

ON THE USAGE OF UML DIAGRAMS IN OPEN SOURCE PROJECTS

Joseph Romeo

Sept 2023

Supervised by
Prof. Dr. Michele Lanza

Co-Supervised by
Dr. Csaba Nagy
Marco Raglianti

Abstract

Every software project requires some form of documentation. Documentation increases the maintainability, comprehensibility, and evolvability of software systems. One important form of documentation is UML diagrams. They help developers comprehend software systems to better evolve and maintain them. They also help developers communicate design ideas and elicit collaboration and discussion.

Although the importance of good diagrams has been shown to have a positive impact on software projects, it is often an afterthought. In the context of open source projects, where communication and collaboration happen asynchronously, good diagrams can be especially valuable. When conversations happen over the span of days, weeks, or even months through issues and pull requests, UML diagrams can offer a way to communicate design ideas in a clear and concise way. Why then is UML underutilized in open source projects?

To shed light on this, we explore the usage of UML in open source repositories. We analyze the commit histories of over 13,000 GitHub repositories to see what types of projects use UML diagrams, and which do not. We explore the formats that UML diagrams are stored in, the tools that are used to create them, and how the popularity of those formats and tools has changed over time. We look at the contributors who create and maintain UML diagrams, and how they differ from other contributors. We do an in-depth analysis of three open source projects, where we examine how UML diagrams are used within these projects, and how that usage has changed over time. We pose the following research questions:

- **RQ1.** How widespread is the use of UML in open source projects?
- **RQ2.** What formats are UML diagrams found in?
- **RQ3.** Who is creating and maintaining UML design diagrams?
- **RQ4.** What types of projects are UML diagrams found in?

By examining these aspects of UML usage in open source projects, we gain insights into why UML is underutilized in open source projects.

Dedicated to my wife, Nona Ebrahimi, for
all of the support she has given me on this
journey these past 2 years. . .

Acknowledgements

A special thanks to my advisor, Prof. Dr. Michele Lanza, for your guidance, support, and vision throughout this journey. Every meeting and conversation was a mix of learning and inspiration, and left me motivated to tackle the next problem. Thank you for your constant guidance in helping me see the forest for the trees, and to always pushing me to elevate my work, and never sell myself short. I will carry the knowledge and skills I have gained working with you for the rest of my life.

A huge thank you to my co-advisors Dr. Csaba Nagy and Marco Raglianti for your endless dedication, for the countless hours spent intensively reading the thesis, for the extremely detailed feedback, and for the many discussions we had that helped shape this work. You helped guide what started as a mess of thoughts with no clear organization into a document that I can be proud of.

Thank you to all of the professors of the MSDE program. The knowledge you have imparted has been invaluable and my capabilities as a software developer and technical writer have exceeded any expectations I had when I started this program.

To my wife, Nona, thank you for all of your support these past 2 years. You supported me through the countless late nights and long weekends spent studying. I could not have survived the last 2 years without you.

To my parents, Joe and Lydia, thank you for your support and encouragement throughout my life. You have always been there for me, told me to follow my dreams, and supported me in all that I do and have done. You inspired my love of learning and taught me to never be afraid to try something new.

To my brother, TJ, to whom I have always looked up to, thank you for showing me the value of perseverance. You taught me that when things are challenging, and are not coming easy, you just have to work harder.

Contents

Abstract	iii
Acknowledgements	vii
1 Introduction	1
1.1 Contributions	2
1.1.1 Tools	2
Drifter	2
AViz	2
Author Merge Suggester Endpoint	2
1.1.2 Approaches	3
UML to Source Mapping	3
Author Anti-Aliasing	3
UML Extension Tagging	3
1.1.3 Data	3
UML Tools	3
1.1.4 Example UML Diagrams	3
1.1.5 Tagged UML Files	3
1.2 Document Structure	4
1.2.1 Chapter 2: Related Work	4
1.2.2 Chapter 3: Approach	4
1.2.3 Chapter 4: Implementation	4
1.2.4 Chapter 5: Research Questions	4
1.2.5 Chapter 6: Case Studies	4
1.2.6 Conclusions	4
2 State of the Art	5
2.1 UML Diagrams	5
2.1.1 Defining UML	5
2.1.2 The Usefulness of UML Diagrams	6
2.1.3 UML Tools	6
2.1.4 UML Diagram Extraction	7
2.2 Software Traceability	7
2.3 Architecture Erosion and Consistency	8
2.4 Author Anti-Aliasing	8
2.5 Conclusions	9
3 Approach	11
3.1 Dataset	12
3.1.1 Repository Selection Criteria	12
3.1.2 Definitions	12
3.1.3 Full Dataset Statistics	12

3.1.4	UML Subset Statistics	13
3.1.5	Summarizing the Dataset	14
3.2	Extension Exploration	14
3.2.1	Generating UML Extension Candidate List	15
3.3	Extension Tagging	17
3.4	Author Anti-Aliasing	18
3.4.1	Model	18
3.4.2	Process	18
3.4.3	Definitions	18
3.4.4	Evaluation	19
False Positive Examples		20
False Negative Examples		21
3.4.5	Results	21
3.5	Author Analysis	22
3.6	UML to Source Mapping	23
3.6.1	Definitions	23
Coverage		23
Method Coverage		23
Attribute Coverage		24
3.7	Summary	24
4	Implementation	25
4.1	Definitions	26
4.2	Gitt	26
4.3	Parsers	27
4.3.1	Java Parser	27
4.3.2	UML Parser	29
4.4	Analyzers	31
4.4.1	ProjectTracer	31
4.4.2	Project Analyzer	31
4.4.3	Author Merge Suggestion Analyzer	32
4.5	Databases	33
4.6	CLI	33
4.6.1	Cloning and Summarizing Repositories	34
4.6.2	Author Analysis	34
4.6.3	Diagram Generation	34
4.7	Drifter	35
4.7.1	Package Visualization	35
4.7.2	Java to UML graph	35
4.7.3	Coverage History	36
4.7.4	File History	37
4.8	Summary	37
5	RQs	39
5.1	RQ1: How Widespread is the use of UML in Open Source Projects?	40
5.1.1	Definitions	40
5.1.2	Evolution of UML use	40
5.1.3	Popularity by Extension	41
5.1.4	Conclusions	44
5.2	RQ2: What Formats are UML Diagrams Found in?	45

5.2.1	Finding UML Diagrams in Candidate Extensions	45
	Step 1: Find Example or Counter-Example for each Extension	45
	Step 2: Search Files in /uml/ Paths	45
	Step 3: Search File Names for Keywords	46
	Step 4: Manual Search	46
5.2.2	Results	47
5.2.3	Conclusions	48
5.3	RQ3: Who is Creating and Maintaining UML Design Diagrams?	49
5.3.1	Definitions	49
5.3.2	Methodology	49
5.3.3	Contribution Period of UML Committers vs non-UML Committers	49
5.3.4	Number of UML Committers versus non-UML Committers	52
5.3.5	Number of Commits by UML Committers vs non-UML Committers	54
5.3.6	Are There Dedicated UML Diagrammers?	55
5.3.7	Conclusions	57
5.4	RQ4: What Types of Projects are UML Diagrams Found in?	58
5.4.1	Methodology	58
5.4.2	UML by Main Programming Language	58
	UML in non-OOP Languages – An Example	59
5.4.3	UML by Activity and Community Size	61
5.4.4	Conclusions	61
5.5	RQ0: Why is UML Underutilized in Open Source Projects?	62
5.6	Summary	62
6	Case studies	63
6.1	Definitions	63
6.2	Orekit: An Impressive Feat of Diagramming	64
6.2.1	Method and Attribute Coverage	67
6.2.2	UML to Java References Graph	69
6.2.3	From the Beginning of Time	73
6.2.4	Where is the UML used?	73
6.2.5	UML Committers	74
6.2.6	Conclusions	75
6.3	Teammates: From PowerPoint to PlantUML	76
6.3.1	A Blip in Time	76
6.3.2	From PowerPoint to PlantUML	76
6.3.3	UML Committers	78
6.3.4	Conclusions	79
6.4	Dataverse: PlantUML from the Start	80
6.4.1	Designing Before Coding	80
6.4.2	Documentation Website	82
6.4.3	Authors	83
6.4.4	Conclusions	83
6.5	Summary	84
7	Conclusion	85
7.1	Discussion	85
7.2	Threats to Validity	86
7.3	Future Work	86
7.4	Epilogue	87

A	UML Tools	89
B	UML Examples	91
C	UML Counter-Examples	95
D	UML Extension Tagging	99
	D.1 .argo and .zargo	99
	D.2 .asta	99
	D.3 .cmof	99
	D.4 .dia	100
	D.5 .diagram	100
	D.6 .ecore	100
	D.7 .gliffy	100
	D.8 .iuml, .puml, .plantuml, .platuml	101
	D.9 .mdj	101
	D.10 .mdzip	101
	D.11 .mmd	101
	D.12 .prj	101
	D.13 .pu	102
	D.14 .session	102
	D.15 .ucls	102
	D.16 .uml	102
	D.17 .umlclass and .umlprofile	102
	D.18 .ump	103
	D.19 .uxf	103
	D.20 .vpp	103
	D.21 .xmi	103
	D.22 .yuml	103
	D.23 .zuml	104
E	Queries	105
	E.1 Extension Exploration	105
	Find all file extensions that contain string	105
	Find all file extensions found in /uml/ paths	105
	Find all repositories where extension exists	105
	Find all commits and files where extension exists for given repository	106
	Find all commits and files in uml paths for given extension	106
	E.2 Commit Exploration	106
	Retrieve commit extension statistics	106
F	Author Anti-Aliasing Algorithm Details	107
	F.1 Names match	107
	F.2 Emails match	108
	F.3 Name matches email	108
G	List of UML Repositories	109

List of Figures

2.1	Simple class diagram	6
3.1	General flow of the approach	11
3.2	Number of active repositories by year	12
3.3	Number of repositories by language	13
3.4	Number of active UML repositories by year	14
3.5	PostgresDB ER diagram	15
3.6	Example of recursive merge suggestion	18
3.7	Example of the author visualization tool	19
3.8	Simple class name example	23
4.1	Package diagram for Drifter, Harvest, and CLI tools	25
4.2	Gitt package class diagram	26
4.3	Java parser class diagram	27
4.4	Class diagram for Java model	28
4.5	Class diagram with various method parameter formats	29
4.6	Class diagram for UML model	30
4.7	ProjectTracer class diagram	31
4.8	Persisted metrics class diagram	31
4.9	Edge metrics class diagram	32
4.10	Author merge suggester HTTP API	32
4.11	Dbms class diagram	33
4.12	Annotated package visualization	35
4.13	Java to UML graph	36
4.14	Coverage history graph – Release view	36
4.15	File history for <i>Instructor.java</i>	37
5.1	Number of repositories with UML	40
5.2	Percentage of repositories with UML	41
5.3	Number of repositories with UML extensions by year	42
5.4	Average contribution periods boxplot of UML committers vs non-UML committers	50
5.5	Average contribution periods scatterplot of UML committers versus non-UML committers	50
5.6	Number of UML committers versus non-UML committers	52
5.7	Proportion of commits by UML committers versus non-UML committers	54
5.8	Average number of commits by UML committers versus non-UML committers	55
5.9	Number of repositories with UML by language	58
5.10	Percentage of repositories with UML by language	59
5.11	Class diagram from arm-software/arm-trusted-firmware	60
6.1	Orekit release view coverage history	64
6.2	Orekit package diagram from release 7.2	65
6.3	Evolution of Orekit Packages	66

6.4	Orekit DSSTCentralBody file history	67
6.5	Orekit time package simplified class diagram	67
6.6	Orekit method coverage release 7.2	68
6.7	Orekit attribute coverage release 7.2	68
6.8	Orekit bodies package attribute coverage release 7.2	68
6.9	Orekit bodies package method coverage release 7.2	68
6.10	Orekit UML to Java references graph release 11.3.2	69
6.11	UnscentedKalmanEstimator diagram history	70
6.12	MatricesHarvester class diagram coverage comparison	70
6.13	MatricesHarvester diagram history	71
6.15	Orekit coverage history commit view	73
6.16	Evolution of UML tools in Orekit	74
6.17	Comparison of author statistics	74
6.18	Teammates release coverage history	76
6.19	Teammates storage class diagram from PowerPoint	77
6.20	Teammates storage class diagram from PlantUML	77
6.21	Evolution of UML tools in Teammates	78
6.22	Comparison of author statistics	79
6.23	Dataverse commit coverage history	80
6.24	Dataverse users and groups UML diagram - commit 06.12.2014	82
6.25	Comparison of author statistics	83

List of Tables

3.1	Definitions used in dataset section	12
3.2	Repository statistics	13
3.3	UML repository statistics	14
3.4	Final UML extension candidates list	16
3.5	Definitions for author anti-aliasing	19
3.6	Author anti-aliasing evaluation	20
3.7	Author anti-aliasing false positive example 0	20
3.8	Author anti-aliasing false positive example 1	20
3.9	Author anti-aliasing false negative example 0	21
3.10	Author anti-aliasing false negative example 1	21
3.11	Aliasing percentage	21
3.12	Example of commit extension statistics for a single commit	22
3.13	Definitions for UML to source mapping section	23
4.1	Definitions for implementation chapter	26
5.1	Definitions used in RQ1	40
5.2	Manually searched UML extensions	46
5.3	Extensions with UML examples	47
5.4	Definitions used in RQ3	49
5.5	Authors for rolisteam/rolisteam	51
5.6	Top 5 authors by number of commits for alsa-project/alsa-lib	51
5.7	UML Committers for kubernetes-sigs/cluster-api	53
5.8	Top 5 committers by number of commits for embox/embox	54
5.9	Dedicated diagraphers	56
5.10	Statistical significance of repository statistics of UML vs non-UML	61
6.1	Definitions used in case studies	63
6.2	Orekit GitHub statistics	64
6.3	Orekit release view statistics	64
6.4	Orekit author statistics	74
6.5	Teammates GitHub statistics	76
6.6	Teammates author statistics	78
6.7	UML Committers for Teammates	79
6.8	Dataverse GitHub statistics	80
6.9	Covered references for Dataverse over time	81
6.10	Dataverse author statistics	83
A.1	Popular UML tools and their supported file extensions	90
B.1	Examples of UML diagrams for each extension	93
C.1	Counter-Examples of UML diagrams for each extension	97

G.1 List of 550 UML repositories used in the evaluation. 121

Chapter 1

Introduction

“Every adventure requires a first step.”

— Cheshire Cat

Andrew Watson said that the history of visual modeling can be divided cleanly into two eras, “Before UML” and “After UML” [53]. The period “Before UML” was a time marked by division and strife. With the introduction of Simula and Smalltalk in the 1960s and 1970s [11] the object-oriented paradigm was born. With the advent of object-oriented ideas came many competing ideas for how code should be designed, modeled and visualized. By the 1990s, hundreds of disparate approaches emerged to model object-oriented software systems, resulting in a clash of ideas notable enough to earn the name “The Method Wars” [3]. This was a problem for both software developers and software managers. Software developers did not want to waste time learning a new modeling language that would potentially be irrelevant at another company or on another project. In the absence of standards, software managers were hesitant to invest in modeling languages that might lack future support or have a limited pool of experienced developers in the market.

Grady Booch, Jim Rumbaugh, and Ivar Jacobson, aka the “Three Amigos”, were creators of three popular object-oriented development approaches of their time [10, 23, 43]. They combined their efforts to create a single, unified modeling language. “The Method Wars” were slowly put to rest when the Unified Modeling Language (UML) was adopted as a standard by the Object Management Group (OMG) in 1997. Thus began the era designated as “After UML”. Since becoming a standard, UML has grown in popularity and is now one of the most prominent modeling languages used in software development. UML can and has been used in a variety of ways, from automatic code generation for embedded systems to creating workflow diagrams for business process to designing, architecting and documenting software systems [17, 22, 25, 35, 39]. In this thesis, we focus on the latter case of UML usage in designing and documenting software systems.

Documentation in all forms is often perceived as tedious, time-consuming, expensive, and no fun to create and maintain [2, 27, 40]. As such, the efficacy of such documentation has often been called into question, especially with the agile movement and the phrase “Working software over comprehensive documentation” [18]. Still, even on agile teams, more than half of developers find documentation important or very important, but feel too little is available [51]. Research continues to show time and again that good, high-quality documentation can be a huge benefit to the success of a software project [16, 31, 52]. This notion is especially true when it comes to UML diagrams and documentation based on visual modeling languages. Software design diagrams have been shown to promote better active discussion, both in homogeneous and cross-functional teams [24]. Developers can achieve better functional correctness when making changes with accurate and up-to-date design diagrams [6, 16].

Unfortunately, even given the vast amounts of research showing the benefits of good, quality design documentation, and the standardization of UML, the low quality of documentation in software projects is still a major problem. It is still perceived as one of the leading contributors to the high cost of software maintenance [50].

Given the importance of documentation, and the benefits of UML diagrams, we want to understand how UML is being used in open source repositories. We examine the extent to which UML is employed in open source projects, and what types of projects tend to incorporate UML. We also study which UML diagram formats are most popular, and how that popularity has changed over time. In addition, we explore which characteristics set apart contributors who actively create and maintain UML diagrams from those who do not. In essence, we seek to find answers to these research questions:

- **RQ1.** How widespread is the use of UML in open source projects?
- **RQ2.** What formats are UML diagrams found in?
- **RQ3.** Who is creating and maintaining UML design diagrams?
- **RQ4.** What types of projects are UML diagrams found in?

In addition to these research questions, we also propose one additional research question:

- **RQ0.** Why is UML underutilized in open source projects?

By answering RQ1-RQ4, we look to provide insights into why UML is not more prevalent in open source projects. After exploring these questions, we look at real world examples of UML usage in open source projects.

1.1 Contributions

To support this thesis, we developed tools, approaches, and curated data. In this section we present those contributions, along with the sections where they are discussed in detail.

1.1.1 Tools

Drifter

We developed Drifter, a web application made to explore the evolution of open source projects and their usage of UML diagrams. It allows visualizing the connections between UML diagrams and the source code they describe, how detailed those diagrams are, and how those connections change over time. The tool can be seen at <https://drifter.si.usi.ch/>, and its features are described in Section 4.7. The power of Drifter can be seen in the case studies in Chapter 6, where we explore the usage of UML in real world open source projects.

AViz

The next tool we developed is called AViz (Author Visualizer), a web-application created to visualize commit authors and their aliases. Properly identifying authors in git repositories can be difficult because of the ability to change author names and emails at any point. Authors often commit under multiple aliases. AViz gives the ability to visualize this information, along with the ability to manual clean author data for a git repository. The tool can be accessed at <https://drifter.si.usi.ch/aviz>.

Author Merge Suggester Endpoint

Given the problem that anti-aliasing can cause, we also developed an HTTP end-point, that given a list of authors with name and email, will suggest which authors are likely to be the same person. The end-point can be accessed at <https://drifter.si.usi.ch/api/authors/suggest-merge> and the request and response schema are described in Section 4.4.3.

1.1.2 Approaches

UML to Source Mapping

We developed an approach to make connections between entities from UML diagrams and entities from source code. By making these connections, we can see how much of a software system is covered by UML diagrams, and how that coverage changes over time. This approach supports the visualizations used in Drifter. The approach is described in Section 3.6.

Author Anti-Aliasing

To support AViz, the merge suggestion end-point, and answering the research questions, we developed an approach for performing author anti-aliasing that is tuned to GitHub repositories. The approach is described in Section 3.4.

UML Extension Tagging

We also developed an approach for tagging UML diagrams in numerous formats. We tag 9875 UML diagrams with 28 different extensions across 550 GitHub repositories. The approach for performing this tagging is described at a high level in Section 3.3 and in detail in Appendix D.

1.1.3 Data

UML Tools

In our exploration of UML usage, we curated a list of 30 popular UML tools and the file extensions that they work with. The list includes the file formats that they can import and export, and also the project file formats that they use. This list can be found in Appendix A.

1.1.4 Example UML Diagrams

We collected examples of UML diagrams in various formats. While exploring which file formats UML diagrams are found in, we collected examples of UML diagrams in those formats. The list includes the file format, the repository name, the commit hash, and the file path of the UML diagrams. This list can be found in Appendix B. In addition to examples, we also collected counter-examples of files that are not UML diagrams. This counter-examples list is provided in the same format as the examples list, and can be found in Appendix C.

1.1.5 Tagged UML Files

The CSV containing the repository name and file name of the 9875 UML diagrams we tagged is available at <https://gitlab.reveal.si.usi.ch/students/2022/romeo-joseph/drifter/-/blob/main/data/authors.csv>. The full archive of tagged UML files is available upon request, but the list of repositories and UML extensions found in them is available in Appendix G.

1.2 Document Structure

In this section we give an overview of the structure of this document, and a short description of the contents of each chapter.

1.2.1 Chapter 2: Related Work

In Chapter 2, we look at the related work regarding UML in software development. We give an overview of the history of UML, and the motivating forces behind its creation. We look at what a UML diagram is, why it is useful, and the tools that are available to create them. After exploring UML, we look at related work in software traceability and architecture erosion and consistency. Finally, we look at research on author anti-aliasing in the context of software repositories.

1.2.2 Chapter 3: Approach

In Chapter 3 we look at the approach we take to answer our research questions and perform case studies. We share the repository selection criteria, and statistics about the repositories we analyze. We present the methods we use to build the dataset, tag UML diagrams, perform author anti-aliasing, and map UML diagrams to source code.

1.2.3 Chapter 4: Implementation

In Chapter 4 we look at the architecture of the tools developed to support the approach. We explore the Java and UML parsers, and the source code and UML diagram analyzers that support the visualizations in Drifter. We look at the schema of the data we collect and analyze, and the database technologies we use to store that data. In addition, we share how the CLI tools we developed can be run, and what their inputs and outputs look like. We end the chapter by describing the visualizations in Drifter, and how they are interpreted.

1.2.4 Chapter 5: Research Questions

In Chapter 5 we present the research questions. For each research question we present the data we collected, the analysis we performed, and the findings we discovered.

1.2.5 Chapter 6: Case Studies

In Chapter 6 we present case studies on the usage of UML in open source projects. We look at three GitHub repositories: `cs-si/orekit`, `teammates/teammates`, and `iqss/dataverse`. We explore how they use UML diagrams, and how that usage changes over time.

1.2.6 Conclusions

In Chapter 7 we conclude the thesis with a discussion of the findings and their potential implications. We also share the threats to validity, ideas for future work, and end with a brief discussion on the future of UML.

Chapter 2

Related Work

"It is indeed a desirable thing to be well-descended, but the glory belongs to our ancestors."

— Lucius Mestrius Plutarchus

Software documentation is an important part of software development that can improve the quality of a software product [27]. Research continues to show the vast benefits and importance of documentation [16, 31, 51, 52]. Software traceability is another hallmark of high-quality software systems [13]. While software traceability looks to create links between natural language documentation and source code, we borrow ideas to make similar links between UML diagrams and source code. Architecture consistency is yet another important quality of software systems. Consistency checks often use software architecture recovery techniques to recover architecture from source code [4]. We leverage ideas from these techniques to recover design from source code, and compare the recovered design to the linked design documentation. Identifying authors and all of their aliases in version control systems can be difficult due to the ease in which authors can commit under multiple names and emails [9, 19, 29].

2.1 UML Diagrams

As hardware capabilities increased through the 1950s and 60s, the potential for what software could be used to do also increased. The increasing possibilities lead to software becoming more and more complex. By the late 1960s, the high complexity of software caused many projects and companies to fail, and lead to the coining of the term “software crisis” [54]. This motivated efforts throughout the 1970s and 80s to develop approaches to tame this high complexity. Among these efforts were dozens of competing object-oriented methods and modeling languages [3]. Object-oriented modeling languages were difficult to adopt because there were many competing ideas and no standardization. To combat this issue, Jim Rumbaugh, Ivar Jacobson, and Grady Booch (the Three Amigos) banded together to create UML. UML 1.1 was proposed to the Object Management Group (OMG) in 1997 and was adopted as a standard that same year. The UML specification has evolved throughout the years and is now on version 2.5.1.¹ Since the specification’s inception in the 1990s, it has continually gained popularity and is now the de facto modeling language standard [21].

2.1.1 Defining UML

What UML is exactly and how it should be used changes depending on who is asked. To some, UML should be used to comprehensively model a system so the model can be used to generate code, and in some cases, be executed directly [44, 48]. To others, it is a tool to communicate conceptual ideas with stakeholders, and elicit collaborative discussions [41]. In all of these cases, it is a notation to describe a system’s structure and behavior in a precise way. Figure 2.1 shows an example of a simple UML class diagram (the

¹<https://www.omg.org/spec/UML>

most used UML diagram type [28]). In this diagram we see an interface `Shape` which is realized (implemented) by the classes `Circle`, `Rectangle`, and `Triangle`. We also see a drawing is a composition of shapes that can be rendered.

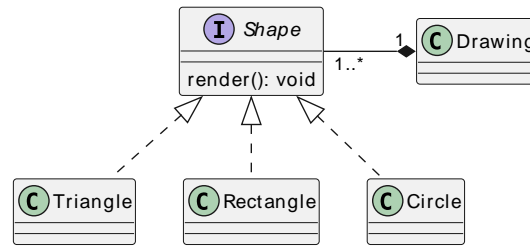


FIGURE 2.1: Simple class diagram

Although there are 13 other types of UML diagrams, we can see the purpose and power of UML in this simple example. Each of the elements in this diagram has a precise meaning. The dotted line with the arrowhead indicates a realization relationship (which can be achieved in Java for example using the `implements` keyword). The diamond with the filled in arrowhead indicates a composition relationship, which tells us the drawing is composed of shapes, and shapes cannot exist without the drawing.

2.1.2 The Usefulness of UML Diagrams

UML diagrams are often used to aid in program comprehension tasks. They can help a developer better create and retain a mental model of the software system. Given the popularity of UML and the continued high cost of software maintenance, the efficacy of using UML diagrams as a program comprehension aid has been studied extensively [6, 16, 20, 45, 46, 47]. Surprisingly, the results of these studies are mixed.

Arisholm *et al.* found that the use of UML diagrams does help with the correctness and design quality of software maintenance tasks, but also increases the time needed to complete those tasks [16]. The additional time needed to use the diagrams is likely worth the cost as it is estimated that over \$500 billion USD was spent on finding and fixing bugs in the U.S. alone in 2020 [30]. Given the cost and time needed for bug finding and fixing, a small increase in time to complete a task is worth the cost if it means better functional correctness and fewer bugs. Although the previous study found UML diagrams to be beneficial, Scanniello *et al.* and Gravino *et al.* instead found using UML diagrams could actually decrease program comprehension and maintainability in certain situations [20, 45].

Through a series of 12 experiments, Scanniello *et al.* [46] found that for diagrams to be most useful, they should be detailed, up-to-date, and closely related to the implementation. As part of our work we capture how the usage of UML diagrams in a system changes over time, including how much of the system they cover, and how detailed that coverage is.

2.1.3 UML Tools

There are a number of tools available to create UML diagrams. These tools support a range of capabilities, from simple drawing, to the generation of code from UML models, to full fledged IDEs that support round-trip engineering where changes in the model are reflected in the code and vice-versa [49]. Ozkaya identified 58 such tools used for UML. Ozkaya also identified which tools support code generation, and which tools support the exporting of diagrams in image formats, XML, and XMI [38]. As part of our work, we tag UML diagrams in open source repositories. Given this, in addition to image formats, XML, and XMI, we are also interested in knowing the project formats each tool supports, and what defining characteristics these formats have. We expand on Ozkaya's work by identifying these additional formats, finding defining characteristics that can be used for tagging, and also identifying new tools not present in Ozkaya's work.

2.1.4 UML Diagram Extraction

UML diagrams come in both text and image formats. Karasneh and Chaudron [26] developed a tool called *Img2UML* which can extract class models from UML diagrams. They were able to successfully detect the rectangles that contained class entities 95% of the time, extract text from the enclosing rectangles correctly 92% of the time, and detect the correct relationships between classes 80% of the time.

Chen *et al.* [12] built on this work and developed a tool called *ReSECDI*. The motivation for developing the tool was to improve relationship detection while also making the tool generalize better to more flavors of UML class diagrams. They were able to detect relationships between classes with precision and recall both above 90%. They also achieved moderately better results for detecting class entities.

We were able to get *ReSECDI* running, and verified the results of Chen *et al.* on the dataset they provide. Unfortunately, we found it did not generalize well with our own dataset. In a sample from our dataset, we found that the tool struggles with diagrams that have certain styling, including the default styling for diagrams created by *PlantUML*. Given that the tool did not apply well to our dataset, we made the decision to focus our work on diagrams in text-based formats.

2.2 Software Traceability

Software Traceability is an important quality of software systems [13]. Gotel *et al.* defines requirement traceability as “*the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)*”. The Center of Excellence for Software Systems Traceability (CoEST) extends this definition from the traceability of requirements to encompass the multi-directional traceability centered around diverse artifacts.² These diverse artifacts include (among many others) requirements documents, design documents, source code, bug reports, and test cases [7]. Software traceability is essential for safety-critical systems where it is important to be able to trace each requirement to its corresponding implementation [37].

Manual creation of traceability links between documentation and source code is untenable in large-scale projects, so many automated techniques for traceability have been developed [5, 7, 33, 34]. Antoniol *et al.* developed a tool that uses Information Retrieval (IR) methods to recover links between free text and source code [5]. The method has two inputs: a set of documents (textual artifacts in natural language) and a query generated from source code (*i.e.*, class, attribute, method, and parameter names). The approach is predicated on the fact that developers use meaningful names, and so the constituent parts of a source code entity can be found in the documentation. They achieved high recall close to 100%, but very low precision of 13%.

To improve on the low precision of IR traceability approaches, Lin *et al.* proposed a deep learning approach leveraging BERT techniques [33]. In their approach, they only looked at the top 3 documents returned by a query, so they did not report a recall metric. They did however report a precision of over 90% when considering the top 3 documents, where precision, in this case, means the number of queries that returned a relevant document in the top 3 divided by the total number of queries.

Although software traceability techniques are aimed at natural language documentation, we use similar ideas (*i.e.*, developers use meaningful names) to link source code to design documentation. We restrict our inputs to UML diagrams, which provide a more formal, non-ambiguous way to retrieve information about the system. By making links between the UML diagrams and the source code, we determine which parts of the system are covered by UML. The deep learning approaches used by Lin *et al.* could be useful for future work to expand from UML diagrams to also consider natural language documentation.

²CoEST: <http://www.coest.org/>

2.3 Architecture Erosion and Consistency

As software systems evolve and grow in size and complexity, they often naturally diverge from their intended architecture. Many studies have described this phenomenon as architecture erosion [4, 8, 32]. This often-cited definition is at odds with the actual metrics used to detect it. Baabad *et al.* performed a systematic mapping study of 43 papers and found nearly 100 different metrics used to detect architecture erosion [8]. In each case, the metrics were computed only on the implemented architecture. Given the metrics used to describe architecture erosion, it would be better defined as the deterioration of the implemented architecture.

The idea of keeping the implemented architecture consistent with the intended architecture is better described under the term architecture consistency. Architecture consistency aims to align the architecture and implementation of the system [4, 42]. Many architecture consistency approaches use software architecture recovery techniques [4]. Ducasse et Pollet identified 3 main flavors of software architecture recovery: bottom-up, top-down, or a combination of both [15]. One of the most cited software architecture recovery techniques is a top-down approach called the reflexion model that was developed by Murphy *et al.* [36]. In the reflexion model, an architect creates a high-level architecture view of a system of interest. Next, a source model (*i.e.*, inheritance hierarchy) is extracted from the source code, and then mapped by the architect to the high-level view. The mapping is then automatically checked for consistency. Although this process yields good results for architecture recovery, it is a very manual and time-consuming process. We do, however, leverage ideas from the reflexion model to recover the design of a system from the source code. Since design is lower level, and more closely represented in the source code, we avoid the manual parts of the reflexion model technique, which involve mapping the low-level design to the high-level architectural views. In our work, we extract a source model similarly to the reflexion model technique, and also a diagram model from the diagrams.

2.4 Author Anti-Aliasing

Mining data from version control systems such as git is subject to the problem of author aliasing [19]. Identifying the true identity of an author can be difficult because authors can commit under multiple aliases (names and emails). Bird *et al.* developed an algorithm, which given a list of identities (name, email pairs), identifies identities that should be merged [9]. The algorithm clusters identities based on three similarities: name similarity, email similarity, and name-email similarity. When considering email, they extract the email base (*i.e.*, the part before the @ symbol), and do not consider the domain. The algorithm then takes the max similarity score among the three similarities and merges identities with a score above a specified threshold.

Kouters *et al.* evaluated the algorithm of Bird *et al.* and found it is sensitive to name ordering (*i.e.*, first name, last name vs. last name, first name), as well as email prefixes with common first names [29]. Gote *et al.* expand on this work by accounting for the ordering of names, and also considering the possibility of usernames in the email base (*i.e.*, Joseph Romeo → jromeo@gmail.com). They calculate similarity metrics against the full name, combinations of the first, penultimate, and last names, the email bases, and combinations of the email bases against first and last names. To avoid the problem of common names present in the algorithm of Bird *et al.*, they take the average of the top two similarity scores as the comparison against the threshold versus the max score. In the case where full names are identical, full emails are identical, or both first and last name appear in the email base, they consider an identity match regardless of the average similarity score. In both the work of Bird *et al.* and Gote *et al.*, preprocessing is done on name and email. Gote *et al.* convert non-ASCII characters to their closest ASCII counterpart, use all lower case, replace delimiting punctuation with spaces, and remove non-alphabetical characters (except the @ symbol), and remove common strings from names.

We expand on the work of Gote *et al.* by tuning the algorithm to GitHub repositories, and also taking into account names in the domain of email addresses. To tune to GitHub repositories, we eliminated names and emails we found were extremely common in our dataset. We do not consider matching the following names: *unknown*, *anonymous*, *anon*, and *none*. We do not consider emails containing the words *unknown*, *anonymous*, *devnull*, *noreply*, *none@none*, and *root@localhost*. In addition, we consider names in the domain of email addresses for those who host their own email accounts (*i.e.*, *mail@jromeo.com*). When extracting the email for comparison, we extract the domain instead of the base if the first or last name appears in the domain.

2.5 Conclusions

In our work, we look to make links between UML diagrams and source code. Software traceability instead looks to make links between natural language documentation and source code. Although software traceability techniques are aimed at natural language documentation, we use similar ideas to link source code to design documentation, which can be used to determine which parts of the implementation are covered by UML diagrams, and how detailed that coverage is.

Architecture consistency is the idea that the intended architecture of a system should be consistent with the implemented architecture. An architecture consistency check is often done with the help of a software architecture recovery technique called reflexion models. Although reflexion model is aimed at recovering the implemented architecture of a system, we can use similar ideas to recover the implemented design instead. Since design is lower level, and more closely represented in the source code, we can avoid the manual, time-consuming parts of the reflexion model technique.

UML diagrams also come in both text and image formats, and there are a number of tools available to create UML diagrams. Where previous work has looked at whether these tools support code generation, XMI, and XML, we look to expand this by also looking at the project formats each tool supports, and what defining characteristics each format has. We focus on textual formats over image formats because we found the tools developed to extract information from image based UML diagrams did not generalize well to our dataset.

Chapter 3

Approach

"Success is a journey, not a destination. The doing is often more important than the outcome."

— Arthur Ashe

In this chapter, we discuss the approach we take to answer our research questions and perform the case studies mentioned in Chapter 1. We can see the general flow of the approach in Figure 3.1. In the first step, we create our dataset, which involves choosing the repositories that we will analyze, cloning them, and summarizing information from those repositories in a DB to be used in later steps. In the next step, we explore UML file extensions and diagramming tools, and tag all the UML files we can find in our dataset. With the tagged UML files, we answer RQ1: how widespread is the use of UML in open source projects, RQ2: what formats are UML diagrams found in, and RQ4: what types of projects are UML diagrams found in, which are discussed in Chapter 5. For step three, we perform git commit author anti-aliasing on repositories with at least one tagged UML file. Given the cleaned author data and the tagged UML files, we analyze the authors and their commits to answer RQ3: who is creating and maintaining UML design diagrams, which is also discussed in Chapter 5. Finally, to perform the case studies in Chapter 6, we developed a tool called Drifter to visualize the usage of UML in open source repositories. The tool is described in Section 4.7, but here we describe the approach used to make connections between UML diagrams and source code that are leveraged by Drifter.

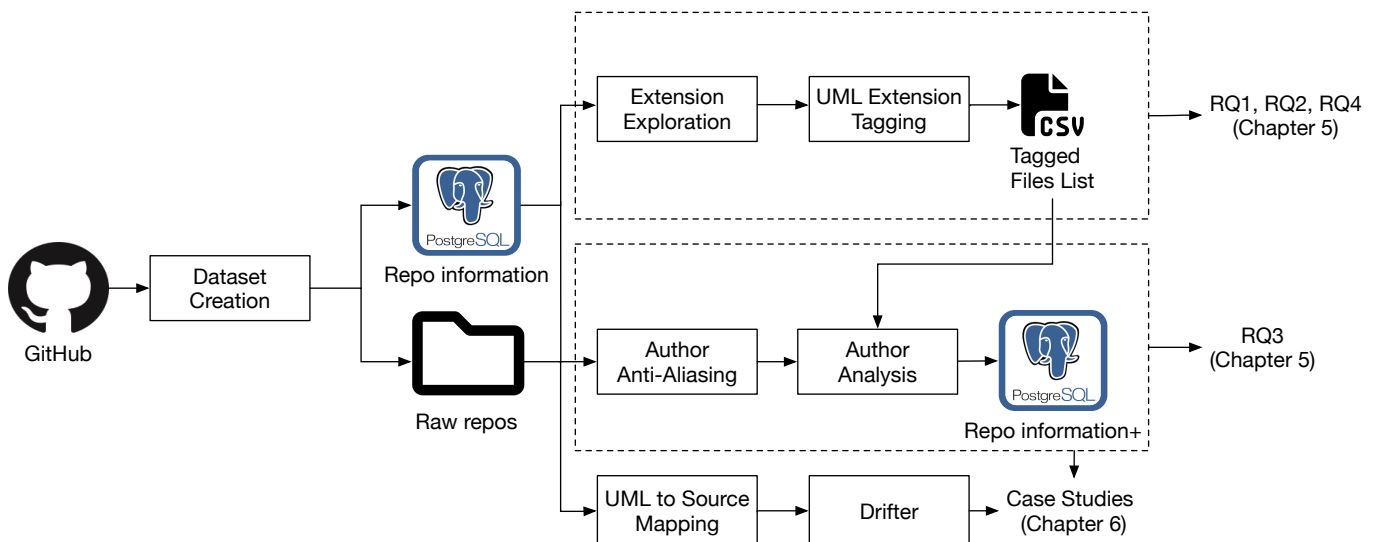


FIGURE 3.1: General flow of the approach

3.1 Dataset

3.1.1 Repository Selection Criteria

We use the SEART GHS tool to gather a relevant set of GitHub projects [14]. We select projects with at least 2,000 commits to eliminate toy projects, which ideally leaves us with projects complex enough to warrant the use of diagrams. We choose projects with at least 10 contributors to ensure the need for collaboration, which increases the utility of diagrams. Last, we take projects with at least 100 stars to ensure the project is considered useful by the open source community, and end up with 13,563 repositories (forks excluded). Each of the 13,563 repositories were cloned over a 24-hour period on April 1st, 2023. These 13,563 represent the full dataset, but for most of the analysis performed in this thesis, we use only a subset of those repositories. The subset is composed of the repositories that we found UML diagrams in, and consists of 550 repositories. Below we present both the full dataset and the subset containing UML.

3.1.2 Definitions

In Table 3.1 we present definitions that are useful throughout the dataset section.

Term	Description
Active Repository	A repository is considered active if it has at least 1 commit in a year (considering the main branch of the repository).

TABLE 3.1: Definitions used in dataset section

3.1.3 Full Dataset Statistics

Figure 3.2 shows the number of active repositories by year. There are a number of repositories active before GitHub was released in 2005 due to the migration from version control tools that existed before GitHub. We also found inconsistent commit dates in a small number of repositories. This can be due to manual override by the committer or discrepancies between the committer’s and the server’s machine clocks. Given that, we filtered out commits after 04.01.2023, the date each repository was cloned, as they should not be possible.

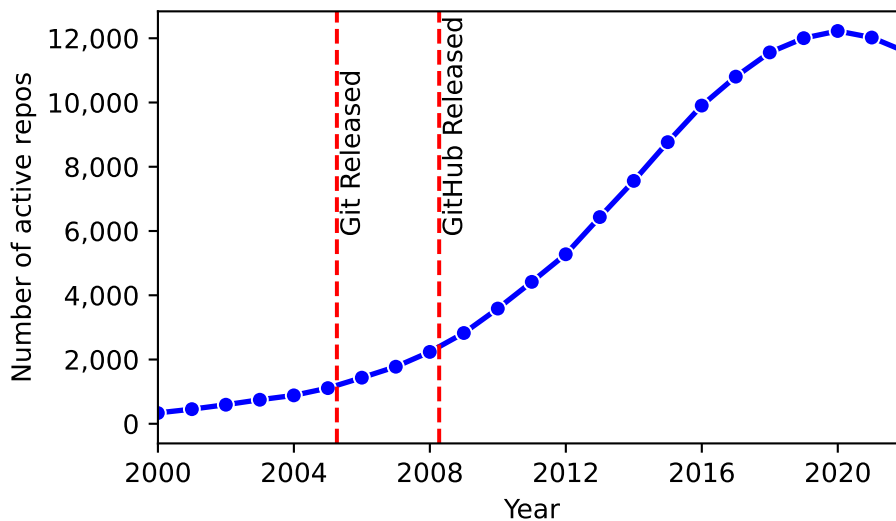


FIGURE 3.2: Number of active repositories by year

Table 3.2 shows general repository statistics which come from the SEART GHS tool. We can see that we have a good mix of projects from moderately popular (100 stars) to extremely popular (321,889 stars), and from moderately sized (2,000 commits) to massive (841,401 commits). We also share Figure 3.3 which shows the number of repositories by main language.

Metric	Min	Median	Max	Mean	Stddev
Stars	100	706	321,889	3,221	8,941
Forks	0	210	91,233	737	2,529
Watchers	0	51	8,428	124	294
Commits	2,000	4,161	841,401	9,457	25,949
Contributors	10	63	12,921	112	235
Issues	0	456	117,484	1,082	2,461

TABLE 3.2: Repository statistics

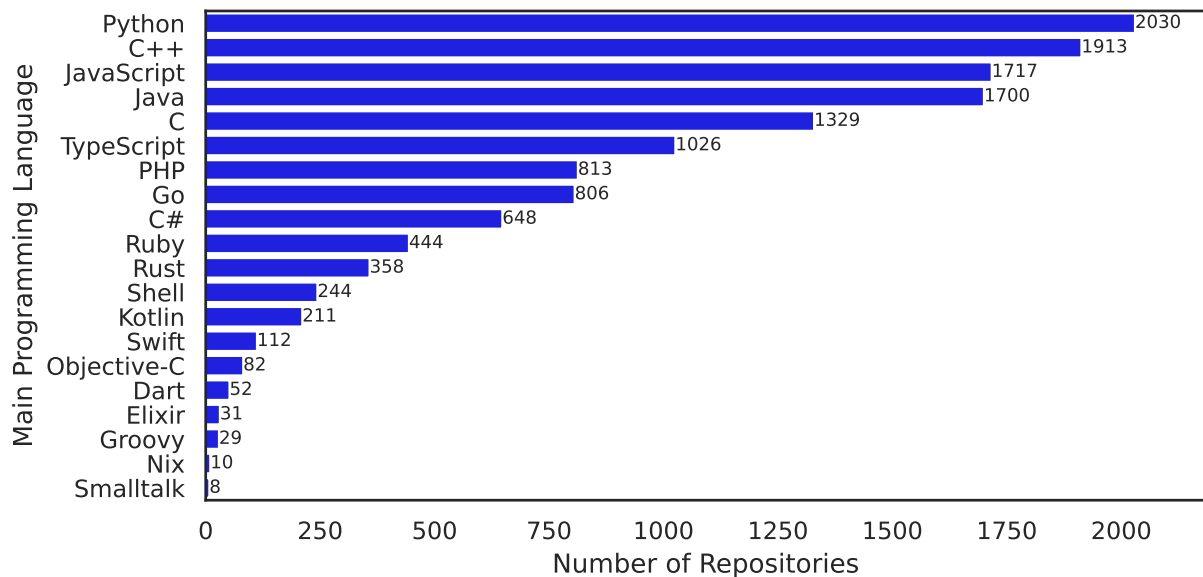


FIGURE 3.3: Number of repositories by language

3.1.4 UML Subset Statistics

Figure 3.4 shows the number of UML repositories active by year. This includes all activity, and not just activity to UML diagrams. The first commit to one of the UML repositories was in 1994, but we show data here and throughout the thesis starting from 2000. The year 2000 marks the first year a commit was made to UML in our dataset. In Table 3.3 we show general statistics for the UML repositories. In Section 5.4 we compare these repository statistics against the full dataset to see if there are any statistically significant differences.

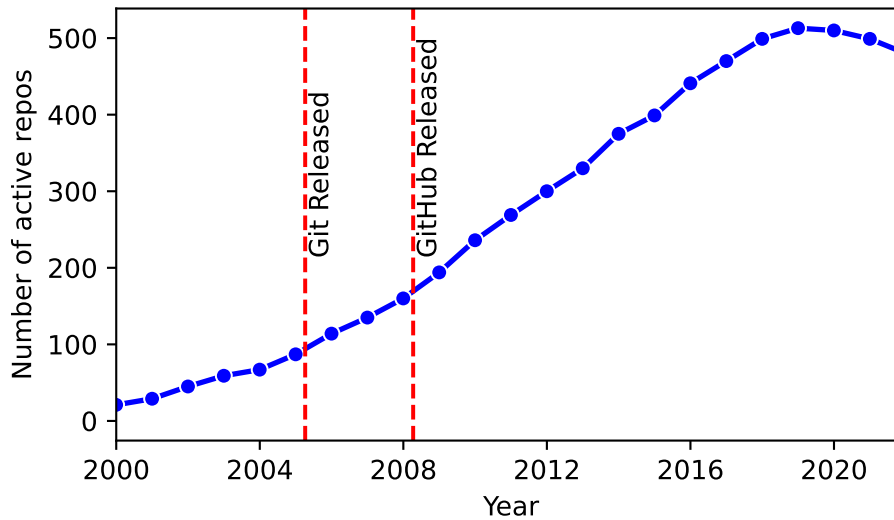


FIGURE 3.4: Number of active UML repositories by year

Metric	Min	Median	Max	Mean	Stddev
Stars	100	535	79,954	2,443	6,980
Forks	11	216	50,497	762	2,788
Watchers	6	51	3,832	122	276
Commits	2,001	6,821	156,849	14,305	20,300
Contributors	10	62	441	102	99
Issues	0	519	27,767	1,308	2,712

TABLE 3.3: UML repository statistics

3.1.5 Summarizing the Dataset

To explore the extensions in the dataset, and discover which are used for UML, we require a quick way to explore the dataset. Given the large number of repositories, and with that, the huge number of files that come with them, running file searches using `find` and `grep` is not feasible. Instead, while cloning each repository, we also gathered additional information about them and stored it in a PostgreSQL. The `repositories`, `file_extensions`, and `file_paths` tables shown in Figure 3.5 are all filled during the cloning step and used extensively in the extensions exploration step. We also filled in the `releases` table during the cloning step. The Drifter tool uses releases from GitHub to select specific versions of a given repository for exploration. GitHub releases can be deleted or modified, and new releases can be added. To avoid inconsistencies between our cloned repository dataset and GitHub, we collected the releases from GitHub upfront. The remaining tables in the ER diagram are filled during the anti-aliasing and author analysis steps.

3.2 Extension Exploration

To answer RQ2: what formats are UML diagrams found in, we take a two-step approach. In the first step we generate a candidate list of UML extensions. In the second step, we search our repository dataset for examples of these extensions. We discuss here the first step of generating a candidate list of UML extensions. Exploring that list and determining which extensions are used for UML is covered in Section 5.2.

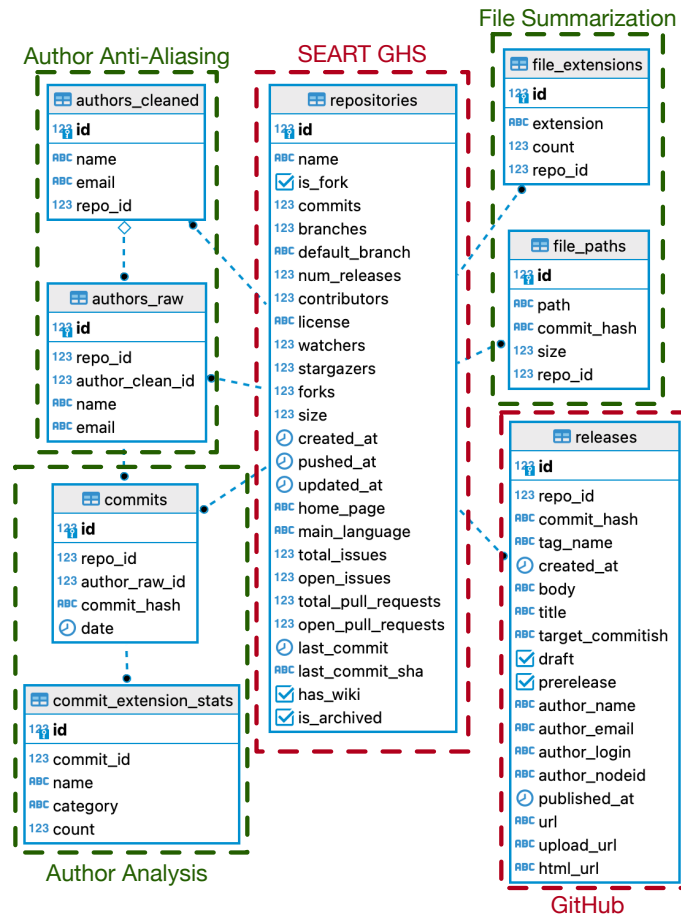


FIGURE 3.5: PostgresDB ER diagram

3.2.1 Generating UML Extension Candidate List

To generate a list of candidate UML extensions we followed a three-step approach. In Step 1, we googled the top UML diagramming tools and identified 30 popular tools used to create UML diagrams. To determine which file extensions each tool works with, we downloaded and installed them. We checked their import, export, and save functionalities to gather the list of extensions that each tool works with. There were a small number of tools that we were not able to successfully install and run, marked with asterisks in Table A.1 in Appendix A. For those we relied on the documentation or YouTube videos of the tools in action to determine which file extensions the given tools work with. The list of importable and exportable extensions for each of these popular UML diagramming tools can be found in Table A.1.

In Step 2, we searched for all extensions with UML in the name which show up in at least two repositories.¹ In the final step, we searched for any file paths with UML in the name.² After generating this list, we removed any extensions that we found no examples of in our dataset,³ along with any extensions that seemed unlikely to be UML extensions (e.g., .java, .jar, .am). Putting all of these extensions together, we generated the list of candidate UML extensions seen in Table 3.4.

¹See query [Find all file extensions that contain string](#) in Appendix E



²See query [Find all file extensions found in /uml/ paths](#) in Appendix E

³Verfied using query [Find all repositories where extension exists](#) in Appendix E

Extension	Repo Count	Classification	Extension	Repo Count	Classification
.md	13,238	UML Extension	.emf	45	UML Extension
.txt	12,442	UML Extension	.ecore	42	UML Extension, Tagged UML Extension
.png	11,916	UML Extension	.mmd	38	UML Extension, Tagged UML Extension
.html	10,400	UML Extension	.session	36	UML Extension, Tagged UML Extension
.xml	8,958	UML Extension	.ucls	33	UML Extension, Tagged UML Extension
.svg	8,446	UML Extension	.vpp	32	UML Extension, Tagged UML Extension
.jpg	7,268	UML Extension	.zargo	32	UML Extension, Tagged UML Extension
.gif	6,432	UML Extension	.uxf	31	UML Extension, Tagged UML Extension
.pdf	3,852	UML Extension	.diagram	30	UML Extension, Tagged UML Extension
.rst	3,541	UML Extension	.pu	20	UML Extension, Tagged UML Extension
.patch	3,335	UML Extension	.mdj	18	UML Extension, Tagged UML Extension
.csv	3,127	UML Extension	.vacuumlo	12	UML Extension
.bmp	1,759	UML Extension	.eap	11	UML Extension
.jpeg	1,598	UML Extension	.umlaut	10	UML Extension
.swf	1,030	UML Extension	.argo	9	UML Extension, Tagged UML Extension
.rtf	965	UML Extension	.umlclass	9	UML Extension, Tagged UML Extension
.xpm	640	UML Extension	.vdx	9	UML Extension
.xlsx	625	UML Extension	.pgml	8	UML Extension
.eps	610	UML Extension	.zuml	7	UML Extension, Tagged UML Extension
.tiff	551	UML Extension	.asta	6	UML Extension, Tagged UML Extension
.docx	521	UML Extension	.iuml	5	UML Extension, Tagged UML Extension
.ps	460	UML Extension	.mdzip	5	UML Extension, Tagged UML Extension
.dia	417	UML Extension, Tagged UML Extension	.vsdm	5	UML Extension
.pptx	378	UML Extension	.yuml	5	UML Extension, Tagged UML Extension
.graffle	373	UML Extension	.mdr	4	UML Extension
.prj	206	UML Extension, Tagged UML Extension	.unt	4	UML Extension
.drawio	191	UML Extension	.edx	3	UML Extension
.vsd	174	UML Extension	.gxml	2	UML Extension
.ppt	167	UML Extension	.platuml	2	UML Extension, Tagged UML Extension
.puml	165	UML Extension, Tagged UML Extension	.umlprofile	2	UML Extension, Tagged UML Extension
.uml	111	UML Extension, Tagged UML Extension	.ump	2	UML Extension, Tagged UML Extension
.xmi	110	UML Extension, Tagged UML Extension	.cmof	1	UML Extension, Tagged UML Extension
.pbm	109	UML Extension	.eddx	1	UML Extension
.vsdx	80	UML Extension	.vssm	1	UML Extension
.wmf	71	UML Extension	.vssx	1	UML Extension
.gliffy	59	UML Extension, Tagged UML Extension	.vstm	1	UML Extension
.plantuml	48	UML Extension, Tagged UML Extension	.vstx	1	UML Extension

TABLE 3.4: Final candidates list (UML Extension, Tagged UML Extension)

3.3 Extension Tagging

To answer RQ1: how widespread is the use of UML in open source projects, and RQ2: what formats are UML diagrams found in, we first need to know which files in our dataset are UML diagrams. We cannot make a decision on what is a UML diagram based solely on the file extension, as many extensions are used for both UML and non-UML purposes (*e.g.*, .png, .svg, .drawio, .graffle). To tag UML files in our dataset, we start with the list of all extensions we discovered were used in UML diagrams during the extensions exploration step. We then eliminated extensions that are too generic (*e.g.*, .html, .md) or are used for document formats and image formats which we do not have the capability of processing automatically. Table 3.4 shows the entire candidate list of extensions, with the extensions that we tagged marked with the  icon, and the extensions we did not tag, but found were used as a UML diagram format marked with the  icon. To tag the files, we first gathered all files in our dataset for each extension, and each repository by unique file name. We end up with a file directory structure as seen below.

```
|-- example-owner0
|   |-- example-repo0
|   |   |-- example1.uml
|   |   |-- example2.uml
|   |-- example-repo1
|   |   |-- somedoc.uml
|-- example-owner1
    |-- example-repo0
        |-- main.uml
```

This file directory structure has two main implications. One, we grab every file throughout the history of the repository with the given file extension, but we take only the first version of it. We make the assumption that if a file is a UML diagram at some point in its history, it is always a UML diagram. The second is that we do not take into account the file path of the file. We make the assumption that if a file with the same name in another path is a UML diagram, then the file in the current path is also a UML diagram. Once the files are gathered, we tag each extension type using regular expressions. For most of the files the regex was applied directly, but there are three special cases. Some extensions are gzip compressed files, so we first decompress them before applying the regex. In other cases they are zipped up directories with a full project in them, so we unzip them before applying regex to the files inside. Lastly, some extensions are SQLite databases, so we run queries on the database and apply the regex to the results. This process is described for each extension in detail in Appendix D.

3.4 Author Anti-Aliasing

Before we can answer RQ3, who is creating and maintaining UML design diagrams, we need to be able to identify the authors of the diagrams. This poses a problem when working with git repositories as authors often go under many aliases throughout the life of a project. A person may have their git username and email set up differently on a home computer versus on a work computer, for instance. They may have switched personal emails overtime, or decided to commit with their github username. For whatever reason a user might have multiple aliases, git commit data can be very noisy when trying to identify commit authors. To address this issue we performed author anti-aliasing on the commit data.

3.4.1 Model

If we refer back to the ER diagram in Figure 3.5 and focus on the `authors_raw`, `authors_cleaned`, and `commits` tables, we can get an idea of what the final output of the author anti-aliasing is. The authors in the `authors_raw` table are taken directly from the git commit data. We group those by name and email. Each raw author has at most 1 "cleaned" author, and a "cleaned" author in the `authors_cleaned` table can have many raw authors. Each raw author also has an associated set of commits in the `commits` table. To find all commits for a cleaned author, we can aggregate all the commits for each raw author associated with the cleaned author. A raw author with no associated cleaned author is an author we found no aliases for.

3.4.2 Process

To perform author anti-aliasing, we follow three main checks to determine whether an author is an alias of another author. The checks are: name match, email match, and email-name match. If any of these checks are positive then we have an alias. The details of these three checks can be found in Appendix F. These three checks are then run recursively for a given author. That is, after getting a set of aliases for an author, we then check each of those aliases for new aliases. An example of this can be seen in Figure 3.6 which was taken from `h2oai/h2o-2`. The authors in red are merged on the first pass of the algorithm by name. The author in green is merged on the second pass of the algorithm because the email matches the second red author. The author in purple is merged on the third pass of the algorithm because the name matches the author in green.



FIGURE 3.6: Example of recursive merge suggestion

3.4.3 Definitions

We developed a frontend visualization tool to view the results of the author anti-aliasing. The tool can be accessed at <https://drifter.si.usi.ch/aviz>. We used the tool to aid in evaluating the efficacy of

the results, which can be seen below in the evaluation section. Here we use a snapshot from the tool in Figure 3.7 to define terms used in the evaluation and results, and also throughout the rest of this thesis.

Term	Description
Raw Author	An author taken directly from the git commit data, grouped by unique names and emails.
Clean Author	An author that represents an aggregation of raw authors who are aliases of each other. These are identified in the graphic by the drop-down arrow on the right, which shows the author has associated aliases.
Alias	A raw author we determine is the same as another raw author.
Real Author	An author we determine as a true representative of an author in the git repository. These are either raw authors who have no aliases, or clean authors which aggregate a set of aliases.
Aliasing Percentage	The percentage of aliases to raw authors ($\frac{RawAuthors - RealAuthors}{RawAuthors} * 100$).

TABLE 3.5: Definitions for author anti-aliasing

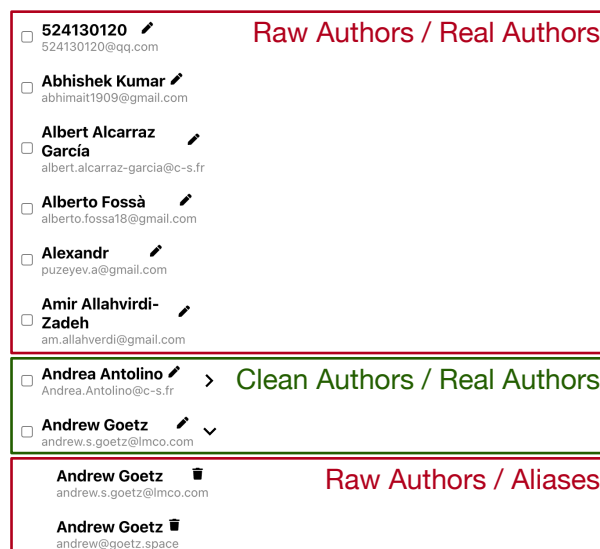


FIGURE 3.7: Example of the author visualization tool

3.4.4 Evaluation

To evaluate the efficacy of the author merging, we took a random sample of 10 repositories. We took the random sample from the repositories that we tagged at least 1 UML file in, rather than the entire set of repositories, because we only ran author merging on the repositories for which we analyzed authors. For each repository in the random sample, we sought to find the number of false positives and false negatives in the author merging. We define a false positive as an author who was merged to another author but should not have been. We define a false negative as an author who should have been merged to another author but was not. To find false positives and false negatives, we used the tool shown in Figure 3.7 to explore the sampled repositories.

Note that since we lack a ground truth for the author merging, we mark false positives and false negatives according to our own best judgement. Also, checking the false positives, in general, is easier than the false negatives. For the false positives, we can look at the authors who were merged and see if they should have been merged. For false negatives, we have to take into account every author in the repository

to determine if we missed any. The larger the repository, the more difficult it becomes to check for false negatives. Table 3.6 shows the summary of the evaluation.

Repo Name	Raw Authors	Real Authors	Aliasing Percentage	Aliases	Clean Authors	FP	FN
controlsystemstudio/cs-studio	201	116	42.3%	128	43	0	5
ehcache/ehcache3	104	66	36.5%	67	29	0	4
opentripplanner/opentripplanner	279	179	35.8%	156	56	2	2
opendds/opendds	134	88	34.3%	73	27	0	1
arm-software/arm-trusted-firmware	623	455	27.0%	186	83	2	0
brightid/brightid	61	45	26.2%	23	7	0	1
kubernetes/enhancements	675	507	24.9%	294	126	5	4
rptools/maptool	87	58	33.3%	46	20	0	2
owncast/owncast	138	125	9.4%	22	9	0	3
nuitka/nuitka	140	131	6.4%	18	9	0	0

TABLE 3.6: Author anti-aliasing evaluation (FP=False Positive, FN=False Negative)

False Positive Examples

Here we show two examples of false positives in Tables 3.7 and 3.8. In the first, the two authors with the name *Thomas Gran* are joined correctly, but *grant-humphries* is incorrectly joined. The bolded parts in the name *Thomas Gran* and the email *grant-humphries@gmail.com* trigger a username match (done as part of email-name check). In this case, one username we consider for *Thomas Gran* is *grant*, which shows up in *grant-humphries@gmail.com*. We see a similar false positive on the right, with *Bo-Chen Chen* matching *chenbaozi@phytium.com.cn*. All false positives we found were triggered by the same matching rule.

Name	Email
Thomas Gran	tgr@capraconsulting.no
Thomas Gran	t2gran@gmail.com
grant-humphries ⊕	grant.humphries@gmail.com

TABLE 3.7: False positive example: opentripplanner/opentripplanner (⊕ false positive)

Name	Email
Rex-BC Chen	rex-bc.chen@mediatek.corp-partner.google.com
Rex-BC Chen	rex-bc.chen@mediatek.com
Bo-Chen Chen	rex-bc.chen@mediatek.com
Chen Baozi ⊕	chenbaozi@phytium.com.cn

TABLE 3.8: False positive example: arm-software/arm-trusted-firmware (⊕ false positive)

The second example we show as a false positive, *Chen Baozi*, highlights one of the difficulties of marking false positives and false negatives without a ground truth. There is some ambiguity as to whether *Chen*

Baozi is an alias of *Bo-Chen Chen* and *Rex-BC Chen*. Maybe BC in *BC Chen* stands for *Baozi Chen*. In these cases we use our best judgement for the marking of false positives and false negatives.

False Negative Examples

Tables 3.9 and 3.10 show examples of false negatives.

Name	Email
xichen	ztyaner11@gmail.com
ztyaner11	ztyaner11@gmail.com
Xihui	ztyaner11@gmail.com
xihui	ztyaner11@gmail.com
Xihui Chen \ominus	chenx1@ornl.gov

TABLE 3.9: False negative example: arm-software/arm-trusted-firmware (\ominus false negative)

Name	Email
tnakamoto	devnull@localhost
Takashi Nakamoto (Cosylab)	takashi.nakamoto@cosylab.com
Takashi Nakamoto	takashi.nakamoto@cosylab.com
tnakamoto	tnakamoto@JCSL_WS001
Takashi Nakaomoto	devnull@localhost
nakamoto \ominus	devnull@localhost

TABLE 3.10: False negative example: controlsystemstudio/cs-studio (\ominus false negative)

3.4.5 Results

Now that we have seen an evaluation of the author anti-aliasing, let us take a look at some raw numbers coming from this process. Table 3.11 shows statistics on the author aliasing percentage. With a median of 28.08%, we can see that in over half of the repositories, at least 1 of 4 authors are aliases. We found only 4 repositories which have an aliasing percentage of 0%. We also found some extreme cases where the aliasing percentage is as high as 66.42%, meaning nearly 2 out of 3 authors is an alias. Even the median case of 1 of 4 authors being an alias could significantly skew our analysis, and shows the importance of performing author anti-aliasing.

Repo Count	Median	Min	Max	Mean	Std Dev
550	28.08%	0.00%	66.42%	28.71%	12.61%

TABLE 3.11: Aliasing percentage $\frac{RawAuthors - RealAuthors}{RawAuthors} * 100$

3.5 Author Analysis

If we refer back to the ER diagram in Figure 3.5, we can see the output of the author analysis step are the `commits` and `commit_extension_stats` tables. Given the input of the tagged UML file list, we gather commit extension statistics for each commit in the repositories in our dataset. An example output of a query to these tables can be seen in Table 3.12.⁴

Repo Name	Commit Hash	Extension	Category	Count
cs-si/orekit	c3465033...	.puml	uml	5
cs-si/orekit	c3465033...	.java	code	119
cs-si/orekit	c3465033...	.md	other	1

TABLE 3.12: Example of commit extension statistics for a single commit

The above table shows a commit where 119 .java files were touched, 5 .puml files were touched, and 1 .md file was touched. We can see the output of UML tagging in the `Category` column, which shows that the .puml file is a UML file type. With the four tables `authors_cleaned`, `authors_raw`, `commits`, and `commit_extension_stats`, we generate the graphs in Section 5.3 for RQ3: Who is creating and maintaining UML design diagrams.

⁴See query [Find all repositories where extension exists](#) in Appendix E

3.6 UML to Source Mapping

Drifter is a tool we developed which provides visualizations that help explore a repositories' usage of UML over time. It is used in Chapter 6 where we share case studies of the usage of UML in GitHub repositories. The visualizations are presented in Section 4.7. Here we present the metrics we use to generate the visualizations and the approach behind calculating those metrics.

3.6.1 Definitions

In Table 3.13, we provide a few useful definitions that will be helpful throughout this section.

Term	Description
Java Reference	A Java class, interface, or enum.
UML Reference	A class, interface, or enum in a UML diagram.
Reference	A Java or UML reference.

TABLE 3.13: Definitions for UML to source mapping section

Coverage

One of the main goals is to capture how well a repository is covered by UML diagrams. To do this we developed the coverage percentage metric. Before understanding what coverage percentage is, we first need to understand what a covered Java reference is. We say a Java reference is covered by a UML diagram if it is referenced by at least one diagram. To determine if a Java reference is referenced by a diagram we compare the fully qualified name of the Java reference from the source code to the fully qualified name of the Java reference from the diagram. In the example in Figure 3.8, the Java reference in both the PlantUML example and Java example would have a fully qualified name of `org.some.pkg.Example`.

```
package org.some.pkg {
    class Example
}
```

LISTING 3.1: Example puml file

```
package org.some.pkg;
public class Example {
}
```

LISTING 3.2: Example Java file

FIGURE 3.8: Simple class name example

We do not require an exact match of the fully qualified name to consider a Java reference covered by a diagram. Instead, we check that the package structure is accurate, but not necessarily complete. For example, given the Java reference in Figure 3.8, we would accept `some.pkg.Example`, `org.pkg.Example`, and `org.some.pkg.Example` as valid matches, but not `pkg.some.Example`.

Method Coverage

Aside from the relationship hierarchy, one of the main aspects we hope to see in the UML class diagram are the methods of the classes. Seeing the methods in the class diagram gives us an idea of what the behaviors of the classes are. For that reason, we added the idea of method coverage. Method coverage starts first with coverage. Once we have determined a Java reference is covered by a diagram, we then check if its methods are also present in that diagram. We check specifically for methods marked in the source code with `public`, `protected`, or `package-private` visibility, as these are the methods which give the most information about

the capabilities of the class. Method coverage is the percentage of methods in the Java class covered in a diagram. We consider 2 methods a match between the Java source code and a UML diagram if they are in the same class and have the same name.

Java references may be covered by multiple diagrams, and with varying degrees of detail. For instance, a class may show up in multiple class diagrams, in 1 where its full details are shown with all methods, and in another just showing the class hierarchy in a high level package diagram. For this reason, we calculate method coverage using three possible aggregations: average, minimum, and maximum. This would be the method coverage percentage calculated over all of the diagrams.

Attribute Coverage

Public attributes are another aspect that can convey interesting information about a Java reference. We added the idea of attribute coverage to our package visualization. Similarly to method coverage, we only check for public, protected, and package-private attributes. We consider an attribute a match between the Java source code and a UML diagram if the names of the attribute match. We have the same aggregations available for attribute coverage as we do for method coverage.

3.7 Summary

In this chapter, we presented our dataset selection criteria and statistics about that dataset. We shared our approach for exploring UML file extensions and tagging UML files in our dataset. We then looked at the approach, evaluation, and results of author anti-aliasing on our dataset. Last, we presented the approach used to make connections between UML diagrams and source code that are leveraged by Drifter. In the next chapter, we present the implementation details of the tools that were developed to support answering the research questions and to perform the case studies that come later.

Chapter 4

Implementation

"Whatever good things we build end up building us."

— Jim Rohn

In this chapter we look at the architecture of the tools built, which can be seen in Figure 4.1. Starting from the left, we have Drifter, a frontend tool developed to visualize GitHub repositories and their usage of UML. It also contains visualizations for viewing the results of the author anti-aliasing. The frontend communicates with the backend through the API package using HTTP. The server package runs Flask, and houses the server configuration and API endpoints used by the frontend.¹ The parsers package contains the code used for parsing Java and UML into usable entities for analysis. The analyzers package contains the code which makes connections between Java and UML entities and calculates various metrics. It also contains the author anti-aliasing code. The database package contains the code used to interact with the PostgresDB and MongoDB databases. The Gitt package contains the code used to interact with locally cloned git repositories and the GitHub API. Finally, the CLI package contains a suite of CLI tools that are used to create, explore, and analyze the dataset. Each of these packages are discussed in detail in the following sections.

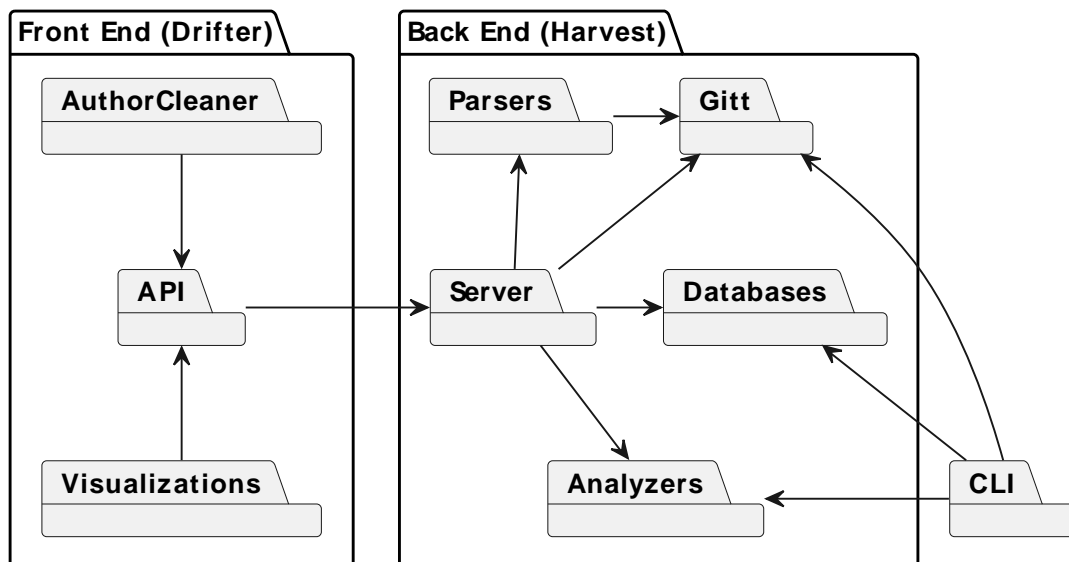


FIGURE 4.1: Package diagram for Drifter, Harvest, and CLI tools

¹Flask, a popular web framework for python: <https://flask.palletsprojects.com/en/2.3.x/>

4.1 Definitions

In Table 4.1, we provide definitions of terms and metrics used throughout this chapter. The full details on the calculations of the metrics below can be found in Chapter 3.

Term	Description
Java Reference	A Java class, interface, or enum.
Coverage	The percentage of Java references that have a corresponding UML diagram.
Method Coverage	For a given Java reference, the percentage of methods covered in a UML diagram. If a class is represented in multiple diagrams, the percentage can be aggregated using min, max, and average across all diagrams.
Attribute Coverage	For a given Java reference, the percentage of attributes covered in a UML diagram. This metric can be aggregated same as method coverage.

TABLE 4.1: Definitions for implementation chapter

4.2 Gitt

The gitt package contains classes which help interact with local git repositories and GitHub.² The class diagram for this package can be seen in Figure 4.2. The GitHub class is a wrapper around the PyGithub library providing an additional polling interface to help deal with the GitHub API rate limits.³



FIGURE 4.2: Gitt package class diagram

The RepoCache class is the main entry for interacting with local git repositories. It is implemented as a python context manager, providing a locking mechanism to ensure that only one thread is accessing the repository at a time.⁴ The API allows repositories to be added to the cache (cloned if they do not already exist), and then accessed by their name.

²The gitt package is named oddly so to avoid name collisions with the python git library.

³PyGithub library: <https://github.com/PyGithub/PyGithub>

⁴Python context manager tutorial: https://book.pythontips.com/en/latest/context_managers.html

The `GitRepository` class is returned by the `RepoCache` when a repository is accessed, and is also implemented as a context manager. The context manager ensures that the repository is always left in a clean state (no uncommitted changes, no untracked files, checked out to the main branch, etc.) when the context is exited. The `GitRepository` class also hosts a number of methods for interacting with local repositories, (e.g., getting every file name that has existed in the repository, getting all authors for the repository and the commits they have created). The parsers package, server, and CLI tools all leverage `GitHub`, `RepoCache`, and `GitRepository` for extracting information from `GitHub` repositories.

4.3 Parsers

To support calculating the various metrics (coverage, method coverage, etc.) discussed in Chapter 3, we need a way to compare Java and UML entities. To do this, we parse Java and UML into models that can be compared. In this section we look at the Java and UML parsers in detail.

4.3.1 Java Parser

In Figure 4.3 we see a class diagram showing the java parsers portion of the parsers package. Looking first at the `JavaParser` class, we see the `parse_project()` method which takes Java source files and returns a `JavaProject`. To parse Java, we leverage the `javalang` library to generate an AST that we then transform into the `JavaProject` model seen in Figure 4.4.⁵ `JavaProject` represents our view of 1 commit for a Java project. The `JavaSnapshot` wraps a `JavaProject` and is used to serialize and store a snapshot of the project in a `MongoDB` database. We can also see a second snapshot class, `JavaFileSnapshot`, which stores another view of the same data in the `MongoDB` database. These snapshots are saved in multiple views in a DB to keep Drifter performant.

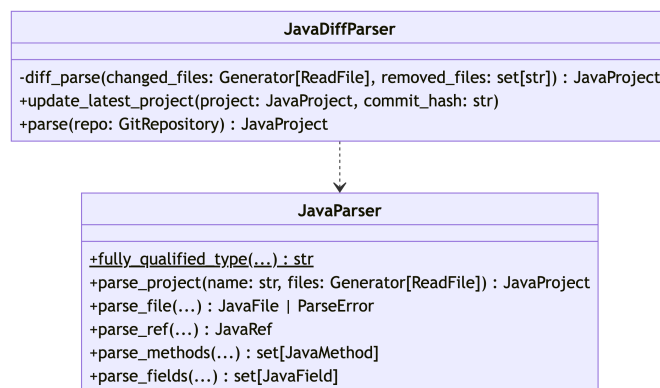


FIGURE 4.3: Java parser class diagram

Parsing a Java project can be an expensive operation. It is especially expensive because generating coverage metrics over time requires parsing many commits. To help ease this burden, we developed the `JavaDiffParser`. `JavaDiffParser` allows us to parse only the files that have changed between 2 commits, given we have a `JavaProject` object for one of the 2 commits. This boosts the performance of our parsing of a project because in many cases only a few files change between commits. Even more, those few file changes may not be source code related.

⁵Javalang library: <https://github.com/c2nes/javalang>

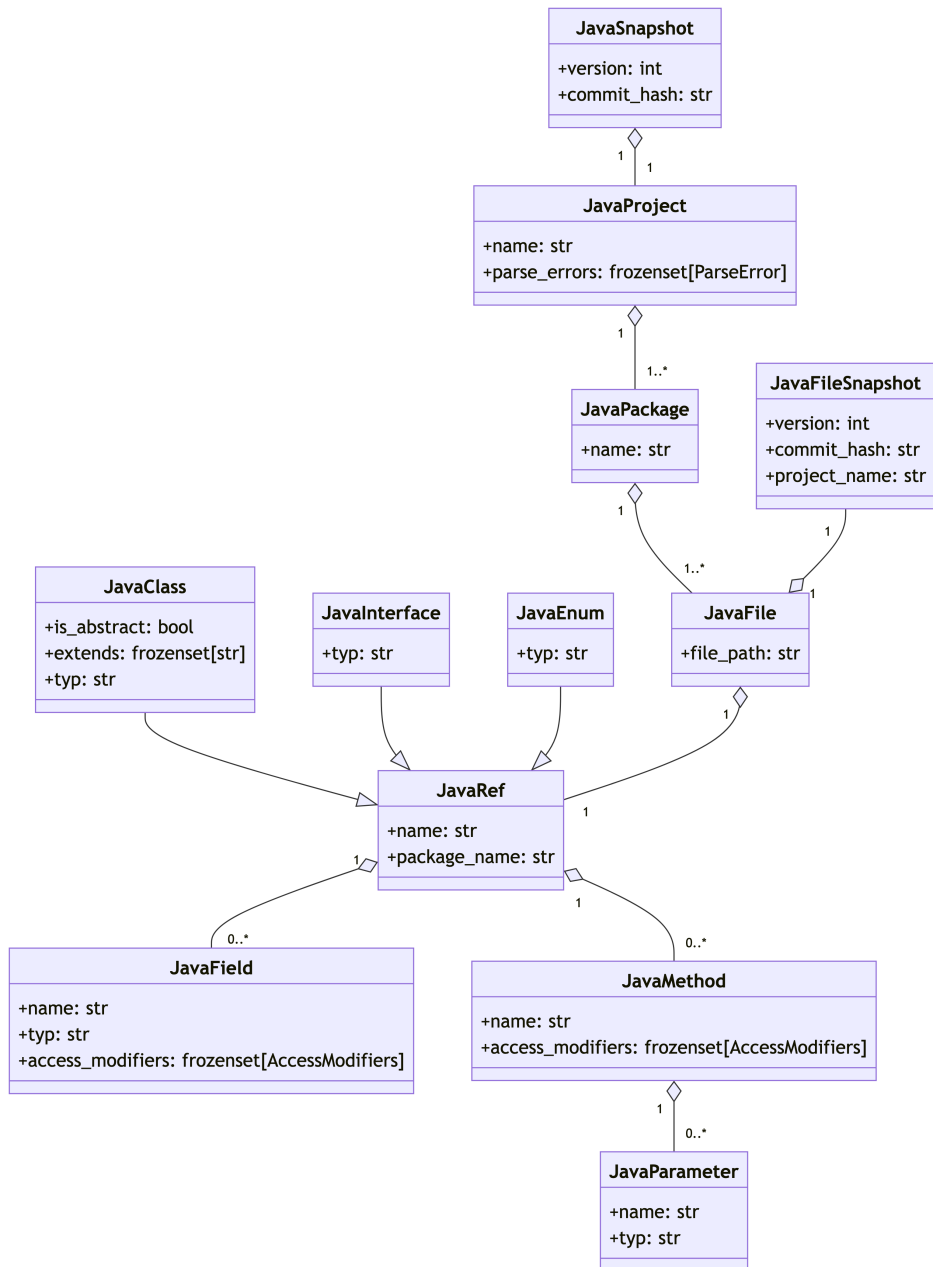


FIGURE 4.4: Class diagram for Java model

4.3.2 UML Parser

UML parsing is challenging because there are many UML diagramming tools and formats. We decided to focus on the PlantUML format because we find it is one of the most used formats in open source repositories in recent years (see Figure 5.3). This poses a challenge though as there are no Python libraries (at time of writing) for parsing PlantUML, and our backend is run as a Python Flask server. We did, however, find a PlantUML parsing library written in NodeJS.⁶ To use this library, we developed a UML parsing service in TypeScript running an Express server.⁷ The service provides an HTTP endpoint that takes a PlantUML diagram as a string and returns a JSON object representing the diagram. For full details of the interaction between Python and the PlantUML server see the source code.⁸ Here we focus on the output of the UML parsing, and some of the difficult scenarios we handle.

Figure 4.6 shows the output of the UML parsing step. The model for a UML snapshot is similar to the Java snapshot. Both have packages, classes, interfaces, enums, references, methods, fields, and arguments. These similarities help when attempting to make connections between Java classes and their associated class diagrams. Although the similarities are numerous, there are some subtle differences between the two models. In the UML model a class diagram has many packages, whereas in the Java model a package has many files. This matches what we expect, as a class diagram may document many packages and classes. Similarly to the Java model, the `UmlProjSnapshot` and `ClassDiagramSnapshot` classes are used to store the model in the MongoDB database.

Now that we have looked extensively at the UML model, let us look at some of the difficulties seen in UML parsing that we do not have with Java parsing. Java has a precise grammar for methods and attributes. UML, on the other hand, is much more flexible. In Figure 4.5 we see how flexible UML diagrams can be. The type can be specified before or after the name. The type can be specified without the name, or vice versa. The arguments may not have a type or name at all.

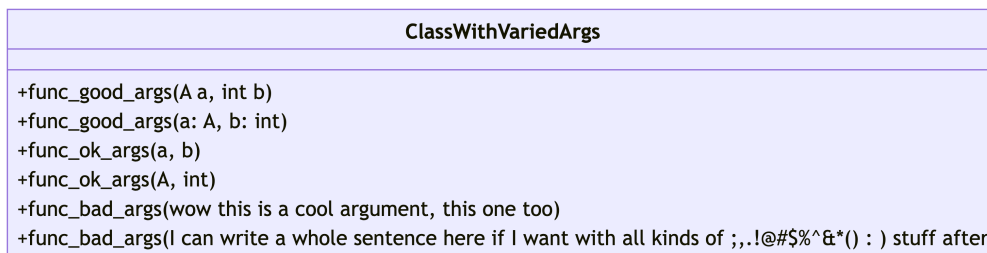


FIGURE 4.5: Class diagram with various method parameter formats

To partially address this issue we use the following rules (splitting arguments by commas):

1. If there are 2 words, we treat the first as the type and the second as the name.
2. If there are 2 words separated by a colon, we treat the first as the name and the second as a type.
3. If there is only 1 word and it starts with a lowercase and is not a primitive type, we treat it as a name with type unknown.
4. If there is only 1 word, and it starts with a capital or is a primitive type, we treat it as a type with name unknown.
5. If there are more than 2 words in the argument, we just return with type and name unknown.

We use similar rules for resolving class attribute names and types.

⁶NodeJS PlantUML Parser: <https://github.com/Enteee/plantuml-parser>

⁷ExpressJS: <https://expressjs.com/>

⁸<https://gitlab.reveal.si.usi.ch/students/2022/romeo-joseph/drifter/> (backend/puml_parser, bakend/harvest/src/parsers)

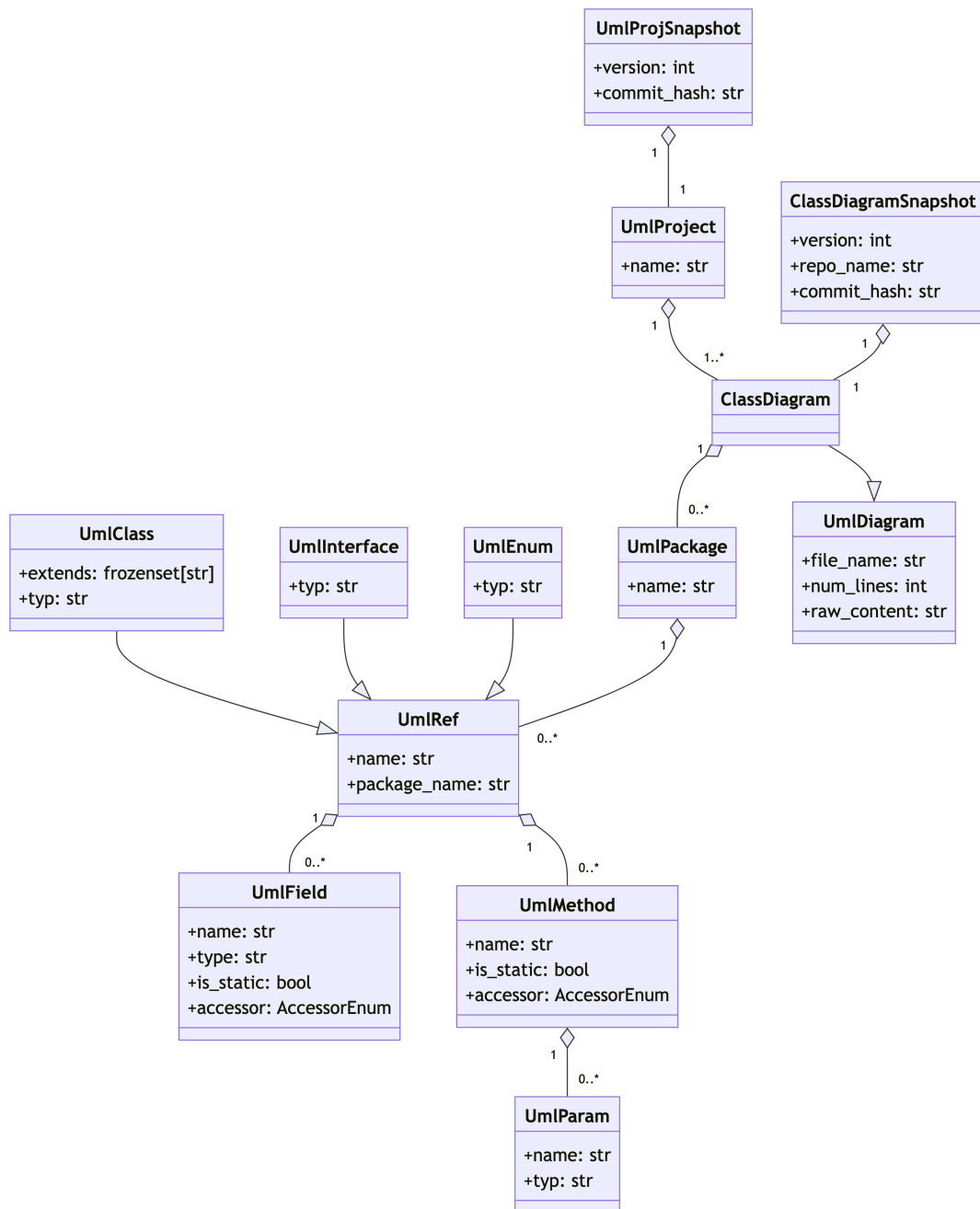


FIGURE 4.6: Class diagram for UML model

4.4 Analyzers

The analyzer package contains a host of methods and classes to analyze various aspects of a GitHub repository. The first to mention, `ProjectTracer`, is a class which given a snapshot of a `UMLProject` and `JavaProject`, makes connections between references in the Java source to references in UML diagrams. The next is the `analyze()` method in the `project_analyzer.py` file. Given a `JavaProject`, `UMLProject`, and the traced connections between them, the `analyze()` method calculates the various coverage metrics. The last analyzer to mention is the `get_author_merge_suggestions()` in the `merge_suggester.py` file, which aids in performing the author anti-aliasing.

4.4.1 ProjectTracer

Figure 4.7 shows the class diagram for the `ProjectTracer` class. It is leveraged by `project_analyzer.py` to retrieve the UML references for a given Java reference. The mapping between Java references and their corresponding UML references are stored as `Edges`.

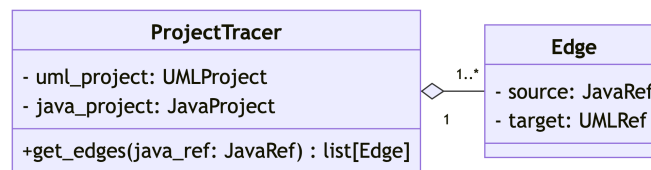


FIGURE 4.7: ProjectTracer class diagram

4.4.2 Project Analyzer

The `analyze()` method in `project_analyzer.py` is the main entry point for calculating various metrics. Figure 4.8 shows the metrics that we calculate which are persisted in a database. `ProjectSnapshotMetrics` represents a snapshot of the metrics for a given commit. It contains the overall coverage percentage for the commit, along with information about the number of references in the project. The `analyze()` method leverages `ProjectTracer` to determine if a Java reference has an edge to a `UmlRef`. If it has an edge then it is covered. These metrics are persisted in the database because we want to be able to visualize their evolution. Although the metrics are quick to calculate for a single commit, it can be expensive to calculate them for every commit in a large repository.

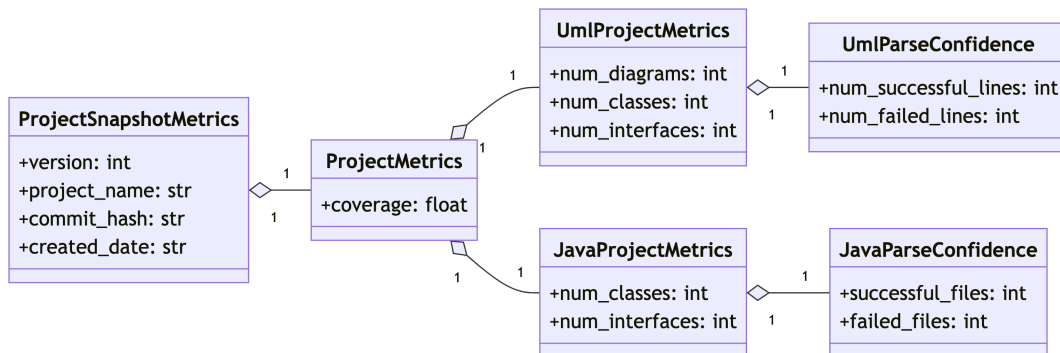


FIGURE 4.8: Persisted metrics class diagram

The next set of metrics we calculate on demand. Given a parsed `JavaProject`, `UMLProject`, and `ProjectTracer`, we calculate additional metrics on each edge between a Java reference and a UML reference. These metrics can be seen in the class diagram in Figure 4.9.

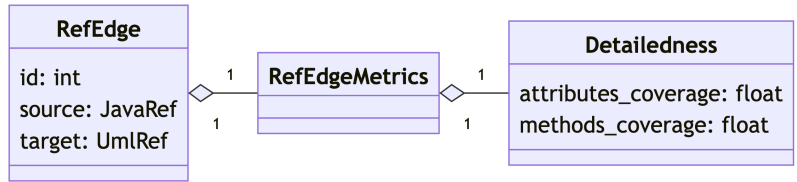


FIGURE 4.9: Edge metrics class diagram

4.4.3 Author Merge Suggestion Analyzer

The final analyzer we look at is the author merge suggester which is leveraged to perform anti-aliasing. We cover the methodology for performing anti-aliasing in Section 3.4 and the full details of the algorithm in Appendix F. Here we present the API endpoint for the author merge suggester. The endpoint is available at <https://drifter.si.usi.ch/api/authors/suggest-merge>. It takes a JSON request as seen in Listing 4.1, which is composed of a list of authors with their name, email, and a unique id. The JSON response can be seen in Listing 4.2. The response is composed of a list of author merge suggestions.

```

{
  "authors": [
    {
      "id": int,
      "name": str,
      "email": str
    },
  ]
}
  
```

LISTING 4.1: JSON Request

```

{
  "suggestions": [
    {
      "author": {
        "name": str,
        "email": str
      },
      "authors_to_merge": [
        {
          "id": int,
          "name": str,
          "email": str
        },
      ],
    },
  ]
}
  
```

LISTING 4.2: JSON Response

FIGURE 4.10: Author merge suggester HTTP API

4.5 Databases

The databases package contains the code used to interact with the PostgresDB and MongoDB databases. As we can see from the class diagram in Figure 4.11 below, the way we interact with the databases is different. In the MongoDB database, we have many functions to query the database. In the majority of the functions, the queries are simple as the collections in MongoDB are directly mapped to python classes in the code. Each of the classes in MongoDB are dataclasses, and we use the mashumaro dataclass serialization library to serialize and deserialize the dataclasses into MongoDB documents.⁹ The schema of the data saved in MongoDB are all the classes with snapshots that we have seen throughout this chapter, as seen in Figures 4.4, 4.6, and 4.8.



FIGURE 4.11: Dbs class diagram

Instead for the Postgres database we use the SQLAlchemy ORM.¹⁰ The schema for this data can be found in Figure 3.5, and is directly mapped to python classes. Access to data in the database is done by directly accessing python objects within a SQLAlchemy session.

We chose to have two databases because the way we interact with the data is different. In the case of the data in MongoDB, we are generally fetching and using the entire data we have for a given commit to be visualized by Drifter: the parsed Java data, the parsed UML data, the project metrics, etc. Instead, for the data in PostgresDB, we are generally fetching a small subset of the information about a repository at a time to answer the various research questions we pose. MongoDB and PostgresDB lend themselves well to these two disparate use cases.

4.6 CLI

We developed a suite of CLI tools to help us build, explore, and summarize the dataset, and generate the graphs that are used throughout this thesis. They can be broken up into 3 main groups: cloning and summarizing repositories, author analysis, and graph generation.

⁹Mashumaro serialization library: <https://github.com/Fatality/mashumaro>

¹⁰SQLAlchemy ORM: <https://www.sqlalchemy.org/>

4.6.1 Cloning and Summarizing Repositories

The first set of CLI tools revolves around cloning and summarizing repositories. The whole process is shown in Listing 4.3. The first step, `download_trees`, takes the `results.json`, which contains the list of repositories we want to analyze (generated by SEART), and clones them to the `repos` directory. In addition, data about the file structure of each repository is stored in pickles in the `trees` directory. After `download_trees` is run, we run `add_releases`, which adds release information from GitHub to the PostgreSQL database. Finally, we run `tree_stats`, which stores information about the file structure and extensions of each repository in the PostgreSQL. At the end of this process, the tables in the File Summarization and SEART GHS boxes in the ER diagram in Figure 3.5 are populated.

```
python -m src.cli.download_trees \
  -r ~/workspace/drifter_data/repos \
  -i ~/workspace/drifter_data/results.json \
  -o ~/workspace/drifter_data/trees/
python -m src.cli.add_releases -r ~/workspace/drifter_data/repos
python -m src.cli.tree_stats -i ~/workspace/drifter_data/trees
```

LISTING 4.3: CLI tool for cloning and summarizing repositories

4.6.2 Author Analysis

The author analysis phase is split into 2 CLI tools. The first part, `analyze_authors` takes the list of UML files tagged (as described in Section 3.3) and a list of repositories with UML diagrams. It performs author anti-aliasing and gathers information about the authors and the commits they make, categorizing commit information by type of commit (UML, other) based on the UML tagged list. Next, `summarize_author_data` summarizes the author data from the database into a CSV file that is used by the diagram generator.

```
python -m src.cli.analyze_authors \
  -rl repo_list.txt \
  -tl ~/workspace/drifter_data/tagged_files.csv \
  -o ~/workspace/drifter_data/analysis/author_analyze
python -m src.cli.summarize_author_data \
  -rl ~/drifter_data/analysis/repo_list.txt \
  -o ~/drifter_data/analysis/author_summary/ \
  -c ~/drifter_data/analysis/author_summary/authors.csv
```

LISTING 4.4: Author analyzer

4.6.3 Diagram Generation

The graphs, plots, and many tables presented throughout this thesis are created by the diagram generator. It uses the information in the PostgreSQL along with the CSV generated by the author summarization step. In addition to the authors CSV, it takes a CSV with the number of repositories by year. This CSV is generated using the `get_commit_dates` script.

```
./get_commit_dates.sh

python -m src.cli.gen_diagrams \
  -i ~/Downloads/author_summarizer_output/authors.csv \
  -o ~/drifter_data/thesis_diagrams \
  -rc ~/drifter_data/analysis/repo_count_by_year.csv
```

LISTING 4.5: Diagram generator

4.7 Drifter

Drifter is a tool we developed to explore a repositories' usage of UML over time. We use it in Chapter 6 where we share case studies of the usage of UML in GitHub repositories. Drifter can be accessed at <https://drifter.si.usi.ch>. The frontend is composed of a set of visualizations that help us explore a given repository. In this section we present the visualizations that are used in the case studies and how to interpret them.

4.7.1 Package Visualization

The first visualization we present is the Java package visualization. As a reminder, coverage is the percentage of Java references that have a corresponding UML diagram. We would like a way, given a commit for a project, to be able to see how much coverage different Java packages have by UML diagrams, and if certain packages have more coverage than others. Figure 4.12a shows an annotated version of the package visualization. Each of the innermost circles represents a Java reference. Green references are covered and white references are not covered. Each circle which contains another circle represents a package.

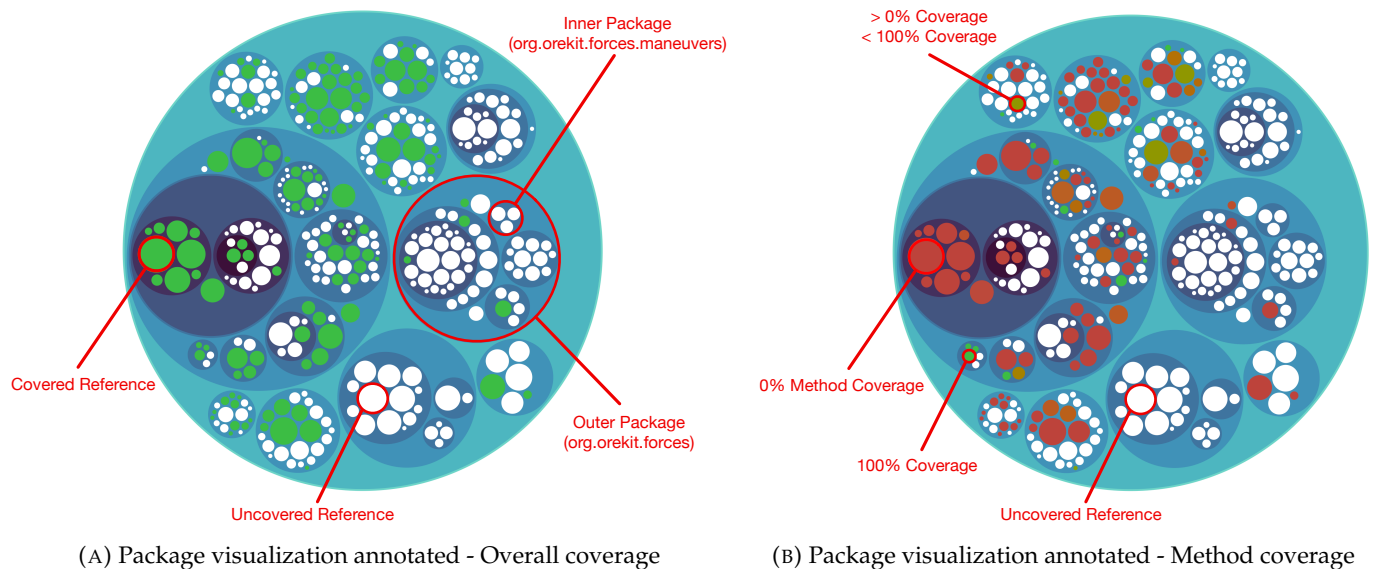


FIGURE 4.12: Annotated package visualization

In addition to seeing the overall coverage, we want to be able to see how detailed that coverage is. As a reminder method coverage is the percentage of methods in a Java reference that are covered by UML diagrams. Figure 4.12b shows an annotated version of the package visualization with method coverage (aggregated by the average, min, or max percentage per diagram). White references still represent uncovered references. Red circles represent references which have no methods covered by diagrams. Green circles represent references which have all of their methods covered by diagrams. The shades in between represent coverages between 0% and 100%. The same rules apply when visualizing attribute coverage.

4.7.2 Java to UML graph

The second visualization we present is the Java to UML graph. We want to visualize the relationship between Java references and UML diagrams. We want to see how many diagrams a Java reference is referenced by, and how many Java references a diagram references. We would also like to see which diagrams are not referencing any Java references. To accomplish this, we used the graph shown in Figure 4.13. Blue entities in the graph represent Java references, with circles representing classes, diamonds abstract classes,

triangles interfaces, and the y-shape enums. The edges between nodes show which UML diagrams Java references are found in.

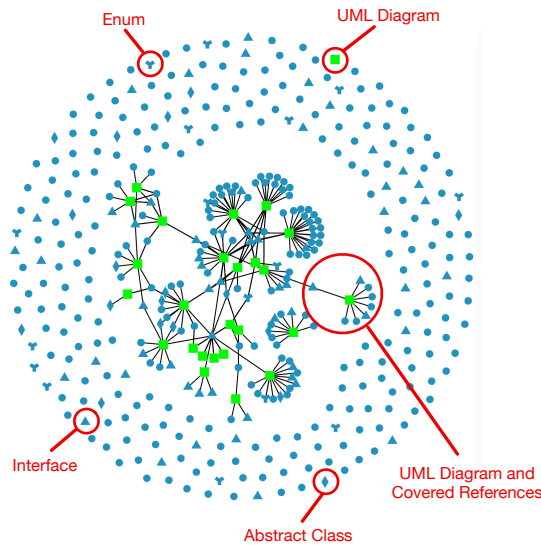


FIGURE 4.13: Java to UML graph

4.7.3 Coverage History

The Java package and Java to UML graph visualizations are useful for exploring a given commit of a repository, but we also would like a way to visualize the evolution of the repository. To do this, we developed the coverage history visualization seen in Figure 4.14, which shows coverage percentage over time.

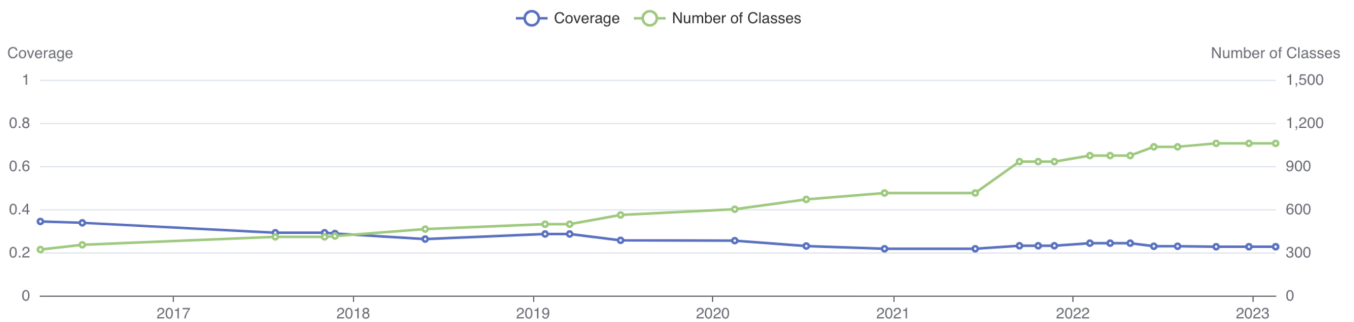


FIGURE 4.14: Coverage history graph – Release view

The above graph shows the coverage history in terms of releases. We choose releases as the unit of time because releases give us an easy sampling point where the project should be in a good, stable, buildable, and parsable state. This does potentially lose some interesting information though as not all projects have releases from the beginning of their history, and others might not have them at all. For this reason, we also offer a view of the coverage history in terms of commits. Parsing every single commit and attempting to order the commit history is a difficult task due to the inherent non-linearity of Git commit histories. There is also the problem that intermediate commits may not always leave the project in a good state. To give as linear a history as possible, while also choosing commits where the project is in a good state, we used the `git log -topo-order -first-parent` command to build the history. This puts commits in the order in which they were made, rather than in date order, and also only follows the first parent of a merge commit. This is a sample of the total commits, but still gives a more expansive view of the history versus the release view.

4.7.4 File History

Another aspect we wanted to be able to visualize is how a Java reference's coverage changes over time. To do this we created the visualization seen in Figure 4.15. The figure shows an annotated and zoomed in view of the file history for a Java file called *Instructor.java*. On the left at the top we see the Java source file for which we want to view the coverage history of. Below the Java file are a series of UML diagrams that have references to the Java file at some point. The line coming from the Java file shows whether or not the Java file is covered by the diagram at that point in time. Green means the Java file is covered, and red means it is not covered. The line coming from UML files shows whether or not that UML file provides coverage for the Java file. The symbols show where the file has been added (squares), modified (circles), or removed (diamonds).

In this history we see a Java file which has no coverage until the *StorageClassDiagram.puml* is added. At some point this file is deleted and re-added (we restart the history of a file when it has been deleted). Even later a file called *Actors.puml* is added which also provides coverage for the Java file.

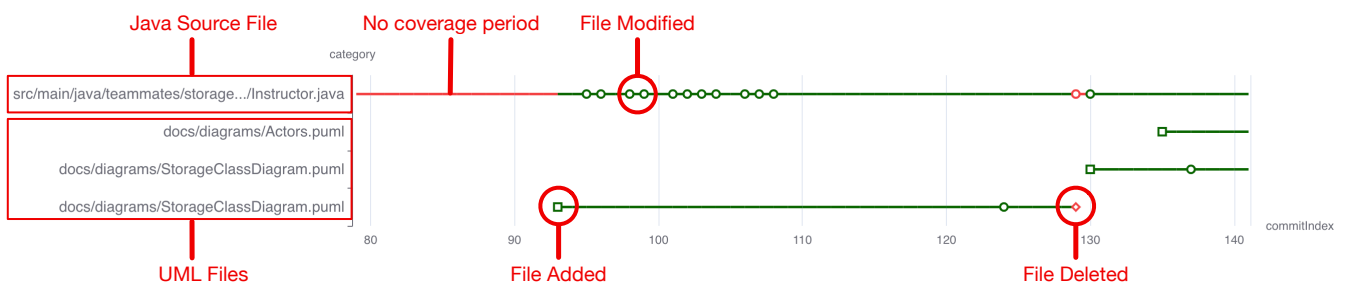


FIGURE 4.15: File history for *Instructor.java*

4.8 Summary

In this chapter we looked in-depth at the architecture of the tools we developed. We looked at the Java and UML parsers, and how we make connections between Java and UML references, and calculate coverage metrics. We also looked at the databases we used to store data and the technologies used to interact with them. We explored the CLI tools built to explore the dataset and generate the diagrams for this thesis. We concluded with a tutorial on Drifter, and how to interpret the visualizations it provides. In the next chapter, we present our analysis of the research questions posed at the beginning of this thesis.

Chapter 5

Research Questions

“Ask the right questions if you’re to find the right answers.”

— Vanessa Redgrave

Given the importance of documentation and the benefits of UML diagrams, we want to understand how UML is being used in open source repositories. Before investigating some repositories in-depth in the case studies section, we want to get a general idea of how UML is being used. In this chapter we explore the repositories in our dataset to answer the following research questions:

RQ1. How Widespread is the use of UML in Open Source Projects?

We look at the number and percentage of repositories that have UML diagrams over time. We also analyze the formats that UML diagrams are stored in, and how the popularity of those formats changes over time.

RQ2. What Formats are UML Diagrams Found in?

We analyze the formats that UML diagrams are found in, and share the methodology we used to find them. We provide a list of each format we found a UML diagram in, and also share which of those formats we found were used solely for UML diagrams.

RQ3. Who is Creating and Maintaining UML Design Diagrams?

We examine the characteristics of authors who create and maintain UML diagrams, and how they compare with those who do not. Analyzing the number of authors, the number of commits authors make, and the number of days they have been on a project, we gain insights into the makeup of authors who create and maintain UML diagrams. We also investigate whether those who create and maintain UML diagrams are typically dedicated to that task, or whether they also usually contribute to source code.

RQ4. What Types of Projects are UML Diagrams Found in?

In the last research question, we look at the activity, community size, and main language of repositories that have UML diagrams versus those that do not.

RQ0. Why is UML Underutilized in Open Source Projects?

Using the results gathered from the previous research questions, we look to provide insights into why the use of UML is not more widespread in open source repositories.

5.1 RQ1: How Widespread is the use of UML in Open Source Projects?

UML is often touted as the de facto standard for modeling software systems. It gives developers a common language to communicate how a system is structured and what it does. It can greatly increase the usability of a system for users, and the maintainability of a system by developers. Given the benefits, we wanted to see how popular the usage of UML is in open source software. In this section we look at the evolution of usage of UML along with the evolution of UML extension usage.

5.1.1 Definitions

Table 5.1 presents a few helpful definitions that will be used throughout this section.

Term	Description
Active Repository	A repository is considered active if it has at least 1 commit in a year (considering the main branch of the repository).

TABLE 5.1: Definitions used in RQ1

5.1.2 Evolution of UML use

The usage of UML in our dataset is not widespread. Out of over 13,000 repositories, we found only 550 repositories (4.1% of our dataset) that contained at least 1 UML diagram at some point in their lifetime. Still, we find it interesting to investigate whether the usage of UML diagrams is increasing or decreasing over time. Figure 5.1 shows the number of repositories with UML diagrams and the number of repositories actively making UML changes over time. Looking at Figure 5.1, we can see that the both the number of repositories with UML diagrams and the number of repositories actively making UML changes has been on an upward trend since 2000.

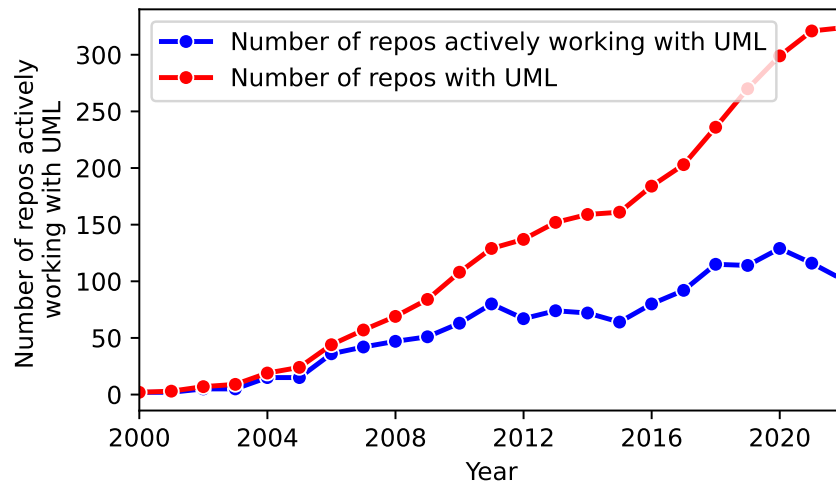


FIGURE 5.1: Number of repositories with UML

However, if we keep in mind that in our dataset the number of active repositories changes over time (*i.e.*, not all repositories were created and active in 2000), we see a different story. In Figure 5.2, the blue line representing the percentage of repositories making UML changes climbs to a peak in 2006 then consistently decreases until 2015, where after climbing back from a local minimum, finally reaches a steady state,

hovering around 1.0% of repositories. If we look to the overall percentage of repositories with UML (red line), we see a similar trend, but we can see that the percentage of repositories with UML climbs starting in 2017. The overall use hits a peak around 2006 and 2007, but in recent years is on the rise again. Even at the peak of use, only 3.5% of repositories have UML and only 2.5% are actively updating UML diagrams.

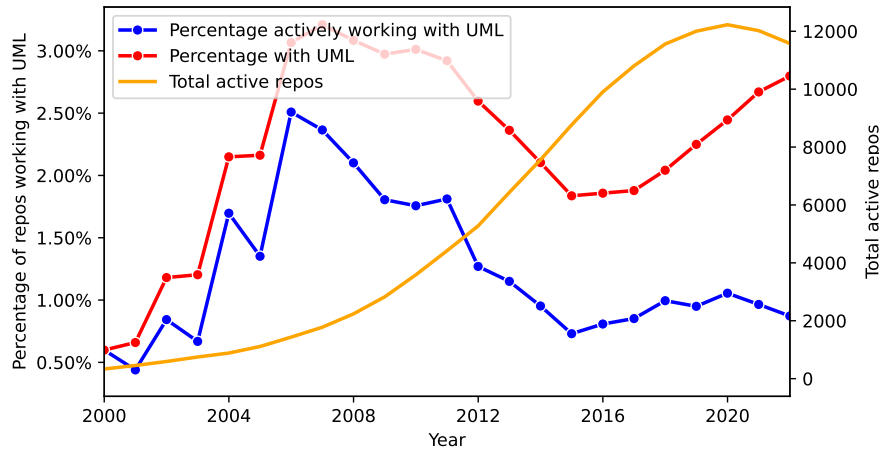


FIGURE 5.2: Percentage of repositories with UML

5.1.3 Popularity by Extension

Figure 5.3 shows the evolution of UML extension usage by repositories. Each year, we plot the number of repositories actively making changes to each UML extension. The red bars represent the top 5 extensions found in the most repositories for that year (some years have less than 5 because there is a tie for 5th place or there are less than 5 extensions). We see some interesting trends. `.xmi`, `.zargo`, `.argo`, and `.pgml` are the only UML extensions used from 2000 to 2003. The `.zargo`, `.argo`, and `.pgml` extensions are all extensions used by ArgoUML (a graphical UML modeling tool), and `.xmi` is a standard used by many graphical UML modeling tools. 2004 to 2007 is still dominated by `.xmi` and `.zargo`, but `.dia` and `.ecore` start to make an appearance. The `.dia` extension is used by Dia, a general purpose diagramming tool and `.ecore` is used by the Eclipse Modeling Framework (EMF), a plugin for Eclipse that gives the ability to model graphically and then generate code from those models. From 2008 to 2015, we see `.xmi`, `.dia`, `.ecore` remain popular and `.ucls`, `.uxf`, and `.uml` start to become popular. The `.uml` extension contains many different types of UML files, including Ecore/EMF files, XMI files and PlantUML files. The `.ucls` the extension is used by the ObjectAid UML Explorer plugin for Eclipse. The `.uxf` extension is the UML eXchange Format (a standard similar to XMI for exchanging diagrams but specifically for UML), but we see it is used only by UMLet, a stand-alone UML modeling tool that also comes with a plugin for Eclipse. From 2016 to 2022 we start to see the previously popular extensions be supplanted by `.mmd`, `.plantuml`, `.puml`, and in the last year PlantUML and mermaid are the most used. The `.mmd` is extension is used by Mermaid and `.plantuml` and `.puml` are the extensions used by PlantUML. PlantUML and Mermaid are two popular text based UML modeling tools.

Given the above, we can see three main time periods. Early on, from 2000 to 2007 we see the usage of graphical UML modeling tools and generic diagramming tools as the most popular. From 2008 to 2016, Eclipse plugins take over as the most popular UML modeling tools. Finally, from 2016 to 2022, we see the rise of text based UML modeling tools as the most popular.

It is interesting to note that although PlantUML did not see widespread use until 2016, it had its first release well before that, in 2009. Mermaid was also released in 2014 before it started to see use in 2016 and before it made it into the top 5 extensions in 2020. So what changed in 2016 that caused these tools to start

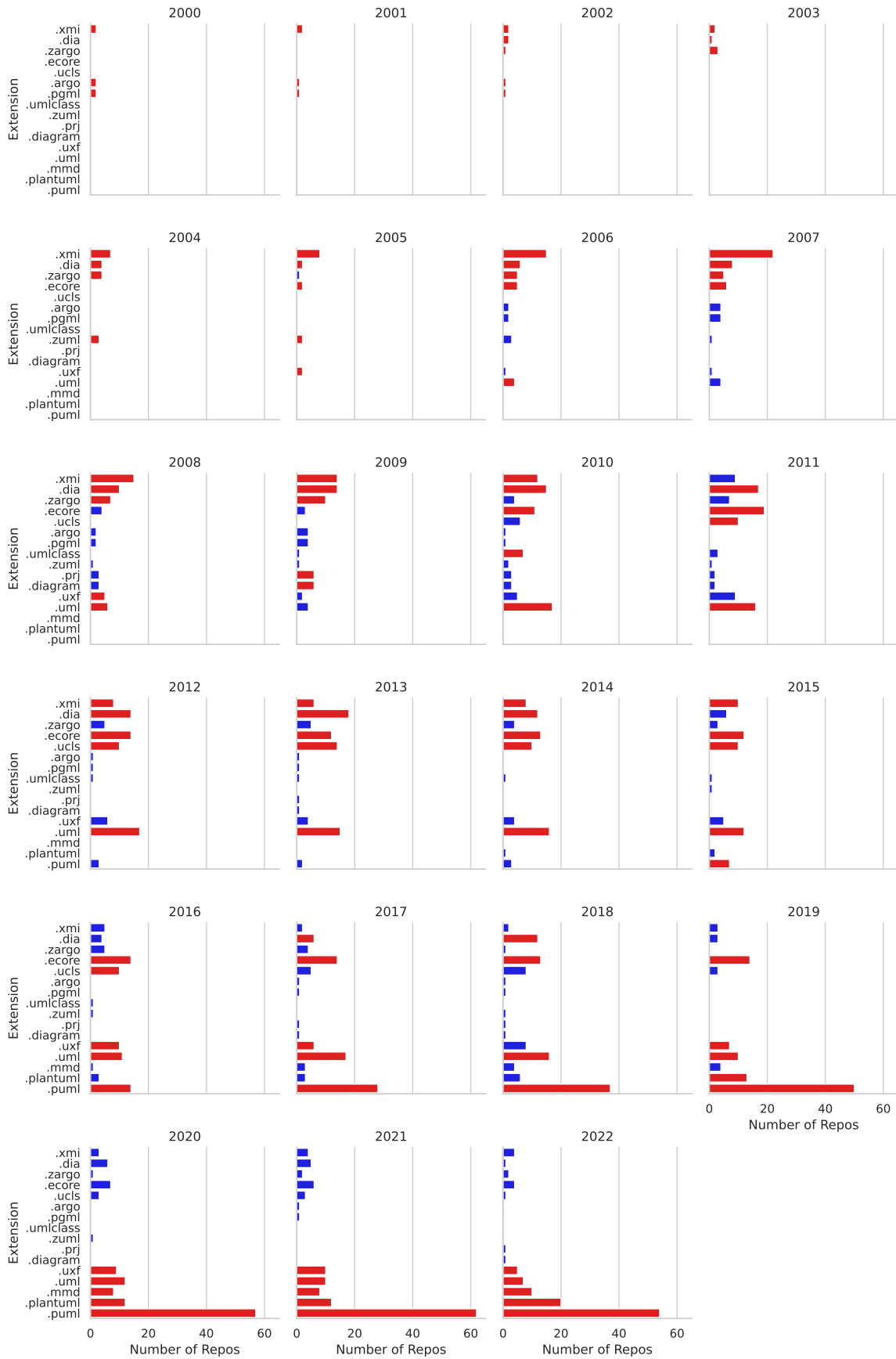


FIGURE 5.3: Number of repositories with UML extensions by year (red are the top 5 extensions for that year – ties not marked)

seeing widespread use? One possible explanation is the release of Visual Studio Code (VSCode) in 2015. It is clear, given how popular Eclipse plugins are from 2008 to 2016, that developers like their UML modeling tools to be integrated into their IDE. Since its release in 2016, VSCode has slowly become the most popular IDE for developers.¹ VSCode has extensions for visualizing UML diagrams for both PlantUML, and Mermaid. UMLet, which also remains popular from 2016 to 2022 also has a VSCode extension. As developers started migrating from Eclipse to VSCode, projects would have to migrate to new UML modeling tools that do not rely on Eclipse.

It is also interesting that from 2016 to 2022, we see a huge uptick in the overall number of repositories actively working with UML, mainly due to the rise of PlantUML. This could be due to advantages that text-based UML modeling tools offer. Compared to graphical UML modeling tools, it is arguably easier to version control text-based UML diagrams. Text-based UML diagrams have meaningful diffs that can be checked for correctness during code reviews. Although .xmi and .ecore are human-readable, if we look at Listings 5.1 and 5.2, we can see that models represented in more recent text-based formats (e.g., PlantUML) are simpler to read and would be easier to compare in a code review.

```
<ecore:EPackage xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    name="camel"
  nsURI="http://camel.apache.org/routing/1.0" nsPrefix="camel">
<eClassifiers xsi:type="ecore:EClass" name="Routes">
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="routes" upperBound="-1" eType="#//Route"
    containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Route">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="from"
  eType="ecore:EDatatype
    http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
    name="processors" upperBound="-1"
    eType="#//Processor" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Processor"/>
<eClassifiers xsi:type="ecore:EClass" name="Send">
  eSuperTypes="#//Processor">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="uri"
    eType="ecore:EDatatype
    http://www.eclipse.org/emf/2002/Ecore#//EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Process"
  eSuperTypes="#//Processor">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="type"
    eType="ecore:EDatatype
    http://www.eclipse.org/emf/2002/Ecore#//EString"/>
</eClassifiers>
</ecore:EPackage>
```

LISTING 5.1: Example ecore package and class

```
package camel {
  class Routes {
    List<Route> routes
  }

  class Route {
    String from
    List<Processor> processors
  }

  class Processor { }

  class Send {
    String uri
  }

  class Process {
    String type
  }

  Send --|> Processor
  Process --|> Processor
}

LISTING 5.2:
Example puml
package and class
```

Another advantage of text-based UML modeling tools is their versatility. Both Mermaid and PlantUML can be used within an IDE through plugins or as a stand-alone tool. They both also offer online editors so new users can contribute without having to install anything. As their main mode for generating images of

¹Stack overflow survey (Integrated development environment): <https://survey.stackoverflow.co/2022>

UML diagrams is through a command line interface, their generation can be easily automated. This last fact is seen in Section 5.3 where we see `iluwatar/java-design-patterns` and `kubernetes-sigs/cluster-api` generate UML diagrams from PlantUML text files and embed the generated images in Markdown files. We see it again in all three case studies in Chapter 6.

Although anecdotal, we found Mermaid and PlantUML much lighter weight and easier to use than the various graphical tools and Eclipse plugins. We even use PlantUML and Mermaid to generate the UML diagrams used throughout this thesis. In the first case study with `cs-si/orekit` we found an example of a project migrating from graphical editors, to an eclipse-based modeler, and finally landing on PlantUML in the end for similar reasons (see Section 6.2 `cs-si/orekit`).

RQ1: How Widespread is the use of UML in Open Source Projects?

Finding 1: The use of UML in open source software is not widespread, with a peak of only 3.5% of repositories having a UML diagram.

Finding 2: UML hit peak usage in 2007, but has been seeing a steady resurgence since 2016.

Finding 3: PlantUML has been the most popular UML diagramming tool since 2016, the same time as the resurgence of UML started.

5.1.4 Conclusions

The use of UML in open source software is not widespread. Even at the peak usage of UML, we see only 3.5% of repositories have a UML diagram, and 2.5% actively work with a UML diagram. The difference between the peak number of repositories with UML diagrams, and those actively working with UML diagrams could mean a few things. First, UML diagrams can be useful even if they are not actively being modified, especially if they document parts of the system that do not change often. There is a chance that those who are not actively maintaining UML diagrams are still benefitting from diagrams that are already created. It could also mean that there are diagrams which are not being actively maintained, and are out-of-date and inaccurate.

Although the peak usage of UML was seen back in 2006 and 2007, we are seeing a steady increase in the number of repositories with UML diagrams since 2016. We also see that this resurgence in UML usage coincides with the rise of text-based modeling tools such as PlantUML and Mermaid.

5.2 RQ2: What Formats are UML Diagrams Found in?

What formats are UML diagrams found in? Unfortunately, it is almost impossible to exhaustively answer this seemingly simple question. If we think about image files, we can see why this question is difficult to answer. Although we are all familiar with .jpg, .png, .svg, .gif, .bmp, etc., there are many other image formats, many being proprietary and tool specific such as .psd (Adobe Photoshop's format). It would be difficult to enumerate every single image format, and even if we did, it would change as a new tool comes out with its own image format. The same is true for UML diagrams. Although there is a UML standard, the standard specifies a modeling language and not how that language should be realized.²

The UML standard does not say that UML diagrams must be realized through text or image files in a certain format. The lack of a standard means that UML diagrams can be found in a potentially limitless number of formats, some potentially proprietary, and many of which may not be specifically for UML diagrams (e.g., .png, .xml, .drawio). There is a standard which can be used for exchanging UML models between tools called XMI.³ However, XMI is not the most popular format for UML diagrams in GitHub repositories. We can also see from Table A.1 which shows the most popular UML tools and their associated extensions, XMI is supported only by a subset of them. Even popular tools like PlantUML list only beta support for XMI export, and no options for XMI import.⁴

Although we cannot answer RQ2 in an exhaustive way, we can find a lower bound on the total possible UML formats. In this section we present the formats we found in our dataset (see Section 3.2 for the approach).

5.2.1 Finding UML Diagrams in Candidate Extensions

Starting from the candidate extension list in Table 3.4, we set out to answer the following questions for each extension in the list:

1. Are there examples of the extension being used as a UML diagram in the dataset?
2. Is the extension specific to UML diagrams?

To answer these questions, we attempted to find an example and a counter-example for each file extension. If we can find an example, then it is a valid UML extension, and if we find a counter-example it is not UML specific. All examples and counter-examples can be found in Appendix B and Appendix C respectively. We used a 4-step approach to find those examples and counter-examples as shown below.

Step 1: Find Example or Counter-Example for each Extension

To start our search, we look at the candidate extensions and get one file for each extension. To obtain these files, we query the PostgresDB for the repository, commit hash, and file path of a file given an extension.⁵ We then retrieve the file from GitHub and inspect it to see if it is a UML diagram (note we can grab the file from our dataset if the GitHub repository has since been deleted). This process leaves us with at least one example or counter-example for each extension.

Step 2: Search Files in /uml/ Paths

After finding examples and counter-examples in the first step, we are left with the more challenging cases. There are over 12,000 repositories which have at least one .txt file, and likely only a small percentage of

²<https://www.omg.org/spec/UML/2.5.1/About-UML>

³<https://www.omg.org/spec/XMI/2.5.1/About-XMI/>

⁴<https://plantuml.com/xmi>

⁵See queries [Find all repositories where extension exists](#) and [Find all commits and files where extension exists](#) in Appendix E

these repositories contain UML diagrams. In the first step we found a counter-example for .txt. Now we must search all of the repositories with .txt to see if we can find an example of a UML file. This is true for all of the extensions, although not all are as ubiquitous as .txt. To avoid needing to search through every repository in the dataset for examples and counter-examples, we use a variety of techniques. The first of these techniques is given an extension, we find the repository, commit hash, and file path of all files with that extension that are in path that contain the string /uml/.⁶ As in step one, we retrieve the file and inspect it to see if it is a UML diagram or not.

Step 3: Search File Names for Keywords

The next technique we use to find examples for extensions is to search for keywords in the file names. We gather files for each extension we still need to find examples for using the file gathering tool. After gathering files for each extension, we search for the following keywords (case insensitive) in the file names: architecture, uml, diagram, sequence, class, usecase, state, activity, component, deployment, object, communication, composite, interaction, timing, collaboration, package, profile, and timing, to find files that are likely UML candidates. This keyword list is comprised of the names of the 14 UML diagram types. We then inspect these files to see if they are UML diagrams or not.

Step 4: Manual Search

The above techniques found examples and counter-examples for most extensions. For the extensions that it was insufficient for, we have one last technique for finding examples, and another one for finding counter-examples. For counter-examples, we used the UML extension tagging discussed in Section 3.3. If every file was tagged as UML by our process, then we treat it as a UML specific extension and found no counter-examples. For those extensions that were not tagged as UML, any file that was not tagged as UML is a valid counter-example. For finding examples, we manually searched the dataset. For the extensions that were too numerous to exhaustively search, we take a statistically significant subsample (confidence level = 90%, margin of error = 5%) of repositories and search all files with the given extension in that subset for UML examples. The extensions that we had to manually search can be seen in Table 5.2.






Extension	Repo Count	Sampled	Extension	Repo Count	Sampled
.csv	3,127	249	.vacuumlo	12	12
.swf	1,030	215	.umlaut	10	10
.rtf	965	212	.edx	3	3
.xlsx	625	189	.mdr	4	4
.eps 	610	188	.gxml	2	2
.tiff	551	183	.eddx	1	1
.docx 	521	179	.vssm	1	1
.ps 	460	171	.vssx	1	1
.vsd 	174	106	.vstm	1	1
.ppt 	167	104	.vstx	1	1
.pbm	109	109			

TABLE 5.2: Manually searched extensions ( UML example found)

⁶See query [Find all commits and files in uml paths for given extension](#) in Appendix E

5.2.2 Results

Table 5.3 shows the extensions we found can contain UML diagrams. Those that are UML specific (every file with that extension was tagged), are marked with a ✓. It is interesting to see that the .uml, .puml, and .plantuml extensions are not UML specific. Although PlantUML is a popular tool for creating UML diagrams, it can also be used to create other types of diagrams (*e.g.*, gantt charts, c4 diagrams). We also find the .uml extension is used for User Mode Linux (UML) kernel configuration files.

Extension	Repo Count	UML Specific	Extension	Repo Count	UML Specific
.md	13,238		.plantuml	48	
.txt	12,442		.emf	45	
.png	11,916		.ecore	42	✓
.html	10,400		.mmd	38	
.xml	8,958		.session	36	
.svg	8,446		.ucls	33	✓
.jpg	7,268		.vpp	32	
.gif	6,432		.zargo	32	✓
.pdf	3,852		.uxf	31	✓
.rst	3,541		.diagram	30	
.bmp	1,759		.pu	20	
.jpeg	1,598		.mdj	18	✓
.xpm	640		.eap	11	
.docx	521		.argo	9	
.ps	460		.umlclass	9	
.dia	417		.vdx	9	
.pptx	378		.pgml	8	✓
.graffle	373		.zuml	7	✓
.prj	206		.asta	6	✓
.drawio	191		.iuml	5	✓
.vsd	174		.mdzip	5	✓
.ppt	167		.vsdm	5	
.puml	165		.yuml	5	✓
.uml	111		.unt	4	
.xmi	110		.platuml	2	✓
.vsdx	80		.umlprofile	2	✓
.wmf	71		.ump	2	✓
.gliffy	59		.cmof	1	✓

TABLE 5.3: Extensions with UML examples (Repo count is number of repositories extension is found in, not how many it is a UML extension in)

RQ2: What Formats are UML Diagrams Found in?

Finding 1: UML diagrams can be found in numerous formats. We found examples of diagrams in 56 different extensions.

Finding 2: A majority of extensions UML diagrams are found in are not used specifically for UML.

5.2.3 Conclusions

Table 5.3 shows the formats we found UML diagrams in. This outlines one of the challenges of analyzing the usage of UML diagrams. UML diagrams come in many formats, and are created using numerous tools. Although there is a standard format for exchanging UML models between tools (XMI), we can see from the table that it is not the most popularly used format.

5.3 RQ3: Who is Creating and Maintaining UML Design Diagrams?

It is important to understand the attributes of contributors who create and maintain UML diagrams to get a sense of how UML is used in open source projects. We would like to see if there are differences between contributors who create and maintain UML diagrams and those who do not. We look at the various attributes of contributors to see if there are any common attributes for contributors who create and maintain UML diagrams. We also look to see if there are any contributors who manage UML diagrams but do not contribute to source code.

5.3.1 Definitions

Here we present a few useful definitions that will be helpful throughout this section.

Term	Description
Contribution Period	The period of time between the first commit and the last commit of a contributor.
Non-UML Committer	Contributors to repositories who have never created or modified a UML diagram.
UML Committer	A UML committer is a contributor who has created or modified at least one UML diagram in a repository. They may or may not have also contributed to other file types.
Dedicated Diagrammer	A UML committer who has never contributed to source code (but may have contributed to other file types).

TABLE 5.4: Definitions used in RQ3

5.3.2 Methodology

Discriminating between UML diagrams and other files exclusively based on file extension is difficult (see Section 5.2). To distinguish between UML committers and non-UML committers, we use the tagged UML diagrams from Section 3.3. We consider UML diagrams in formats that we tagged (*e.g.*, .argo, .uml, .plantuml, .xmi). As the tagged UML diagrams are a subset of the total UML diagrams in our dataset, the numbers reported here are a lower bound for the total number of UML committers in our dataset. Given the tagged UML diagrams and the history of contributors and their commits to the repositories in our dataset, we analyze attributes of the commit authors to assess who is and who is not managing UML diagrams.

5.3.3 Contribution Period of UML Committers vs non-UML Committers

The first contributor attribute we look at is the average contribution period. Figure 5.4 shows three box plots with the distribution of average contribution period per repository of UML committers and non-UML committers. The top is the distribution with no filtering, the middle is the distribution with filtering out contributors with less than 2 commits, and the bottom is the distribution with filtering out contributors with less than 10 commits. We can see between the three, the UML committer box plots are relatively unaffected by filtering compared to the non-UML committer box plots. The median contribution period of the non-UML committer box plots moves from 0 days with no filtering to 801 days when filtering out contributors with less than 10 commits. This huge move is due to the large number of contributors who have only committed a few times. In the remaining analysis in this section, we show the results after filtering out contributors with less than 10 commits.

When we consider only contributors with at least 10 commits, the median contribution period of UML committers is 1,735 days versus 801 days for non-UML committers. We confirmed that the difference between contribution periods is statistically significant using the Mann-Whitney U test. The test returned a p-value of 4.52e-58, signifying a statistically significant difference between the two groups. We also

calculated the effect size using Cliff's delta, and got a value of 0.56, signifying the difference in contribution period between UML committers and non-UML committers is large.

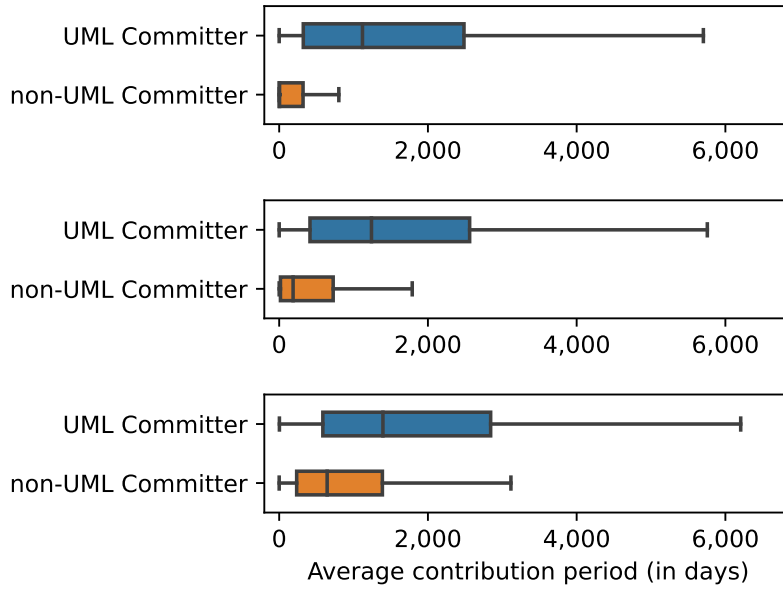


FIGURE 5.4: [

Average contribution periods of UML committers vs non-UML committers]Average contribution periods of UML committers vs non-UML committers (top: all contributors, middle: contributors with at least 2 commits, bottom: contributors with at least 10 commits)

In Figure 5.5, we see a scatter plot of the same average contribution as shown in the previous box plots. The y-axis is the average contribution period of UML committers and the x-axis is the average contribution period of non-UML committers. Each point represents a git repository, and any point above the red line is a repository where the average contribution period of the authors of UML commits is greater than the average contribution period of the other authors.

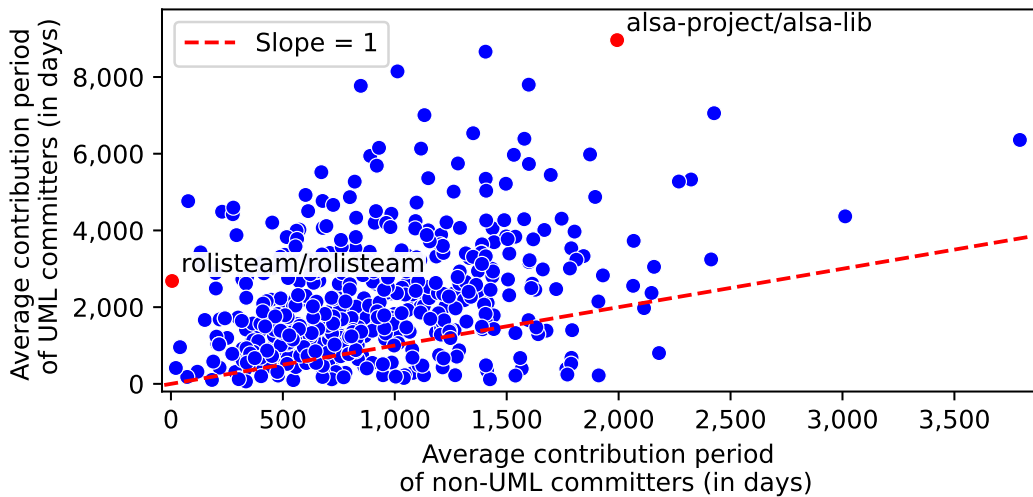



FIGURE 5.5: Average contribution periods of UML committers versus non-UML committers

As we can see, in a majority of cases, the average contribution period of UML committers is greater than the average contribution period of other authors. This suggests that the average UML committer is around longer in repositories than the average non-UML committer.

We marked two outlier repositories in red in the scatter plot. The first we look at is the repository with the lowest average contribution period of non-UML committers while still having a high average contribution period of UML committers, `rolisteam/rolisteam`. The authors for `rolisteam/rolisteam` can be seen in Table 5.5, with the UML committers marked with the  icon. We can see the repository has essentially a single maintainer, *Renaud Guezennec*. The majority of the remaining contributors had all of their commits on the same day (author contribution period of 0). In this repository, the main maintainer is the only person who has created or modified any UML diagrams, which explains the high average contribution period of UML committers and the low average contribution period of non-UML committers.



Author Name	Contribution Period	Number of Commits
Renaud Guezennec 	2,682	5,558
Vladar4	56	2
Etienne	8	3
Tyler Schmidt	8	3
Tomaz Canabrava	5	35
Milan Irigoyen	3	4
Paul Brown	0	4
Gissu	0	5
Ben Cooksley	0	4
Yann Escarbassiere	0	3
Patrick José Pereira	0	3
Grégoire Barbier	0	1
IBPX	0	1

TABLE 5.5: Authors for `rolisteam/rolisteam` ( UML Committers)

The next we look at is the repository with the highest average contribution period of UML committers, `alsa-project/alsa-lib`. Table 5.6 shows the top 5 authors by number of commits for this repository.


Author Name	Contribution Period	Number of Commits
Jaroslav Kysela 	8,962	1,967
Takashi Iwai	8,027	925
Clemens Ladisch	4,764	126
Takashi Sakamoto	2,870	151
Abramo Bagnara	1,227	355

TABLE 5.6: Top 5 authors by number of commits for `alsa-project/alsa-lib` ( UML Committers)

The project is similar to `rolisteam/rolisteam` since only 1 contributor has created or modified any UML diagrams, but in this case there are many other contributors who have been active for a significant amount

of time and made a significant number of commits. The box plots and scatter plots show that UML committers tend to be among the longer-standing members of their respective projects in contrast to non-UML committers.

5.3.4 Number of UML Committers versus non-UML Committers

We would like to see whether in most cases where UML is used, a majority of contributors are UML committers, or if only a small subset of contributors are UML committers. To do this we look at the number of UML committers versus non-UML committers in the scatter plot in Figure 5.6. In almost every single case, the number of non-UML committers is higher than the number of UML committers (any repositories under the red line). To confirm there was a statistically significant difference between the number of UML committers and non-UML committers, we used the Mann-Whitney U test. The test returned a p-value of $8.84e-160$, signifying a statistically significant difference between the two groups. We also calculated the effect size using Cliff's delta, and got a value of -0.93 , signifying the difference in number of UML committers and non-UML committers is large. Given that UML committers are almost always vastly outnumbered by non-UML committers, we were interested in exploring a few repositories where the number of UML committers is greater or close to the number of non-UML committers. We highlight `iluwatar/java-design-patterns`, `umple/umple`, and `kubernetes-sigs/cluster-api` for this reason. We also highlight `embox/embox` to explore a typical case.

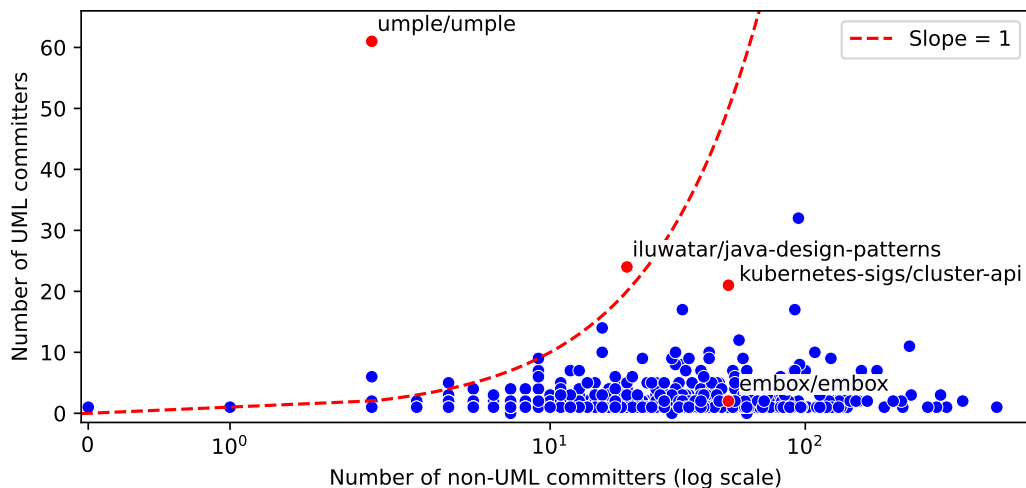


FIGURE 5.6: Number of UML committers versus non-UML committers

If we take a deeper look at each marked repository, we can see why they have a relatively high number of UML committers. The `iluwatar/java-design-patterns` project is an educational repository used to teach Java design patterns. Every design pattern follows the same template: a Java example, a UML diagram, and a readme with a description of the pattern and the UML diagram embedded. Given that every design pattern has an associated UML diagram, it makes sense that any author who wants to add or update a design pattern must also do the same for a corresponding UML diagram. Another interesting project is `umple/umple`. It is the repository for the Umple model-oriented programming language.⁷ Since it is a model-oriented programming language, every source file is essentially a UML diagram.

We highlight another interesting project, `kubernetes-sigs/cluster-api`.⁸ This project houses a set of APIs for managing Kubernetes clusters, and they keep an up-to-date Markdown book which documents

⁷<https://cruise.umple.org/umple/>

⁸<https://github.com/kubernetes-sigs/cluster-api>

the APIs and how they can be used, extended and how new APIs can be implemented.⁹ It also contains proposals for new features and enhancements to the APIs. Both the documentation and proposals contain a wealth of UML diagrams to support the text. If we look at the authors who are UML committers in Table 5.7, we can see that the diagramming work is spread out among many contributors. Fei Yang, for instance, picks up a GitHub issue to create documentation that is listed as a "good first issue", which includes generating diagrams for the documentation.¹⁰ pablochacin put together a proposal for a feature he wanted to see, which involved UML diagrams of the feature.¹¹ We see the diagramming and documentation work in this repository is spread among both the low contribution developers and high contribution developers.

Author Name	Number of Commits	Number of Commits to UML
Vince Prignano	609	3
Stefan Bueringer	484	3
fabriziopandini	440	1
killianmuldoon	256	1
Chuck Ha	129	2
Yuvaraj Kakaraparthi	117	1
Jason DeTiberus	105	1
Oscar Utbult	97	4
Warren Fernandes	85	2
Cecile Robert-Michon	81	2
Naadir Jeewa	70	1
Andy Goldstein	60	3
Joel Speed	49	2
Alberto García Lamela	43	1
Carlos Panato	36	1
Sagar Muchhal	31	1
Daniel Lipovetsky	29	2
Matt Boersma	23	1
Prankul	22	1
Kazuki Suda	18	2
Zhecheng Li	17	2
Juan-Lee Pang	6	1
pablochacin	4	1
Yassine TIJANI	4	1
Anusha Ramineni	3	1
Moshe Immerman	2	1
relyt0925	1	1
Fei Yang	1	1

TABLE 5.7: UML Committers for kubernetes-sigs/cluster-api

⁹Kubernetes Markdown Book - <https://cluster-api.sigs.k8s.io/>

¹⁰<https://github.com/kubernetes-sigs/cluster-api/issues/5475>

¹¹<https://github.com/kubernetes-sigs/cluster-api/issues/833>

We highlight the above repositories because they are in stark contrast with the typical repository in our dataset. Most repositories have a small number of UML committers compared to non-UML committers. We highlight one typical case, `embox/embox`. This repository has 154 total authors, and only 2 of them have actively worked on UML diagrams. If we look at the top 5 authors by number of commits in Table 5.8, we see both UML committers are within the top 5 by number of commits. This is more in line with what we saw when looking at `alsa-project/alsa-lib` and `rolisteam/rolisteam` in the previous analysis of contribution period. This brings us to our next analysis, number of commits by UML committers versus non-UML committers.




Author Name	Number of Commits	Number of Commits to UML
Anton Bondarev	5,581	0
Anton Kozlov 	3,154	28
Alex Kalmuk	2,811	0
Denis Deryugin	2,584	0
Eldar Abusalimov 	2,252	41

TABLE 5.8: Top 5 committers by number of commits for `embox/embox` ( UML Committers)

5.3.5 Number of Commits by UML Committers vs non-UML Committers

We wanted to see whether UML committers make more contributions to projects versus non-UML committers. We hypothesized that UML committers are more likely to be main contributors than those who are not for a host of reasons: They are more familiar with the project and so more capable of making diagrams, they want developers to be able to easily contribute so they provide diagrams that help onboard, they want users to use their project and diagrams can be a helpful tool to convey how it works. In the previous section, we highlighted a case where main contributors were not the main UML diagrammers (`kubernetes-sigs/cluster-api`) and one where they were (`embox/embox`). We highlighted specifically what looked like outliers in the dataset in the previous section, so here we look to see if there is a general trend between number of commits by UML committers versus non-UML committers. In Figure 5.7 we see the distribution of proportion of commits made by UML committers versus non-UML committers by repository.

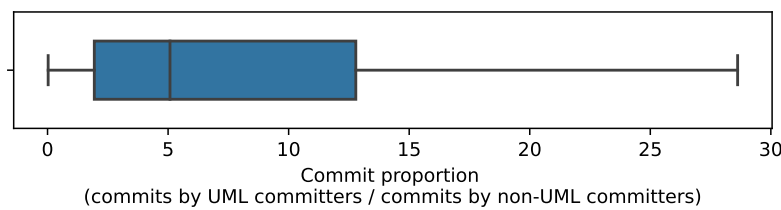


FIGURE 5.7: Proportion of commits by UML committers versus non-UML committers

From this plot, we can see in the median case, UML committers make 5 times more commits than non-UML committers. To confirm there is a statistically significant difference between number of commits made by UML committers vs non-UML committers, we used the Mann-Whitney U test. The test returned a p-value of 1.64e-86, signifying a statistically significant difference between the two groups. We also calculated the effect size using Cliff's delta, and got a value of 0.69, signifying the difference in number of commits between UML committers and non-UML committers is large. Note that we show the boxplot without outliers to focus on the general trend, but we see repositories where the UML committers make as much as

534 times more commits than non-UML committers. The repositories that make up these high proportions are similar to `rolisteam/rolisteam`, where there is 1 contributor who makes up the majority of commits and is a UML-committer.

To get a better idea of how many repositories have UML committers committing more than other committers, we can look at Figure 5.8. The y-axis is the average number of commits by UML committers and the x-axis is the average number of commits by non-UML committers. This scatter plot and the box plot before shows that in most cases, UML committers make more commits than non-UML committers.

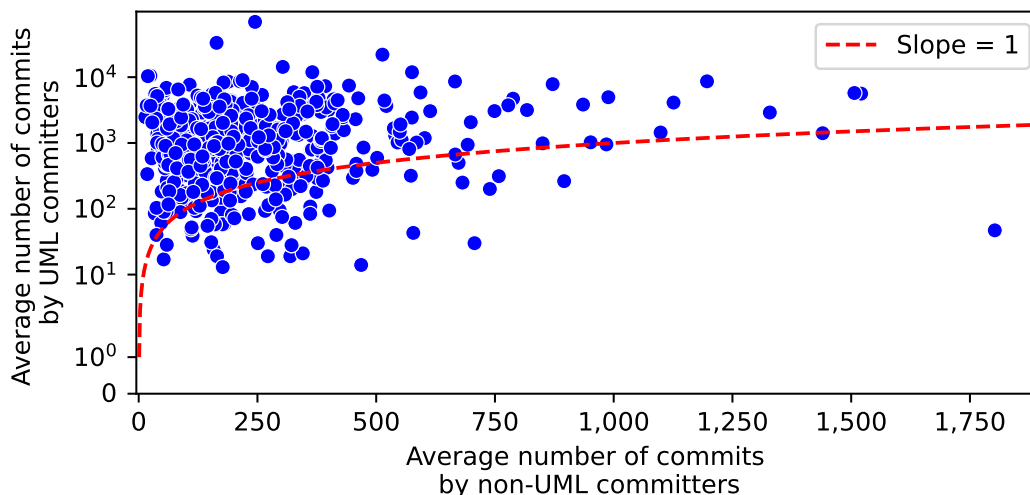


FIGURE 5.8: Average number of commits by UML committers versus non-UML committers

5.3.6 Are There Dedicated UML Diagrammers?

The last question we look to answer in this section is whether there are often dedicated UML diagrammers (those who modify UML diagrams but do not modify source code). We start from a list of UML committers who did not modify any source code files (see Table 5.9). We filter out those with less than 10 commits so that we only see authors who are more than just one-time contributors. The table shows the 14 repositories we found. The short answer is that there are almost never dedicated UML diagrammers (only 2.5% of repositories have authors who modify UML diagrams without modifying source code).

Nevertheless, we took a deeper look at the data. First, we see some interesting repositories. The repositories marked in bold are repositories that contain little to no source code. `w3f/polkadot-spec` is a specification repository, `progit/progit` is a book, `edmcoun/fibo` holds a specification for an ontology, and `kubernetes/enhancements`, `hyperledger/aries-rfcs`, and `openshift/enhancements` are repositories used to discuss enhancements or features for other repositories. The `wix/detox` repository is a source code repository but at closer look to the author who modified only UML diagrams, it was a bot account created solely for publishing documentation, so it is not a real UML committer.

One example we see of someone who commits to UML diagrams and not source code in a repository with source code is *Daria Lavrenova* in the `adorsys/xs2a` repository. Her entire commit history is related to documentation and roadmap planning. Her GitHub profile shows she is a program manager (PM), and based on her commit history, looks like she is doing program management work for `adorsys/xs2a`. *Olga Levandovska* of `adorsys/xs2a` also has a similar commit history to *Daria Lavrenova*, and also has PM listed in her GitHub profile, so we can assume she is also doing program management work for `adorsys/xs2a`. *Dora Nziali* of `adorsys/open-banking-gateway` also has a similar commit history to *Daria* and *Olga*, but does not have any reference to PM activities in her GitHub profile. Still, given the similarity in commit history, it

looks like adorsys has program managers who are dedicated to updating roadmaps and documentation which includes UML diagrams.

We can say that in our dataset there are only a few cases where there are those dedicated to UML diagramming without also contributing to source code, which indicates that it is rare in open source projects.

Repository Name	Author Name	Number of commits
w3f/polkadot-spec	Fabio Lama	1378
wix/detox	wixmobile	343
progit/progit	Igor Murzov	156
programmevitam/vitam	Clemence Boyer	64
edmcouncil/fibo	Mike Bennett	85
kubernetes/enhancements	Patrick Ohly	61
hyperledger/aries-rfcs	Brent	52
adorsys/xs2a	Daria Lavrenova	50
hyperledger/aries-rfcs	ashcherbakov	41
hyperledger/aries-rfcs	Vinomaster	32
programmevitam/vitam	edith	32
distribution/distribution	Mary Anthony	29
deegree/deegree3	Danilo Bretschneider	26
progit/progit	Anthony Gaudino	24
apache/isis	Alexander Schwartz	21
uportal-project/uportal	Christian Cousquer	21
openshift/enhancements	Enxebre	17
adorsys/open-banking-gateway	Dora Nziali	16
adorsys/xs2a	Olga Levandovska	16
distribution/distribution	John Mulhausen	15

TABLE 5.9: Dedicated diagrammers

RQ3: Who is Creating and Maintaining UML Design Diagrams?

Finding 1: There is a large and statistically significant difference between the average contribution periods of UML committers versus non-UML committers.

Finding 2: There is a large and statistically significant difference between the average number of UML committers versus non-UML committers.

Finding 3: There is a large and statistically significant difference between the number of commits UML committers make versus non-UML committers.

Finding 4: It is rare to have someone committing to UML who is not also working on source code.

5.3.7 Conclusions

We found a large and statistically significant difference between the contribution periods of UML committers versus non-UML committers. In the median case, UML committers have a contribution period of more than double non-UML committers, at 1,735 days versus 801 days for non-UML committers. This implies that the average UML committer sticks around a project longer than the average non-UML committer. This may be true for a variety of reasons. One reason is that those who stick around longer are likely core developers who understand the system and are better able to create UML diagrams than casual contributors. Another potential reason is that documentation is generally not a fun task, so those who are around a project for a long time and committed to its success are more likely to do documentation work even if it is not fun. Lastly, those who are around a project for a long time have more opportunity to work on UML diagrams.

We also found a large and statistically significant difference between the number of UML committers versus non-UML committers. UML committers are usually vastly outnumbered by non-UML committers. In our dataset, we see an average of 41.76 non-UML committers per repository, versus an average of only 2.54 UML committers per repository. This implies that UML committers are usually a small subset of the total contributors to a project. We also found a large and statistically significant difference between the number of commits made by UML committers versus non-UML committers. In the median case, UML committers make 5 times more commits than non-UML committers.

The fact that UML committers are usually a small subset of the total contributors and also generally make more commits than non-UML committers likely points to the fact that UML committers are generally core developers for a project. Last, we found that UML committers are almost always also involved in the development of source code in a project as well. We found only 12 repositories where there was a UML committer who did not also contribute to source code.

5.4 RQ4: What Types of Projects are UML Diagrams Found in?

In this section we explore the characteristics of repositories with UML diagrams, and compare them to repositories without UML diagrams. We examine the effect of programming language, amount of activity, and community size on the likelihood of a repository having UML diagrams. We also look at some examples of OOP centric UML diagrams, and how they can be warped to fit in non-OOP languages

5.4.1 Methodology

In this section, when we discuss which projects have UML diagrams, we refer to repositories which have had at least one UML diagram in their history. We consider diagrams which were tagged during the UML extension tagging process described in Section 3.3. The repository statistics (e.g., main language, number of commits) are those from SEART GHS.

5.4.2 UML by Main Programming Language

The discussion on which programming language is best is a long-standing one which often sparks passionate debates. Frameworks, tooling, and even philosophies are built around the choice of a programming language. It stands to reason then that the choice of programming language might also have an effect on the choice of modeling language. To explore this, we can start by looking at Figure 5.9 which shows the number of repositories with UML by main language (Nix, Smalltalk, and Elixir are excluded as there were no repositories with UML in those languages). We see Java, C++, and Python in the top spots for total number of repositories with UML.

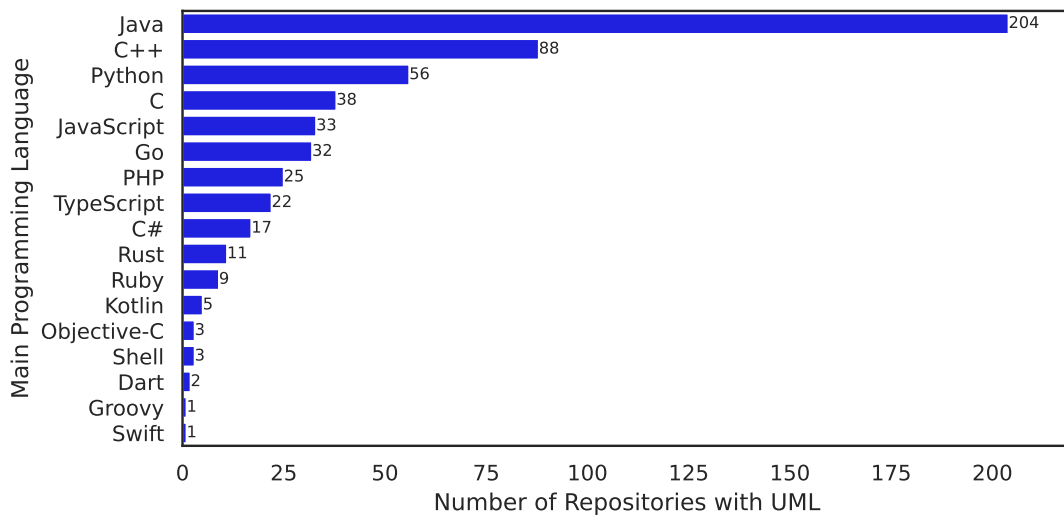


FIGURE 5.9: Number of repositories with UML by language

Given our dataset is not evenly distributed amongst the programming languages, we also look at the percentage of repositories with UML by language (Figure 5.10). We still see Java and C++ in first and second respectively, but Python is replaced by Go (with Python moving to 10th on the list). From both of these figures, we can see the choice of programming language does have an effect on how likely a repository is to have UML. Mainly, Java repositories are much more likely to have UML than repositories in other languages, where the percentage of Java repositories with UML is more than double C++ which has the second highest percentage. After Java, only a few percentage points separate the remaining languages, and we find it difficult to identify distinguishing characteristics of languages which are more likely to have UML diagrams. Java and C++ which support class based OOP are in the top spots, but are followed by Go

at number three which does not have classes. We also find Kotlin, another language which supports class based OOP near the bottom of the list. Also, given UML was built to support OOP methodologies, it is interesting to find C which is not an OOP language above Python, C#, and Kotlin which do support OOP.

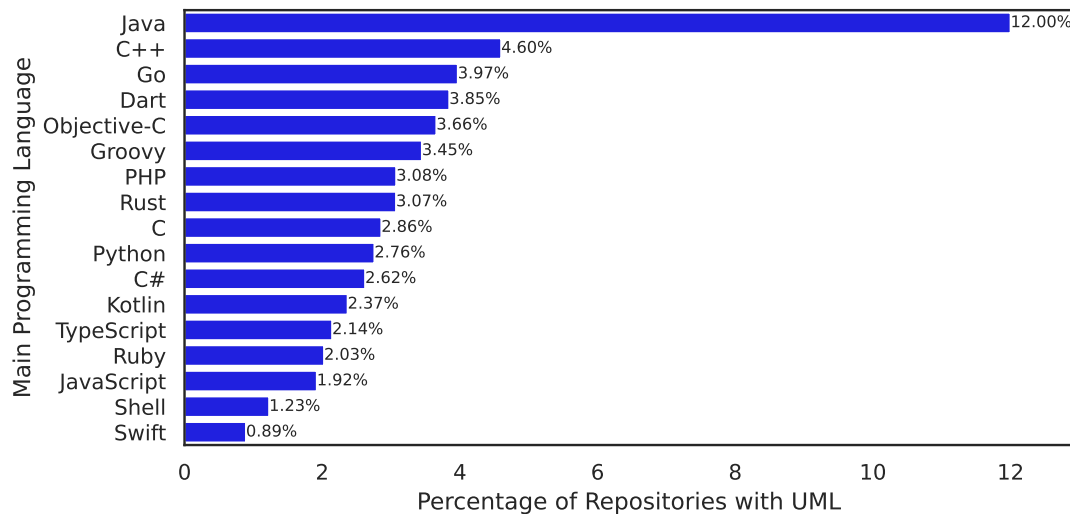


FIGURE 5.10: Percentage of repositories with UML by language

UML in non-OOP Languages – An Example

UML was born out of the need to standardize the many object-oriented approaches that existed before UML. Although originally intended for OOP, many UML diagrams lend themselves well to non-OOP languages. Use case diagrams for instance are often used to model at a high level how users can interact with a system, and are not always tied to the underlying objects in the system. State machine diagrams are another example of a diagram which can be easily used outside of the OOP paradigm. Some diagram types are truly tailored to OOP though, such as class diagrams which are used to model the structure of classes and their relationships with other classes. We explore the dataset to see if there are instances of class diagrams in repositories in non-OOP languages.

Figure 5.11 shows a class diagram we found in the `arm-software/arm-trusted-firmware` repository. The `arm-software/arm-trusted-firmware` repository contains a reference firmware implementation for Arm architecture, and as such is written mainly in C. We see the use of interfaces, classes, packages, and inheritance in the diagram, concepts that do not usually exist in C. Given that, we want to see what each of these concepts means in the context of this repository, and if each concept consistently means the same thing.

First we look at what classes represent in this class diagram. We found it had 3 different meanings: a C file, a C struct, and a C macro. The C files are `arm_io_storage`, `io_driver`, and `io_storage`. For these files, class attributes are a mix of global variables (accessible everywhere) and static global variables (accessible only in the file) and class operations are functions. We found `plat_io_policy` is a C struct used in an interesting way. The struct defines a set of attributes and function pointers that should exist. The `arm_io_storage` file then contains an array of policies which can be indexed using the macros `FIP_IMAGE_ID`, `BL2_IMAGE_ID`, etc. In this way `plat_io_policy` functions similar to an interface in Java, where the attributes and methods that need to be implemented are defined, and may be better represented as an interface in the class diagram.

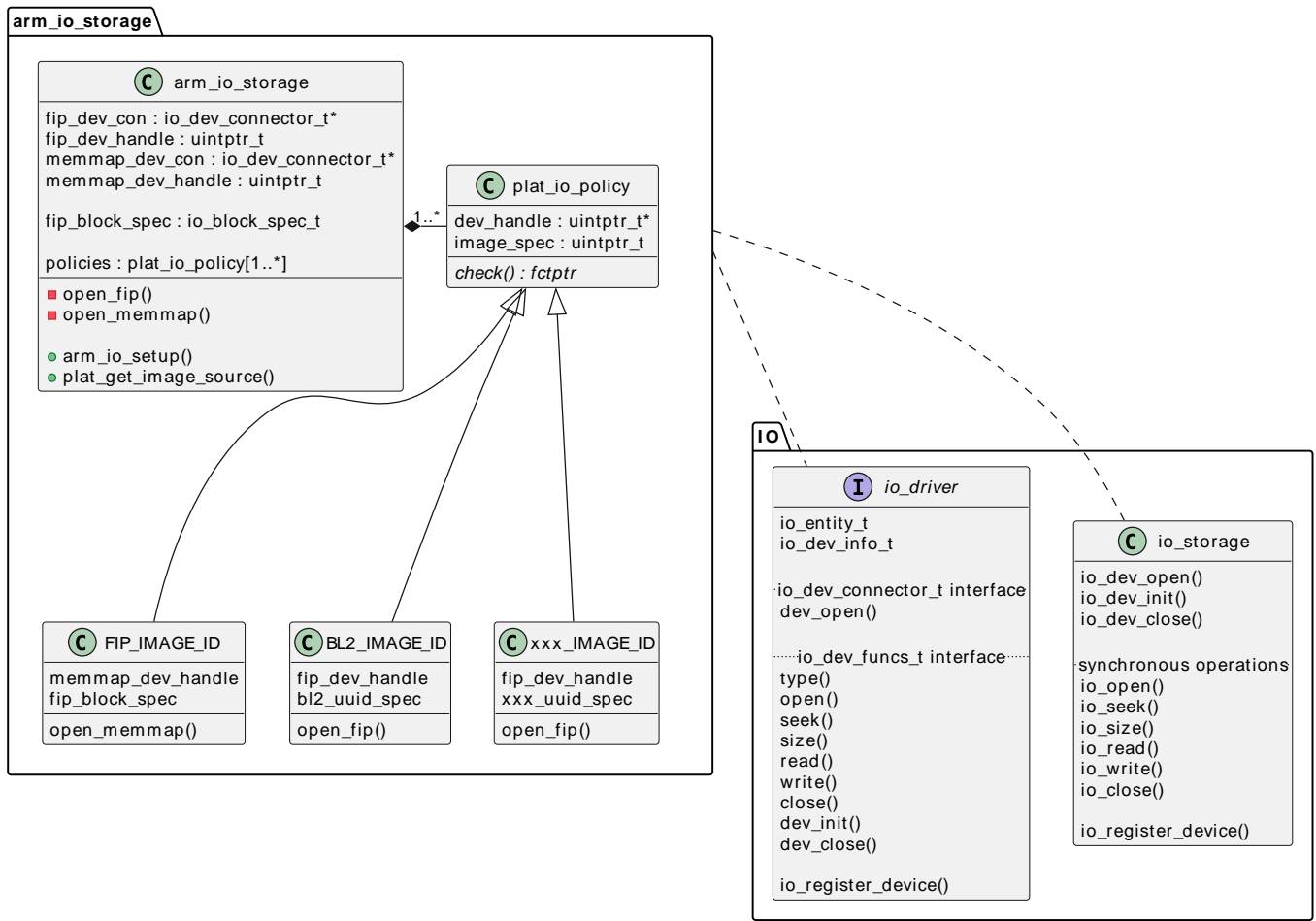


FIGURE 5.11: Class diagram from arm-software/arm-trusted-firmware

Next we look at what the interface in this class diagram represents. The `io_driver` is a header file which contains a set of functions that must be implemented for a device driver to be valid. This set of functions is provided by the `io_dev_funcs_t` structure. Finally, we look at the packages `arm_io_storage` and `IO` package in the class diagram. In this case `IO` has meaning as there is a folder named `io` containing the files shown in the class diagram. The `arm_io_storage` package has less of a clear meaning. We find that the `xxx_IMAGE_ID` are defined in the file `/include/export/common/tbbr/tbbr_img_def_exp.h`, but `arm_io_storage` and `plat_io_policy` are defined across dozens of files with the naming pattern `/plat/<platform>/common/<platform>_io_storage.c`.

Given this really odd example, does the use of a class diagram make sense in a non-OOP context? One of the main benefits of UML is that it provides a standard way to communicate how a system is structured and behaves in a precise way, so a developer can understand the system without having to resort to reading the code. A lot of that is lost here in this example. It was not immediately clear without exploring the code what a class represents, and even after exploring, we found the meaning of a class was not consistent. Even given these shortcomings, the use of a class diagram in this context is not completely without merit. The diagram guided us through exploring the repository, and the use of inheritance, interfaces, and associations helped us understand the intention of the code, even if it was not implemented in a strictly object-oriented way. Given that, the flexible use of UML class diagrams here looks beneficial, but does lose clarity compared to being used in an object-oriented context.

5.4.3 UML by Activity and Community Size

Given UML diagrams are used as a tool to describe various aspects of a system, we hypothesized that repositories with lots of activity and large communities would be more likely to have UML diagrams. We analyzed the values of 10 repository attributes with and without UML diagrams (Table 5.10). Five of them had a statistically significant difference, but only number of commits has a non-negligible effect size.

Attribute	Mann Whitney U		Cliff's Delta	
	p-value	Statistically Significant	Effect Size	Interpretation
commits	<.0001	✔	0.295	small
total pull requests	<.0001	✔	0.130	negligible
open pull requests	<.0001	✔	0.136	negligible
stargazers	<.0001	✔	-0.101	negligible
open issues	<.005	✔	0.072	negligible
forks	0.931			
contributors	0.929			
releases	0.763			
watchers	0.565			
total issues	0.744			

TABLE 5.10: Statistical significance of repository statistics of UML vs non-UML

RQ4: What Types of Projects are UML Diagrams Found in?

Finding 1: Java is the most likely programming language to use UML, with 12% of repositories seeing UML usage at some point in their history.

Finding 2: There is a small but statistically significant difference between the number of commits a repository has made for repositories with UML vs those without (with UML having more).

Finding 2: Outside of number of commits, there is little difference in activity and community size between repositories with UML and those without.

5.4.4 Conclusions

We found that the choice of programming language does have an effect on how likely a repository is to have UML. Java sees nearly 12% of repositories with UML at some point, whereas Swift sees less than 1% of repositories with UML. Although we see a clear difference in the likelihood of a repository having UML by language, we do not see a pattern in the types of languages which are more likely to have UML diagrams. In the future, it would be interesting to look at the types of diagrams used by each language.

We also found there are little differences in amount of activity and community size of UML versus non-UML repositories. Although we found a small difference in the number of commits, the difference is not large enough to support our hypothesis that repositories with lots of activity and large communities would be more likely to have UML diagrams.

5.5 RQ0: Why is UML Underutilized in Open Source Projects?

In this section we look at the results from the previous research questions and provide insights into why UML is being underutilized in open source projects. First, we confirm that it is underutilized. We see only 4.1% of repositories in our dataset have ever used UML diagrams.

One reason why the usage of UML is so low could be due to the numerous tools that can be used to generate UML diagrams. In RQ2, we identified 30 different tools that can be used to generate UML diagrams. Although XMI is supposed to provide a standard exchange format for UML diagrams, we found many tools do not support it. Even when XMI is supported, we found that the XMI files generated by different tools are not always compatible with each other. This means if a developer wants to contribute to the UML diagrams of a project, they may have to set up and learn a new tool. Given documentation is already perceived as a chore by many developers, having a difficult barrier to entry may be enough to turn away developers.

In RQ3, we identified that UML committers are typically a small fraction of the total contributors to a project. This could have a few implications. The small percentage of UML committers could be due to the reasons mentioned above – that there are many tools, and many have a steep learning curve. If it is not quick to set up and use the diagramming tool of choice, it may only be worth it for those who are truly invested in a project to learn. The recent uptick in the use of lighter weight UML diagramming tools seen in RQ1 may be due to the smaller barrier to entry for casual contributors.

Another contributing factor to the modest percentage of UML committers might lie in the fact that only a small portion of contributors are knowledgeable enough to create relevant UML diagrams. Agrawal *et al.* found while studying open source projects from GitHub, that over 77% of projects exhibit the “Hero pattern”, the pattern that 20% of the total contributors complete 80% of the contributions [1]. If we distill this further, to have an individual who will create UML diagrams, that individual must not only belong to this 20% who are deeply versed in the project, but must also have proficiency in UML, recognize the utility of UML diagrams, and must be motivated enough to create them. With each of these successive requirements, the pool of potential UML contributors narrows further and further.

5.6 Summary

In this chapter we looked at a zoomed out view of UML usage in open source repositories. We looked at the numerous formats that UML diagrams are stored in and how the popularity of these formats has changed over time, with PlantUML being the most popular format in recent years. We have seen that authors who contribute to UML diagrams are generally a small minority of the total project members, and they also tend to be the most active and long-standing members. We also saw that Java projects are the most likely to use UML diagrams.

In the next chapter we take a zoomed in look at 3 repositories to see how UML is used in practice. Given Java is the most likely to have UML diagrams, and PlantUML is the most popular format in recent years, we look at Java repositories with PlantUML diagrams. We explore the usage of UML in these repositories, and how the diagrams evolve with the code.

Chapter 6

Case Studies

In this chapter we look at examples of the usage of UML in open source repositories. Using Drifter (<https://drifter.si.usi.ch/>), we explore how the repositories use UML diagrams, and how that usage changes over time. We look at three case studies:

1. Orekit, a low-level space dynamics library.
2. Teammates, a web-based student peer evaluation system.
3. Dataverse, a software platform for sharing, finding, citing, and preserving research data.

6.1 Definitions

There are many definitions that were introduced throughout the thesis so far. Table 6.1 provides a recap of the important ones that will be used in the case studies.

Term	Description
Contribution Period	The period of time between the first commit and the last commit of a contributor.
Non-UML Committer	Contributors to repositories who have never created or modified a UML diagram.
UML Committer	A UML committer is a contributor who has created or modified at least one UML diagram in a repository. They may or may not have also contributed to other file types.
Raw Author	An author taken directly from the git commit data, grouped by unique names and emails.
Alias	A raw author we determine is the same as another raw author.
Clean Author	An author that represents an aggregation of raw authors who are aliases of each other.
Real Author	An author we determine as a true representative of an author in the git repository. These are either raw authors who have no aliases, or clean authors which aggregate a set of aliases.
Java Reference	A Java class, interface, or enum.
Coverage	The percentage of Java references that have a corresponding UML diagram.
Method Coverage	For a given Java reference, the percentage of methods covered in a UML diagram. If a class is represented in multiple diagrams, the percentage can be aggregated using min, max, and average across all diagrams.
Attribute Coverage	For a given Java reference, the percentage of attributes covered in a UML diagram. This metric can be aggregated same as method coverage.

TABLE 6.1: Definitions used in case studies

6.2 Orekit: An Impressive Feat of Diagramming

The first project we examine is Orekit,¹ a low-level space dynamics library written in Java. It has been used by many governmental agencies, including but not limited to the U.S. Naval Research Laboratory, the Swedish Space Corporation, and the European Space Agency.² Given the gravity of precision required in the space industry, we expect they launch well documented and maintained releases. We selected this project because it had just that – some of the most comprehensive PlantUML diagrams when it comes to low-level detail in our dataset. At the time of building our dataset, Orekit had the GitHub stats shown in Table 6.2.

Created	Stars	Forks	Watchers	Releases
2014-08-11	126	68	19	24

TABLE 6.2: Orekit GitHub statistics

It is still an active project and has commits all the way until the day we froze our dataset, 04.01.2023. To get an idea of how active the UML diagramming is, we can take a look at the release view coverage history shown in Figure 6.1. At first glance, we might only notice that the overall coverage percentage is going down over time. If we take a closer look though, we see that the number of references goes from 300 to over 1000, more than tripling, but the coverage percentage only drops 10%.

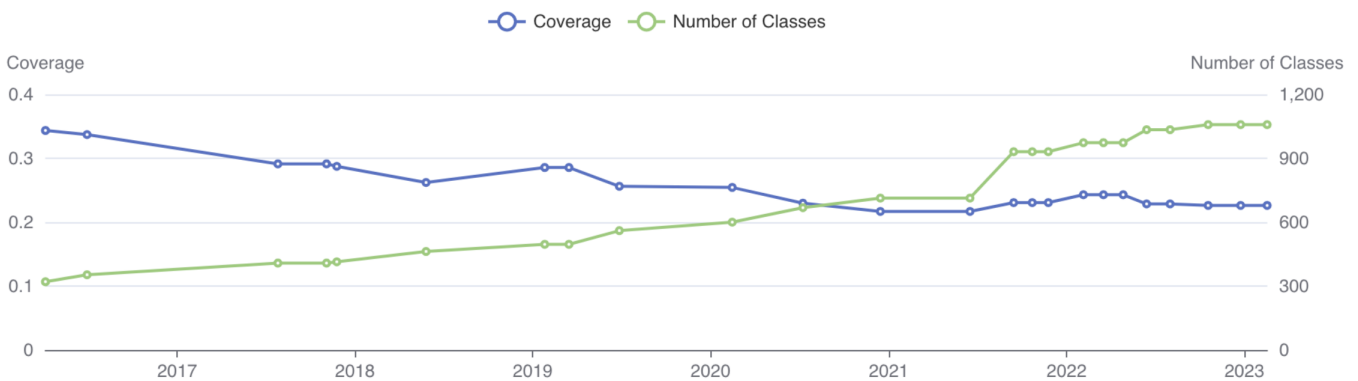


FIGURE 6.1: Orekit release view coverage history

If we take a look at Table 6.3 which shows the first and last releases, we can see that the number of references (classes, interfaces, enums) covered more than doubles from the first release to the last release.

Release version	Release date	References	Covered references	Diagrams
7.2	2016-04-05	396	145	26
11.3	2022-10-18	1344	304	55

TABLE 6.3: Orekit release view statistics

Let us take a look at the package diagram from the first release seen in Figure 6.2. As a reminder, inner most circles are Java references, green represents covered references, white represents uncovered references, and circles containing other circles are packages. We highlight a few packages in red that appear to be well

¹<https://github.com/cs-si/orekit>

²Orekit, who is using it: <https://www.orekit.org/>

covered in the first release. We wanted to see if a package that is already well covered would remain well covered in future releases. We chose one small package (`org.orekit.propagation.integration`), one medium-sized package (`org.orekit.propagation.semianalytical.dsst.forces`), and one large package (`org.orekit.time`), whose histories can be seen in Figures 6.3a, 6.3b, and 6.3c respectively. We show only snapshots where the package changed in some way.

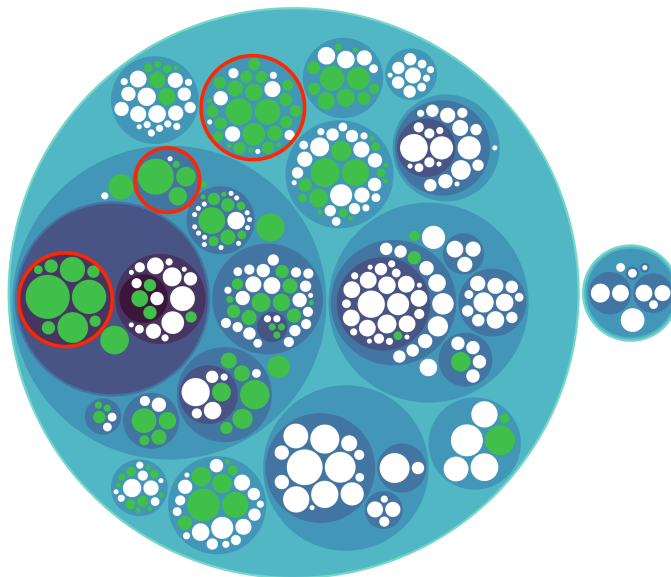


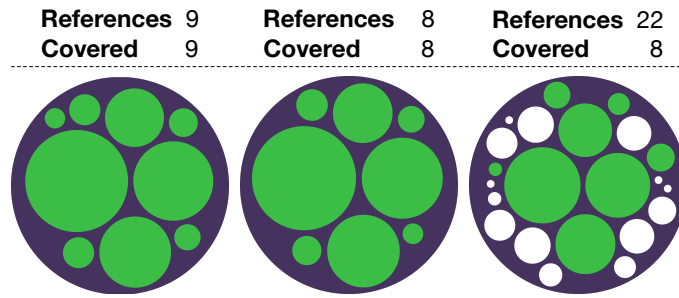
FIGURE 6.2: Orekit package diagram from release 7.2

In general, the number of references increases, but the number of references covered either stays the same, or in some cases, goes down. In the `forces` package history Figure 6.3b, we can see one of the few cases where the number of references decreases. We can see the history of the reference which was removed, `DSSTCentralBody`, in Figure 6.4 (see Section 3.6 for description of diagram). This shows that the `DSSTCentralBody` was deleted, and then subsequently, the diagram was updated. Manual inspection confirms that the diagram was updated correctly to remove the reference to `DSSTCentralBody`.

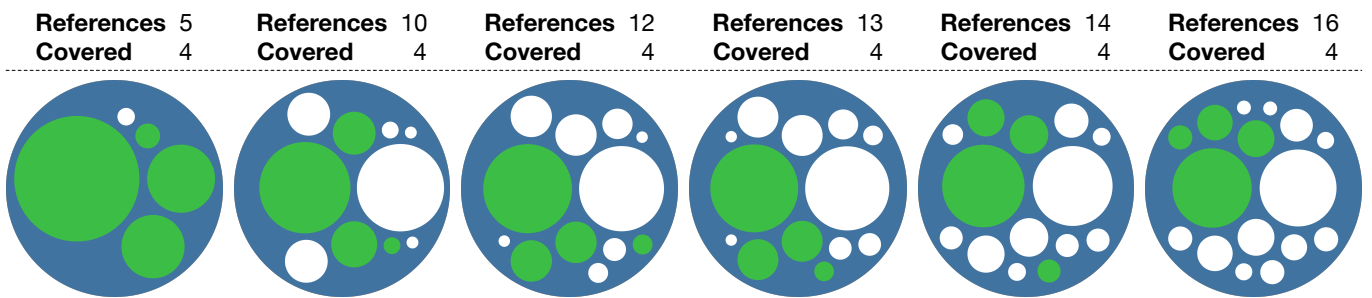
We did a similar investigation to see why there was the big drop in covered references in the `time` package between the first two changes, where the number of covered references goes from 23 to 17. We found the references `GMSTScale`, `GPSScale`, `QZSSScale`, `TCBScale`, `TCGScale`, `TDBScale` were in java code and covered by `time-class-diagram.puml` in the first release, but were not in a diagram in the second release. In Figure 6.5, we see `time-class-diagram.puml` before and after the references were removed. The top diagram is from the first release (7.2) where the references existed, and the bottom is from the release where the coverage went down (8.0). These are abbreviated versions of the full class diagrams. We took the original PlantUML diagram source³⁴ and removed entities unrelated to the missing references. In the top diagram, there is an interface called `TimeScale` which is implemented by more than 10 classes. In the bottom diagram, many of the scales are purposely omitted to make the diagram more readable. Even though this change causes the coverage to go down, the diagram looks more organized and readable after the change. This highlights a few interesting points. The first is that a more detailed diagram is not always necessarily a better diagram. The second is that these diagrams do not look like they are auto-generated. The third is that they are likely being used by somebody as they took the time to update an old diagram to be more usable.

³Release 7.2: <https://github.com/cs-si/orekit/blob/46abd9/src/design/time-class-diagram.puml>

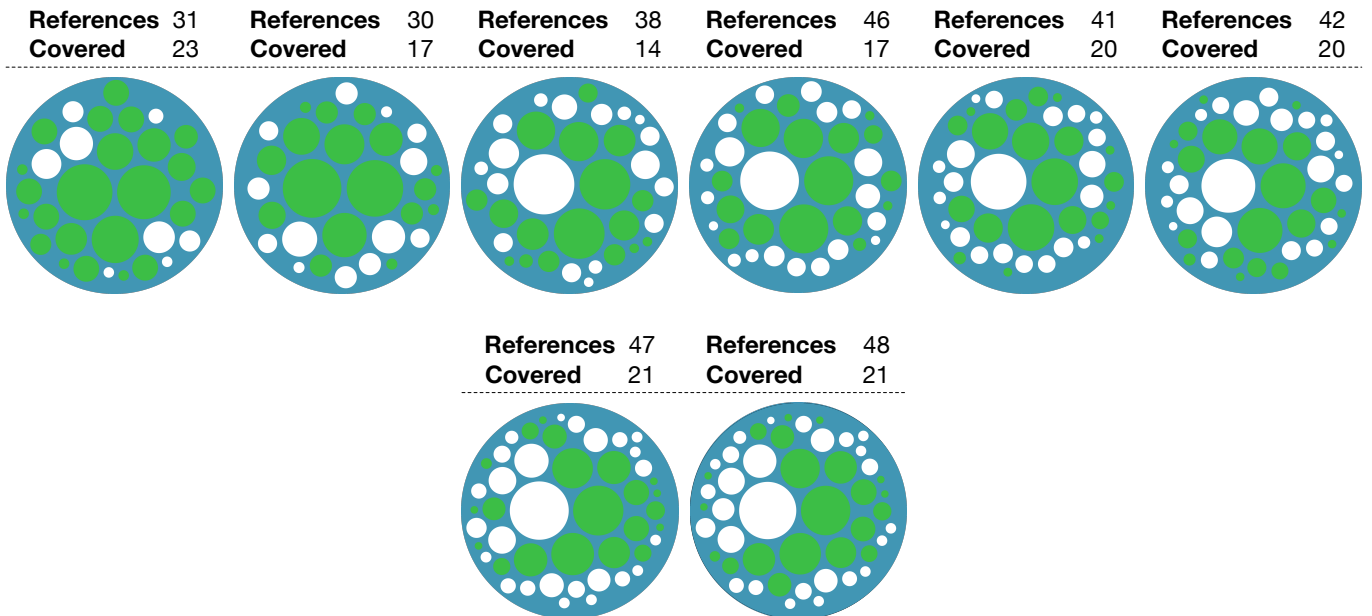
⁴Release 8.0: <https://github.com/cs-si/orekit/blob/16e633/src/design/time-class-diagram.puml>



(A) Orekit forces package history



(B) Orekit integration package history



(C) Orekit time package history

FIGURE 6.3: Evolution of Orekit Packages

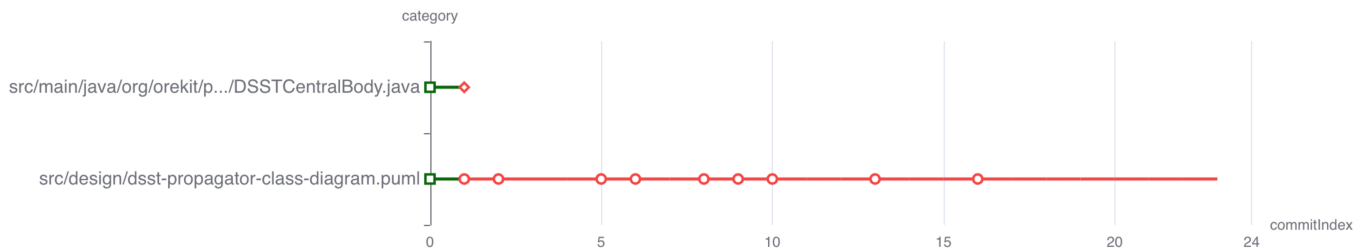


FIGURE 6.4: Orekit DSSTCentralBody file history

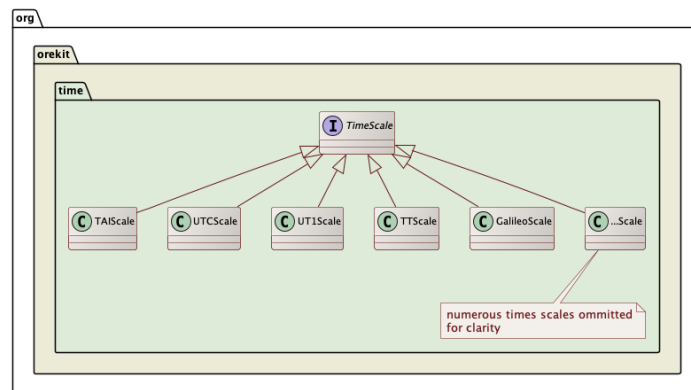
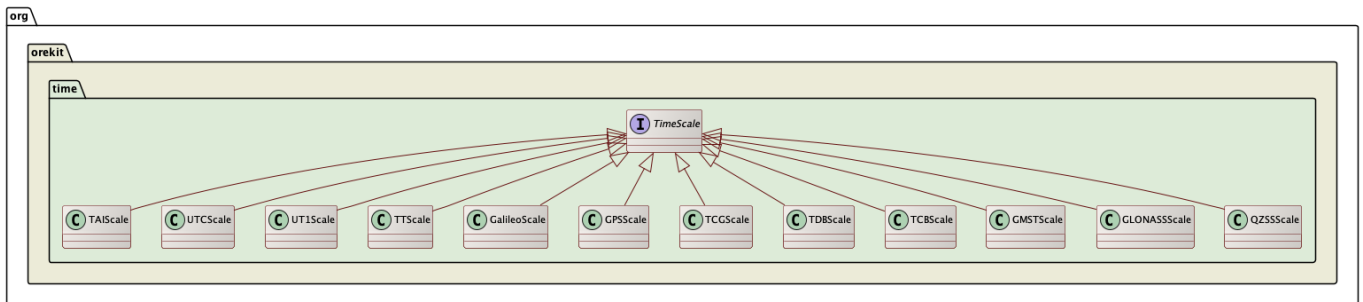


FIGURE 6.5: Orekit time package simplified class diagram (top release 7.2, bottom release 8.0)

6.2.1 Method and Attribute Coverage

So far we have looked at the overall coverage of Java references, but we have not looked at the detailedness of that coverage. We show two metrics for diagram detailedness: method coverage and attribute coverage. At first glance it looks like attribute coverage is much higher than method coverage. We also get more shades of coverage for methods versus attributes.

To explore this phenomenon, we highlight `GeodeticPoint` in the attribute coverage diagram in Figure 6.8 and the method coverage diagram in Figure 6.9. If we look at the attribute coverage diagram and Listing 6.1 which covers it, we see that the diagram does not contain any attributes for `GeodeticPoint`, but the attribute coverage is green (100%). If we take a closer look at the source code, we see that `GeodeticPoint` has no public attributes.

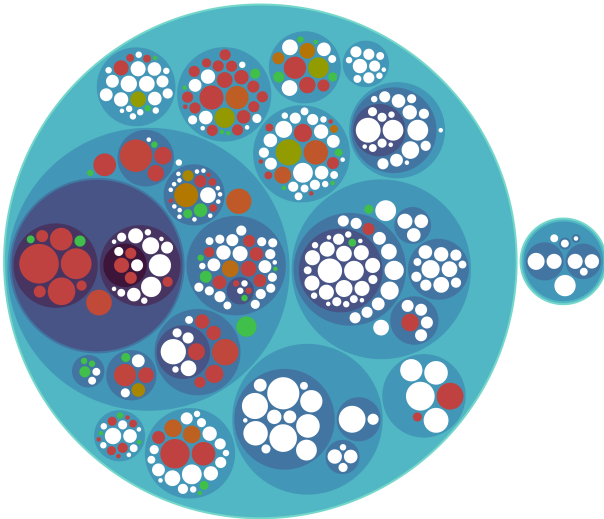


FIGURE 6.6: Orekit method coverage release 7.2

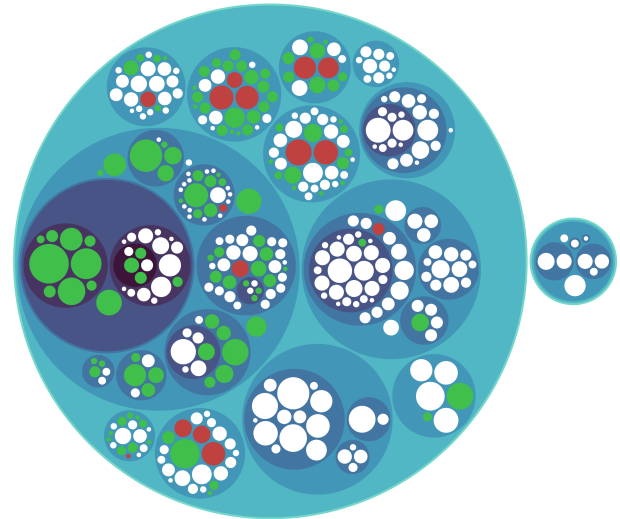


FIGURE 6.7: Orekit attribute coverage release 7.2

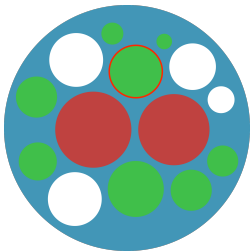


FIGURE 6.8: Orekit bodies package attribute coverage release 7.2



FIGURE 6.9: Orekit bodies package method coverage release 7.2

```

package org.orekit #ECEBD8 {
  package bodies #DDEBD8 {
    ...
    class GeodeticPoint {
      +double getLatitude()
      +double getLongitude()
      +double getAltitude()
      +Vector3D getZenith()
      +Vector3D getNadir()
      +Vector3D getNorth()
      +Vector3D getSouth()
      +Vector3D getEast()
      +Vector3D getWest()
    }
    ...
  }
}

```

LISTING 6.1: Abbreviated bodyshape-class-diagram.puml from release 7.2

We find that many of the references with 100% attribute coverage are similar cases, where the reference has no public attributes. As expected, most references have at least one public method, and only a small percentage of those methods look to be covered in the average case. For the 4 Java references that show 100% method coverage in Figure 6.6, we find they are all interfaces. When exploring the coverage graph we found this is a trend. Nearly every single reference that has 100% method coverage is an interface. It looks like special care is taken to document the methods of interfaces, which makes sense given that an interfaces main purpose is to define the contract (in the form of methods) that implementing classes must follow.

6.2.2 UML to Java References Graph

If we turn our attention now to a snapshot of the UML to Java references graph for release 11.3.2 in Figure 6.10, we see a graph with lots of connections in the center, and many unconnected components on the periphery. Even in this very well documented system we still see a lot of references with no documentation. In the graph we highlight a few interesting cases. In release 11.3.2, the latest release at the time of cloning, we see that out of the 55 diagrams, only 3 have no connections to any Java references.

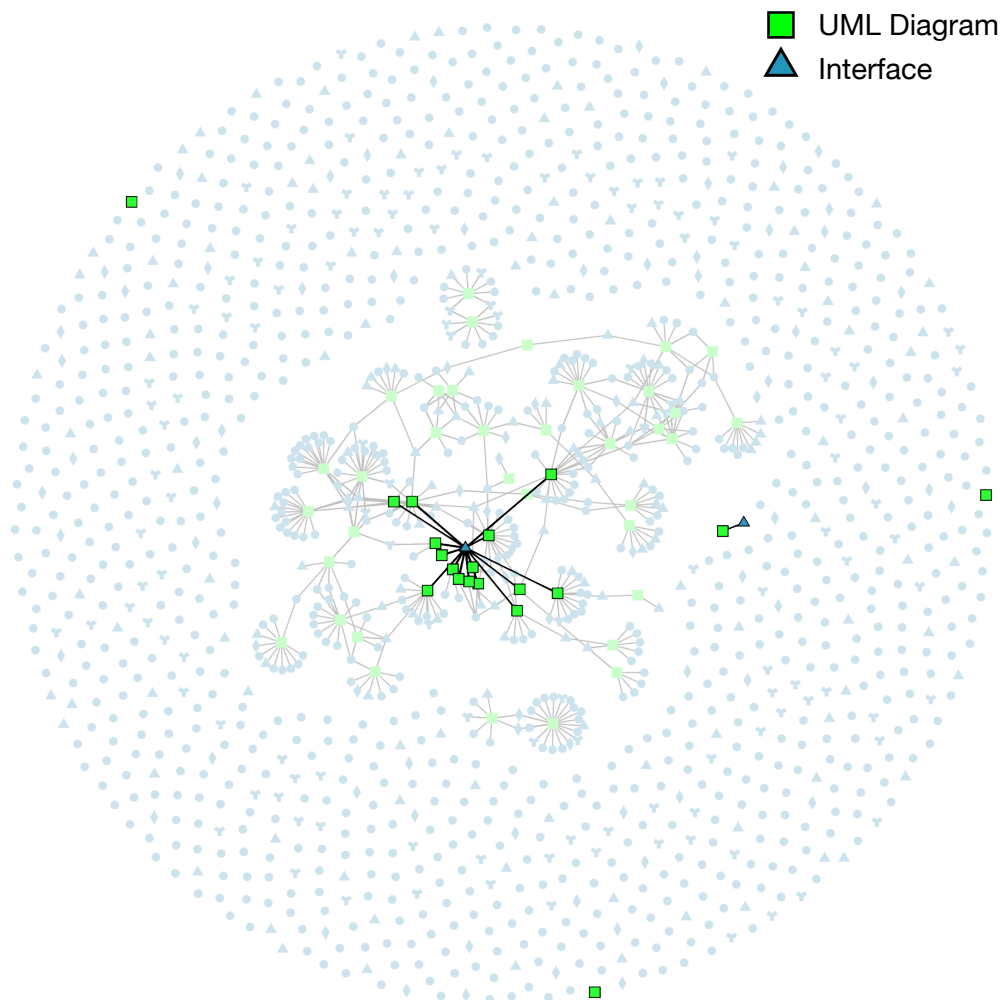


FIGURE 6.10: Orekit UML to Java references graph release 11.3.2

The first unconnected diagram we look at is called `top-packages.puml`. As the name might suggest it is a package diagram, so it does not cover any classes. We can see a small extract of the contents of `top-packages.puml` in Listing 6.2.

```
package org.orekit #ECEBD8 {...
package forces #DDEBD8 { }
package propagation #DDEBD8 { }
package estimation #DDEBD8 { }

estimation --> propagation
propagation --> attitudes
propagation --> forces
```

```
...}
```

LISTING 6.2: top-packages.puml

The next unconnected diagram we look at is called `unscented-kalman-filter-diagram.puml` whose relevant content can be seen in Listing 6.3. When searching the package diagram, we find the three classes modeled in the puml diagram in the project, but the package for the class is `org.orekit.estimation.sequential`. The diagram has an incorrect package name, `unscented`, so our rules do not pick it up as a covered class.

```
package org.orekit #ECEBD8 {
  package estimation.sequential.unscented #DDEBD8 {
    class UnscentedKalmanEstimator { }
    class UnscentedKalmanEstimatorBuilder { }
    class UnscentedKalmanModel { }
  }
}
```

LISTING 6.3: unscented-kalman-filter-diagram.puml

If we look at the file history for the `UnscentedKalmanEstimator`, we see that both the diagram and class were added at the same time, but the diagram is always out of sync with the class (red line).

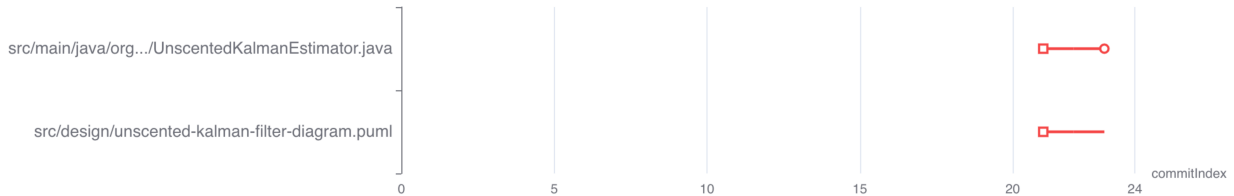


FIGURE 6.11: UnscentedKalmanEstimator diagram history

The last unconnected diagram we look at is called `dsst-partial-derivatives-class-diagram.puml` and its relevant content can be seen in Listing 6.4. This diagram is similar to the previous one. We again do not mark the classes as covered by this diagram because the package is modeled incorrectly. There is another class diagram which does model the package correctly and it is shown in Listing 6.5.

```
package org.orekit #ECEBD8 {
  interface Propagator { }
  interface MatricesHarvester { }
}
```

LISTING (6.4) dsst-partial-derivatives-class-diagram.puml

```
package org.orekit.propagation #ECEBD8 {
  interface Propagator { }
  interface MatricesHarvester { }
}
```

LISTING (6.5) partial-derivatives-class-diagram.puml

FIGURE 6.12: MatricesHarvester class diagram coverage comparison

This can be seen in the file history for the `MatricesHarvester` class in Figure 6.13. After being added to source, the `MatricesHarvester` class was immediately added to and covered by `partial-derivatives-class-diagram.puml`. The `dsst-partial-derivatives-class-diagram.puml` diagram is never fixed to reflect the correct package name.

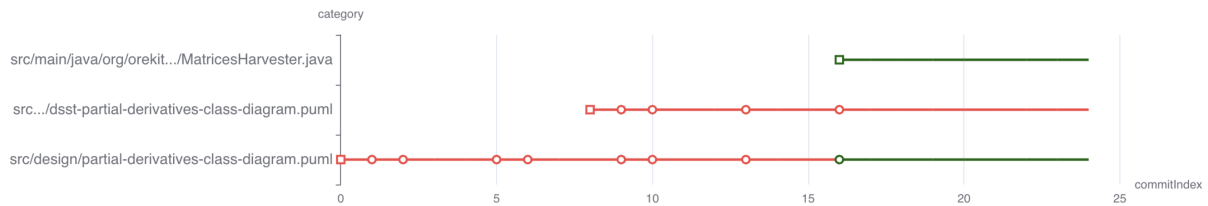


FIGURE 6.13: MatricesHarvester diagram history

The next odd diagram we highlight is `field.puml`, which is the highlighted diagram with only 1 connected reference (interface in this case). In the most recent release of Orekit, the average number of Java references covered per diagram is 11.08. We wanted to know why this diagram, with only 1 reference, was created and if it is actually useful. A snippet of the diagram can be seen in Listing 6.6. The diagram shows interaction with an external library, Hipparchus.⁵ Although the class diagram does show a some classes, we only see one connected reference because we do not take into consideration external libraries.

```

package org.hipparchus #ECEBD8 {
    interface "FieldElement<T>" as FieldElement_T_ { }
    interface "CalculusFieldElement<T>" as CalculusFieldElement_T_ { }
}

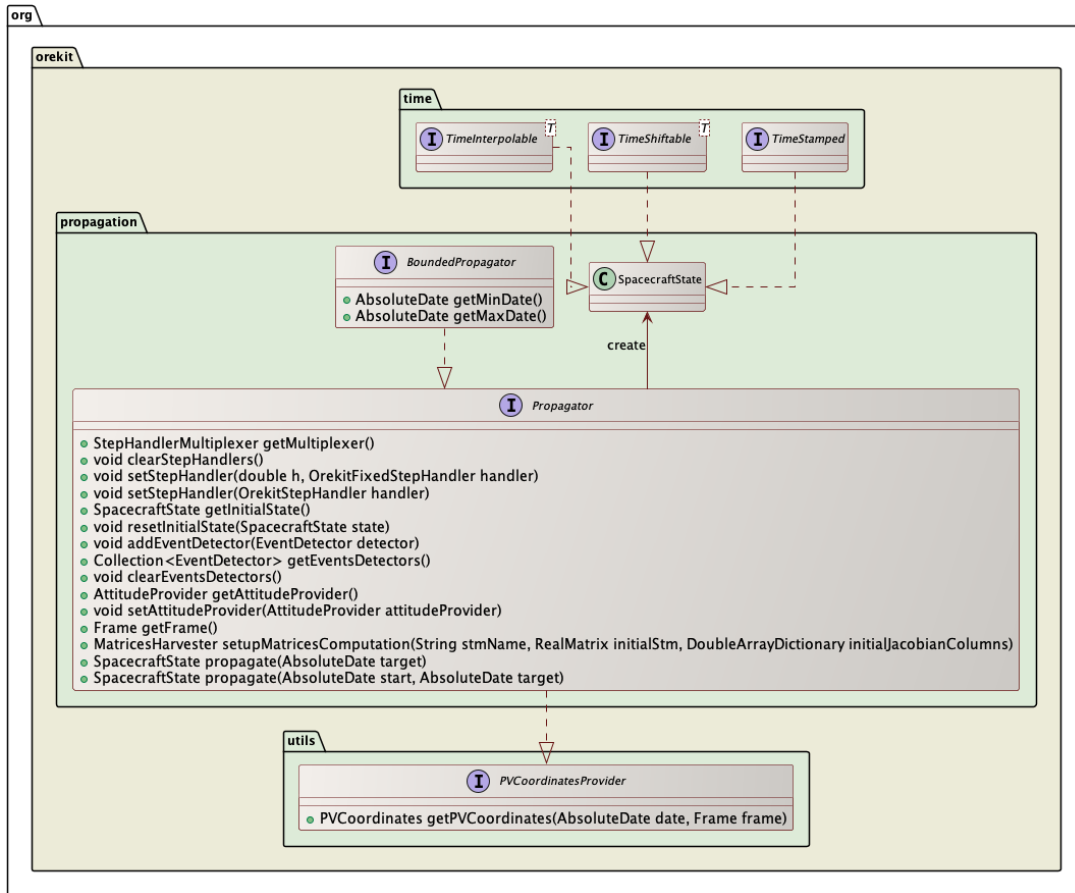
package org.orekit.propagation #ECEBD8 {
    interface "FieldPropagator<T>" as FieldPropagator_T_ { }
    CalculusFieldElement_T_ <-- FieldPropagator_T_
}

```

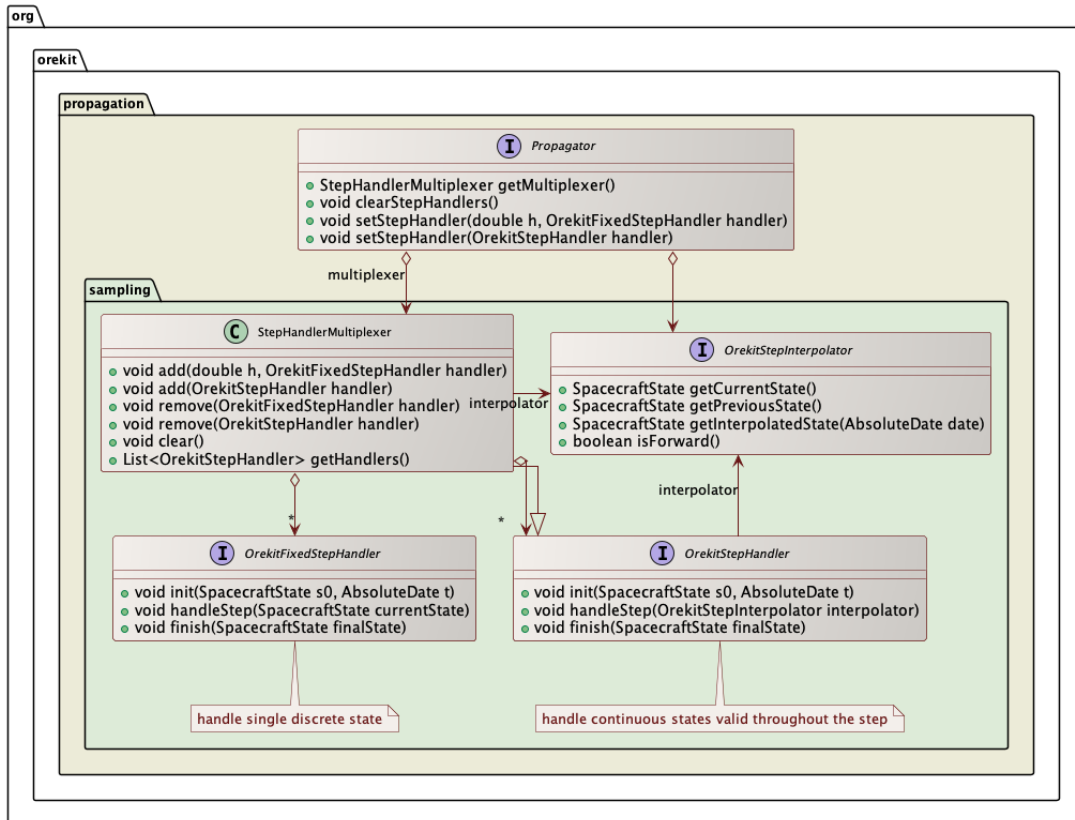
LISTING 6.6: field.puml

The final reference we would like to highlight from Figure 6.10 is the Propagator interface in the center that has connections to 15 diagrams. On average in release 11.3.2 of Orekit, the average reference is connected to 2.12 diagrams. This number drops to 1.72 references per diagram if we exclude interfaces. Although interfaces in general are more connected in the Orekit diagrams, this interface is still an exceptional case. We found it connected to 6 sequence diagrams and 9 class diagrams. When exploring the diagrams, we can see the interface is described with varying degrees of detail. If we look at Figure 6.14a, the main class diagram for Propagator, we see it covers all of the methods. The `sample-class-diagram.puml` shown in Figure 6.14b instead shows just a few methods that are relevant for the interaction between Propagator and the other classes and interfaces.

⁵<https://hipparchus.org/>



(A) propagation-class-diagram.puml



(B) sampling-class-diagram.puml

6.2.3 From the Beginning of Time

So far we have seen what the history of Orekit looks like from the release point of view. Their first GitHub release version was in 2017, but the history traces all the way back to 2003, before both GitHub and Git existed. To get a better look at the evolution of the system, we can examine the commit history shown in Figure 6.15. Although it is still an incomplete history as we lose commits when trying to linearize the history, we see some interesting stories missing from the release view. First there is what appears to be 0 coverage up until 2012.

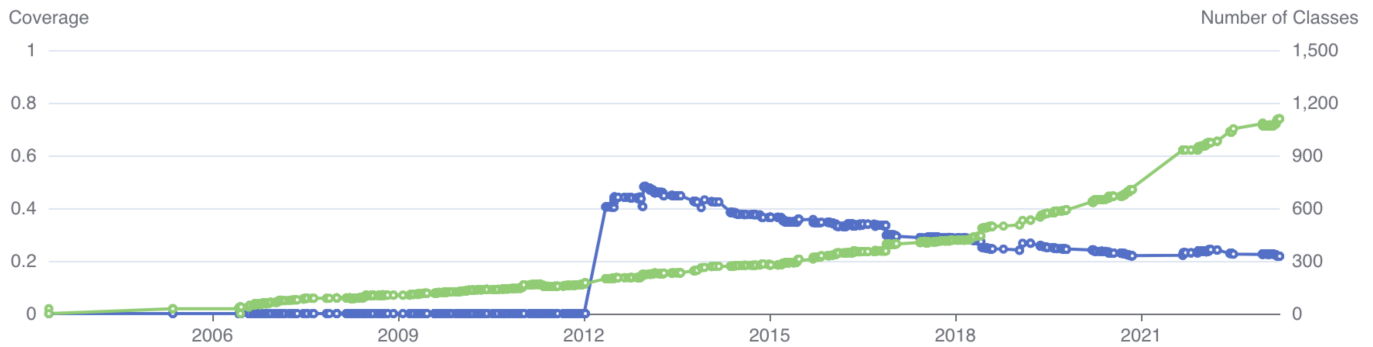


FIGURE 6.15: Orekit coverage history commit view

A deeper analysis shows an evolution in the way diagrams are created and maintained. We see the first diagram committed on 2008-06-25.⁶ It is committed as a PNG and there is no supporting file committed along with it. A month later, supporting files for the images were committed⁷ in .odg (OpenDocument Graphic) format. In 2010, new diagrams⁸ were added with the .di2 extension, whose source comes from Papyrus.⁹ Finally, in 2012 a migration from .di2 to .puml occurred,¹⁰ and the project has been actively using .puml (PlantUML) since. We can see this evolution in Figure 6.16, which shows the number of commits per UML extension by year. From 2012 on, after the switch to PlantUML, there is much more activity in diagramming overall. This is likely because the Orekit developers found it was easier to maintain the diagrams with PlantUML versus with Papyrus or in ODG format. We make this assertion based on the comment of the commit containing the first PlantUML diagrams: *“Updated design and documentation, with new UML diagrams. For diagrams creation, we have switched to PlantUML, which can be directly integrated into the maven build and eclipse and allow very simple diagram creation and update without heavy tools.”*

6.2.4 Where is the UML used?

So we have seen that the Orekit diagrams are fairly comprehensive and maintained, but where are these diagrams being used? Do they only exist as text files for developers to look at, or are they being used in some other way? We found all of the diagrams in the Orekit maven site documentation.¹¹ The documentation website contains a lot of information, among which is a description of the architecture and design. The documentation contains the package diagram shown earlier in Listing 6.2 in the overview section, and then each package has its own page. Each page contains the various UML diagrams for that package, along with descriptions giving more details. We found this use of the diagrams very useful for understanding the system, and the natural language and diagrams have good synergy.

⁶<https://github.com/CS-SI/Orekit/blob/bb2d94c151/src/site/resources/images/attitudes.png>

⁷<https://github.com/CS-SI/Orekit/blob/a816c0be60/src/site/resources/images/attitudes.odg>

⁸<https://github.com/CS-SI/Orekit/blob/45133524d2/src/design/OrekitModel.di2>

⁹<https://eclipse.dev/papyrus/>

¹⁰<https://github.com/CS-SI/Orekit/tree/d310347ba7/src/design>

¹¹<https://www.orekit.org/site-orekit-development/>

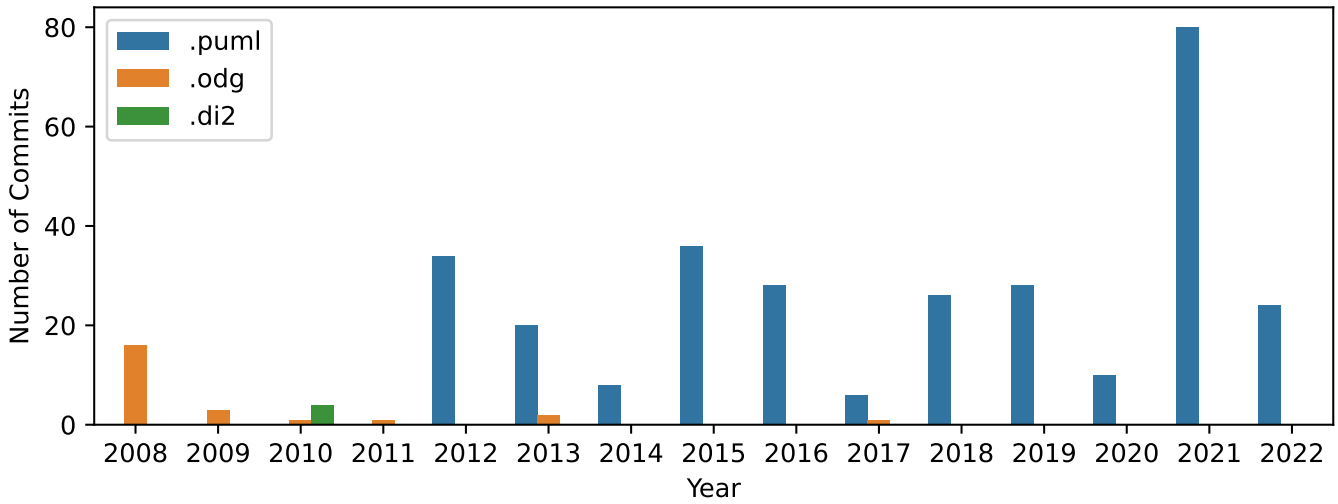


FIGURE 6.16: Evolution of UML tools in Orekit

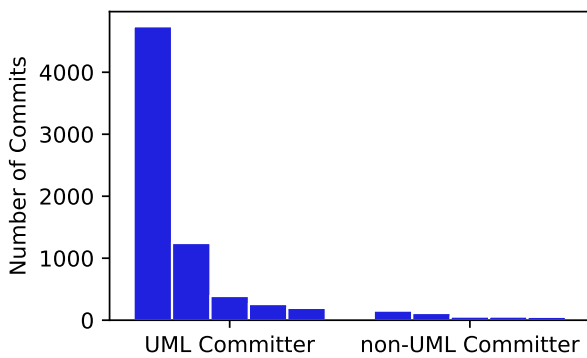
6.2.5 UML Committers

We have seen that new UML diagrams are still being created, and old maintained in the Orekit project. Now we look to see if those who are maintaining the diagrams have the same attributes as discussed in the research questions. We first start looking at some basic stats for authors in Table 6.4. As expected, the number of people diagramming is much smaller than the total number of authors.

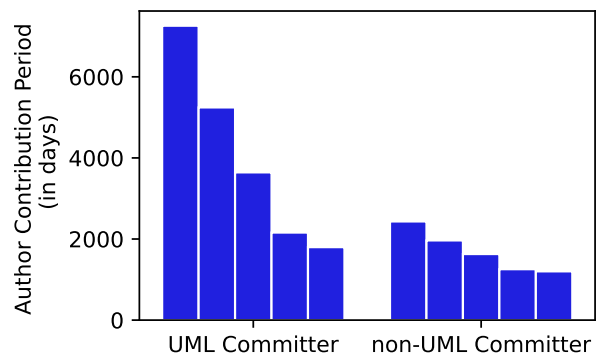
Raw Authors	Authors	Aliases	Authors with Aliases	UML Committers	Non-UML Committers
120	81	63	24	6	75

TABLE 6.4: Orekit author statistics

Looking to Figures 6.17a and 6.17b, we see the contribution period and number of commits for the top 5 UML and non-UML committers. Just as we saw in the average case in Section 5.3, UML committers are those who contribute the most to the project, and are among the oldest contributors.



(A) Top 5 authors by number of commits - UML vs non-UML committers



(B) Top 5 authors by contribution period - UML vs non-UML committers

FIGURE 6.17: Comparison of author statistics

6.2.6 Conclusions

In this case study we looked in-depth at the Orekit project and its usage of UML. We saw that Orekit adopted UML early on in its project history, but the tools they used to create and maintain UML diagrams changed over time. They started by storing diagrams in image format with no supporting diagram file, then moved to ODG (OpenDocument Graphic) format, then to Papyrus, and finally to PlantUML. They settled on PlantUML because it is lightweight, and easy to integrate into their build system and IDE of choice. After migration to PlantUML, Orekit had huge uptick in the number of commits to UML diagrams.

At its peak, Orekit hits an impressive 50% of Java references covered by UML diagrams, and although the coverage percentage drops to 23% in the latest release, the number of references covered more than doubled from the first release. We found that the diagrams are being used alongside natural language documentation on a documentation website. We wrapped up by seeing that the UML committers are among the most active and longest-standing contributors to the project, following the trends we identified in RQ3 (Section 5.3) when we looked at characteristics of UML committers. We now move on to our second case study, Teammates, where we see a similar journey in finding the right UML tooling.

6.3 Teammates: From PowerPoint to PlantUML

The next repository we look at is Teammates.¹² Teammates is a feedback management system for students and teachers used by more than 800,000 users from over 1,110 universities around the world.¹³ It is implemented as a web application and is used in universities to facilitate peer feedback. Some statistics for the repository from GitHub are shown in Table 6.5.

Created	Stars	Forks	Watchers	Releases
2014-05-02	1298	2615	99	152

TABLE 6.5: Teammates GitHub statistics

6.3.1 A Blip in Time

Figure 6.18 shows the coverage history of the Teammates project. The x-axis shows the releases over time, ordered by date. The decrease in number of references with the subsequent rebound around the V7.0-alpha pre-release is a consequence of this date ordering. We can see the official release with the changes that causes the decrease comes later, in release V7.0-beta and beyond. In contrast to Orekit in Section 6.2 where the number of references consistently grew, the number of references for this project stays relatively flat from the first release, and in the case of V7.0, the number of references decreases. Upon further inspection, we see the drop in number of classes is a due to a refactor that moves from JSP (JavaServer Page)¹⁴ to Angular,¹⁵ causing some Java code to move to Typescript.

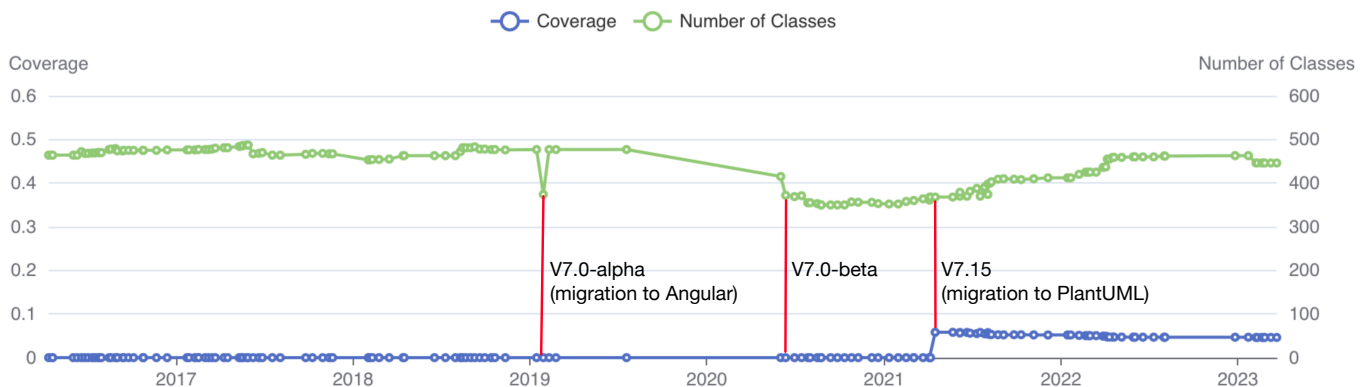


FIGURE 6.18: Teammates release coverage history

6.3.2 From PowerPoint to PlantUML

If we turn our attention now to V7.15, we see that coverage percentage jumps from 0 to 6%. 6% is the max coverage percentage this project reaches (although it is possible it reaches higher before migrating to PlantUML), far from the 50% coverage percentage peak we saw in Orekit. We did however find that similarly to Orekit, Teammates used a different tool for diagramming before migrating to PlantUML. In this case, Teammates used PowerPoint to manage their diagram creation before moving to PlantUML.¹⁶

¹²Teammates GitHub: <https://github.com/teammates/teammates>

¹³Teammates about: <https://teammatesv4.appspot.com/web/front/home>

¹⁴https://en.wikipedia.org/wiki/Jakarta_Server_Pages

¹⁵<https://angular.io/>

¹⁶<https://github.com/TEAMMATES/teammates/blob/e36c2fdf39/docs/images/packageDiagram.pptx>

We found a GitHub issue (Migrate design diagrams from Powerpoint to PlantUML #10623) was created to migrate from PowerPoint to PlantUML in 2020.¹⁷ To summarize from the issue, Teammates wanted to migrate to PlantUML for the following reasons: PlantUML is free, and usable without users needing to install software (through the web interface), PlantUML diagrams are version control friendly, PlantUML makes it easier to use UML notation consistently. These sentiments are similar to those found in Orekit, and adds some evidence as to why PlantUML has become popular in recent years.

One sentiment mentioned in the GitHub issue that was not pointed out in Orekit is that PlantUML makes it easier to use UML notation consistently. Figure 6.19 shows a class diagram from the Teammates project created in PowerPoint, and Figure 6.20 shows the same class diagram created in PlantUML after the migration. The diagram coming from PowerPoint has a few issues regarding UML notation. First, the double lined arrows used here to represent associations are not standard UML. Second, the filled in arrow head is also not used to define an association in standard UML. In the PlantUML version, we can see the associations are represented in standard UML notation.

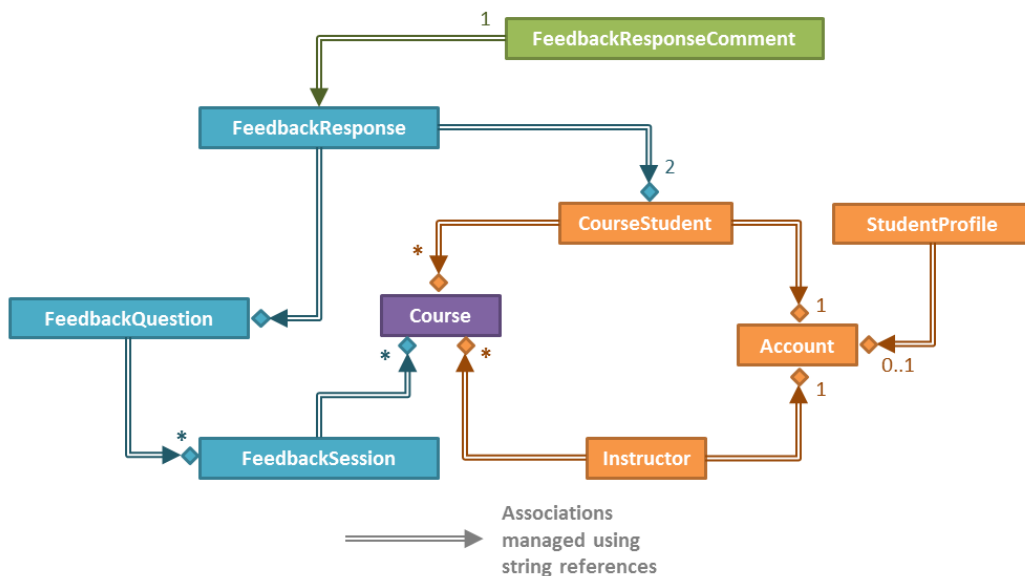


FIGURE 6.19: Teammates storage class diagram from PowerPoint

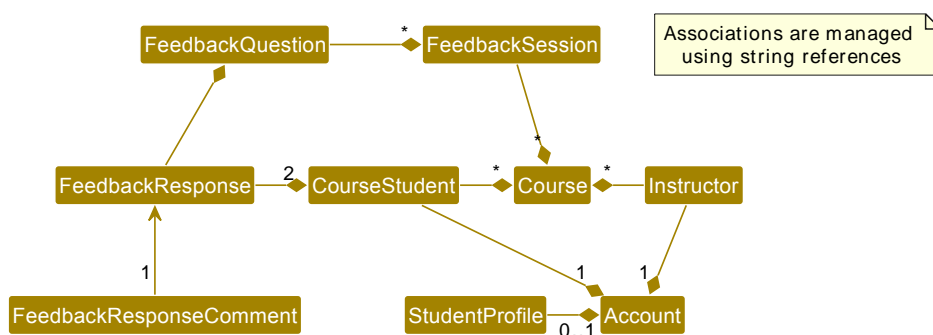


FIGURE 6.20: Teammates storage class diagram from PlantUML

Given Orekit saw a rise in activity after the migration to PlantUML, we wanted to see if the same was true for Teammates. Figure 6.21 shows the number of commits to diagrams in PowerPoint and PlantUML over

¹⁷<https://github.com/TEAMMATES/teammates/issues/10623>

time. For PowerPoint, we consider the diagrams which were eventually migrated to PlantUML, and not every PowerPoint file as they have many PowerPoint documents not used for diagramming efforts. Unlike Orekit, we do not see a big jump in the number of commits to UML diagrams after the migration.

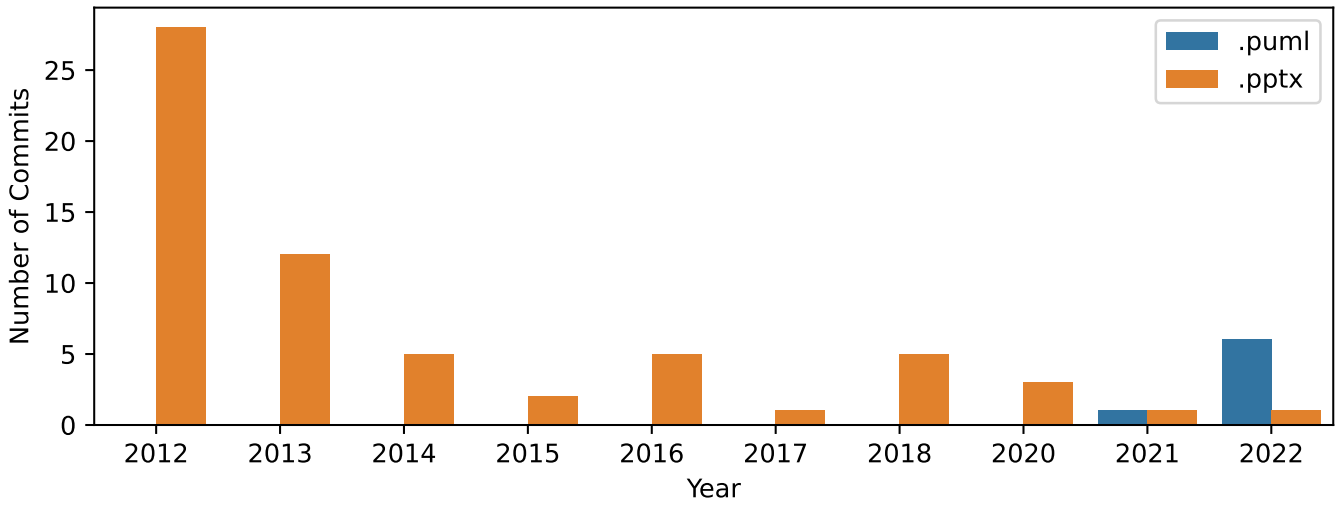


FIGURE 6.21: Evolution of UML tools in Teammates

6.3.3 UML Committers

We now look at the composition of UML committers versus non-UML committers. Table 6.6 shows stats based on the anti-aliasing and number of UML committers in the project. This data includes commits to the PowerPoint files that were eventually migrated to PlantUML (even if they did not use proper UML notation before the migration). We can see that UML committers are a tiny percentage (2%) of the total committers.

Raw Authors	Authors	Aliases	Authors with Aliases	UML Committers	Non-UML Committers
727	620	180	73	12	608

TABLE 6.6: Teammates author statistics

If we look now to Figure 6.22, we see the top 5 authors by number of commits and contribution period. We can see that the top UML committers by number of commits are also the top 3 committers for the repository. In general the UML committers look to be among the contributors with the most commits and longest contribution periods. The 5th place UML committer by number of commits, however, looks to have less commits than the 5th place non-UML committer, and there are still 7 more UML contributors to account for.

Table 6.7 shows the 12 UML committers along with their number of total commits, commits to diagrams, and contribution periods. We can see that the top 2 UML committers are actually the top 2 committers for the repository in terms of contribution period and number of commits, and they handle the majority of the design diagrams. The bottom three contributors in terms of commits, *Tan Yi Guan*, *Hannah*, and *Syasya Azman*, helped migrate the diagrams from PowerPoint to PlantUML, and integrate PlantUML into a tool chain to generate diagrams and place them in developer documentation. Like Orekit, the design documentation can be found on a documentation website made for developers.¹⁸

¹⁸<https://teammates.github.io/teammates/design.html>

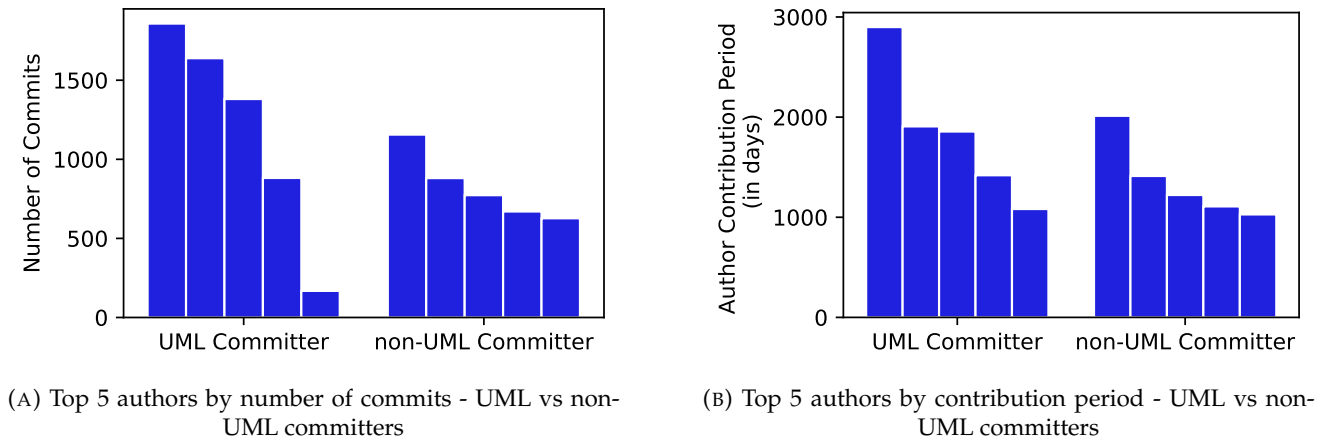


FIGURE 6.22: Comparison of author statistics

Name	Total Commits	Commits to UML	Contribution Period
Damith Rajapakse	1858	40	1856
Wilson Kurniawan	1639	12	2899
Kang Hong Jin	1381	1	900
Thyagesh Manikandan	882	3	1082
Kenny	168	5	250
Xiao Pu	106	1	1419
aldrianobaja.m@gmail.com	80	1	71
Samuel Fang	45	2	1907
Howard Liu	18	1	117
Tan Yi Guan	4	3	136
Hannah	2	1	4
Syasya Azman	1	2	0

TABLE 6.7: UML Committers for Teammates

6.3.4 Conclusions

In this section we looked at the Teammates project and its usage of UML. Like Orekit before, it also used a different diagramming tool before migrating to PlantUML. We found the migration was done for similar reasons to Orekit, but for the added reason that PlantUML makes it easier to use UML notation consistently. Unlike Orekit, we did not see a big jump in the number of commits to UML diagrams after the migration. Teammates also sees only a small percentage of committers contributing to UML diagrams, with the top committers in the repository also being UML committers. Unlike Orekit, we did see some casual contributors committing to UML diagrams, but the majority of diagramming is still done by the top committers. Now that we have looked at two case studies, we move to the final case study, where we look at the Dataverse project.

6.4 Dataverse: PlantUML from the Start

The final repository we look at is Dataverse.¹⁹ Dataverse is a software platform for sharing, finding, citing, and preserving research data, and is managed by the Institute for Quantitative Social Science (IQSS) at Harvard University. GitHub statistics for the repository are shown in Table 6.8.

Created	Stars	Forks	Watchers	Releases
2013-11-01	737	413	67	59

TABLE 6.8: Dataverse GitHub statistics

6.4.1 Designing Before Coding

Figure 6.23 shows the commit coverage history for Dataverse. The annotated points on the graph show when the coverage percentage increases, which we will refer to as coverage events. The first commit to the project was made on 11.01.2013, and the first coverage event happens 01.16.2014. In contrast to Orekit and Teammates, Dataverse started using PlantUML as their first diagramming tool, so we can get a more complete picture of how the UML diagrams evolve alongside of the project. From the figure, we can see that coverage percentage hits a peak on 08.15.2014. From there, the coverage drops until 09.07.2014, where we then see another spike in coverage percentage. After 09.07.2017, the coverage percentage slowly drops over time because the system grows but the number of covered references stays the same.

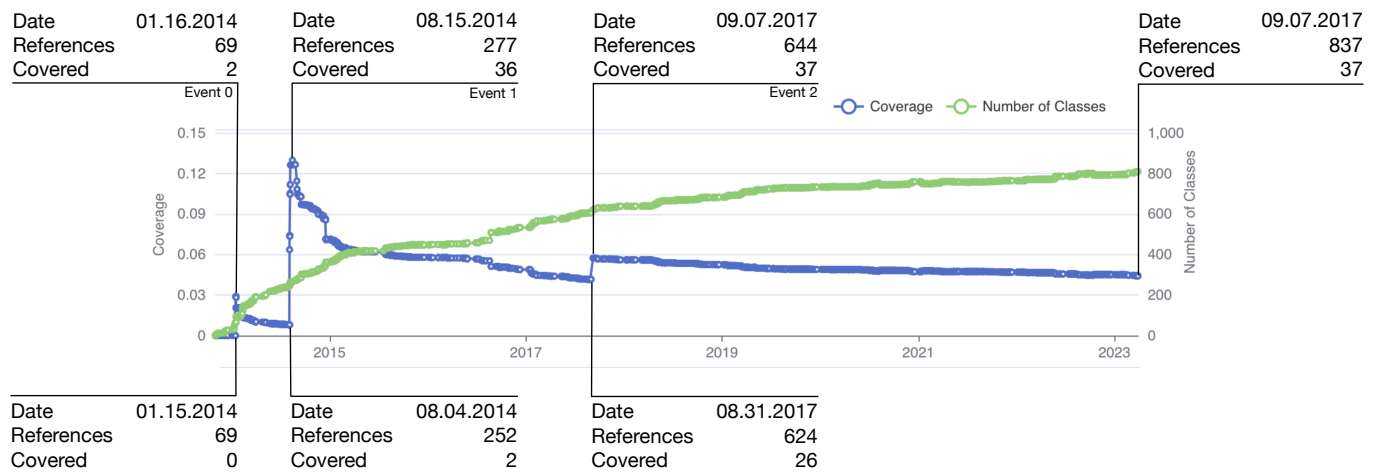


FIGURE 6.23: Dataverse commit coverage history

At each coverage event, we see that the number of references increases along with the number of covered references, so it is not clear whether the events are focused on newly added parts of the system, or already existing parts of the system. It is also unclear whether diagrams were created, and then source code was added to cause the increase in coverage, or whether the source code was implemented, with diagrams added later for documentation.

To disambiguate the two situations, we can refer to Table 6.9. The table shows how the covered references change over time at coverage events. Each item in the table is a covered reference. The **+** symbol signifies that an increase in coverage occurred because a change was made to the source code. The **⊕** symbol signifies that an increase in coverage occurred because a change was made to a diagram. The **-** symbol signifies that a decrease in coverage occurred because a change was made to the source code.

¹⁹<https://github.com/iqss/dataverse>

Event 1		Event 2	
08.15.2014 (+ 25, + 9)	08.31.2017 (- 11)	09.07.2017 (+ 5, + 6)	03.29.2023
AbstractCommand +	AbstractCommand	AbstractCommand	AbstractCommand
AbstractGroup +	-		
AccessRequest +	AccessRequest	AccessRequest	AccessRequest
AllUsers +	AllUsers	AllUsers	AllUsers
ApiKey +	-		
AssignRoleCommand +	AssignRoleCommand	AssignRoleCommand	AssignRoleCommand
AuthenticatedUser +	AuthenticatedUser	AuthenticatedUser	AuthenticatedUser
AuthenticatedUserLookup +	AuthenticatedUserLookup	AuthenticatedUserLookup	AuthenticatedUserLookup
AuthenticatedUsers +	AuthenticatedUsers	AuthenticatedUsers	AuthenticatedUsers
AuthenticationManager +	-		
AuthenticationProvider +	AuthenticationProvider	AuthenticationProvider	AuthenticationProvider
Command +	Command	Command	Command
CreateRoleCommand +	CreateRoleCommand	CreateRoleCommand	CreateRoleCommand
DataFile	DataFile	DataFile	DataFile
		Dataset +	Dataset
Dataverse	Dataverse	Dataverse	Dataverse
DataverseRole +	DataverseRole	DataverseRole	DataverseRole
DataverseUser +	-		
DvObject +	DvObject	DvObject	DvObject
		DvObjectContainer +	DvObjectContainer
ExplicitGroup +	ExplicitGroup	ExplicitGroup	ExplicitGroup
ExplicitGroupCreator +	-		
		Failure +	Failure
Group +	Group	Group	Group
GroupCreator +	-		
GroupException +	GroupException	GroupException	GroupException
GroupRow +	-		
GuestUser +	GuestUser	GuestUser	GuestUser
IpGroup +	IpGroup	IpGroup	IpGroup
LocalAuthenticationProvider +	-		
		Pending +	Pending
Permission +	Permission	Permission	Permission
		PublishDatasetCommand +	PublishDatasetCommand
RoleAssignee +	RoleAssignee	RoleAssignee	RoleAssignee
RoleAssigneeDisplayInfo +	RoleAssigneeDisplayInfo	RoleAssigneeDisplayInfo	RoleAssigneeDisplayInfo
RoleAssignment +	RoleAssignment	RoleAssignment	RoleAssignment
ShibAuthenticationProvider +	ShibAuthenticationProvider	ShibAuthenticationProvider	ShibAuthenticationProvider
ShibGroup +	ShibGroup	ShibGroup	ShibGroup
User +	User	User	User
UserLister +	-		
UserRoleAssignments +	-		
		Workflow +	Workflow
		WorkflowContext +	WorkflowContext
		WorkflowStep +	WorkflowStep
		WorkflowStepData +	WorkflowStepData
		WorkflowStepResult +	WorkflowStepResult
		WorkflowStepSPI +	WorkflowStepSPI

TABLE 6.9: Covered references for Dataverse over time
 (+ diagram increases coverage, + source increases coverage, - source decreases coverage)

At Event 1 in Table 6.9, a majority of the increases in coverage are due to source code changes. This means that the diagrams are not just being created to document already implemented source code, but is also being used to help design the system. On 07.12.2014, a commit was made which added one of the UML diagrams used for design work.²⁰ A subset of the diagram can be seen in Figure 6.24. The classes marked in red are ones that were never added to the source code. Almost a month later, on 08.04.2014, a commit was made which implemented the classes found in the diagram.²¹ The commit shows, for example, that Assignee was never implemented, and was instead implemented as RoleAssignee. The design diagram was used in a flexible way, as a starting point, and the diagram in this case updated to reflect the changes made to the source code. This is why we see a mix of diagrams increasing coverage, and source code increasing coverage in Table 6.9.

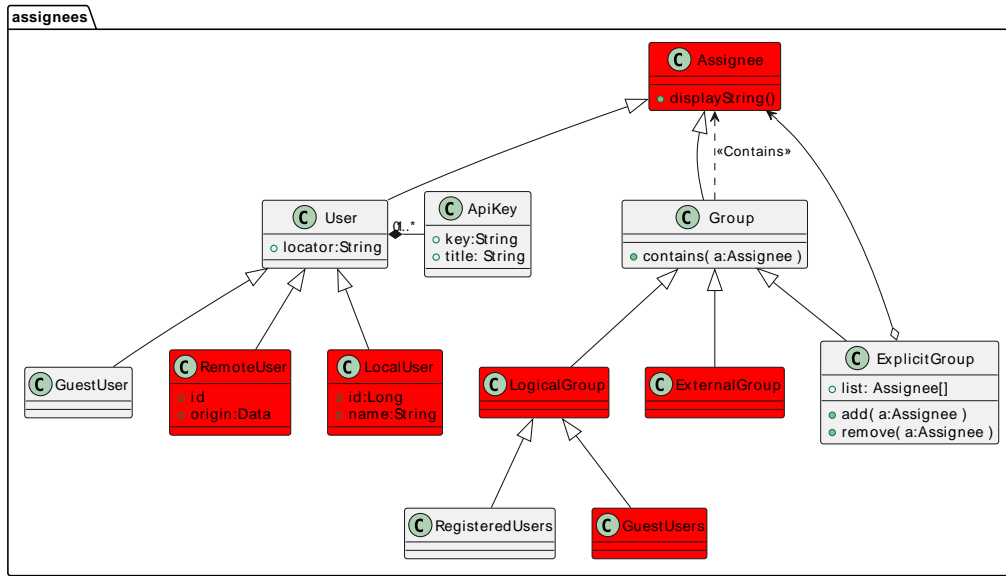


FIGURE 6.24: Dataverse users and groups UML diagram - commit 06.12.2014

Referring back to the beginning of Event 2 in Table 6.9, we see many decreases in coverage due to source changes. The references were removed from source code, but as of writing this thesis almost 6 years after the references were removed, they can still be found in diagrams in the repository.²² Sometimes the diagrams are updated to reflect changes in the source code, and other times they are not.

6.4.2 Documentation Website

Like Orekit and Teammates, Dataverse also hosts a documentation website.²³ The documentation website is generated using Sphinx,²⁴ and many of the UML diagrams can be found scattered throughout the documentation.

²⁰<https://github.com/IQSS/dataverse/commit/36381f9427>

²¹<https://github.com/IQSS/dataverse/commit/936271ba52>

²²<https://github.com/IQSS/dataverse/blob/5f7fd5b421/doc/Architecture/auth-classes.uml>

²³<https://guides.dataverse.org/en/latest/developers/index.html>

²⁴<https://www.sphinx-doc.org/en/master/usage/quickstart.html>

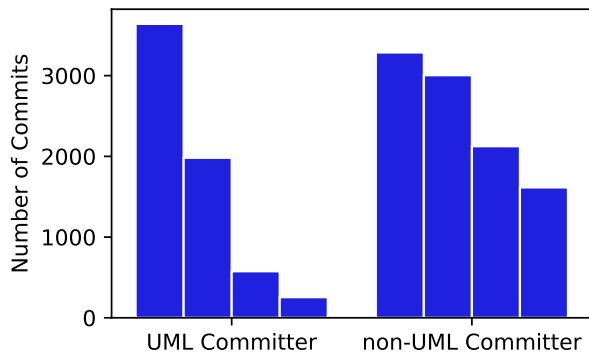
6.4.3 Authors

Table 6.10 shows author statistics for Dataverse. As in the other case studies, the number of UML committers is a small subset of the total number of committers.

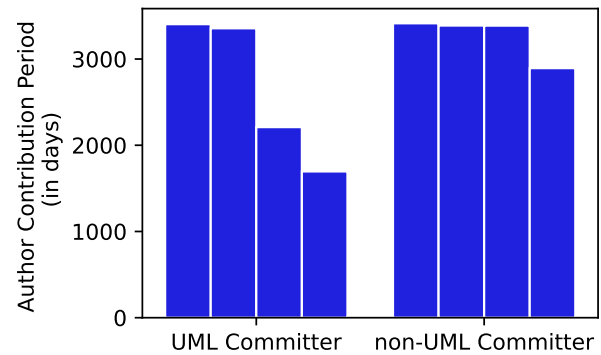
Raw Authors	Authors	Aliases	Authors with Aliases	UML Committers	Non-UML Committers
341	183	222	64	4	179

TABLE 6.10: Dataverse author statistics

Figure 6.25a shows the top 4 authors by number of commits (we choose 4 instead of 5 as there are only 4 UML committers). The top UML committer by number of commits is also the top committer overall in terms of number of commits. The 2nd UML committer is still among the top contributors, but the 3rd and 4th UML committers are not. A similar trend can be seen in the top 4 authors by contribution period, although the difference is less pronounced.



(A) Top 4 authors by number of commits - UML vs non-UML committers



(B) Top 4 authors by contribution period - UML vs non-UML committers

FIGURE 6.25: Comparison of author statistics

6.4.4 Conclusions

In this section, we looked at our final case study, Dataverse. Unlike Orekit and Teammates before, Dataverse used PlantUML as their first UML diagramming tool. We saw that Dataverse used UML diagrams to design parts of the system before implementing them. During implementation, changes to the design were made, and the diagrams were sometimes updated to reflect the changes. We found examples of diagrams references to classes that had not existed for almost 6 years.

Like Orekit and Teammates, Dataverse also hosts a documentation website, where some of the UML diagrams can be found alongside natural language documentation. We also saw that the number of UML committers is a small subset of the total number of committers, and that the top UML committer is also the top committer overall in terms of number of commits.

6.5 Summary

In this chapter, we presented case studies of the usage of UML in open source repositories. We started with Orekit, which has an impressive amount of UML diagram coverage, and at one point in its history had 50% of the system covered by UML diagrams. Even in the Orekit project, where great effort was put into the diagramming (in the latest commit, 55 diagrams with over 304 references covered), parts of the diagrams do not accurately reflect the code. Dataverse also has issues with diagrams that have out of date information, with at least one diagram showing references that have been removed from the source code over 6 years ago. Both Teammates and Orekit both migrated from other tools to PlantUML because it is a lightweight, version control friendly tool that integrates well with their workflow. One downside of these tools is that they do not support traceability and round-trip engineering by default like many traditional UML tools, so greater care needs to be taken to ensure diagrams are up-to-date. This becomes a monumental task as the size of the system grows.

For each system in our case study, the UML diagrams do not exist solely in the repository. Each system has a documentation website which contains the UML diagrams. The diagrams compliment the natural language documentation well, and are a valuable resource for understanding the architecture and design of the systems.

In every case study, the top committer by contribution period, and the top committer by number of commits are also UML committers. In general UML committers are among the most active and longest standing contributors to their respective projects, but we did find some exceptions. UML committers are also a small subset of the total number of contributors in each system.

In each case study, only subsets of the system are diagrammed. The coverage percentage also generally hits a peak and then declines over time, either because the system is no longer being actively diagrammed, or because the system is growing faster than it is diagrammed.

Now that we have answered the posed research questions, and explored the usage of UML in open source repositories, we conclude the thesis with a discussion of the results, threats to validity, and future work in the next chapter.

Chapter 7

Conclusion

"I may not have gone where I intended to go, but I think I have ended up where I needed to be."

— Douglas Adams

7.1 Discussion

In this thesis we have looked at the usage of UML in open source repositories. We saw that the use of UML in open source repositories is not widespread, with only 4.1% of repositories in our dataset containing UML diagrams at some point in their history. We found that the popularity of UML peaked in 2007, and has not reached the same level since. So why is the use of UML not more widespread in open source projects? We believe this may be due to a number of factors.

The first is the learning curve associated with UML. UML is a large and complex specification, whose latest version comes in the form of an over 700-page document.¹ One of UML's main benefits is its ability to communicate design ideas between developers in a clear, concise, and non-ambiguous way. This benefit is lost if developers do not have a good understanding of UML, and the misuse of UML can have a negative impact on a project. The scope of UML has grown to be large and complex, and this complexity has made it harder to understand and use the small subset of the specification that most developers use. Documenting systems is already perceived as a chore by many developers, and adding the complexity of UML to the mix may be enough to turn many developers away.

Another factor is that tooling for UML is often cumbersome and unintuitive to use. First, while exploring the tooling available for creating and maintaining UML diagrams, we found many tools were difficult to install. Many tools were not self-contained, and required the tracking down and installation of many dependencies. Other tools only support Windows, and so required us to set up a Windows virtual machine just to run them. Once they are running, many of them are overly complicated for the needs of the average developer. These types of issues are not problems at large companies, where management can mandate the use of specific tooling. If an open source project decides to use a tool that is difficult to install and use, they may lose potential contributors.

Although we saw a decline in the use of UML in open source projects from 2007 to 2015, we have seen a recent resurgence in the use of UML. This resurgence coincides with the fall in popularity of Eclipse plugins for UML and the rise in popularity of text-based UML diagramming tools. Text-based UML diagramming tools offer a lighter weight alternative to traditional UML diagramming tools that also lend themselves well to version control. We found two examples in our case studies of projects that migrated from Eclipse plugins and generic modeling tools to PlantUML. It could be that many developers who were turned away by the complexity of traditional UML diagramming tools are now willing to try the lighter weight text-based alternatives.

¹<https://www.omg.org/spec/UML/2.5.1/PDF>

The last potential factor that may have caused the decline in UML usage is the rise of agile development methodologies. The agile manifesto was published in 2001, and called for working software over comprehensive documentation. An overall shift towards less documentation may have led to a decline in the use of UML. Although developers may have been quick to jettison documentation, recent research has shown that a majority of agile practitioners still find documentation important, but that there is not enough of it [51].

7.2 Threats to Validity

As part of our data collection, we looked at a subset of the total GitHub repositories. We use the GitHub search API for dataset collection, which analyzes 1.25 million repositories of the 100+ million public repositories on GitHub. We added additional constraints to filter out repositories with less than 100 stars, 2000 commits, and 10 contributors. The exclusion of smaller repositories with smaller communities may hide an interesting population of repositories that use (or do not use) UML.

In addition, we only considered UML diagrams in formats that we could identify and tag. In the context of how popular UML is and the types of projects its found in, the inclusion of additional formats we were not able to tag may have an effect on the final outcome.

When deciding whether a repository is active, and whether a commit was made to UML or not, we did not filter out bot accounts, commits that only update copyright years, and whitespace formatting only commits. These types of commits are common in open source repositories, and may have an effect on the final numbers.

7.3 Future Work

Here we discuss some potential avenues for future work that can build on the work presented in this thesis.

1. **Tagging more UML formats:** We were able to tag UML files in a number of textual formats. We did not tag UML files in image formats (*e.g.*, .png, .jpg, .svg), or in Markdown files (*e.g.*, .md, .rst). Chen *et al.* [12] found that 74% of UML diagrams in GitHub projects are stored in image formats, versus 26% in text-based. They likely under-counted text-based formats as they considered only .xmi and .uml extension, and we have identified many others, but it still shows that many UML diagrams come in image based formats. We also found examples of UML diagrams in Markdown files and xml files, but did not tag them. Given the popularity of MermaidJS and PlantUML as text based UML diagramming tools, and their ability to be embedded and rendered in Markdown files, it seems likely that there are many UML diagrams to be tagged in Markdown files.
2. **Tagging diagram types:** Adding the type of diagram to the tagged UML files would allow for some interesting analysis. For example, we looked at UML diagrams by programming language, and found a significant number of C repositories with UML diagrams. We suspect that the usage of class diagrams is much lower in C than in Java or C++ for instance. Tagging diagram types would allow for these types of comparisons.
3. **Support more Programming Languages:** In our case studies, we looked only at Java projects as our tooling was built around Java. Extending support to more languages would allow for a more complete picture of UML usage in open source projects. It would also allow for the ability to see how UML is used in projects that use multiple languages.
4. **Support more UML Diagram Types:** We focused on class and sequence diagrams in PlantUML. There are many other tools for creating UML diagrams, and many other types of UML diagrams.

Expanding to support more diagram types and tools would allow for the inclusion of more projects in our analysis.

5. **Mining Coverage:** We used Drifter to visualize the coverage of UML diagrams in some projects. Expanding this to run on a larger set of projects would allow for some interesting analysis. Looking at how coverage changes over time
6. **Natural Language Documentation:** The most ubiquitous form of documentation we found while exploring the various repositories in our dataset were natural language ones in Markdown. Even the UML diagrams are often found in Markdown files alongside natural language. Applying the visualizations for UML diagrams could be interesting, especially for the Java2UML graph, which would show links between Markdown files and Java references.
7. **Relaxing Coverage Constraints:** In our coverage analysis, we used strict constraints when making connections between UML diagrams and source code. We did minimal string preprocessing, and we required package structures to be accurate. Relaxing these constraints would allow for more connections to be made, and being able to compare the relaxed versus strict constraints would unveil inaccuracies in diagrams.

7.4 Epilogue

We have taken both a broad and deep look at the usage of UML in open source repositories. Although UML is not widely used, and the usage has generally declined over the years, we have seen a recent resurgence in the use of UML. Will this resurgence continue into the future, or will it be short-lived? The future of UML in open source projects is uncertain, but what is certain is that it faces an uphill battle.

With the advent of lighter weight, version control friendly UML diagramming tools, we believe UML has become more accessible. These tools also come with challenges, both new and old. Like many tools before PlantUML, it does not readily support traceability and round-trip engineering. We have seen that even projects with active diagramming practices have inaccuracies, which can diminish the effectiveness of UML diagrams.

Another challenge that is new to diagramming tools like PlantUML is the limitations in layout. PlantUML works great for small diagrams, but those who have worked with PlantUML likely know the struggle of making moderately sized diagrams look good. After spending hours toiling away to make a diagram look perfect, adding 1 component can cause the whole layout to tangle into an unreadable mess. These types of issues can cause developers to spurn UML, and choose simpler forms of documentation.

It will be interesting to see if despite the current issues with tools like PlantUML, if the use of UML usage in open source continues to trend upwards, and if the current limitations will be fixed.

Appendix A

UML Tools

Tool	Importable Extensions	Exportable Extensions
ArgoUML	.argo, .zargo	.argo, .zargo, .pgml, .xmi, .png, .gif, .svg, .ps, .eps
Astah	.asta	.asta, .png, .jpg, .rtf, .html
Bouml	.prj, .xmi	.prj, .diagram, .html, .svg, .xmi
Cacoo		.png, .svg, .pdf, .ps, .ppt
ConceptDraw	.cddz, .vsd, .vsdx, .vdx, .png, .jpg	.swf, .png, .jpeg, .gif, .bmp, .tiff, .pdf, .svg, .pptx
Creately	.png, .svg, .jpeg, .csv, .pdf	.png, .svg, .jpeg, .csv, .pdf
Dia*	.dia	.dia, .svg
DrawIO	.drawio	.xml, .png, .svg, .html
EdrawMax	.vdx, .vsd, .vsdx, .vsdm, .svg, .eddx, .edx	.eddx, .pdf, .png, .jpg, .vsdx, .html, .docx, .xlsx, .pptx, .svg, .tiff, .pbm, .ps, .eps
EnterpriseArchitect	.qea, .qeax, .feap, .xml, .xmi, .eap, .eapx, .eadb, .csv, .rtf, .pdf, .html	.qea, .qeax, .feap, .xml, .xmi, .eap, .eapx, .eadb, .csv, .rtf, .pdf, .html
GenMyModel	.xmi	.svg, .png, .jpg, .xmi, .docx, .pdf
Gleek.io		.png, .svg, .pdf
Gliffy	.gliffy	.png, .jpg, .svg, .gliffy
Lucidchart	.graffle, .graffle.zip, .vdx, .vsd, .vsdx, .vsdm, .gxml, .gliffy, .xml, .drawio	.pdf, .png, .jpeg, .svg, .csv, .vsdx, .vdx
MagicDraw*	.mdr, .mdzip	.mdr, .mdzip, .xmi, .xml, .ecore, .cmof, .emof
Mermaid	.mmd	
Microsoft Visio	.vsdx, .vsdm, .vssx, .vssm, .vstx, .vstm	.vsdx, .vsdm, .vssx, .vssm, .vstx, .vstm, .pdf, .svg, .png, .emf
ModelIO*	.xmi, .uml	.svg, .png, .gif, .jpeg, .xmi, .uml
Moqups		.png, .pdf, .html
PlantUML		.png, .svg, .eps, .pdf, .vdx, .xmi
Software Ideas Modeler*	.simp	.simp, .jpg, .png, .gif, .bmp, .tiff, .svg, .wmf, .emf, .pdf, .xmi
StarUML	.mdj, .mfj, .uml, .xmi	.mdj, .mfj, .png, .jpg, .svg, .html, .uml, .xmi
TextUML	.tuml	
Umbrello*	.xmi	.xmi, .png, .jpg, .svg, .pdf
UML Designer	.uml	.uml
UMLetino	.uxf	.uxf, .pdf, .png
Umple	.ump	.ump, .tuml, .yuml, .svg, .xmi
Visual Paradigm	.vpp	.png, .html
Yuml	.yuml	.yuml, .png, .svg, .jpg, .pdf

TABLE A.1: Popular UML tools and their supported file extensions (* indicates tool was not successfully installed and run, extensions come from documentation)

Appendix B

UML Examples

Extension	Repo Name	Commit Hash	File Path
.argo	apache/commons-math	05195b77ca	src/mantissa/src/org/spaceroots/mantissa/linalg/design/linalg.argo
.asta	opendds/opendds	4eac1ec78e	docs/design/UML/OpenDDS_UML_sketches.asta
.bmp	orbiternassp/NASSP	6ae708ff94	TVD2MXF/TVD2MXF/lib/SimMetricsv1.5/SimMetrics/SimilarityClasses/edit%20distance/Edit%20Distance.bmp
.cmof	bpmn-io/bpmn-js	b7733572a0	resources/bpmn/cmof/BPMN20.cmoF
.dia	adoptium/temurin	8b5106821a	docs/images/sequence.dia
.diagram	eclipse/sumo	61b58fbf78	docs/modules/sumo/128048.diagram
.docx	gbif/ipt	7e0b210c8b	gbif-ipt-docs/downloads/ipt-architecture_1.1.docx
.drawio	apache/camel-k	2903656f49	docs/diagrams-source/camel-k-state-machine-build.drawio
.eap	cqframework/clinical_quality_language	f1865e8e02	Src/cql-lm/uml/elm.eap
.ecore	b2ihealthcare/snowowl	bd6b51ebdc	snomed/-com.b2international.snowowl.snomed.etl/model/generated/Etl.core
.emf	sommer/veins	135d963b30	doc/interfaces/PhyLayer/tex/images/emf/modelling/Air-Frame_members.emf
.eps	cpc/openasip	33e56b42f9	tce/doc/specs/design/UniversalMachine/eps/classdiagram.eps
.gif	apache/commons-dbcP	eac7b3de69	src/site/resources/images/uml/ConnectionFactory.gif
.gliffy	akeneo/pim-communit	09bf43ee51	src/Oro/Bundle/DataGridBundle/Resources/doc/backend/diagrams/datagrid_base_uml.gliffy
.graffle	cashmusic/platform	4f1191a7f8	vendor/swiftmailer/swiftmailer/doc/uml/Encoders.graffle
.html	sgothel/jogl	6e54fba3bb	doc/uml/html/index.html
.iuml	chef/automate	1ca3deb611	dev-docs/diagrams/authz-sequence.iuml
.jpeg	iluwatar/java-design-patterns	8524c75ba6	double-checked-locking/etc/ double_checked_locking.jpeg
.jpg	iluwatar/java-design-patterns	932836f68b	servant/src/etc/mediator.jpg
.json	ls1intum/artemis	9dc0d4fca5	src/test/resources/de/tum/in/www1/artemis/service/-compass/umlmodel/deployment/deploymentModel2.json
.md	aelfproject/aelf	ba3daca8a3	dev/docs/development/main-sequence.md
.mdj	btccom/btcpool-abandoned	249df9d268	docs/Eth/uml.mdj
.mdzip	jsettlers/settlers-remake	41caf8362d	doc/uml/networklib.mdzip
.mmd	apache/arrow	dd52b384cb	docs/source/format/FlightSql/CommandPrepared-StatementQuery.mmd
.odg	cs-si/orekit	321233898f	src/site/resources/images/antenna-frames.odg
.pdf	nasa/fprime	053fe23ada	docs/Architecture/FPrimeSoftwareArchitecture.pdf

.plantuml	apache/syncope	4814ebf2b8	src/main/asciidoc/images/sra-request.plantuml
.platuml	kubernetes-sigs/cluster-api	7303333fa3	docs/proposals/images/cluster-spec-crds/figure2.platuml
.png	nasa/fprime	a2744296b7	docs/Architecture/ComponentTree.png
.ppt	docgroup/ace_tao	fd904693d4	DAnCE/docs/OMG-DnC-Tutorial.ppt
.pptx	nasa/fprime	053fe23ada	docs/Architecture/FPrimeSoftwareArchitecture.pptx
.prj	eclipse/sumo	61b58bf78	docs/modules/sumo/sumo.prj
.ps	siconos/siconos	50e2df5543	Docs/Dev/Kernel_DDD/figure/class_diag_DDD_DS_XML.ps
.pu	mudita/muditaos	e433e8dd03	module-cellular/modem/doc/scripts/class_channel.pu
.puml	cs-si/orekit	5ef673c5da	src/site/resources/images/attitude-class-diagram.puml
.rst	bareos/bareos	f4826a68ff	docs/manuals/source/DeveloperGuide/jobexec.rst
.session	sgothel/jogl	fa00349682	doc/uml/jogl/69.session
.svg	adoptium/temurin-build	8b5106821a	docs/images/sequence.svg
.txt	staged-project/udisks	05caa69296	doc/uml/classes.txt
.ucls	apache/cloudstack	893a88d225	engine/storage/storage.ucls
.uml	opendds/opendds	4eac1ec78e	docs/design/UML/history/DDS.uml
.umlclass	activiti/activiti	d571cbc794	modules/activiti-engine/doc/persistence.model.umlclass
.umlprofile	opendds/opendds	27ad9eab0f	tools/modeling/plugins/org.opendds.modeling.resources/profiles/OpenDDS.umlprofile
.ump	umple/umple	e492ca969d	Umplificator/UmplifiedProjects/weka-umplified-0/src-umple/FastVector.ump
.unt	opendds/opendds	4eac1ec78e	docs/design/UML/history/Analysis.unt
.uxf	armmbed/mbed-os	c2d849133f	features/storage/FEA-TURE_STORAGE/cfstore/doc/design/umlet/configuration_store_hld.uxf
.vdx	resiprocate/resiprocate	9e3b6b6b2b	resip/dum/doc/dum-fsms1.vdx
.vpp	alexo/wro4j	ce7993a1d4	wro4j-examples/wro4j-demo/doc/WebResourceOptimizer.vpp
.vsd	eclipse-aspectj/aspectj	329a415e48	docs/developer/compiler-weaver/dev-guide-diagrams.vsd
.vsdm	azuread/microsoft-authentication-library-for-js	fca271e588	extensions/docs/diagrams/visio/msal-node-extensions-persistence.vsdm
.vsdx	azure/iotedge	1a35477d74	doc/resources/EdgeHubDiagrams.vsdx
.wmf	activeloopai/deeplake	674196a82c	deeplake/tests/dummy_data/images/crown.wmf
.xmi	kdd/calligra	723cc438e5	kexi/migration/xbase/doc/design.xmi
.xml	iluwatar/java-design-patterns	932836f68b	servant/src/etc/mediator.xml
.xpm	gnome/dia	1707d4d23e	objects/UML/umlclass.xpm
.yuml	kratosmultiphysics/kratos	2eb0483902	applications/ShapeOptimizationApplication/optimization_process.yuml
.zargo	alexo/wro4j	ce7993a1d4	wro4j-examples/wro4j-demo/doc/Diagram.zargo
.zuml	apache/openoffice	cdf0e10c4e	main/svx/doc/UML/-grid_control_implementation.zuml

TABLE B.1: Examples of UML diagrams for each extension
https://github.com/<repo_name>/tree/<commit_hash>/<file_path>

Appendix C

UML Counter-Examples

Extension	Repo Name	Commit Hash	File Path
.argo	ALIGN-analoglayout/ ALIGN-public	db92f3903a	Cktgen/Argo/route.argo
.bmp	andreikop/enki	d573cda3c4	win/portrait-logo.bmp
.diagram	arvidn/libtorrent	9a31c45d3c	docs/img/read_disk_buffers.diagram
.drawio	snailclimb/javaguide	3e8b402cc7	docs/high-performance/images/message-queue/message-queue-pub-sub-model.drawio
.eap	FreeRADIUS/ freeradius-server	f1ee539f43	share/dictionary/freeradius/dictionary.freeradius.internal.eap
.eddx	rdkmaster/jigsaw	fbaa40ea14	docs/implement-sudoku-puzzle-with-table-and-its-render-system/event-graph.eddx
.edx	kissyteam/kissy	9831969f64	src/dom/sub-modules/selector/docs/linkedlist.edx
.emf	apache/openoffice	863a930eea	main/extras/source/gallery/diagrams/Section-Pasters02-Blue.emf
.gif	3liz/lizmap-web-client	f7d333c7ce	lizmap/www/assets/css/images/download_layer.gif
.gliffy	GoogleContainerTools/skaffold	d5db3264d4	docs-v2/diagrams/skaffold.gliffy
.graffle	actor-framework/ actor-framework	6c08fe2dec	doc/graffle/mailbox.graffle
.html	qt/qtwebengine	8787c53297	examples/webenginewidgets/printme/data/index.html#L4
.jpeg	RestComm/Restcomm-Connect	c67f142339	restcomm/restcomm.docs/sources-asciidoc/src/main/asciidoc/tutorials/images/ussdpull.jpeg
.jpg	1024pix/pix	aafb074d8f	orga/public/coming-soon.jpg
.md	AElfProject/AElf	ba3daca8a3	README.md
.mdr	Gwion/Gwion	382f82d117	docs/02_Reference/00_Types/03_Typedefs.mdr
.mmd	github-linguist/linguist	916bd8f3df	samples/Mermaid/gitgraph.mmd
.patch	coolsnowwolf/lede	404209f6c4	target/linux/uml/patches-5.4/102-pseudo-random-mac.patch
.pbm	a1ive/grub	d52357bd38	grub-core/bits-deps/python/Lib/test/imghdrdata/python.pbm
.pdf	OLNetworkCommunity/libra	b0ce13c7d6	documentation/contributing/individual-cla.pdf
.plantuml	testingisdocumenting/znai	afff408226	znai-docs/znai/visuals/gantt.plantuml
.png	benawad/dogehouse	794eba762f	kibbeh/src/img/dogehouse.png

.pptx	apache/poi	5c29cfc058	test-data/slideshow/2411-Performance_Up.pptx
.prj	3liz/lizmap-web-client	f7d333c7ce	extra-modules/lizmapdemo/qgis-projects/demoqgis_intranet/data/VilleMTP_MTP_FilaireVoies_2011.prj
.pu	ax-molengine/ax-mol	062b510c38	tests/cpp-tests/Content/Particle3D/scripts/UVAnimation.pu
.puml	apache/fineract	40eb547a5c	fineract-doc/src/docs/en/diagrams/release-schedule.puml
.rst	apache/arrow	caa94a446a	docs/source/format/ADBC.rst
.session	FreeRTOS/FreeRTOS	0d95aca202	FreeRTOS/Demo/T-HEAD_CB2201_CDK/RTOSDemo_CDK/.cdk/RTOSDemo_CDK.session
.svg	benawad/dogehouse	794eba762f	.redesign-assets/dogehouse_logo.svg
.txt	stored-project/udisks	c640ad72a2	modules/lsm/TEST_NOTE_lsm.txt
.uml	van-hoefm/fragat-tacks	abf9b9bd8b	tests/hwsim/vm/kernel-config.uml
.umlaut	DragonFlyBSD/DragonFly-BSD	d83cfc8d85	share/me/test/test.umlaut
.ump	ufoaiorg/ufoai	d9aa2b00ab	base/maps/construction.ump
.unt	ProteoWizard/pwiz	bc8a3ada92	pwiz_tools/Bumbershoot/bumberdash/Tests/Data/BrukerTest.d/Sample_1-A%2C1_01_985.unt
.vacuumlo	postgres/postgres	6e414a171e	contrib/vacuumlo/README.vacuumlo
.vdx	angular/angular.js	7e5a12e2c1	images/docs/Diagrams.vdx
.vpp	istoreos/istoreos	e106f25ee7	target/linux/omap35xx/gumstix/defconfig.vpp
.vsdm	apache/tika	48132cfd7	tika-parsers/tika-parsers-standard/tika-parsers-standard-modules/tika-parser-microsoft-module/src/test/resources/test-documents/testVISIO.vsdm
.vsdx	andreikop/enki	d573cda3c4	tests/test_plugins/preview_sync_source_above_target.vsdx
.vssm	apache/tika	48132cfd7	tika-parsers/tika-parsers-standard/tika-parsers-standard-modules/tika-parser-microsoft-module/src/test/resources/test-documents/testVISIO.vssm
.vssx	apache/tika	48132cfd7	tika-parsers/tika-parsers-standard/tika-parsers-standard-modules/tika-parser-microsoft-module/src/test/resources/test-documents/testVISIO.vssx
.vstm	apache/tika	48132cfd7	tika-parsers/tika-parsers-standard/tika-parsers-standard-modules/tika-parser-microsoft-module/src/test/resources/test-documents/testVISIO.vstm
.vstx	apache/tika	48132cfd7	tika-parsers/tika-parsers-standard/tika-parsers-standard-modules/tika-parser-microsoft-module/src/test/resources/test-documents/testVISIO.vstx
.xmi	act-boy168/YDWE	90f0a56f5a	ThirdParty/RadGameTools/MSS/6.1a/Examples/media/demo.xmi
.xml	benawad/dogehouse	794eba762f	pilaf/android/app/src/main/res/values/colors.xml
.xpm	BestImageViewer/geeqie	5a961050ab	src/icons/gq-marker.xpm

TABLE C.1: Counter-Examples of UML diagrams for each extension
https://github.com/<repo_name>/tree/<commit_hash>/<file_path>

Appendix D

UML Extension Tagging

In this appendix we provide a brief description of each of the UML extensions we tagged, and the method we used to tag them. There are three main types of files we tagged: text files, zip folder, and gzip files. For text files, we used regex to search for characteristics of the file that would indicate it was a UML diagram. For gzip we unzipped the file and then used regex to search for characteristics. For zip files, we unzipped the folder, and searched the files for discernible characteristics. For some file types we were not able to find any discerning characteristics, but the number of files was few. In those cases we manually tagged the files.

D.1 .argo and .zargo

.argo and .zargo are the two extensions we see used for the ArgoUML¹ diagramming tool. .argo is an XML file that contains various pointers to the PGML and XMI files that make up the project. Since ArgoUML is a UML specific diagramming tool, we tagged all .argo file containing the "<argo>" tag as a UML diagram. The .zargo file extension instead is a zip archive containing the .xmi, .argo and .pgml files that contain all of the diagram information. For .zargo files, we tagged all zips that contained .argo, .xmi and .pgml files.

D.2 .asta

.asta is an extension used by another UML specific diagramming tool called AstaUML². It is a zip file that unzips into a java serialization data object. In the end, there were only a few files to check, so instead of deserializing the data object and looking for characteristics, we manually opened each in asta to ensure it whereas a proper project file. Each file was, so we tagged all .asta files as UML diagrams.

D.3 .cmof

The .cmof extension is based on the MOF standard, which is a metamodel used to describe other metamodels.³ It is used to describe the UML metamodel, and has a subset of the notations used in UML. For that reason we considered each .cmof file to be a UML diagram and tagged them as such. For the same reason as .asta above, there were only four total CMOF files, so we checked them manually to ensure they were all the same format.

¹<https://argouml-tigris-org.github.io/tigris/argouml/>

²<https://astah.net/products/astah-uml/>

³<https://www.omg.org/spec/MOF/>

D.4 .dia

The .dia extension is used by the Dia⁴ diagramming tool. Unlike AstaUML and ArgoUML it is not UML specific, and can be used for other types of diagramming. We found .dia extensions with diagrams saved in 2 ways, as an XML file and as a gzipped XML file. To tag each .dia file, we checked if it was a gzip file, and if it was then we unzipped it. After unzipping, the checks are the same for both the XML and unzipped XML files. We search the XML file using the following regex to determine if it is a UML diagram or not.

```
"UML[[:space:]-]*\*(Class|Note|Dependency|Realizes|Generalization|
  Association|Implements|Constraint|SmallPackage|LargePackage|
  Actor|Usecase|Lifeline|Object|Message|Component|Component Feature|
  Node|Classicon|State Term|State|Activity|Branch|Fork|Transition\)"
```

This covers the parts of the UML diagram that we saw when creating an example UML diagram in Dia. Below is an example from one of the files in our dataset that the regex finds.

```
<dia:object type="UML - Class" version="0" id="00">
  <dia:attribute name="obj_pos"> <dia:point val="49.6866,16.777"/> </dia:attribute>
  <dia:attribute name="name"> <dia:string>#TestCase#</dia:string> </dia:attribute>
</dia:object>
```

D.5 .diagram

The .diagram extension is used by a few different tools. A majority of the files we found with the .diagram extension were a companion to the .prj file used by the BOUML⁵ diagramming tool. To tag the BOUML files, we used the regex below.

```
"classcanvas|classinstance|activitynodecanvas"
```

D.6 .ecore

The .ecore extension is used by the Eclipse Modeling Framework (EMF)⁶. It uses a subset of the UML notation to allow for the generation of code from UML models. To tag .ecore files, we used the following regex.

```
"EClass|EPackage"
```

D.7 .gliffy

Gliffy⁷ is a general purpose diagramming tool that can be used to create UML diagrams. Although it is capable of making UML diagrams outside of class diagrams, we found in the cases where it used UML entities in other diagrams (like a UML actor for instance), it was used within a diagram that was not a UML diagram. Those with class entities were consistently UML diagrams following proper notation. Given that, to tag .gliffy files, we used the following regex.

```
"com\.gliffy\.shape\.uml\.uml_v1\.default\.(class|simple_class|package)"
```

⁴<https://wiki.gnome.org/Apps/Dia>

⁵<http://bouml.fr/>

⁶<https://www.eclipse.org/modeling/emf/>

⁷<https://www.gliffy.com/>

D.8 *.iuml, .puml, .plantuml, .platuml*

The *.iuml* extension is used by PlantUML. In our dataset, we see it is typically used for as an include file for diagram styling, actors for various use cases, re-usable classes and sequences, etc.. Given that, we tag the *.iuml* files the same as we tag the *.puml*, *.plantuml*, and *.platuml* files. For the tagging criteria, we skipped the types of diagrams that PlantUML supports that are not UML diagrams that we found in our dataset, including archimate and c4 diagrams, gantt charts, mind maps, network diagrams, and wireframes. We do tag files that are purely for styling as changes in the styling would still update their respective UML diagram. Tagging these we use a different method than in the previous cases where we looked using inclusive regexes. In the case of the *.iuml*, *.puml*, *.plantuml* and *.platuml* extensions, we did not find any files that were not PlantUML files, so the following regex we use to tag them is an exclusive regex.

```
"@(startgantt|startmindmap|startsalt)|nwdiag|include.*c4|c4-plantuml"
```

D.9 *.mdj*

The *.mdj* extension is a json file used by the StarUML⁸ diagramming tool to store UML model data. We searched for files to tag using the following regex.

```
"UMLModel"
```

D.10 *.mdzip*

The *.mdzip* extension is a zip file used by the MagicDraw⁹ diagramming tool which zips up the content of an mdxml file. The mdxml file is actually an xml file supporting the xmi standard. To tag *.mdzip* files, we unzip them and use the following regex to check if it is a valid magicdraw file.

```
"uml:Class"
```

D.11 *.mmd*

Mermaid¹⁰ is another diagramming tool similar to PlantUML that can be used to generate diagrams from text. Rendering mermaid graphs is supported in many flavors of markdown, including GitHub, which means mermaid diagrams in READMEs are directly rendered when viewing on GitHub. The extension used for Mermaid files is *.mmd*. Mermaid supports 3 types of diagrams: sequence diagrams, class diagrams, and state machine diagrams. In mermaid, the type of diagram is specified before the rest of the markdown. We search using the following regex to tag mermaid files (noting that *zenuml* and *sequenceDiagram* are 2 different syntaxes for sequence diagrams in mermaid).

```
"sequenceDiagram|classDiagram|stateDiagram"
```

D.12 *.prj*

The *.prj* extension is a very generic extension and used in a variety of ways. Since we found this extension in a folder with UML, and found that it was BOUML file, we searched for the characteristics found in

⁸<http://staruml.io/>

⁹<https://www.nomagic.com/products/magicdraw>

¹⁰<https://mermaid.js.org/>

BOUML project files. To tag .prj files, we found BOUML files have a comment at the top for the start of the settings for the project. We use the following regex to tag using this settings comment.

```
"// class settings"
```

D.13 .pu

We found the .pu extension used for two different purposes. A bit more than half of the files were files used as a scripting tool for 3D graphics. The rest were plantuml files. Given that, we use the same exclusive regex mentioned in the .iuml section above, but we also add an explicit check for the @startuml tag.

D.14 .session

As you might expect, we found files with the .session extension are typically used for session information for a variety of cases (gnome sessions, db sessions, etc.). The BOUML tool uses the .session extension as another companion file to the .prj file, along with the .diagram extension mentioned earlier. To tag BOUML .session files we use the following regex.

```
"(class|sequence)diagram"
```

D.15 .ucls

This was one of our UML specific tags, and is created and used by a tool called UML ObjectAid Explorer, an eclipse plugin for creating UML diagrams. It is a diagram in XML format, and we were able to use the below regex to tag it. This regex tagged all files in our dataset with the .ucls extension.

```
"<class-diagram"
```

D.16 .uml

We found the .uml extension was used by a variety of UML tools. The main 3 UML formats with the .uml extension in the dataset are PlantUML, XMI, and XML. We first eliminated PlantUML files that were not UML diagrams using the same exclusion regex mentioned in the .iuml, .puml, .plantuml, and .platuml section above. After excluding any non UML PlantUML files, we then tagged the remaining .uml files as UML diagrams using the following regex.

```
"uml:model\\|uml:package\\|uml:profile\\|umlproject\\|
schemas.microsoft.com/dsltools/UmlModelLibrary\\|<Category>Methods</Category>\\
|<Category>Fields</Category>\\|<ID>java</ID>\\
|skinparam\\|participant\\|type\\s*=\\s*[\\\"']umlclass\\w*[\\\"']"
```

D.17 .umlclass and .umlprofile

The UML class files represented XML files implementing Eclipse UML 2¹¹. The UML profile files provide style for the same UML class files. The following regex was used to tag these files.

```
"type\\s*=\\s*[\\\"'](umlclass|umlprofile)[\\\"']"
```

¹¹<https://projects.eclipse.org/projects/modeling.mdt.uml2>

D.18 .ump

The .ump extension is mainly used by the Umple model-based programming language.¹² We also found it as an extension used for storing video game maps. We did not find any distinguishable characteristics, but we found only 2 projects with the .ump extension. One was the Umple project itself, and the other was a video game project. We tagged all files with the .ump extension as UML diagrams.

D.19 .uxf

The .uxf extension is used by the UMLet¹³ tool which can be used as a standalone tool, an eclipse plugin or a vscode plugin to model UML diagrams. The diagram data is stored in xml format. The following regex was used to tag these files.

```
"program="umlet"\|program="umletino"\|umlet_diagram"
```

D.20 .vpp

The .vpp extension is used by the Visual Paradigm¹⁴ tool which is used for enterprise architecture modeling, including many features beyond just UML diagramming. We found this extension in two main formats, ZIP and SQLite. For zip we unzipped the folder and used the following regex to see if it contained any of the supported UML diagram types.

```
"ClassDiagram|ComponentDiagram|InteractionDiagram|UseCaseDiagram"
```

For SQLite, we ran the following query, and applied the regex above to the results.

```
sqlite3 $file "SELECT * FROM DIAGRAM"
```

D.21 .xmi

XMI is a standard for exchanging model data between tools. It is not specific to UML diagrams, but is often used for this case. To find the XMI files that were UML related, we looked for the UML standard reference in the XMI header, looked for a reference to a uml tag, or looked to see if the metamodel being described had the name UML. We used the following regex to tag these files.

```
"org\.omg\/standards\/uml|schema.omg.org\/spec\/uml|
  <uml:(model|class|attribute|namespace|interface|method)|
  <xmi\.metamodel.*name\s*=\s*['\"]UML['\"]"
```

D.22 .yuml

We did not find any discernible characteristics for the .yuml extension. There are only 10 files in our dataset with the .yuml extension, so we tagged them manually.

¹²<https://www.umple.org/>

¹³<https://www.umlet.com/>

¹⁴<https://www.visual-paradigm.com/>

D.23 .zuml

The .zuml extension is used for zipped UML project files. Each zip contained a .project file, and an .xmi file written out from Netbeans XMI writer¹⁵. We used the following regex on the unzipped .xmi file.

```
"UML:GraphNode"
```

Appendix E

Queries

E.1 Extension Exploration

Find all file extensions that contain string

```
SELECT extension, COUNT(*) AS extension_count
FROM file_extensions
WHERE extension ILIKE '%<string>h%'
GROUP BY extension
HAVING COUNT(*) > 1;
```

Output: A list of all file extensions that contain the specified string and the number of repositories that contain each extension.

Find all file extensions found in */uml/* paths

Notes: This is a fairly long running command and takes about 20 minutes to run on the server we are using. It is still much faster than trying to grep for the same data.

```
SELECT LOWER(SUBSTRING(path FROM '\.([\^.]*)$')) AS extension,
COUNT(DISTINCT repo_id) AS repo_count
FROM file_paths WHERE path ILIKE '%/uml/%'
GROUP BY LOWER(SUBSTRING(path FROM '\.([\^.]*)$'))
HAVING COUNT(DISTINCT repo_id) > 1;
```

Output: A list of all file extensions found in */uml/* paths and the number of repositories that contain files with that extension.

Find all repositories where extension exists

```
SELECT r.name, LOWER(fe.extension)
FROM repositories AS r
JOIN file_extensions AS fe
ON r.id = fe.repo_id
WHERE LOWER(fe.extension)
IN ('.<ext>')
```

Output: A list of repositories that the extension exists in.

Find all commits and files where extension exists for given repository

```
SELECT fp.path, fp.commit_hash FROM file_paths AS fp
JOIN repositories AS r ON r.id = fp.repo_id
WHERE r.name = '<repo_name>' AND fp.path ILIKE '%.<extension_name>';
```

Output: A list of file paths and commit hashes for files with given extension.

Find all commits and files in uml paths for given extension

```
SELECT r.name, fp.path FROM repositories AS r
JOIN file_paths AS fp
ON fp.repo_id = r.id
WHERE fp.path ILIKE '%/uml%/%.<ext>';
```

Output: A list of file paths and commit hashes for files with given extension.

E.2 Commit Exploration

Retrieve commit extension statistics

```
SELECT r.name AS repo_name, c.commit_hash, ces.name, ces.category, ces.count
FROM commits c
JOIN commit_extension_stats ces ON c.id = ces.commit_id
JOIN repositories r ON c.repo_id = r.id
WHERE r.name = '<repo_name>' and c.commit_hash = '<commit_hash>';
```

Output: A list of extensions, their categories, and number of files touched in commit.

Appendix F

Author Anti-Aliasing Algorithm Details

F.1 Names match

Algorithm 1: names_match(author_a, author_b)

Data: author_a, author_b

Result: Whether names match

```

  /* Check author has a name, and is not unknown, anonymous, anon, none */
1 if not has_valid_name(author_a) or not has_valid_name(author_b) then
2   | return False;
3 end

4 name_a, name_b ← split(author_a, author_b); // Split into first and last name
  /* remove punctuation, accents, and emojis. all lower case, convert to utf-8 */
5 name_a, name_b ← normalize(name_a, name_b);

  /* Check if names match, or if names match but first and last names are swapped */
6 if name_a.last is not None and name_b.last is not None then
7   | if (name_a.first == name_b.first and name_a.last == name_b.last)
8     | or (name_a.first == name_b.last and name_a.last == name_b.first) then
9       | return True;
10    | end
11 end

  /* Repeat next check swapping name_a and name_b */
12 if name_a.last is not None and name_b.last is None then
13   | if is_squished_name(name_a, name_b.first) or is_user_name(name_a, name_b) then
14     | return True
15   | end
16 end

  /* If neither has last name, check for first name match, eliminating short names */
17 if name_a.last is None and name_b.last is None and len(name_a.first) > 6 and len(name_b.first) > 6 then
18   | if name_a.first == name_b.first then
19     | return True;
20   | end
21 end
22 return False;

```

F.2 Emails match

Algorithm 2: emails_match(author_a, author_b)

Data: author_a, author_b

Result: Whether emails match

```

1 if not has_valid_email(author_a) or not has_valid_email(author_b) then
2 |   return False;
3 end

4 if author_a.email == author_b.email then
5 |   return True;
6 end

7 username_a ← extract_username(author_a);
8 username_b ← extract_username(author_b);
9 if len(username_a) > 4 and len(username_b) > 4 and username_b == username_a then
10 |  return True;
11 end
12 return False;

```

F.3 Name matches email

Algorithm 3: name_matches_email(author_a, author_b)

Data: author_a, author_b

Result: Whether name matches email

```

1 name_a ← split(author_a); // Split into first and last name
2 name_a ← normalize(author_a); // remove punctuation, accents, etc.
3 email_b ← normalize(author_b); // remove punctuation, accents, etc.

4 if name.first in email and name.last in email then
5 |   return True;
6 end

7 potential_users ← [{"name.first[0]}{name.last}", "{name.last[0]}{name.first}"];
8 if any(username in email for username in potential_usernames) then
9 |   return True;
10 end
11 return False;

```

Appendix G

List of UML Repositories

Repository	UML Extension List
0xd34df00d/leechcraft	{.yuml}
1024pix/pix	{.puml}
18f/tock	{.puml}
389ds/389-ds-base	{.dia}
4minitz/4minitz	{.puml}
accord-net/framework	{.uml}
activiti/activiti	{.umlclass, .uml}
adorsys/open-banking-gateway	{.puml}
adorsys/xs2a	{.puml}
aelfproject/aelf	{.puml}
agoric/agoric-sdk	{.puml}
aidenlab/juicebox	{.uml}
akeneo/pim-community-dev	{.gliffy}
alexo/wro4j	{.zargo, .vpp}
alphagov/whitehall	{.puml}
alsa-project/alsa-lib	{.puml}
altinity/clickhouse-operator	{.xmi}
andbible/and-bible	{.puml}
angular/angular	{.puml}
anotheria/moskito	{.uml}
aosp-mirror/platform_system_core	{.dia}
apache/apex-core	{.uml}
apache/arrow	{.mmd}
apache/atlas	{.uml}
apache/attic-apex-core	{.uml}
apache/attic-apex-malhar	{.uml, .zargo}
apache/camel	{.ecore}
apache/camel-spring-boot	{.ecore}
apache/celix	{.pu, .puml}
apache/cloudstack	{.ucls}
apache/commons-math	{.pgml, .xmi, .puml, .argo}
apache/fineract	{.puml}

apache/gobblin	{.ucls}
apache/incubator-gobblin	{.ucls}
apache/incubator-streampipes	{.ucls}
apache/isis	{.ucls, .uxf, .puml, .dia}
apache/jackrabbit	{.uxf}
apache/jackrabbit-oak	{.uml, .uxf, .puml}
apache/james-project	{.uml}
apache/lucene	{.uml, .uxf, .puml}
apache/lucene-solr	{.uml, .uxf, .puml}
apache/netbeans	{.pgml, .zargo, .dia, .argo, .xmi}
apache/openoffice	{.zuml, .xmi}
apache/poi	{.pgml, .zargo, .xmi, .argo}
apache/solr	{.uml, .uxf, .puml}
apache/syncope	{.plantuml}
apache/trafficserver	{.plantuml, .uml}
apache/velocity-engine	{.pgml, .xmi, .argo}
arangodb/arangodb	{.zargo}
arcemu/arcemu	{.uml}
arm-software/arm-trusted-firmware	{.puml, .dia}
armmbed/mbed-os	{.uxf}
asynkron/protoactor-go	{.puml}
atk4/atk4	{.xmi}
audiveris/audiveris	{.uxf}
aurelia/aurelia	{.mmd}
automatic/simplenote-android	{.umlclass, .uml}
automatic/wp-calypto	{.puml}
autotest/autotest	{.xmi}
aws/amazon-freertos	{.pu}
aws/eks-anywhere	{.plantuml, .uml, .puml}
awslabs/djl	{.puml}
azure/azure-iot-sdk-csharp	{.puml}
azure/iotedge	{.plantuml}
b2ihealthcare/snow-owl	{.ecore}
ballerina-platform/ballerina-lang	{.plantuml}
bareos/bareos	{.plantuml, .xmi, .puml}
bentoboxworld/bentobox	{.puml}
betonquest/betonquest	{.puml}
betterthantomorrow/calva	{.plantuml}
bh107/bohrium	{.dia}
biolink/biolink-model	{.yuml}
bmwcarit/joynr	{.uxf, .puml}
bndtools/bnd	{.ecore}
bonigarcia/webdrivermanager	{.ucls}
boost-ext/di	{.uml}
bpmn-io/bpmn-js	{.cmof}

brightid/brightid	{.mmd}
broadleafcommerce/broadleafcommerce	{.umlclass, .uml}
btccom/btcpool-abandoned	{.mdj}
build-trust/ockam	{.mmd, .puml}
bundy-dns/bundy	{.dia}
burtonator/polar-bookshelf	{.puml}
c3nav/c3nav	{.puml}
caliopen/caliopen	{.uml, .puml}
camunda/camunda-bpm-platform	{.umlclass, .uml}
carrot2/carrot2	{.zargo}
catmaid/catmaid	{.dia}
cdapio/cdap	{.ucls}
cdk/cdk	{.pgml, .xmi, .argo}
cellularprivacy/android-imsi-catcher-detector	{.uml}
cgal/cgal	{.mdj}
chainsql/chainsql	{.uml, .pu, .puml}
chamilo/chamilo-lms	{.dia}
chef/automate	{.iuml, .puml}
chef/chef-workstation	{.puml}
cloudsimplus/cloudsimplus	{.mdj}
clougence/hasor	{.ucls}
clrfund/monorepo	{.mmd}
cocoalumberjack/cocoalumberjack	{.mdj}
code-dot-org/code-dot-org	{.puml}
comit-network/comit-rs	{.puml}
comit-network/xmr-btc-swap	{.puml}
common-workflow-language/cwltool	{.dia}
conda/conda	{.puml}
consulo/consulo	{.uml}
contiv/vpp	{.puml}
controlsystemstudio/cs-studio	{.umlclass, .ucls, .zargo, .uml}
coreboot/coreboot	{.plantuml}
coreemu/core	{.plantuml}
corfudb/corfudb	{.puml}
cosmicpython/book	{.puml}
cosmos/cosmos-sdk	{.puml}
counterfactual/monorepo	{.mmd}
cpc/openasip	{.xmi, .dia}
cqframework/clinical_quality_language	{.xmi}
cs-si/orekit	{.uml, .puml}
cyberbotics/webots	{.dia}
d-ronin/dronin	{.dia}
dayatang/dddlib	{.uml}
dcs4cop/xcube	{.puml}
de-labtory/it-chain	{.mdj}

deegree/deegree3	{.yuml}
deepjavalibrary/djl	{.puml}
deeppavlov/deeppavlov	{.uml}
demoiselle/framework	{.ucls}
dgta-team/dgtal	{.dia}
diracgrid/dirac	{.uml}
distributedcollective/sovryn-smart-contracts	{.puml}
dlsc-software-consulting-gmbh/workbenchfx	{.asta}
docgroup/ace_tao	{.uml}
doctrine/annotations	{.xmi, .dia}
doctrine/common	{.xmi, .dia}
doctrine/dbal	{.xmi}
doctrine/orm	{.xmi}
dolibarr/dolibarr	{.uml, .dia}
duniter/duniter	{.pu}
eclipse-archived/smarthome	{.ucls}
eclipse-cdt/cdt	{.ecore}
eclipse-iceoryx/iceoryx	{.puml}
eclipse/birt	{.ecore}
eclipse/buildship	{.ecore}
eclipse/capella	{.ecore}
eclipse/gef	{.ucls, .uml, .ecore}
eclipse/jetty.project	{.puml}
eclipse/org.aspectj	{.zargo}
eclipse/sumo	{.zuml, .prj, .diagram, .uxf}
eclipse/vorto	{.ecore}
eclipse/xacc	{.plantuml, .uml}
eclipse/xtext-core	{.ecore}
edgetx/edgetx	{.dia}
edmcouncil/fibo	{.mdzip}
egroupware/egroupware	{.zargo}
ehcache/ehcache3	{.puml}
ehsan/ogre	{.prj, .session, .diagram}
elastic/cloud-on-k8s	{.puml}
elastic/rally	{.puml}
elektrainitiative/libelektra	{.mmd, .xmi}
embox/embox	{.ecore}
enalean/tuleap	{.zargo}
enmasseproject/enmasse	{.puml}
enthought/mayavi	{.xmi}
eprosima/fast-dds	{.plantuml}
erlang/erlide_eclipse	{.uxf}
estatio/estatio	{.ucls, .uml}
eugenp/tutorials	{.puml}
exelearning/iteexe	{.xmi}

ezsystems/ezpublish-legacy	{.xmi}
f4exb/sdrangel	{.mdj}
featureide/featureide	{.zargo, .ecore}
fedict/eid-mw	{.uml, .puml}
fenics/dolfinx	{.dia}
finmath/finmath-lib	{.ucls}
finos/waltz	{.puml}
firewalld/firewalld	{.dia}
flowable/flowable-engine	{.umlclass, .uml}
flutter/packages	{.puml}
fonttools/fonttools	{.puml}
forgeessentials/forgeessentials	{.uxf}
freecad/freecad	{.uml}
freeseer/freeseer	{.dia}
freesurfer/freesurfer	{.xmi}
freeyourgadget/gadgetbridge	{.xmi}
friendupcloud/friendup	{.dia}
galoymoney/galoy	{.iuml}
gama-platform/gama	{.uml, .ecore}
gamefoundry/bsf	{.uml}
gaphor/gaphor	{.xmi}
garux/netradiant-custom	{.pgml, .zargo, .xmi, .argo}
geonetwork/core-geonetwork	{.uml}
geosx/geos	{.mmd, .plantuml}
geotools/geotools	{.ucls, .ecore}
github/linguist	{.mmd, .iuml, .puml}
gluufederation/oxauth	{.vpp}
gnome/pygobject	{.dia}
godlikepanos/anki-3d-engine	{.xmi}
googlefonts/glyphslib	{.uml}
googleforgames/agones	{.puml}
gradle/gradle	{.puml}
gradle/kotlin-dsl-samples	{.puml}
grafana/loki	{.plantuml}
guardianproject/chatsecureandroid	{.ucls}
gunet/openeaclass	{.plantuml}
h2oai/h2o-2	{.uml}
habitat-sh/habitat	{.iuml}
halestudio/hale	{.uml, .xmi}
halo-dev/halo	{.puml}
hapifhir/hapi-fhir	{.puml}
hashicorp/consul	{.mmd}
hawkular/hawkular-metrics	{.uml}
hdfgroup/hdf5	{.plantuml}
helidon-io/helidon	{.uml}

heremaps/gluecodium	{.dia}
hibernate/hibernate-orm	{.zargo}
hibernate/hibernate-search	{.dia}
highperformancedecoder/minsky	{.xmi}
hipparchus-math/hipparchus	{.pgml, .xmi, .puml, .argo}
hpi-information-systems/metanome	{.ucls, .uml}
htmlunit/htmlunit	{.uxf}
hurence/logisland	{.plantuml, .mdj}
hyperledger-archives/composer	{.uml}
hyperledger-labs/blockchain-carbon-accounting	{.puml}
hyperledger/aries-cloudagent-python	{.uml, .puml}
hyperledger/aries-rfcs	{.puml}
hyperledger/cacti	{.puml}
hyperledger/indy-node	{.puml}
hyperledger/indy-plenum	{.puml}
hyperledger/indy-sdk	{.puml}
hzi-braunschweig/sormas-project	{.ucls}
ibinti/bugvm	{.uml}
idempiere/idempiere	{.ecore}
igvteam/igv	{.uml}
iluwatar/java-design-patterns	{.ucls, .puml}
imglib/imglib2	{.dia}
inception-project/inception	{.ucls}
indeedeng/proctor	{.puml}
infiniteorg/infinite	{.puml}
input-output-hk/daedalus	{.uml}
insolar/insolar	{.uml}
instructure/canvas-lms	{.plantuml}
intermine/intermine	{.zargo, .xmi}
internetarchive/heritrix3	{.zargo}
inveniosoftware/invenio	{.plantuml}
iotaledger/wasp	{.mdzip, .uxf, .puml}
iqss/dataverse	{.uml, .puml}
isc-projects/kea	{.uml, .dia}
ivmartel/dwv	{.puml}
jabref/jabref	{.uml}
jamesagnew/hapi-fhir	{.puml}
java110/microcommunity	{.puml}
javalite/javalite	{.uxf}
javaparser/javaparser	{.puml}
jenetics/jenetics	{.uml, .xmi}
jenkinsci/warnings-ng-plugin	{.puml}
jetbrains/intellij-sdk-docs	{.puml}
jpgcri/gcam-core	{.zuml, .zargo, .xmi}
jiangwenyuan/nuster	{.dia}

jmetal/jmetal	{.ucls}
jmri/jmri	{.puml}
jplag/jplag	{.ecore}
jsettlers/settlers-remake	{.mdzip, .zargo}
jumpmind/symmetric-ds	{.dia}
kaltura/mwembed	{.puml}
kbengine/kbengine	{.dia}
kde/amarok	{.xmi}
kde/calligra	{.xmi}
kde/kdevelop	{.xmi}
kde/krita	{.xmi}
kde/marble	{.uml, .xmi}
kedro-org/kedro	{.puml}
kerbalism/kerbalism	{.plantuml}
kermitt2/grobid	{.zargo}
khronosgroup/gltf-blender-io	{.uxf}
kiegroup/jbpm	{.uml, .ecore}
kiegroup/jbpm-designer	{.uml}
kiegroup/kie-tools	{.ecore}
kiegroup/kogito-runtimes	{.uml, .dia, .ecore}
kiegroup/optaplanner	{.asta, .dia}
knowagelabs/knowage-server	{.ecore}
koding/koding	{.uml}
kratosmultiphysics/kratos	{.yuml}
kubernetes-sigs/cluster-api	{.plantuml, .platuml, .puml}
kubernetes-sigs/cluster-api-provider-aws	{.plantuml, .platuml}
kubernetes-sigs/cluster-api-provider-azure	{.plantuml}
kubernetes/enhancements	{.puml}
kubernetes/test-infra	{.mmd}
kubevirt/kubevirt	{.plantuml}
kulshexhar/ts-jest	{.puml}
lf-lang/lingua-franca	{.ecore}
librepilot/librepilot	{.dia}
libreplan/libreplan	{.xmi}
libretime/libretime	{.zuml}
liferay/liferay-docs	{.xmi}
ligato/vpp-agent	{.puml}
linutronix/elbe	{.dia}
locationtech/udig-platform	{.ucls, .ecore}
logisim-evolution/logisim-evolution	{.ecore}
ls1intum/artemis	{.puml}
manoelcampos/cloudsim-plus	{.mdj}
manoelcampos/cloudsimplus	{.mdj}
mantidproject/mantid	{.uxf, .puml}
maplibre/maplibre-gl-js	{.plantuml}

mathics3/mathics-core	{.dia}
matsim-org/matsim-libs	{.ucls}
mbeddr/mbeddr.core	{.puml, .ecore}
mercedes-benz/sechub	{.plantuml, .puml}
mermaid-js/mermaid	{.mmd}
microsoft/appcenter-sdk-dotnet	{.uml}
microsoft/mwt-ds	{.uml}
microsoft/vscode-dev-containers	{.plantuml}
mimblewimble/grin	{.puml}
minvws/nl-covid19-notification-app-ios	{.mdj}
mitk/mitk	{.xmi}
mixcore/mix.core	{.mmd}
mlreef/mlreef	{.uml, .puml}
mne-tools/mne-cpp	{.mdj}
modeshape/modeshape	{.ecore}
mongodb/mongo-java-driver	{.uml}
mozilla/fx-private-relay	{.mmd}
mudita/muditaos	{.uml, .mdj, .puml, .pu}
mulesoft/mule	{.ecore}
mullvad/mullvadvpn-app	{.puml}
mupen64plus-ae/mupen64plus-ae	{.ucls}
myrobotlab/myrobotlab	{.ucls, .uml}
named-data/ndn-cxx	{.vpp}
nasa/europa	{.dia}
nasa/fprime	{.mdzip, .puml}
near/nearcore	{.puml}
neo4j/neo4j	{.ucls}
neorazorx/facturascripts	{.mdj}
nest/nest-simulator	{.uxf}
netdata/netdata	{.puml}
netflix/genie	{.ucls}
netty/netty	{.uml}
neuralensemble/python-neo	{.dia}
nextcloud/deck	{.yuml}
nfs-ganesha/nfs-ganesha	{.dia}
nhibernate/nhibernate-core	{.zargo}
nightingale-media-player/nightingale-hacking	{.uml}
nihlus/remora.discord	{.puml}
nitorcreations/nflow	{.plantuml}
noear/solon	{.puml}
noobaa/noobaa-core	{.puml}
nosqlbench/nosqlbench	{.puml}
nrel/energyplus	{.plantuml}
nrfconnect/sdk-nrf	{.uml}
nuitka/nuitka	{.plantuml}

nuxeo/nuxeo	{.puml}
objectcomputing/opensdds	{.asta, .ecore, .uml, .umlprofile, .dia, .xmi}
ockam-network/ockam	{.mmd, .puml}
odamex/odamex	{.asta}
odoo/documentation	{.dia}
odpi/egeria	{.puml}
ofs/opae-sdk	{.mmd}
ogrecave/ogre	{.puml, .prj, .session, .diagram}
ogrecave/ogre-next	{.prj, .session, .diagram}
omegat-org/omegat	{.vpp}
omnetpp/omnetpp	{.ecore}
onebusaway/onebusaway-iphone	{.mdj}
onlyliuxin/coding2017	{.gliffy}
ontop/ontop	{.ucls}
opae/opae-sdk	{.mmd}
openapitools/openapi-generator	{.plantuml}
openconnect/ocserv	{.dia}
opendatacube/datacube-core	{.plantuml, .puml}
opensdds/opensdds	{.asta, .ecore, .uml, .umlprofile, .dia, .xmi}
openems/openems	{.mmd}
openhab/openhab-addons	{.mdj}
openhab/openhab1-addons	{.ecore}
openl-tablets/openl-tablets	{.pgml, .zargo, .ecore, .argo, .xmi}
openmdao/openmdao-framework	{.dia}
opennebula/one	{.xmi}
openolat/openolat	{.uml, .zargo}
openrocket/openrocket	{.umlclass, .uml, .uxf}
opensearch-project/opensearch-dashboards	{.puml}
openshift/enhancements	{.plantuml}
opensourcebim/bimserver	{.umlclass, .uml, .ecore}
openssl/openssl	{.plantuml}
openstack/tacker	{.pu}
opensuse/libzypp	{.zargo, .dia}
openthread/openthread	{.uml}
opentripplanner/opentripplanner	{.uxf}
openturns/openturns	{.xmi}
opm/resinsight	{.plantuml}
oracle/helidon	{.uml}
orangehrm/orangehrm	{.xmi}
oroinc/platform	{.gliffy}
os-guild/tweek	{.puml}
osgeo/grass	{.dia}
osgeo/proj	{.xmi}
osm-search/nominatim	{.plantuml}
otsaloma/gaupol	{.dia}

owncast/owncast	{.plantuml}
owtf/owtf	{.plantuml}
pagopa/io-app	{.puml}
pbiggar/phc	{.zargo}
pcsx2/pcsx2	{.puml}
pdal/pdal	{.uml}
pdepend/pdepend	{.zargo}
pentaho/mondrian	{.zargo}
pentaho/pentaho-kettle	{.uxf}
photoprism/photoprism	{.mmd}
phpdocumentor/phpdocumentor	{.puml}
pietermartin/sqlg	{.uml}
pimcore/pimcore	{.uml}
pingcap/tiflow	{.puml}
pixelated/pixelated-user-agent	{.puml}
pjsip/pjproject	{.dia}
powershell/powershell	{.puml}
pretix/pretix	{.puml}
progit/progit	{.dia}
programmevitam/vitam	{.dia}
projectcontour/contour	{.uml}
provenance-emu/provenance	{.mdj}
publicmapping/districtbuilder-classic	{.dia}
pulumipulumi/pulumipulumi	{.uml}
puppetlabs/puppet	{.ecore}
pycqa/pylint	{.mmd, .puml}
pyglet/pyglet	{.dia}
pythonarcade/arcade	{.puml}
pytorch/serve	{.puml}
qcodes/qcodes	{.puml}
qgis/qgis	{.xmi}
qmcpack/qmcpack	{.xmi}
qos-ch/logback	{.uml}
ranger/ranger	{.prj, .session, .diagram}
rbfx/rbfx	{.dia}
rdflib/rdflib	{.plantuml}
rdkmaster/jigsaw	{.eddx}
realm/realm-dotnet	{.uml}
remkop/picocli	{.uxf}
remora/remora.discord	{.puml}
restcomm/restcomm-connect	{.puml, .dia}
restcomm/sip-servlets	{.plantuml, .uml}
restlet/restlet-framework-java	{.uml, .ecore}
revive-adserver/revive-adserver	{.zuml, .xmi}
rexray/rexray	{.puml}

riot-os/riot	{.puml}
ripple/rippled	{.uml, .pu, .puml}
ripple/xrpl-dev-portal	{.uxf}
rolisteam/rolisteam	{.puml}
ros-planning/moveit	{.uxf}
ros-planning/moveit2	{.uxf}
ros-planning/navigation2	{.pu}
rose-compiler/rose	{.zargo}
rpm-software-management/libdnf	{.uxf}
rptools/maptool	{.puml}
rstudio/rstudio	{.puml}
samsung/gearvrf	{.puml}
sap/cloud-security-xsuaa-integration	{.puml}
sarl/sarl	{.asta, .ecore}
savoirfairelinux/jami-client-android	{.dia}
savoirfairelinux/jami-daemon	{.xmi, .dia}
savoirfairelinux/ring-client-android	{.dia}
savoirfairelinux/ring-daemon	{.xmi}
scada-lts/scada-lts	{.puml}
sdwebimage/sdwebimage	{.mdj}
seccubus/seccubus	{.uml}
securecodebox/securecodebox	{.uxf, .puml}
segs/segs	{.vpp}
servicetitan/stl.fusion	{.mmd}
sfttech/openage	{.uxf}
sgothel/jogl	{.prj, .session, .diagram}
shopware/platform	{.puml}
siconos/siconos	{.xmi}
simgrid/simgrid	{.uml, .zargo}
simpletest/simpletest	{.dia}
skycoin/skycoin	{.puml}
slicer/slicer	{.xmi}
slicer/slicergitsvnarchive	{.vpp, .xmi}
smallrye/smallrye-mutiny	{.puml}
snyk/driftctl	{.puml}
sofa-framework/sofa	{.prj, .session, .diagram}
soluto/tweek	{.puml}
sourcefabric/airtime	{.zuml}
sourcefabric/newscoop	{.xmi, .dia}
sphinx-doc/sphinx	{.puml}
starviewer-medical/starviewer	{.xmi}
storj/storj	{.plantuml}
supertux/supertux	{.dia}
symfony/symfony	{.puml}
syndesisio/syndesis	{.plantuml}

talend/tdi-studio-se	{.ecore}
talend/ui	{.puml}
teammates/teammates	{.puml}
testingisdocumenting/znai	{.mmd, .plantuml}
theforeman/foreman	{.mmd}
thesandboxgame/sandbox-smart-contracts	{.mmd, .puml}
thirdweb-dev/typescript-sdk	{.mdj}
thsmi/sieve	{.dia}
thymeleaf/thymeleaf	{.zargo}
tng/archunit	{.puml}
tony19/logback-android	{.uml}
tribe29/checkmk	{.uml, .iuml, .puml}
triplea-game/triplea	{.puml}
twisted/twisted	{.dia}
ufoai/ufoai	{.zargo}
ultimaker/uranium	{.xmi}
ultimate-pa/ultimate	{.ucls}
ultrastar-deluxe/play	{.plantuml}
umlet/umlet	{.uxf}
umple/umple	{.ucls, .ump, .uxf, .ecore, .uml, .xmi}
ungleich/cdist	{.dia}
unidata/netcdf-java	{.mdzip}
uninett/nav	{.dia}
uportal-project/uportal	{.mmd}
urho3d/urho3d	{.dia}
usetheource/rascal	{.zargo}
ushahidi/platform	{.uml}
uwescience/myria	{.uml}
uyuni-project/uyuni	{.dia}
valora-inc/wallet	{.plantuml}
vcmi/vcmi	{.prj, .diagram}
vectordotdev/vector	{.pu}
vendure-ecommerce/vendure	{.puml}
veracruz-project/veracruz	{.puml}
videojs/http-streaming	{.plantuml, .puml}
viking-gps/viking	{.dia}
vnpv/vnpy	{.puml}
vocadb/vocadb	{.dia}
volttron/volttron	{.puml}
vprover/vampire	{.xmi}
vsrinivas/fuchsia	{.dia}
w3f/polka-dot-spec	{.puml}
wala/wala	{.ecore}
wargus/stratagus	{.xmi}
wazuh/wazuh	{.plantuml, .puml}

webanno/webanno	{.ucls}
werf/werf	{.puml}
wesnoth/wesnoth	{.xmi, .ecore}
wicketstuff/core	{.uml}
wikieducationfoundation/wikiedudashboard	{.zargo}
wix/detox	{.mmd, .uml}
worldbrain/memex	{.puml}
wro4j/wro4j	{.zargo, .vpp}
wurstscript/wurstscript	{.ecore}
wwbn/avideo	{.plantuml}
xfce-mirror/thunar	{.xmi}
xrplf/rippled	{.uml, .pu, .puml}
xrplf/xrpl-dev-portal	{.uxf}
yakindu/statecharts	{.umlclass, .uml, .ecore}
yamcs/yamcs	{.dia}
yubico/java-webauthn-server	{.plantuml}
yuliskov/smarttubelegacy	{.uml}
yuliskov/smartyoutubetv	{.uml}
zanata/zanata-platform	{.zuml}
zetaops/ulakbus	{.puml}
zim-desktop-wiki/zim-desktop-wiki	{.puml}
zom/zom-android-xmpp	{.ucls}
zycgit/hasor	{.ucls}

TABLE G.1: List of 550 UML repositories used in the evaluation.

Bibliography

- [1] A. Agrawal, A. Rahman, R. Krishna, A. Sobran, and T. Menzies. We don't need another hero? the impact of "heroes" on software development. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '18*, page 245–253, New York, NY, USA, 2018. Association for Computing Machinery.
- [2] A. Aimar. Introduction to software documentation. Technical report, CERN, 1998.
- [3] S. S. Alhir. *UML in a nutshell: a desktop quick reference*. " O'Reilly Media, Inc.", 1998.
- [4] N. Ali, S. Baker, R. O'Crowley, S. Herold, and J. Buckley. Architecture consistency: State of the practice, challenges and requirements. *Empirical Software Engineering*, 23(1):224–258, 2018.
- [5] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.
- [6] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6):365–381, 2006.
- [7] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *Proceedings of the 32nd International Conference on Software Engineering*, volume 1, pages 95–104. IEEE, 2010.
- [8] A. Baabad, H. B. Zulzalil, S. Hassan, and S. B. Baharom. Characterizing the architectural erosion metrics: A systematic mapping study. *IEEE Access*, 10:22915–22940, 2022.
- [9] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR '06*, page 137–143, New York, NY, USA, 2006. Association for Computing Machinery.
- [10] G. Booch. *Object-Oriented Analysis and Design with Applications (2nd Ed.)*. Benjamin-Cummings Publishing Co., Inc., USA, 1993.
- [11] L. F. Capretz. A brief history of the object-oriented approach. *SIGSOFT Softw. Eng. Notes*, 28(2):6, mar 2003.
- [12] F. Chen, L. Zhang, X. Lian, and N. Niu. Automatically recognizing the semantic elements from UML class diagram images. *Journal of Systems and Software*, 193:111431, 2022.
- [13] J. Cleland-Huang, O. C. Z. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman. Software traceability: Trends and future directions. In *Proceedings of the Future of Software Engineering*, page 55–69. Association for Computing Machinery, 2014.
- [14] O. Dabic, E. Aghajani, and G. Bavota. Sampling projects in github for MSR studies. In *Proceedings of the International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021.

- [15] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.
- [16] W. J. Dzidek, E. Arisholm, and L. C. Briand. A realistic empirical evaluation of the costs and benefits of UML in software maintenance. *IEEE Transactions on Software Engineering*, 34(3):407–432, 2008.
- [17] A. M. Fernández-Sáez, D. Caivano, M. Genero, and M. R. Chaudron. On the use of UML documentation in software maintenance: Results from a survey in industry. In *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 292–301. IEEE, 2015.
- [18] M. Fowler, J. Highsmith, et al. The agile manifesto. *Software development*, 9(8):28–35, 2001.
- [19] M. Goeminne and T. Mens. A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8):971–986, 2013. Special section on software evolution, adaptability, and maintenance & Special section on the Brazilian Symposium on Programming Languages.
- [20] C. Gravino, G. Scanniello, and G. Tortora. Source-code comprehension tasks supported by UML design models: Results from a controlled experiment and a differentiated replication. *Journal of Visual Languages and Computing*, 28:23–38, 2015.
- [21] I. Hadar and O. Hazzan. On the contribution of UML diagrams to software system comprehension. *Journal of Object Technology*, 3(1):143–156, 2004.
- [22] P. Hruby. Specification of workflow management systems with uml. In *Proceedings of the OOPSLA Workshop on Implementation and Application of Object-oriented Workflow Management Systems*, volume 2, 1998.
- [23] I. Jacobson. *Object-oriented software engineering: a use case driven approach*. Pearson Education India, 1993.
- [24] R. Jolak, M. Savary-Leblanc, M. Dalibor, J. Vincur, R. Hebig, X. L. Pallec, M. Chaudron, S. Gérard, I. Polasek, and A. Wortmann. The influence of software design representation on the design communication of teams with diverse personalities. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, pages 255–265, 2022.
- [25] A. Kalnins and V. Vitolins. Use of UML and model transformations for workflow process definitions. *arXiv preprint cs/0607044*, 2006.
- [26] B. Karasneh and M. R. Chaudron. Img2uml: A system for extracting UML models from images. In *Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications*, pages 134–137, 2013.
- [27] N. J. Kipyegen and W. P. K. Korir. Importance of software documentation. *International Journal of Computer Science Issues (IJCSI)*, 10(5):223–228, 09 2013.
- [28] H. Koç, A. M. Erdoğan, Y. Barjakly, and S. Peker. UML diagrams in software engineering research: a systematic literature review. In *Proceedings of the 7th International Management Information Systems Conference*, volume 74, page 13. MDPI, 2021.
- [29] E. Kouters, B. Vasilescu, A. Serebrenik, and M. G. Van Den Brand. Who’s who in gnome: Using lsa to merge software repository identities. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 592–595. IEEE, 2012.
- [30] H. Krasner. The cost of poor software quality in the us. *Consortium for Information and Software Quality*, 2020.

- [31] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE software*, 20(6):35–39, 2003.
- [32] R. Li, P. Liang, M. Soliman, and P. Avgeriou. Understanding architecture erosion: The practitioners' perceptive. In *Proceedings of the IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 311–322, 2021.
- [33] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang. Traceability transformed: Generating more accurate links with pre-trained bert models. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pages 324–335. IEEE, 2021.
- [34] A. Mahmoud, N. Niu, and S. Xu. A semantic relatedness approach for traceability link recovery. In *Proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 183–192, 2012.
- [35] T. G. Moreira, M. A. Wehrmeister, C. E. Pereira, J.-F. Petin, and E. Levrat. Automatic code generation for embedded systems: From UML specifications to vhdl code. In *Proceedings of the 8th IEEE International Conference on Industrial Informatics*, pages 1085–1090. IEEE, 2010.
- [36] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28, 1995.
- [37] P. Mäder, P. L. Jones, Y. Zhang, and J. Cleland-Huang. Strategic traceability for safety-critical projects. *IEEE software*, 30(3):58–66, 2013.
- [38] M. Ozkaya. Are the UML modelling tools powerful enough for practitioners? a literature review. *IET Software*, 13(5):338–354, 2019.
- [39] M. Ozkaya and F. Erata. A survey on the practical use of UML for different software architecture viewpoints. *Information and Software Technology*, 121:106275, 2020.
- [40] D. L. Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering*, pages 125–148. Springer, 2011.
- [41] M. Petre. UML in practice. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 722–731, 2013.
- [42] J. Rosik, A. Le Gear, J. Buckley, M. A. Babar, and D. Connolly. Assessing architectural drift in commercial software development: a case study. *Software: Practice and Experience*, 41(1):63–86, 2011.
- [43] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. E. Lorensen, et al. *Object-oriented modeling and design*, volume 199. Prentice-hall Englewood Cliffs, NJ, 1991.
- [44] B. Rumpe. Executable modeling with UML. a vision or a nightmare? *arXiv preprint arXiv:1409.6597*, 2014.
- [45] G. Scanniello, C. Gravino, M. Genero, J. A. Cruz-Lemus, and G. Tortora. On the impact of UML analysis models on source-code comprehensibility and modifiability. *ACM Trans. Softw. Eng. Methodol.*, 23(2), apr 2014.
- [46] G. Scanniello, C. Gravino, M. Genero, J. A. Cruz-Lemus, G. Tortora, M. Risi, and G. Doderio. Do software models based on the UML aid in source-code comprehensibility? aggregating evidence from 12 controlled experiments. *Empirical software engineering*, 23(5):2695–2733, 2018.

- [47] G. Scanniello, C. Gravino, and G. Tortora. Investigating the role of UML in the software modeling and maintenance—a preliminary industrial survey. In *International Conference on Enterprise Information Systems*, volume 2, pages 141–148. SCITEPRESS, 2010.
- [48] B. Selic. UML 2: A model-driven development tool. *IBM Systems Journal*, 45(3):607–620, 2006.
- [49] H. H. Smith. On tool selection for illustrating the use of UML in system development. *J. Comput. Sci. Coll.*, 19(5):53–63, may 2004.
- [50] M. Sousa and H. Moreira. A survey on the software maintenance process. In *Proceedings of the International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 265–274, 1998.
- [51] C. J. Stettina and W. Heijstek. Necessary and neglected? an empirical study of internal documentation in agile software development teams. In *Proceedings of the 29th ACM International Conference on Design of Communication, SIGDOC '11*, page 159–166, New York, NY, USA, 2011. Association for Computing Machinery.
- [52] E. Tryggeseth. Report from an experiment: Impact of documentation on maintenance. *Empirical Software Engineering*, 2(2):201–207, 1997.
- [53] A. Watson. Visual modelling: past, present and future. *White paper UML Resource Page*, 28, 2008.
- [54] N. Wirth. A brief history of software engineering. *IEEE Annals of the History of Computing*, 30(3):32–39, 2008.