilizes the Pattern INference and recOvery Tool (PINOT) [10] to identify and localize design patterns for visualization. PINOT utilizes static program analysis for detection and has been used to detect design patterns in major Java packages and applications (e.g., Apache Ant, Java AWT) [10]. Alternative tools for design pattern detection include DPDT [11], a graph matching approach to design pattern detection, and GEML [12], a grammar-based evolutionary machine learning approach for design pattern detection. Although we use PINOT to support the PIE visualization, PIE could also be modified to utilize other design pattern detection tools.

B. Design Pattern Visualization

Early work on visualizing design patterns has focused on the graphical representation of design pattern instances, often using UML diagrams [5]. These approaches do not consider how design patterns evolve and change over the lifetime of the software and are simply representations of a design pattern instance at a particular moment in time. More recent work on visualizing object-oriented software using a city metaphor has included the visualization of object-oriented design patterns as part of a broader visualization into object oriented mechanisms including overloading and inheritance [13], [14]. In a city metaphor visualization, structural elements of buildings or building colour can indicate the presence of a design pattern or object-oriented mechanisms. Similar to the earlier design pattern visualization work, this work is focused on a particular software instance and does not specifically emphasize the evolution of design patterns in software over time.

III. Approach

The PIE visualization tool architecture includes components or tools to mine project commit data, detect design pattern instances, and visualize the life cycle of design pattern instances over the entire commit history of a project (see Fig. 1).

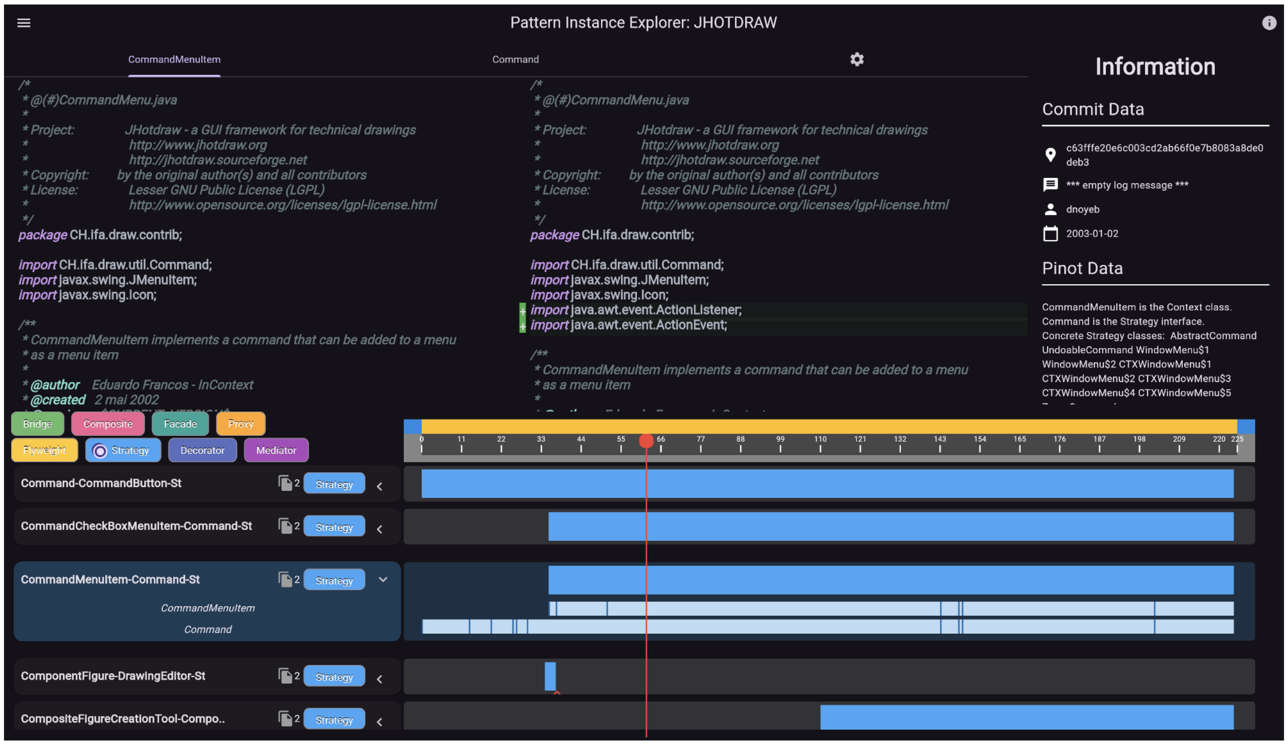
A. Mining Project Commit Data

PIE is designed to mine the commit data of existing open-source projects hosted on GitHub. We used *PyDriller*, a Python framework for mining software repositories, to automate this process [15]. For performance reasons, a second iteration of mining is done to directly acquire all of the file versions that relate to the design patterns found by PINOT. The design pattern instance data and corresponding file data is processed into intervals in which the pattern persists. By creating intervals of the pattern instances and file modifications we can more easily visualize them across a timeline.

B. Detecting Design Pattern Instances

A number of design pattern detection tools exist in the literature [9] and we made the decision to use one of these existing tools to support the PIE visualization. We evaluated multiple design pattern detection tools to find one that was suitable for our needs. We ended up selecting PINOT [16] because of its consistency, speed and robustness – many of the other design pattern detection tools were no longer maintained or no longer available. PINOT did require some minor modifications to work on our target repositories [1], however because the PINOT source code was available and open sourced we were able to adapt it to the needs of our PIE visualization tool. When PINOT detects a design pattern it reports the *design pattern instance* as a set of unique files that make up a single pattern. The files for each pattern instance are concatenated and stored in a single file that is named using an alphabetically listing of the involved source files followed by a design pattern tag. For example, the design pattern instance file *AWTEvent-EventQueue-St* would contain a instance of

[1]Our modifications to PINOT are documented and made available as part of the PIE GitHub repository – https://github.com/seer-lab/PIE



```
/*
 * @(#)CommandMenu.java
 *
 * Project:		JHotdraw - a GUI framework for technical drawings
 *				http://www.jhotdraw.org
 *				http://jhotdraw.sourceforge.net
 * Copyright:	by the original author(s) and all contributors
 * License:		Lesser GNU Public License (LGPL)
 *				http://www.opensource.org/licenses/lgpl-license.html
 */
package CH.ifa.draw.contrib;

import CH.ifa.draw.util.Command;
import javax.swing.JMenuItem;
import javax.swing.Icon;

/**
 * CommandMenuItem implements a command that can be added to a menu
 * as a menu item
 *
 * @author	Eduardo Francos - InContext
 * @created	2 mai 2002
```

```
/*
 * @(#)CommandMenu.java
 *
 * Project:		JHotdraw - a GUI framework for technical drawings
 *				http://www.jhotdraw.org
 *				http://jhotdraw.sourceforge.net
 * Copyright:	by the original author(s) and all contributors
 * License:		Lesser GNU Public License (LGPL)
 *				http://www.opensource.org/licenses/lgpl-license.html
 */
package CH.ifa.draw.contrib;

import CH.ifa.draw.util.Command;
import javax.swing.JMenuItem;
import javax.swing.Icon;
+ import java.awt.event.ActionListener;
+ import java.awt.event.ActionEvent;

/**
 * CommandMenuItem implements a command that can be added to a menu
 * as a menu item
 *
```

Information

Commit Data

c63fffe20e6c003cd2ab66f0e7b8083a8de0deb3
*** empty log message ***
dnoyeb
2003-01-02

Pinot Data

CommandMenuItem is the Context class.
Command is the Strategy interface.
Concrete Strategy classes: AbstractCommand UndoableCommand WindowMenu$1 WindowMenu$2 CTXWindowMenu$1 CTXWindowMenu$2 CTXWindowMenu$3 CTXWindowMenu$4 CTXWindowMenu$5

Fig. 2. The complete PIE visualization interface with the JHotDraw project

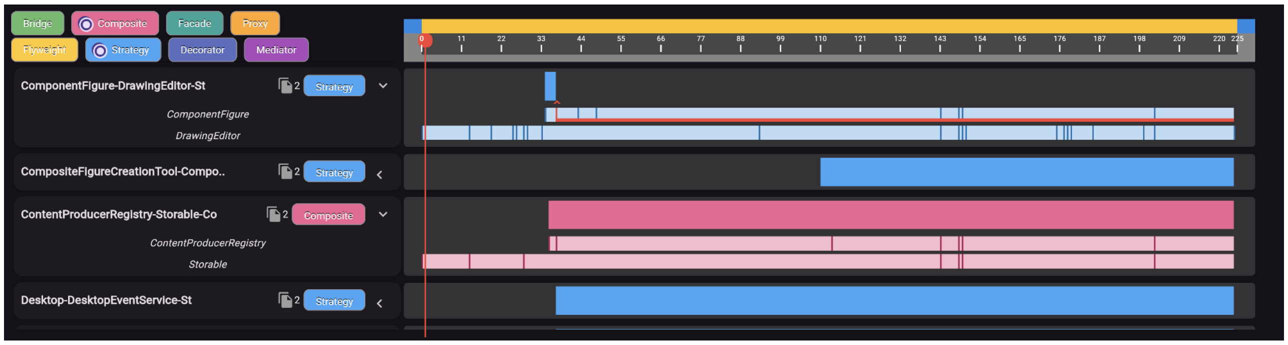


Fig. 3. Example of the PIE timeline with the JHotDraw project where design pattern instances are color-coded by type

the *Strategy* pattern involving the files *AWTEvent.java* and *EventQueue.java*. We mention the naming convention now as we reuse it later in the PIE visualization.

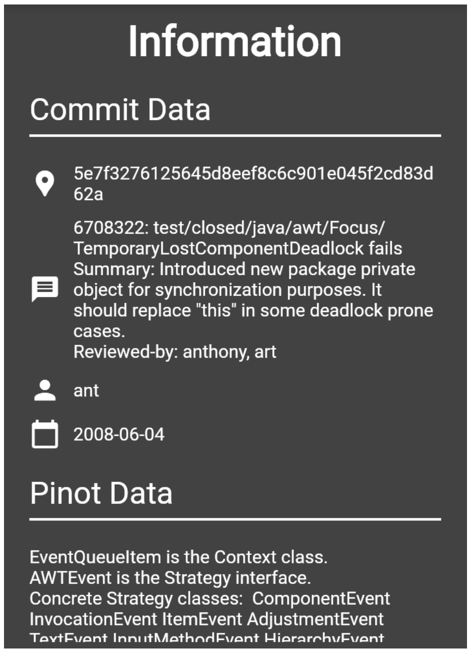
In PIE, each commit is analyzed using PINOT to detect design pattern instances which are stored in a database with relevant commit information. PIE analyzes the detected design pattern instances over all of a project's commits and reassembles instances from different commits into a single history of that design pattern instance.

C. Visualizing Design Pattern Instances Over Time

The web-based visualization component of PIE was designed to support developers in better understanding the use of design patterns in their projects. It is intended to be an exploratory tool which also lends itself to other use cases including supporting researchers understand the use of design patterns in a given project and supporting new project members explore a project's historic design decisions.

More specifically, PIE was designed to allow for exploration of the following questions with respect to a specific project:

- What design patterns have existed over the lifetime of the project?
- When have design patterns been created, broken, and removed?
- What code changes resulted in a design pattern breaking?
- How have design patterns evolved, both in scope or into other patterns?



Information

Commit Data

5e7f3276125645d8eef8c6c901e045f2cd83d62a

6708322: test/closed/java/awt/Focus/TemporaryLostComponentDeadlock fails
Summary: Introduced new package private object for synchronization purposes. It should replace "this" in some deadlock prone cases.
Reviewed-by: anthony, art

ant

2008-06-04

Pinot Data

EventQueueItem is the Context class.
AWTEvent is the Strategy interface.
Concrete Strategy classes: ComponentEvent InvocationEvent ItemEvent AdjustmentEvent TextEvent InputMethodEvent HierarchyEvent

Fig. 4. The PIE information panel for the currently selected design pattern instance and commit in the Java AWT project

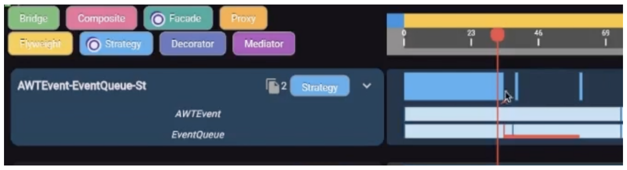


Fig. 5. Timeline for the *AWTEvent-EventQueue-St* Pattern Instance in the Java AWT project

All of the questions are focused on the primary use case of PIE which is understanding the life cycle of design patterns. We will now present the PIE visualization interface (see Fig. 2) which includes the following interface components: timeline, code viewer and information panel.

1) Timeline: The timeline is located at the bottom of the PIE interface, with the pattern instance names presented vertically to the left (see Fig. 3). Each pattern instance has a commit-level timeline (color-coded to match the design pattern color) which shows the commits where the pattern instance was detected. The pattern instance visualization can be expanded to further reveal file-level timelines which show vertical lines to denote commits where a file has been modified. Areas with no color indicate periods where the file does not exist. The colors used for the different design patterns were selected to distinguish the patterns by type. The file-level timelines are lighter tints of the corresponding design pattern colors to make the relationship between timelines clear.

Across the top of the timeline component is the timeline controller. It lists the commit numbers as well as a red marker and a yellow slider. The red marker (denoted by a red circle with a line extending veritcally down) can be moved to select a specific commit which will adjust the information displayed in the top portion of the visualization window. The yellow slider has blue handles that can adjust the scale of the timeline, while moving the yellow slides directly will move the timeline's position along the commits.

To assist the developer identify pattern breaks we have also added a red arrow notation to the bottom of the design pattern instance timeline where a break is more likely to have occurred (see Fig. 3).

2) Code Viewer: The code viewer displays relevant commit-level code changes of the currently selected pattern instance. The content displayed can be changed by moving the red timeline marker to adjust the time and by selecting a different pattern instance to bring up it's code data. If the code data spans multiple files you can select between the files at the top of the code viewer. The code viewer presents commit-level code changes with added code denoted by a '+' and deleted code denoted by a '-'. The code viewer supports both an inline and split view of the code changes to help users analyze the differences.

3) Information Panel: The information panel is above the timeline and to the right side of the code viewer (see Fig. 4). It includes information relevant to the currently selected pattern instance (PINOT analysis data) and commit data (e.g., commit message).

IV. Case Study

We conducted a preliminary assessment of PIE using two projects that are prevalent in the design pattern literature: JHotDraw [17] and Java AWT [18]. Due to space limitations, we'll only highlight some of the observations in the visualization of Java AWT.

One use case for PIE is to identify and understand pattern breaks. By understanding the cause of a design pattern break it can be fixed and possibly further breaks can be prevented in the future. One example of a pattern break was in the *AWTEvent-EventQueue-St* pattern instance (see Fig. 5). Recall that using the PINOT naming convention this is an instance of the Strategy design pattern involving the *AWTEvent* and *EventQueue* classes. The Strategy pattern is part of the GoF catalog and is a behavioural pattern that encapsulates algorithms in different objects to make it easier to use them interchangably [2]. We can see in the Information Panel that *EventQueueItem* is the context class. If we move the red timeline marker to the commit corresponding with the pattern break the commit message reads: *"Unify EventQueue EventQueueItem and SunToolkit.EventQueueItem classes."* This illuminates the reason why the pattern was broken, and we can drill further into the code responsible for that change in the code viewer. At the top of the *EventQueue* file an *EventQueueItem* package was added and the original class was removed. By removing the context class from this file, this set of files no longer represents a strategy design pattern, and is shown as broken.

We also noticed a few cases where patterns expand in terms of the number of files (see Fig. 6) or reduce to fewer files.

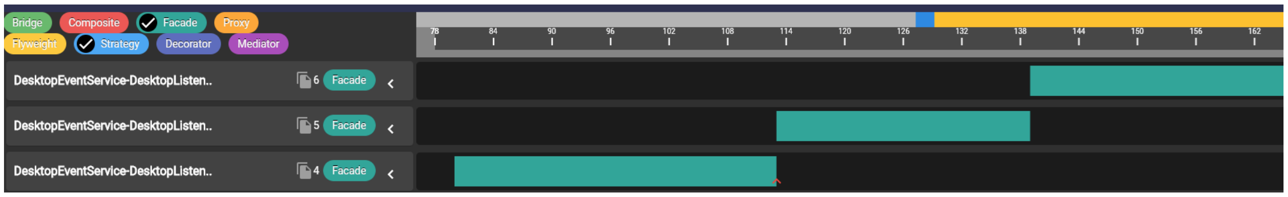


Fig. 6. Example of design pattern instance growth visualized in the PIE timeline

These design pattern behaviours were most apparent with the facade design pattern, a structural pattern that provides an interface for a subsystem of objects [2]. An added benefit of using the same color for instances of the same design pattern in the PIE timeline is that it makes it easier to identify this type of pattern evolution.

V. Conclusions

In this paper we have described the PIE, a visualization tool for exploring and understanding design pattern instance life cycles in open source projects. PIE uniquely visualizes patterns' throughout a project's lifespan and provides insights into the use of design patterns that is not observable in traditional design pattern visualizations.

In our preliminary evaluation we were not able to always determine if a pattern continued to exist in other files because of limits in the design pattern detection process with PINOT. We should also note another limitation is that not all pattern/file breaks in the timeline are real breaks. This is due to limitations in the data mining (we only have information on the main/master branch and not branch merges) and design pattern detection with PINOT (a static analysis approach that can include spurious results). The addition of the red arrow notation was added to mitigate this limitation.

In the future we would like to explore the benefits of the visualization by conducting a user study. Such a study would also help identify areas of future improvement in PIE.

References

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language*. Oxford University Press, 1977.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[3] G. Scanniello, C. Gravino, M. Risi, and G. Tortora, "A controlled experiment for assessing the contribution of design pattern documentation on software maintenance," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1852786.1852853

[4] C. Gravino, M. Risi, G. Scanniello, and G. Tortora, "Does the documentation of design pattern instances impact on source code comprehension? results from two controlled experiments," in *Proceedings of the 18th Working Conference on Reverse Engineering*, 2011, pp. 67–76.

[5] J. Dong, S. Yang, and K. Zhang, "Visualizing design patterns in their applications and compositions," *IEEE Transactions on Software Engineering*, vol. 33, no. 7, pp. 433–453, 2007.

[6] Z. A. Kermansaravi, M. S. Rahman, F. Khomh, F. Jaafar, and Y.-G. Guéhéneuc, "Investigating design anti-pattern and design pattern mutations and their change- and fault-proneness," *Empirical Softw. Engg.*, vol. 26, no. 1, jan 2021. [Online]. Available: https://doi.org/10.1007/s10664-020-09900-0

[7] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta, "An empirical study on the evolution of design patterns," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 385–394. [Online]. Available: https://doi.org/10.1145/1287624.1287680

[8] D. "Shilintsev and G. Dlamini, "A study: Design patterns detection approaches and impact on software quality," in *Frontiers in Software Engineering*, G. Succi, P. Ciancarini, and A. Kruglov, Eds. Springer International Publishing, 2021, pp. 84–96.

[9] M. Al-Obeidallah, M. Petridis, and S. Kapetanakis, "A survey on design pattern detection approaches," *"International Journal of Software Engineering (IJSE)"*, vol. 7, no. 3, pp. 41–59, 2016.

[10] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 123–134.

[11] J. Singh, S. R. Chowdhuri, G. Bethany, and M. Gupta, "Detecting design patterns: a hybrid approach based on graph matching and static analysis," *Information Technology and Management*, vol. 23, no. 3, pp. 139–150, Sep. 2022. [Online]. Available: https://doi.org/10.1007/s10799-021-00339-3

[12] R. Barbudo, A. Ramirez, F. Servant, and J. R. Romero, "GEML: A grammar-based evolutionary machine learning approach for design-pattern detection," *Journal of Systems and Software*, vol. 175, p. 110919, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221000169

[13] J. Mortara, P. Collet, and A.-M. Dery-Pinna, "Visualization of object-oriented variability implementations as cities," in *Proceedings of the Working Conference on Software Visualization (VISSOFT)*, 2021, pp. 76–87.

[14] J. Mortara, P. Collet, and A. M. Dery-Pinna, "Visualization of object-oriented software in a city metaphor: Comprehending the implemented variability and its technical debt," *Journal of Systems and Software*, vol. 208, p. 111876, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223002716

[15] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 908–911. [Online]. Available: https://doi.org/10.1145/3236024.3264598

[16] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from Java source code," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 123–134.

[17] "JHotDraw," Webpage: https://github.com/wrandelshofer/jhotdraw (Last accessed: August 20, 2024).

[18] "Java AWT," Webpage: https://github.com/JetBrains/jdk8ut (Last accessed: August 20, 2024).