Summer 2021 CS 497 Capstone Project Progress Report
Lime: A Model to Identify Antipatterns

Gibran Herrera Lopez
Advisor: Hee Jung (Sion) Yoon, Ph.D.
BS in Computer Science
School of Technology & Computing (STC)
City University of Seattle (CityU)
herreragibran@cityuniversity.edu, yoonhee@cityu.edu

Abstract

Software designing is a technique that is not well-engineered as developers are still being involved with their experience and knowledge in order to create compelling solutions for complex problems. Two schools of thought were created to overcome the issue, antipatterns, and patterns. Antipatterns are an effort to avoid proven not suitable solutions to common issues, even though, their identification process also involves developer time and human error factors that may lead to accidental bugs and underperformance solutions. The lime model is a code-based tool that makes usage of ASTs (Abstract Syntax Trees) to identify syntax Python code following its features and rules, as well as the Flake8 API to create node visitors for the trees to check if an antipattern was introduced, and if so, the model provides a compelling explanation of the problem and a possible solution for the antipattern. The results of the lime model are impressive as the average time to check for three antipatterns in four open-source projects with several lines of code was 9.86 seconds with a precision of 98.35 percent. The lime model can be implemented as a static analysis tool within IDEs, and CI/CD pipelines. Also, it opens the possibility to enforce coding guidelines and grammar checking for several human languages.

Keywords: Antipatterns, Software Design, Software Development, Design Patterns.

1. INTRODUCTION

Software is part of modern society, and it is difficult to imagine a world without all the commodities it brings, from online stores to mass communication networks. But software normally is distributed as a craft instead of a well-engineered product with specific processes behind it. It is understandable since software is a created product with many layers like APIs, frameworks, and libraries.

The necessity of good quality software is constantly increasing as its importance is deeper within critical systems like healthcare, security, and even financial. Fortunately, nowadays, developers understand the importance of thinking about how to make the product before coding and designing, even though designing is a skill that developers tend to grow with time and expertise within their careers. Thankfully, software design patterns and antipatterns were created to help developers to look for previously acquired knowledge of how to craft good software. These practices help software to reach its goal, be maintainable, extensible, and reusable.

Problem Statement

The software has the liberty of being written in many ways. The quantity of options of how to solve a specific problem is one of its significant advantages. But as time is passing, developers have found that some implementations may not be a good way to solve a specific problem, antipatterns. Often, they make the mistake of using them without realizing, leading to unsafe, slow, and low-quality software. Also, on the other side, they have found good ways to approach a problem, design patterns.

To benefit from design patterns, a developer must have a thorough understanding of and a good practice with design patterns to identify the appropriate one to instantiate for its application. The absence of such a high level of expertise often results in antipatterns. (Fourati et al., 2011).

Motivation

Software should be created with all the expertise and knowledge learned from previous development iterations and generations of developers. Though, the absence, or lack of awareness by implementing a proven wrong solution is not perceived until problems arise. One way to identify them is by antipatterns, which have been conceived to address performance issues since the early phases of the software life cycle. (Arcelli & Di Pompeo, 2017). Their detection would help developers to avoid them before they make an impact on a delivered product. The antipatterns that constitute harmful practices must be avoided to reduce the risk of failure in software projects (Bolaños et al., 2011).

Approach

Models to detect antipatterns are divided by its approach: code-based, design-based, and search-based. Each one of them has its advantages, but the key common factor in all of them is that it involves the developer's time and expertise to either detect antipatterns or define them.

The Lime model wants to prevent developers' involvement, by proposing an automated tool with growing capability using ASTs to detect code behavior. The advantage of using this code-based approach involves the easiness of implementation and maintenance. Nevertheless, the disadvantage is the programming language dependency, as ASTs vary between them.

Conclusions

The expected benefits of using an AST automated tool are to be easily maintainable, implementable, and expandible. Also, the tool should be able to adapt the current development tools for each team without impacting the software development cycle. Even though, one of the expected flaws of the solution is, as being a code-based tool, highly programming language coupled.

2. BACKGROUND

Software designing is a process where developers put their expertise and experience to create suitable solutions for complex problems. Most of this knowledge was lost until design patterns were created, which establish solutions for concurrent problems, and as their appearance was acknowledged, developers also created a list of concurrent solutions that may not be suitable, antipatterns.

Even after the creation of multiple compilations that detail patterns and antipatterns, developers are involved in their detection process, which would lead to human error factors. For example, the Automated Reporting of Antipatterns and decay in Continuous Integration by Vassallo et al. (2019) involves the pair review of developers outside the development team in the form of project reports and validation.

Other solutions may not lead to direct human involvement, but they are implemented in slow processes that need human validation like the Preliminary Approach of Applying Design Patterns to Remove Software Performance Antipatterns by Arcelli & Pompeo (2017) which proposed a model to detect performance antipatterns in code by a ranking system. But the final decision is made by the developer.

There is a need for automated tools adaptable to the software development cycle and the development team with constant cases of tight deadlines.

3. RELATED WORK

Software design quality could be achieved in many forms, but one of the best practices in the field is to apply design patterns, which is previously acquired knowledge of how to craft good software. But even by applying the best techniques in the field, developers may also apply the other face of design patterns, antipatterns, which are poor solutions to concurrent issues in object-oriented design. The goal of this paper is to create a model that would help the developer to detect these implementations and correct them.

On the side of the Preliminary Approach of Applying Design Patterns to Remove Software Performance Antipatterns, Arcelli & Pompeo (2017) proposed a model to detect performance antipatterns in code by a ranking system according to a formula built on the number of calls and time it takes to make operations. Their approach is to use model-based and code-based artifacts. The model also covers a cycle process to correct any of the antipatterns it detects by the involvement of developers' models and validation. One of the constraints of the project is that it is tightly thought on three antipatterns: session façade, batching, and aggregate entity pattern.

In the case of the Compendium of Bad Practices in Software Development Process, Bolaños & Medina (2011) include a set of antipatterns definitions, processes, and feature descriptions of how to identify antipatterns without being language-specific. Their goal is to create a compendium of antipatterns. The downfall of the paper is that it does not include a logic-specific way to identify antipatterns.

A metric-based antipattern detection in UML design is proposed by Fourati et al. (2011) with the focus on detecting antipatterns in the design phase by using specific metrics: coupling cohesion, complexity, and inheritance. Its detection is driven by detecting specific structures like the Swiss Army Knife or BLOB structs. The key focus of the paper is to use a design-based approach. The highlight of the project is that it is highly dependable on the usage of UML diagrams, which are language agnostic. Even though one of the disadvantages of the model is that it was not implemented or tested with real-life models, then its effectiveness is not well-proven.

The usage of ASTs (Abstract Syntax Trees) for code-specific behavior is researched and realized with the Muupi project made by Liu (2017). Its approach is to create a mutation testing tool that detects twenty-two mutation operators designed for Python 2.x programs, which defer from the antipatterns detection code-base model of Lime. Even though, the Muupi project presented an exceeded validation of AST usage for detecting specific behavior on code, which would be useful for the antipattern detection.

A detection of Operator Errors in Cloud Computing Using Antipatterns was proposed by Vetter (2019). It defines a TOSCA template based on XML to create an implementation architecture for detecting immediately next, immediately precedence, state-condition, value alternative, alternative absence, and occurrence antipatterns in SQL code. The antipattern detection focuses on specific SQL antipatterns, and the usage of TOSCA-based SML process models to create a Cloud Computing solution, which is a more complicated approach as it involves a later on process during the development software cycle to detect antipatterns and correct them.

On the Automated Reporting of Antipatterns and decay in Continuous Integration, Vassallo et al. (2019) had a different approach in the detection of antipatterns in code which involves the pair review of developers outside the development team in the form of project reports and validation. The biggest flaw of the model is the human error factor present in all the phases of the model. Also, its approach adds a new phase to the continuous integration process that would increase the cost and time of release. Also, the model, in the beginning, is limited to the catalog of antipatterns defined by the organization.
In the case of the automated technique for analysis of orthogonal variability models based on antipatterns detection using DL reasoning that creates the crowd-variability tool by Oyarzun et al. (2019) have a design-based approach, even though the main highlight of the model is the usage of description logic that it helped to create a language-agnostic model that not depends on other technologies or models like UML for designing. However, the approach presents some limitations associated with the complexity of the models because the antipatterns detection is at one level of depth.

The implementation of an AST representation in the form of logic predicates was presented by Stoianov & Sora (2010). It detects a set of patterns and antipatterns with the simplicity of defining Prolog predicates able to describe structural and behavioral types. One of the downfalls of the project is the lack of support to creational types and the rigid adaptation to other structural and behavioral that would be less or complexed than the ones presented. Also, the project is focused on the Java AST implementation.

An automated repair tool is presented by Tan et al. (2016) using a search-based approach that searches for a program fix within a given repair space. The tool is based on two main tools Gen-Prog and SPR and makes usage of a deterministic adaptative search algorithm to control potential randomness. The main advantage of the tool is that it detects a set of antipatterns that are language agnostic. But on the other side, it presents many issues as not being able to test the complete set of antipatterns proposed and the incomplete semantic equivalence that proposes many fixes for a particular fault.

The case of SMAD (Smart Aggregation of Antipatterns detectors) created by Barbez et al. (2019) utilizes a machine-learning-based ensemble method to aggregate various antipatterns detection approaches based on their internal detection rules. One of the constraints of the tool is the limitation of antipattern detection into two, good class and feature envy. Also, the machine learning algorithm was provided with a low amount of data, eight java projects, which could affect the authenticity and effectiveness of the proposed solution.

The OOPDTool was developed by Correa et al. (2000) to support the reengineering of legacy systems and systems that are under development to detect antipatterns and suggest the replacement of more adequate code. Even though, the tool was created as an add-in to the OO CASE tool, which is currently supported for windows systems in a 32-bit operating system. Also, the

tool has an important constraint to only detect antipatterns in C/C++ programs.

SODA-R (Service Object-Oriented Detection for Antipatterns in REST) is a proposed tool by Palma et al. (2014) that detects patterns and antipatterns in RESTful systems. The tool was used for twelve sets of widely used APIs including Best Buys, Facebook, and Dropbox. The proven average precision with the tool was 89.42% and recall of 94%.

Review Conclusions

Tools to detect antipatterns are divided into three categories, code-based, design-based, and search-based. Each one of them has its advantages and constraints. On the side of design-based, it has the highlight to reduce the cost of detected flaws and ease of change, as well as it is characterized to be the most language-agnostic approach as it can be adapted to any programming language.

In the case of a code-based solution, there are different approaches of how to detect anti-patterns, from machine learning algorithms to manual developer interpretation. Even though one of the most useful tools to detect code behavior was AST used in the MUUPI testing tool. The downfall of this approach is that it is highly language dependable. But on the other side, the solution is easier to implement, and it could be adapted to other automated tools for quality assurance.

Finally, a search-based approach is an innovative way to detect anti-patterns as it can also provide a patch to the detected flaw. Even though, it also provides many cons to its implementation as the algorithm to provide patches is complicated to develop and requires a large number of rules and data to be useful.

4. APPROACH

Antipatterns can be detected in numerous ways. Nonetheless, automated tools would be a preferable way as they save developers time and reduce the human error factor. These tools can be divided into three different categories depending on their method: code-based, design-based, and search-based.

The Lime model aims to use a code-based method due to the advantages proved by the MUUPI project by Liu (2017), which helped to detect code behavior, and the Stoianov & Sora (2010) model for the pattern detection in Java. Also, the model is going to use ASTs to detect specific code semantics. AST (Abstract Syntax Tree) are models of software that represent software artifacts using data structures that represent the types of language constructs, their compositional relationships to other language constructs, and a set of direct and derived properties associated with each language construct (Ulrich & Newcomb, 2010). That would help to detect specific situations and behavior leading to antipatterns.

The ASTs are going to be implemented alongside a Flake-8 based linter to create an automated static tool, easy to implement, maintain and, use for developers as they can introduce it into their development life cycle with the minimum impact as well as integrate it into their CI/CD pipeline.

The major disadvantage of the model, as being code-based, is the high dependency of the programming language to be implemented, in this case, Python. In other words, the linter will only detect antipatterns in Python code. But other linters can be developed for different programming languages later on as part of future work.

The main objective of this project is to create a model that ensures the non-implementation of antipatterns and helps developer teams to reach the software goal.

5. DATA COLLECTION

The Lime model, as being a code-based solution, makes usage of three algorithms. Each of them is a specific implementation to detect each antipattern. The antipatterns selected were lambda assigning, with enforcement, and type enforcement. Each of them was divided in The Little Book of Python Antipatterns (2020) into three sections respectively:

- Correctness
- Maintainability
- Readability

Every one of the antipatterns makes usage of ASTs data structures. Even though the Lime project also enforced software development techniques like Test-Driven-Development and Object-Oriented Design.
To define the effectiveness of the Lime approach, the model was assessed in two categories:

- **Performance**: The implemented algorithms perform with considerably

low detection times, i.e., the average time should be in some seconds.

- **Accuracy**: The detection heuristics should have an average detection between the 80% and 100%

The code bases selected to validate the results of the Lime implementation are four open-source projects that have Python as their primary programming language:

1. Manaim: Animation engine for explanatory math videos.
2. Textual: Text User Interface framework for Python.
3. ParlAI: Python framework for sharing, training, and testing dialogue models from open-domain chitchat to task-oriented dialogue, to visual question answering.
4. AlphaFold: Implementation of the interface pipeline of AlphaFold v2.0

Then, the performance was based on the total amount of time it takes to check all Python files, whereas the accuracy was collected by the total amount of antipatterns against the ones detected.

6. DATA ANALYSIS

The results obtained over the lime model implementation are divided according to the violations found. A violation is when the algorithm recognizes an antipattern in the codebase. Each violation has a distinctive name according to it:

- Correctness: Lambda Assigning Violation.
- Maintainability: WithOpen Violation.
- Readability: Type Usage Violation.

On the side of accuracy, the lime model obtained the results shown in Figure1, Figure2, and Figure3 in the Appendix section.

Among all projects, the PerlAI is the one with the highest number of antipatterns, and on the other hand, Textual has the least.

Figure5 highlights the lambda assigning antipattern was the least numerous above all projects, being Manim the project with the most occurrences. Lime had an outstanding precision percentage for the Lambda Assigning Violation in almost all the projects it was applied. But it is important to highlight that in the case of Alphafold, there was only one single occurrence missed. The average precision of the lambda assigning algorithm is 96.43%.

Figure5 demonstrates, in the case of the WithOpen violation for the with enforcement antipattern, it presented a similar behavior. Although the number of occurrences was high in the PerlAI project. Nevertheless, there was only a single instance not detected. The percentage of accuracy remains high, with the average being 99.40%.

Finally, Figure 6 shows the type of enforcement antipattern represented by the type usage violation follows the same tendency of high accuracy average percent as being 99.21%. But in this case, there were three antipatterns missed.

Changing the focus on the other key category to assess the effectiveness of the lime model, the time taken on each algorithm is presented in seconds in Figure 7, Figure 8, and Figure 9.

On each iteration the time taken is around the range of 3.8 to 6.95 seconds, which demonstrates the rapidness of the tool. Even though there has to be considered that each of the algorithms was measure alone, although the model has the ability to check all algorithms in the same cycle.

For that reason, there was also measured the time it takes to run the whole model in each project.

As shown in Figure10, the average time for the Lime model to perform the detection of all antipatterns separated is 14.81 seconds, which is in comparison to the 9.86 seconds it takes to verify all at once. That reveals that checking up several antipatterns at once reduces the overwork it takes to iterate n times being n the number of algorithms.

7. FINDINGS

The Lime model is highly effective due to it excelling in two of the assessed categories: performance, and precision. The model has an accuracy of 98.35 percent and an average of 9.86

seconds, respectively checking all antipatterns at once.

Also, an interesting point of comparison is the amount of time in seconds it takes to check all antipatterns in each codebase against the time it takes to check each antipattern. The lime model had an average of 14.81 seconds individually and 9.86 collectively.

8. CONCLUSIONS

Antipatterns, proven wrong solutions for design problems, is one way to achieve software quality goals, being extensible, reusable, and maintainable. But a developer being able to identify all these flaws have years of experience, and even with it, some would be implemented due to the human error factor.

One way to ensure antipatterns detection was presented by the Lime model, its objective was to be able to detect three key antipatterns divided into three categories:

- Correctness: Lambda Assigning.
- Maintainability: WithOpen.
- Readability: Type Usage.

The way to ensure its effectiveness was to measure its (1) performance, the time it takes to check all the codebase, and (2) accuracy, the quantity of the antipatterns detected against the total of them over four open-source projects.

The results were an outstanding 98.35% average accuracy and 9.86 seconds total time in average checking all the antipatterns at once.

The Lime model demonstrated to be a great static analysis tool with high accuracy and low time of execution. This may help developers to increase code quality and implement them on their software development cycle without impacting their time.

9. FUTURE WORKS

The Lime model has two main flaws, the number of antipatterns that check and the Python language dependency.

In the first case, the project would greatly adapt numerous antipatterns, as its adaptation is ensured by the usage of a common API. And future expandability would help to the objective of the model, increase code quality.

For the second part, the dependency is highly coupled since it uses the Flake8 API, which can only check for Python code. Although, it can be explored the usage of another framework, API, or even plugins that would help to cover other programming languages.

Also, another big opportunity that brings the Lime model is to adapt it into other objectives such as checking code grammar in multiple languages or ensure code guidelines.

10. REFERENCES

Arcelli, D., & Di Pompeo, D. (2017). Applying Design Patterns to Remove Software Performance Antipatterns: A Preliminary Approach. Procedia Computer Science, 109, 521–528. doi:10.1016/j.procs.2017.05.330

Bolaños, S., González, C., & Medina, V. (2011). Antipatterns: A Compendium of Bad Practices in Software Development Processes. Special Issue on Computer Science and Software Engineering. Retrieved from https://www.ijimai.org/journal/bibcite/reference/2367

Fourati, R., Bouassida, N., & Abdallah, H. B. (2011). A Metric-Based Approach for Anti-pattern Detection in UML Designs. Computer and Information Science 2011, 17–33. doi:10.1007/978-3-642-21378-6_2

Liu, X. (2017). muupi: An Abstract Syntax Tree based Mutation Testing Tool for Python 2.x Programs. : Oregon State University. Retrieved from: https://ir.library.oregonstate.edu/concern/graduate_projects/0p0968522

Vetter, A. (2019). Online Detection of Operator Errors in Cloud Computing Using Anti-patterns. JIMD Reports, Volume 45, 1–24. doi:10.1007/978-3-030-11638-5_1

Vassallo, C., Proksch, S., Gall, H. C., & Di Penta, M. (2019). Automated Reporting of Anti-Patterns and Decay in Continuous Integration. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). doi:10.1109/icse.2019.00028

Oyarzun, A., Braun, G., Cecchi, L., & Fillottrani, P. R. (2019). An Automated Technique for Analysis of Orthogonal Variability Models

based on Anti-patterns Detection using DL reasoning. In *XXV Congreso Argentino de Ciencias de la Computación (CACIC)(Universidad Nacional de Río Cuarto, Córdoba, 14 al 18 de octubre de 2019)*.

Stoianov, A., & Sora, I. (2010). Detecting patterns and antipatterns in software using Prolog rules. 2010 International Joint Conference on Computational Cybernetics and Technical Informatics. doi:10.1109/icccyb.2010.5491288

Tan, S. H., Yoshida, H., Prasad, M. R., & Roychoudhury, A. (2016). Anti-patterns in search-based program repair. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016. doi:10.1145/2950290.2950295

Barbez, A., Khomh, F., & Guéhéneuc, Y.-G. (2019). A Machine-learning Based Ensemble Method for Anti-patterns Detection. Journal of Systems and Software, 110486. doi:10.1016/j.jss.2019.110486

Correa, A. L., Werner, C. M. L., & Zaverucha, G. (2000). Object Oriented Design Expertise Reuse: An Approach Based on Heuristics, Design Patterns and Anti-patterns. Lecture Notes in Computer Science, 336–352. doi:10.1007/978-3-540-44995-9_20

Palma, F., Dubois, J., Moha, N., & Guéhéneuc, Y.-G. (2014). Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach. Lecture Notes in Computer Science, 230–244. doi:10.1007/978-3-662-45391-9_16

Ulrich W. & Newcomb P. (2010). Information Systems Transformation. In The MK/OMG Press. doi.org/10.1016/B978-0-12-374913-0.00003-2.

11. APPENDIX

Lambda Assigning Violation	Alphafold		Manim		ParlAI		Textual	
	Detected	Total	Detected	Total	Detected	Total	Detected	Total
	6	7	9	9	1	1	0	0

Figure 1: Amount of Lambda Assigning Violations found in all open-source projects.

WithOpen Violation	Alphafold		Manim		ParlAI		Textual	
	Detected	Total	Detected	Total	Detected	Total	Detected	Total
	0	0	1	1	41	42	1	1

Figure 2: Amount of WithOpen Violations found in all open-source projects.

Type Usage Violation	Alphafold		Manim		ParlAI		Textual	
	Detected	Total	Detected	Total	Detected	Total	Detected	Total
	1	1	16	16	92	95	1	1

Figure 3: Amount of Type Usage Violations found in all open-source projects.

Lambda Assigning Violation	Alphafold	Manim	ParlAI	Textual
	Precision Percentage	Precision Percentage	Precision Percentage	Precision Percentage
	85.71	100.00	100.00	100.00

Figure 4: Precision percentage of Lambda Assigning Violations found in all open-source projects.

WithOpen Violation	Alphafold	Manim	ParlAI	Textual
	Precision Percentage	Precision Percentage	Precision Percentage	Precision Percentage
	100.00	100.00	97.62	100.00

Figure 5: Precision percentage of WithOpen Violations found in all open-source projects.

Type Usage Violation	Alphafold	Manim	ParlAI	Textual
	Precision Percentage	Precision Percentage	Precision Percentage	Precision Percentage
	100.00	100.00	96.84	100.00

Figure 6: Precision percentage of Type Usage Violations found in all open-source projects.

Lambda Assigning Violation	Alphafold	Manim	ParlAI	Textual
	4.31	4.17	4.57	3.85

Figure 7: Time in seconds of Lambda Assigning Violations in all open-source projects.

WithOpen Violation	Alphafold	Manim	ParlAI	Textual
	4.71	4.56	5.15	4.22

Figure 8: Time in seconds of WithOpen Violations in all open-source projects.

Type Usage Violation	Alphafold	Manim	ParlAI	Textual
	5.45	6.87	6.95	4.44

Figure 9: Time in seconds of Type Usage Violations in all open-source projects.

Lime Model	Alphafold	Manim	ParlAI	Textual
	10.51	9.12	11.28	8.53

Figure 10: Time in seconds for the overall lime model in all open-source projects.

Link to source code:
https://github.com/GibranHL0/lime-lynter