Is Self-Admitted Technical Debt a Good Indicator of Architectural Divergences?

Giancarlo Sierra*, Ahmad Tahmid, Emad Shihab*, Nikolaos Tsantalis
Department of Computer Science and Software Engineering
*Data-driven Analysis of Software (DAS) Lab
Concordia University, Montreal, Canada
{g_sierr, a_tahmid, eshihab, tsantalis}@encs.concordia.ca

***Abstract*—Large software systems tend to be highly complex and often contain unaddressed issues that evolve from bad design practices or architectural implementations that drift from definition. These design flaws can originate from quick fixes, hacks or shortcuts to a solution, hence they can be seen as Technical Debt. Recently, new work has focused on studying source code comments that indicate Technical Debt, i.e., Self-Admitted Technical Debt (SATD). However, it is not known if addressing information left by developers in the form source code comments can give insight about the design flaws in a system and have the potential to provide fixes for bad architectural implementations. This paper investigates the possibility of using SATD comments to resolve architectural divergences. We leverage a data set of previously classified SATD comments to trace them to the architectural divergences of a large open source system, namely ArgoUML. We extract its conceptual and concrete architectures based on available design documentation and source code, and contrast both to expose divergences, trace them to SATD comments, and investigate their resolution. We found 7 high-level divergences in ArgoUML and 22 others among its subsystems, observing that merely 4 out of 29 (14%) divergences can be directly traced to SATD. Although using SATD as an indicator of architectural divergences is viable, the effort of doing so is time-intensive, and in general, will not lend to a significant reduction of architectural flaws in a software system.**

***Index Terms*—Self-Admitted Technical Debt, Software Re-Engineering, Software Architecture, Architecture Recovery**

I. Introduction

As part of the evolution of large software systems, they tend to become increasingly complex and often contain unaddressed issues that are rooted in bad design practices or architecture implementations that drift from definition. This trailing problem can be seen as Technical Debt [4], a commonly used metaphor in software engineering nowadays. TD refers to the debt caused by quick fixes, shortcuts or non-optimal implementations introduced into a system to benefit in the short-term, but that has to be paid off in the long-term with an increased cost. More in the context of this paper, we observe TD as those implementations that are not aligned with its design or that violate coding practices. This debt grows over time and has to be paid eventually by ever-increasing maintenance tasks.

Recently, research on technical debt has taken a new direction by studying debt found in source code comments. Potdar and Shihab [18] studied this phenomenon and referred to it as *Self-Admitted Technical Debt (SATD)* since debt comments are consciously-introduced by developers. As found by Maldonado et al. [12], SATD can be of different types (test, defect, design, documentation or test debt). Different approaches have emerged for the detection and classification of SATD [18], [12], [13], [8]. A common finding indicates that one of the most occurring types of SATD in software systems is design debt [18], [12], [13]. However, the usefulness and practicality of addressing design SATD has not been investigated yet. Continuing the research on SATD, we are motivated to study the connection between design debt found in source code comments, and the design flaws of a system in the form of architectural divergences. In this paper, we study if SATD comments can be traced to architectural divergences in a software system, and investigate if using these debt comments can lead to the removal of system divergences. Throughout this paper, we refer to **architectural divergences** as any dependency between subsystem components that is not part of a system's conceptual architecture.

Following a recent study on SATD by Maldonado et al. [13], we have chosen to work with the open source project ArgoUML, since it was found to be rich in technical debt comments [12], and have the most occurrences of design SATD among 10 large open source systems. We use the data set made publicly available by Maldonado et al., which contains classified SATD comments to trace debt instances to architectural divergences in ArgoUML. To aid in the tracing process, we built a SATD identification tool that statically analyzes source code comments with a pattern-based approach derived from previous work by Potdar and Shihab [18] and Maldonado et al. [13]. We investigate the files that create each divergence and check: a) if the comments inside them match with SATD comments we identified; and b) if the divergences can be resolved by addressing SATD comments.

As preliminary work for our investigation, we identify the architectural divergences that have occurred over time in ArgoUML. We achieve this by obtaining and contrasting the concrete and conceptual architectures of the system based on available documentation and the source code of the project. We replicate techniques from previous work [3], [15], [7]; where alongside the goals and findings of each study, their authors were able to expose many unexpected and missing dependencies between subsystems in an architecture (architectural divergences). Bowman and Brewster [3] pointed out that many of the unexpected dependencies they found in Linux could not be explained by rationale and their occurrence is due to developers bad practices or expediency. Since identifying SATD exposes problems acknowledged by developers, we investigate if SATD can be a good indicator of said unwanted links between subsystems.

We examine the following research questions for our study:

- **RQ1 - How many architectural divergences can be traced to SATD comments?** With an available data set of design SATD and a set of identified architectural divergences in ArgoUML, we investigate if it is possible to relate introduced debt (in the form of SATD comments) to architectural divergences. We want to investigate the amount of divergences that are introduced along with SATD. If architectural divergences with a few file dependencies are found, looking at SATD comments could provide insight to determine if the divergence was introduced as design debt by developers expediency or error during development.
- **RQ2 - How many architectural divergences can be fixed by addressing SATD?** Not all the architectural divergences are the same in nature, some might have been introduced by error and occur because of a few dependency files, while others can happen due to a large number of dependencies between components with deep roots in the system. For divergences that can be traced to SATD, we aim to investigate if addressing their debt comments can help to resolve the divergence or not. In either case, we want to investigate how many divergences can be fixed in ArgoUML's architecture by applying changes directly suggested in SATD comments.

By contrasting the conceptual and concrete architectures of ArgoUML, we exposed 7 high-level divergences between different layers of subsystems, and 22 additional divergences that occur within them. We investigated each divergence and found that only 4 out of 29 divergences (14%) can be directly traced to SATD comments. When inspecting these 4 divergences, we found that addressing their debt comments directly results in fixing the divergences. However, the amount of divergences that can be traced to SATD and fixed is not representative enough. Moreover, this task is time-intensive and in general, will not lead to a significant architectural improvement.

This paper is organized as follows: Section II introduces the recovery of the concrete and conceptual architectures of ArgoUML, and the approach to identify SATD. Section III shows the approach to identify the architectural divergences in ArgoUML by contrasting its architectures. Section IV presents the results of tracing the divergences to SATD. Section V introduces the related work for our study, while we discuss the threats to the validity of our work in Section VI. Finally, Section VII concludes the paper and mentions future work.

II. Case-Study Setup

For our study, we require the conceptual and concrete architectures of ArgoUML and the SATD found in its source code comments. To obtain a conceptual architecture of the system, we attempted to derive from its domain reference architecture, however, were not able to find one. Therefore, we studied and followed the system's online design documentation instead. In the case of ArgoUML's concrete architecture, we extracted it directly from source code. The approach for obtaining both architectures is described below in this section. We also explain the procedure to obtain the SATD instances in ArgoUML and the mappings to their corresponding source files, aiming to facilitate the tracing of architectural divergences to SATD. To be consistent with the data set we are using, we target version 0.34 of ArgoUML, which is the same working version used by Maldonado et al. [13]. In this release, ArgoUML has 2,609 Classes, 176,839 SLOC, and 67,716 comments.

A. Conceptual Architecture

To extract the conceptual architecture, we followed the approach of the study by Hassan and Holt [6], where the authors used available documentation of a system to derive its conceptual architecture. We used the public online documentation of ArgoUML, which describes its general design and subsystems. In particular, we followed the contents of the project's design wiki [1], and a conceptual architecture defined by Castro et al. [5]. The first two authors of the paper followed the systems' design documentation to identify the main components of ArgoUML independently, adding the interactions between subsystems and layers of subsystems as described in the documentation. When the two separate conceptual architectures were ready, both authors iteratively discussed the resulting diagrams and interactions until an agreement was reached. By the end of this process, the authors had a consensus on the acquired architecture, determining it was true and in concordance to the conceptual design as documented on the wiki by ArgoUML developers.

Once we derived the architecture that we decided to proceed with, we found that it was missing several subsystems and that the official wiki of the project is not consistent in describing each component. In some cases, there was missing documentation for the components, or its contents were contradictory. Nevertheless, both guides were enough to confirm that each subsystem resides in a single file directory. With this in mind, we were able to obtain a final list of the 22 subsystems that compose ArgoUML.

These subsystems can be grouped into 4 architectural layers, where each subsystem belongs to a single layer. Table I presents ArgoUML subsystems organized per conceptual architectural layer. Each of these layers defines the design concept for its contained subsystems as follows:

Top Level: Initializes other subsystems on which it depends, but no other subsystem depends on it.

View and Control: These subsystems are initiated from the top level and depend on the GUI and Model subsystems.

Loadable Subsystems: These subsystems can only be connected through interfaces from other subsystems and can be enabled or disabled by the Module Loader subsystem.

Low Level: Any subsystem can depend on them, and they do not depend on any other subsystem. We use these conceptual

TABLE I
CONCEPTUAL ARGOUML SUBSYSTEMS GROUPED BY ARCHITECTURAL LAYER.

Arch. Layer	Subsystems	
Top Level	- Application	
Loadable	- Critics and Other Cognitive Tools - Java Code Generation & Reverse Engineering	- Other Source Languages - OCL
View and Control	- Code Generation - Explorer - Module loader - Notation - Persistence	- Property Panels - GUI - Diagrams - Reverse Engineering - Profile
Low Level	- Task Management - Help System - Internationalization - Logging	- Model - Configuration - Swing Extensions

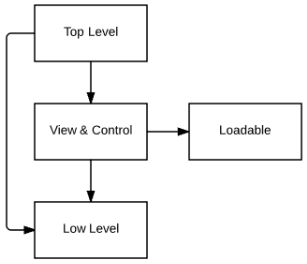

Fig. 1. Conceptual Layered Architecture of ArgoUML.

layer definitions to define the expected dependencies in ArgoUML, hence any dependency violating this design can be considered unexpected.

Due to space limitations, we present two visualizations that summarize the conceptual architecture of ArgoUML. Figure 1 shows a top view that we name Conceptual Layered Architecture, which abstracts all the subsystems in layers and their interactions. Given the fact that the subsystems in the Application, Low Level, and Loadable layers do not conceptually interact between each other, we do not present their detailed conceptual architecture. Figure 2 presents the conceptual architecture of the View and Control layer subsystems and their interactions.

B. Concrete Architecture

To extract the concrete architecture of ArgoUML, we followed the approach of Bowman et al. [3] to obtain the system's architecture from its source code. We downloaded and built the version 0.34 of ArgoUML and used the tool *Sci-Tools Understand*, an IDE for source code comprehension and static code analysis [19]. We used the IDE to create a UDB file for the project's source code; this file contains all the entities, and more importantly, the file dependencies of the project. To collect all the file dependencies, we created a Python script and queried the UDB file using Understand's API. This allowed us to get a landscape output with 3 elements: i) the files in the system; ii) the folder structure for the files; iii) the dependencies between the files. We input the landscape file into *lsedit*, a landscape visualization tool previously used in [3], that allows us to view and organize the extracted files and dependencies from ArgoUML.

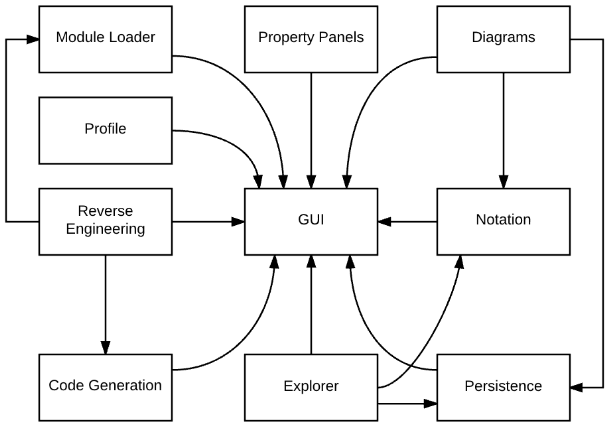

Fig. 2. Conceptual Architecture of the View and Control layer of ArgoUML.

We began to organize the project following its available documentation, which for several cases mentioned the folder structure of a subsystem. For example, the documentation of the Application (top level) subsystem, indicated that its files were located at *org.argouml.application*, which gave us a good insight into the structure of the project after looking at all the subsystem's documentation and repeating this organization process. Several subsystems did not have such documentation, in which case we relied on a mix between inspecting each file in detail to get a grasp of their utility and following the naming conventions used in the project, which we find to be very intuitive. We observe that for most files in the system, naming conventions and folder structure applied well to divide them into their respective subsystems, with some exceptions. We acknowledge that using simple naming conventions and folder structures do not necessarily reflect on a systems real architecture, however, we noticed few cases where these conventions were violated in ArgoUML.

In total, the organization of the landscape file with the concrete architecture of the system took 40 hours of work between the first two authors of the paper. Since the concrete architecture is too complex to visualize as a whole, we present the architecture of each layer separately below:

1) Top Level Layer: This layer is composed of only the Application subsystem, however, using the landscape file we were able to observe that all of the subsystems in the View and Control, and Loadable layers have dependencies to the Application subsystem. We inspected the connections in more detail and found that the Application subsystem has 6 internal modules to which all the other subsystems mentioned above depend on; namely, API, Kernel, UML, Events, Language, Pattern. We observe that all the files in the Application subsystem depend on these 6 modules, and there are many connections between the modules, but they do not depend on any other file in the Application subsystem. Because of this, we were able to group these modules into a new

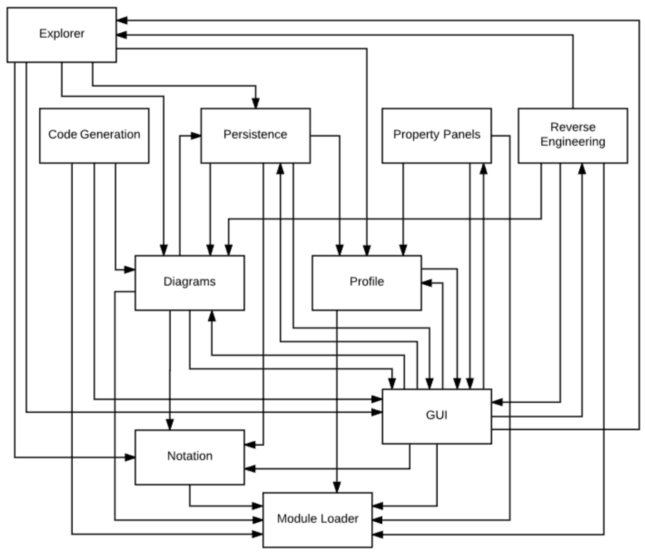

Fig. 3. Concrete Architecture of the View and Control Layer of ArgoUML.

subsystem that we named *Core*. When the Core subsystem is seen independently, the concrete architecture reveals that only the Application subsystem depends on it, and that most dependencies mentioned before from the View and Control, and Loadable layers are connected to the Core, and not the Application subsystem.

2) View and Control Layer: The View and Control layer is composed of 10 subsystems, we focus on this layer since the connections between its components are the most interesting for this system. Figure 3 shows the concrete architecture of this layer, from where we can observe far more dependencies between the subsystems than expected from the conceptual architecture. However, we notice that the directions of the dependencies do follow the conceptual design in most cases. All the unexpected dependencies found in this layer are discussed ahead in the paper. When seen as a whole, this section of the concrete architecture shows at least one dependency per subsystem to the Application layer; this behavior was not expected. Additionally, the subsystems in this layer depend freely on the Loadable and Low Level layer subsystems (as there is no conceptual or design restriction for them to do so).

3) Loadable Layer: Once we extracted the concrete architecture of the system, there were 2 subsystems missing completely; they are the Java Code Generation & Reverse Engineering, and Other Source Languages subsystems. When inspecting the source code closely, we could not find any trace of them. We believe that they may have been removed from the version of ArgoUML we are working on; in which case, the documentation of these 2 subsystems should be updated accordingly. Additionally, it is important to mention that when organizing the landscape files into subsystems, we did observe several files and two sub-folders which we could not map to any of the conceptual subsystems. Luckily, they had intuitive names such as "utilEvent", and were being used by several subsystems across the whole architecture. Based on this, we decided to merge all those unmapped files and sub-folders into a new subsystem that we simply named *Utility*, which we identify as being part of the Loadable layer for how they interact in the system. By design, all the subsystems in this layer can be used by any other subsystem in ArgoUML, except the ones from the Low Level layer. The concrete architecture of this layer revealed that there are several dependencies between subsystems, that are acceptable in design. Because of this and for space restrictions, we do not present this architecture as a figure.

4) Low Level Layer: This layer's subsystems are designed to i) be depended on by the Top Level, and View and Control layers; and ii) not depend on any other subsystem (in any layer). By looking at the concrete architecture we find that in reality, there are 2 unexpected dependencies between the subsystems of this layer, which we have coded as follows:

- L1 - Helper System $\longrightarrow$ Internationalization.
- L2 - Helper System $\longrightarrow$ Module.

C. SATD Identification

Maldonado's et al. [13] dataset contains the SATD comments of 10 open source systems classified by type; where ArgoUML is the richest of them in technical debt comments with 1,413 occurrences. Moreover, 56.68% of these comments were tagged as design SATD. This is over twice the amount of debt instances found in the remaining systems of the data set. Reading through the classified comments surfaced some good indicators that further motivated the study, e.g.: *"TODO: This can't depend on projectbrowser. it needs to get the current drawing area from the diagram subsystem or gef"*; *"TODO: move to diagram subsystem?"*; *"TODO: Lets move this behind the model interface"*.

To identify SATD in ArgoUML, we created a set of 3 Java command-line applications and compiled them as runnable JAR files, with the names: "CX", "CF", and "SATDF" to extract, filter and identify SATD in source code comments, respectively. The following describes the extraction process, while Fig. 4 depicts it. We query the UDB file we created previously with the Python API of Understand to mine all the Java file names in the project. We format the file names and output them as a list in a text file. The set of file names is then given as an input to the CX JAR file, which is in charge of two tasks: a) invoking a process of the tool srcML [20], which decorates the source code of the Java files; and b) parse the decorated code with the Java SAX Parser [17] to extract all line and block comments. This generates another text file with the raw comments only.

Next, we filtered the comments following the approach of Maldonado et al. [13] which uses 5 heuristics to a) remove all License comments (written above a class declarations); b) remove Javadoc comments, c) group consequent single line comments to consider them as 1 comment; d) remove auto-generated comments, which can be "Auto-generated method stub", "Auto-generated constructor stub" or "Auto-generated catch block"; and e) remove commented Java source code, as it most likely is unused code and does not contain SATD. It's important to note that to avoid removing Javadoc or License

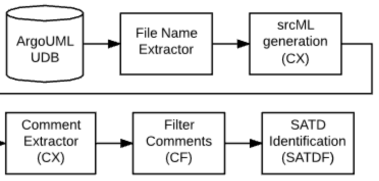

Fig. 4. SATD collection process.

comments that could contain SATD, we avoid the removal if they contain the "TODO:", "FIXME:" or "XXX:" task annotations. Lastly, the SATDF JAR file is used to identify the SATD instances in the set of filtered comments. This JAR file leverages the set of 62 patterns belong to the set made available by Potdar and Shihab [18] for SATD detection. In total, we found 3,863 cases of SATD with static analysis in 1,132 files, from a total of 3,503 files in ArgoUML and successfully mapped all 1,413 classified SATD comments to their source files.

III. Identifying Architectural Divergences

Before being able to answer our research questions, we need to identify the architectural divergences of ArgoUML. Once we obtained both conceptual and concrete architectures of ArgoUML, we followed the approach of Murphy et al. to find divergences, convergences and absences in the system and create contrasting models [16]. We define the elements of a contrasting model for our work as follows:

- **Divergences:** dependencies between subsystem components that do not exist in the conceptual architecture, but appear in the concrete architecture of the system.
- **Convergences:** Dependencies between subsystem components that exist in both, the conceptual and concrete architectures of the system.
- **Absences:** Dependencies between subsystems components that exist in the conceptual architecture, but not in the concrete architecture of the system.

We contrasted both architectures of ArgoUML and observed that there are divergences occurring from one layer of subsystems to another, which we call *high-level divergences*, and several divergences that occur within a layer, namely inside the View and Control, and Low Level layer. In this section, we present a high-level contrasting model of the layered architecture of ArgoUML. Since we found a high number of divergences due to multiple unexpected interactions between subsystems of the View and Control layer, we introduce and focus on a contrasting model of this layer with further detail. Lastly, we describe two divergences found in the Low Level layer of the system.

A. High-Level Contrasting Model

Figure 5 shows the high-level contrasting model of the architecture layers in ArgoUML. In this model, we introduce the *Core* as a divergent layer with dependencies to other layers. As we described previously, from the concrete architecture of the system, we found that inside the Application Subsystem there were several modules to which other subsystems

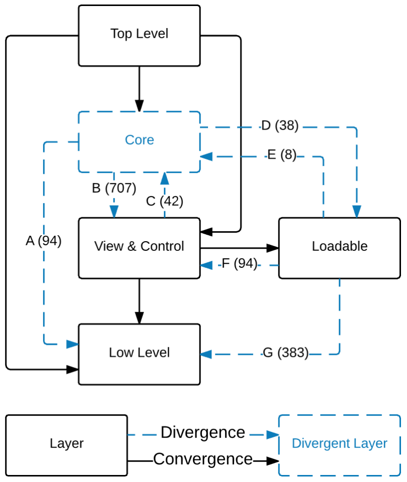

Fig. 5. Contrasting model of ArgoUML Layered Architecture. High-level divergences labeled A to G with their corresponding number of dependencies.

depended on, from different layers. After quantifying these dependencies, we found that their numbers are too high to be considered a design flaw. We included an identifier to the high-level divergences (A-G) in Figure 5, as well as the respective number of dependencies between modules. Moreover, while obtaining the concrete architecture from the system's source code, we noticed that the subsystem interactions of those modules allow the Core layer to be seen as independent. The introduction of the Core layer also makes the Top Level layer and its modules to compile with the conceptual design of the system. Hence, we have added the Core as a complete divergent layer.

To get a better understanding of these connections, we also included the amount of dependencies per divergence. From this table, we can observe that the amount of dependencies varies greatly among the high-level divergences. In this case, the higher the number of dependencies, the less likely that a divergence was introduced by fault or violating ArgoUML's design. For example, we can see that in divergence *B* from the Core layer, there are 707 file dependencies to the View and Control layer, suggesting that this is not the result of simple developers' expediency. Similarly, we found several dependencies of the Loadable layer to the Core, and View and Control layers; namely divergences *E* and *F*, which are originated by the OCL, and Critics and other Cognitive Tools subsystems. We inspected these dependencies at the source code level and found that these subsystems seem to be highly coupled to each other, in particular, the *F* divergence that has 94 dependencies. While divergence *E* has only 8 file dependencies, nothing indicates an error in them, rather the involved subsystems appear coupled.

B. View and Control Layer Contrasting Model

Figure 6 shows the contrasting model for the View and Control layer in ArgoUML; here we observe 20 divergences, surpassing the amount of convergences in the model. We numbered these divergences (1-20) to identify them throughout our work (see Table II). The model shows 2 absences that

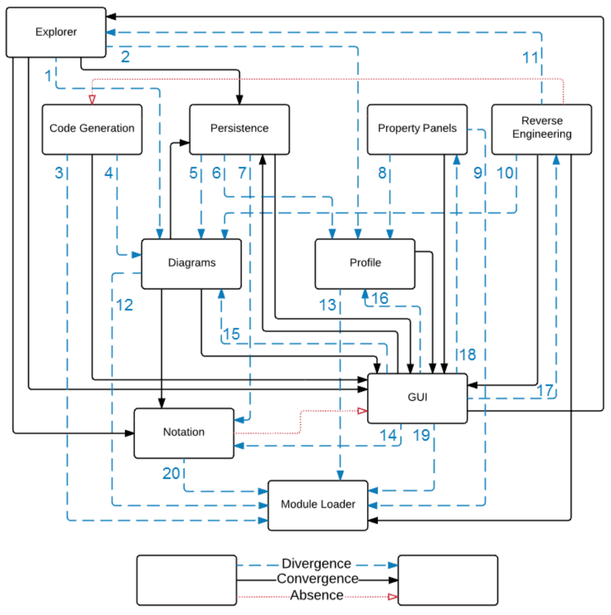

Fig. 6. Contrasting Model of the View and Control layer.

TABLE II
DEPENDENCIES PER VIEW AND CONTROL LAYER DIVERGENCE

	View & Control Divergence	**No. of Dependencies**
1	Explorer ⟶ Diagram	19
2	Explorer ⟶ Profile	11
3	Code Generation ⟶ Module Loader	1
4	Code Generation ⟶ Diagrams	1
5	Persistence ⟶ Diagrams	12
6	Persistence ⟶ Profile	4
7	Persistence ⟶ Notation	1
8	Property Panels ⟶ Profile	2
9	Property Panels ⟶ Module Loader	1
10	Reverse Engineering ⟶ Diagrams	7
11	Reverse Engineering ⟶ Explorer	1
12	Diagrams ⟶ Module Loader	4
13	Profile ⟶ Module Loader	1
14	GUI ⟶ Notation	71
15	GUI ⟶ Diagrams	66
16	GUI ⟶ Profile	16
17	GUI ⟶ Reverse Engineering	8
18	GUI ⟶ Property Panels	14
19	GUI ⟶ Module Loader	1
20	Notation ⟶ Module Loader	1

were expected from the conceptual architecture but were not found in the file dependencies from the source code; they are Reverse Engineering to Code Generation and Notation to GUI. To better explain the divergences found, similar to our previous reflexion model, we present the quantified dependencies per divergence in Table II. Based on the number per divergence, we can see there are several cases with a low number of dependencies. For example, 8 of the 20 cases only occur because of 1 file dependency, which could be traced to a SATD comment. Lastly, we observe that the GUI subsystem has the highest number of dependencies, 176 in total, to 6 different modules.

C. Low Level Layer Divergences

Only two interactions are seen between the subsystems of the Low Level layer in the concrete architecture. Following the layer's conceptual design, only subsystems from other layers should depend on low level subsystems, and no dependencies should occur among them. Therefore, both interactions are architectural divergences; we identify them as L1 and L2. L1 is caused by 1 file dependency from the Help System to the Internationalization subsystem; while L2 is caused by 5 file dependencies from the Help System to the Model subsystem.

D. Architectural Divergences

Contrasting the concrete and conceptual architectures of ArgoUML and inspecting its source code surfaced several architectural divergences and 2 absences. There are 7 high-level divergences that occur between layers of subsystems and 1 divergent layer, the Core. 20 other divergences were found inside the View and Control layer, where divergences are more prevalent than convergences. Lastly, there are 2 divergences inside the Low level layer, labeled L1 and L2. Note that high-level divergences tend to be composed of more file dependencies than those inside the View and Control, and Low level layers. Often in the later, only 1 or 2 file dependencies are responsible for the architectural divergences. This observation leads us to investigate each divergence of ArgoUML at the source code level.

IV. CASE-STUDY RESULTS

Our goal is to investigate if the architectural divergences of ArgoUML can be traced to SATD, and if so, to explore if addressing the debt can resolve these implementation drifts that occur over time. After the preliminary work to identify the architectural divergences of ArgoUML, we proceed to: inspect how many divergences can be traced to SATD (RQ1); investigate the cases where addressing SATD comments can fix architectural divergences (RQ2).

A. RQ1 - How many architectural divergences can be traced to SATD comments?

Once we have contrasted ArgoUML's architectures and found its divergences from concept to implementation, we trace them to SATD by inspecting the source code of the divergences and our available set of classified SATD comments. We are motivated to see if divergences can be explained by a SATD instance, and possibly fixed by addressing it directly. To do this, we inspected each of the architectural divergences in the system; counted the number of files per divergence and the number of SATD instances found per file. More importantly, we looked at the files in detail to see if: a) the source code comments are able to explain a divergence; and b) there is a way to resolve the divergence by addressing those comments in the source code.

We began by inspecting the high-level divergences, however, we found several factors that limit tracing them to SATD. First, the divergences A-G are caused by up to 707 file dependencies; while inspecting these divergences we noted that most of them are dependencies well rooted in the system and originated in highly coupled subsystems. As we explained previously while contrasting the high-level models of the system, divergence E has the lowest number of dependencies; nevertheless, they are highly coupled and do not indicate that they were introduced by error or a design flaw. Furthermore, we could not find SATD that explains this divergence, or to the best of our knowledge, a realistic way to refactor or re-engineer the divergence. This scenario was repeated for the 7 high-level divergences in ArgoUML. We stopped the inspection of a divergence after several of the file dependencies lacked a feasible resolution. Moreover, there is a lack of SATD comments that could explain or help to resolve these divergences. Because of this, we do not consider tracing SATD to these high-level divergences to be feasible.

On the other hand, when we inspected the 20 divergences found inside the View and Control, and 2 inside the Low Level layers, we found several encouraging cases where SATD could help to resolve divergences. This is because several of them are caused by a low number of file dependencies (see Table II) that contain SATD comments. For example, we observed that 10 divergences of the View and Control layer originated in only 34 files that contain 139 SATD comments; a similar scenario was repeated for the remaining divergences. Since the 20 architectural divergences found inside the View and Control, and 2 inside the Low Level layers are good candidates for SATD tracing, we focus on them and provide a more detailed analysis of what we found by looking at their source code. We grouped the analyzed divergences in 3 categories below: i) divergences that can be directly traced to SATD; ii) divergences that can be resolved but where SATD comments are not helpful; iii) divergences that cannot be resolved.

1) Divergences that can be traced to SATD: We were able to trace 4 architectural divergences to SATD successfully, they are 4, 7, 8, and 11. The SATD comments within them explain the divergences and provide good insight to a potential fix for the architectural divergence. We provide an in-depth analysis of each below:

Code Generation to Diagrams (4): The method *setPorts(Layer, FigEdge)* inside the Code Generation subsystem was directly copied from a class inside the Diagram subsystem as expressed in its following SATD comment:

"// TODO: Copied from UmlDiagramRenderer"

This method requires *FigEdge* as a parameter which creates a dependency to the *DiagramSetting* class of the Diagrams subsystem. Re-writing this method according to the strict needs of this class fixes the divergence.

Persistence to Notation (7): There is a dependency from the *ArgoParser* class to the *NotationSettings* class in the Persistence and Notation subsystems, respectively. This dependency is caused by multiple function calls which are absolutely necessary to implement the functionalities of *ArgoParser*. Because of this, there is no way to remove these connections. *NotationSettings* is tightly coupled with the Notation subsystems. Therefore, there is only one way to remove this dependency, which is in fact suggested by a SATD comment inside the problematic file:

"// Maybe this can be implemented in the PersistenceManager?"

Thus, if we relocate *ArgoParser* inside Persistence subsystem, this divergence is resolved.

Property Panels to Profile (8): The *GetterSetterManagerImpl* class inside the Property Panels module uses deprecated methods from the *Profile* class and throws *ProfileException* which is inappropriate for the type of exception it produces. This issue is revealed by the following multi-line SATD comment.

"// TODO: We need the property panels to have some
// reference to the project they belong to instead of using
// deprecated functionality
// Get all classifiers in all top level packages of all profiles
// TODO: We need to rethrow this as some other exception
// type but that is too much change for the moment."

The method *getChoices* of the *GetterSetterManagerImpl* class should be moved because it was a hack solution and the super classes it extends belong to the Kernel subsystem. A proper new type of exception should be introduced instead of using *ProfileException*; such fix will eliminate the divergence.

Reverse Engineering to Explorer (11): This divergence can be identified and fixed by following the following particular SATD comment:

"// TODO: Send an event instead of calling Explorer
// directly"

In the *ImportCommon* class of the Reverse Engineering subsystem, the developer called '*ExplorerEventAdaptor.getInstance().structureChanged()*' directly instead of using an event which violated core design principles of ArgoUML. The developer admitted this fault in a SATD comment and suggested a solution in the same line. Nevertheless, the problem has not been addressed and the divergence remains.

2) Divergences that can be resolved without SATD: We were able to find a fix for 7 of the divergences we inspected; however, SATD comments did not provide an explanation or resolution for them. These divergences are 3, 6, 9, 12, 13, 20, L1. We found that they had between 1 and 4 dependencies only, and the class files that caused them were not tightly coupled and could be relocated. Most of the time applying a simple re-engineering or refactoring task would provide a fix for the divergence. We also found that only divergences 6, 12 and L1 had SATD comments; having 1, 5 and 2 comments, respectively. The remaining divergences 3, 9, 13 and 20 had no SATD comments.

As a practical example, divergence 12 (Diagrams to Module Loader) had 4 dependencies caused by 4 class files; inspecting them revealed that all of them contained the following SATD:

"// TODO: Remove the casting to DiagramFactoryInterface2

Although this comment was repeated in each class file that caused the divergence, it did not provide an explanation or was useful to fix the divergence. In fact, we noticed that these classes inside the Diagrams subsystem could be simply relocated inside the Module Loader subsystem as no other classes in Diagrams depended on them. The remaining divergences in this category faced a similar scenario; we were able to find fixes for them, but SATD comments were not useful or provide any insight for doing so.

3) Divergences without resolution: In total, we inspected other 11 divergences that did not have a resolution, or could be explained by SATD comments; they are 1, 2, 5, 10, 14-19, and L2. By looking at the source code of the divergences we found several reasons that prevented their resolution. First, all of the dependencies created by these divergences were often spread across several files and subsystems. Second, their files had tight coupling between subsystems, avoiding the relocation of files to solve divergences. A good example of this is divergences 14-19 that occur from the GUI subsystem to 6 subsystems (Notation, Diagrams, Profile, Reverse Engineering, Property Panels and Module Loader). These 6 divergences are caused by 172 file dependencies and originated in 234 source files. While inspecting each case, we noticed that because of their high number of dependencies and coupling nature, they fall into the same case as the high-level divergences that we inspected previously, showing no straightforward solution to fix the divergences. A specific example is divergence 19, which has only 1 dependency. Inspecting this dependency revealed that it can not be removed as the *AboutBox* class of the GUI uses information from the *ModuleLoader2* class to describe the available modules in the system. Similar observations were repeated for the remaining divergences in this category. The detailed analysis and inspection of these divergences did not surface any SATD comments that could explain or help to resolve these architectural divergences.

Results: We found that 4 of the architectural divergences inside the View and Control layer (4, 7, 8, and 11) can be successfully traced to SATD comments. Reading through these comments can provide insight on each of the divergences, and directly addressing them can lead to their resolution. Inspecting the architectural divergences of ArgoUML revealed that not all them can be traced to SATD. Looking at the 7 high-level divergences of the system we found that due to the amount of dependency files that cause them, and their roots in the system, attempting to resolve them is not feasible. Besides the high-level divergences, we inspected 22 others that exist between the subsystems of ArgoUML, 20 in the View and Control layer, and 2 in the Low Level layer. We found that 11 of these divergences fall in the same case of the high-level ones. Moreover, we did not find SATD comments that could explain or help to fix these divergences, nor a simple re-engineering or refactoring task that could be applied to resolve them. In contrast to the previous, we also found 7 divergences that could be fixed by simple tasks such as relocating class files; however, their resolution was not related to their SATD comments. Overall, 80% of the divergences that we inspected contained SATD comments, however not all of them were helpful to provide a fix or explain a divergence.

B. RQ2 - How many architectural divergences can be fixed by addressing SATD?

After analyzing all the divergences, we focused on the 4 divergences that were traced to SATD. We explored if addressing the debt comments withing these divergences can actually provide an architectural fix. We manually implemented the changes as suggested by SATD comments (as indicated below) for each traced divergence. To validate a fix, we simply checked that it does not introduce an error to the project and that the system builds successfully.

Code Generation to Diagrams (4): We changed the *setPorts(Layer, FigEdge)* method to *setPorts(Layer, Transition)*. The new method does not require *FigEdge* as a parameter; instead it calls the original *setPorts(Layer, FigEdge)* method in *UmlDiagramRenderer* for setting ports.

Persistence to Notation (7): Instead of calling methods in *NotationSettings* directly, we used *PersistenceManager* to get the fall-back notation value.

Property Panels to Profile (8): Instead of throwing a *ProfileException* from *GetterSetterManagerImpl.java*, we introduced new kind of exception called *PropertyPanelException*.

Reverse Engineering to Explorer (11): We triggered a *UmlChangeEvent* in the *ImportCommon* class of the Reverse Engineering subsystem instead of calling *ExplorerEventAdaptor.getInstance().structureChanged()*, which is a deprecated method that created a divergence.

Only 4 out of 29 (14%) architectural divergences found in ArgoUML can be traced to SATD and fixed by directly addressing the debt comments.

V. Related Work

Our work traces architectural divergences to SATD comments found in source code. As preliminary work, we surface ArgoUML divergences by contrasting its concrete and conceptual architectures. Consequently, we divide our related work into two subsections: work that addresses the study of SATD; and work related to the recovery and contrast of concrete and conceptual architectures.

A. Work Related to SATD

In 2014, Potdar and Shihab [18] first used source-code comments to identify technical debt and introduced the term of SATD. Potdar read through over 100 thousand comments to manually identified technical debt. The authors extracted 62 patterns for SATD identification from the studied comments and found that SATD existed in up to 31% of the files in a system. Maldonado and Shihab. [12] used these patterns to detect and analyze over 33 thousand comments from 5 open source systems, and classified them into 5 types of SATD: design, defect, documentation, requirement, and test debt. They found that design is the most common types of SATD, conforming 42% to 84% of all SATD comments. Bavota and Russo [2] also used the pattern-based approach introduced by Potdar and Shihab [18] to identify and analyze SATD in the complete change history of 159 open source systems.

This large-scale study confirmed that SATD is prevalent and constantly increasing in software systems; that it remains in the code base for long periods of time; and that it is mostly removed by experienced developers.

Later Maldonado et al. [13] investigated the use of Natural Language Processing techniques to automatically detect design and requirement debt. They extracted and manually classified source comments from 10 software systems and trained a maximum entropy classifier that automatically extracts key features (i.e., words) from a classified training dataset. The classifier then uses these keywords to identify design and requirement SATD based on features that contribute positively or negatively a comment's classification. This NLP approach outperforms the pattern-based approach for SATD identification. We leverage the classified data set of SATD comments that resulted from this study. A recent work by Zampetti et al. [22] also used the same data set to train and evaluate a machine learning approach that recommends when developers should self-admit design technical debt by leveraging method-level source code features. More recently, Huang et al. [8] presented an approach for automated SATD identification using text mining techniques. The authors used feature selection to extract key features from a different set of projects and train a classifier for each. They merged the classifiers to construct a composite cross-project classifier that outperforms previous NLP and pattern-based techniques.

We did not use NLP techniques or text mining for SATD identification; instead, we used a pattern-based approach leveraging the 62 patterns presented by Potdar and Shihab [18], combining them with the top 10 features for SATD identification surfaced by the NLP classifier from the work of Maldonado et al. [13]. Recent work has also used the pattern-based approach to detect SATD, for example, Mensah et al. [14] used the 62 patterns found by Potdar and Shihab [18] to identify and estimate the effort needed in LOC to resolve a SATD instance based on term weights. Finding that on average, 13 to 32 LOC are needed to resolve a SATD instance. Kamei et al. [9] also used a similar data set to study the evolution of product metrics related to SATD introduction and removal, finding that 42% to 44% of SATD incurs in positive interest over time.

Other works studied additional aspects of SATD; Wehaibi et al. [21] investigated the relation between SATD and software quality by looking at the defect proneness of files. They observed no relation between SATD and defects but found that the introduction of SATD results in more complex changes in the future. Maldonado et al. [11] further investigated how SATD is removed from source code, finding that the majority of it is removed and that most of the time, it is done by the same developer who introduced the SATD. This highlights that removing SATD is of importance for developers and that its removal is common practice.

B. Work Related to Architecture Recovery

Many large software systems do not have their system architecture properly documented. As a result, it becomes difficult for developers and maintainers to understand and improve these systems. To address this problem, Bowman et al. [3] decided to examine the architecture of a large software system (Linux Kernel) by extracting it from source code. At first, they formed a conceptual architecture of Linux Kernel based on how developers see the system and how different subsystems work and depend on each other. After that, they extracted the actual architecture of the source code, known as, concrete architecture. This concrete architecture reflects exactly how the system is implemented, what are the main components of that system and how these components depend on each other. After contrasting these two architectures, they found that, in reality, a software contains many more undocumented dependencies which do not exist in the original design. This deterioration from original design makes it difficult for developers to understand their systems. This study shows how the architecture of an undocumented system can be extracted and how much its concrete architecture can deteriorate from the architecture that developers designed initially.

Murphy et al. [16] introduced the concept of Reflexion Model that computes and shows the differences between the high-level conceptual model and source level concrete model of software. Their reflexion model enables engineers to get a global overview of their system's structures in a few hours. To generate a reflexion model, the developer has to provide a mapping which associates one or more components of the concrete model, such as, file or directory to one component of the conceptual model. Using this mapping, a graph representation of the system is presented where convergences, divergences, and absences are shown using different types of edges. By performing a case study on NetBSD and Microsoft Excel Spreadsheet products Murphy et al. [16] found that Reflexion Model can be very helpful for understanding a new system or planning re-engineering tasks for an existing system.

Many techniques have been proposed in the literature to automatically or semi-automatically recover software architectures from software code bases. Lutellier et al. [10] compared nine variants of six existing architecture recovery techniques to understand and evaluate their effectiveness. In our work, we do not use automated or semi-automated approaches for software recovery since we strive for a manual and detailed inspection of the subsystems in a single project. Instead, we leverage the manual approaches utilized by Bowman et al., and Murphy et al. with the intention of acquiring a deeper understanding of our study subject and its components.

VI. Threats to Validity

The conceptual architecture we present is an abstraction based on the available design documentation of ArgoUML, to which several contributions have been done over time by the open source community. Hence it may vary from the one created originally by the architects. We followed the design wiki of ArgoUML thoroughly, and believe that the conceptual architecture we present consistently reflects the design documentation maintained by ArgoUML developers. We are aware that developers often have to move away from their original design to match changing requirements, and in many cases, they do not update the documentation to match the new changes. This results in a threat, as ArgoUML's documented architecture may not reflect what developers actually followed to develop the software. To mitigate this, we relied on the primary conceptual documentation of the system, which refers to the architectural layers we defined in section II-A, and their interactions. Therefore, the architectural divergences we detected are only those that that violate the main conceptual design of the system. Even when considering the worst case scenario, where the documentation we used was indeed outdated, at least the core design of the extracted conceptual architecture should remain consistent.

We acknowledge that the pattern-based approach we used to identify SATD might not find all debt instances in the source code. To dampen this threat, we used two recent sources of SATD patterns and extended the set of patterns to 100 based on reading the classified source code comments in the data set at our disposal. Although the patterns will most likely miss some SATD instances in the project releases, this is still one of the best replicable approaches for SATD identification. Moreover, the whole purpose of the ad-hoc process for SATD identification was only to find the source files to which the classified SATD comments in our data set belonged to. We rely on the data set of classified comments, which used a more advanced approach with NLP techniques for SATD identification. Lastly, we also relied on Sci-Tools Understand for creating the UDB files that we used to find the file dependencies from source code. While inspecting the source code of ArgoUML, we discovered that sometimes Understand reported dependencies that did not exist in the source code. We manually checked all the cases that fell into this scenario and removed them from the UDB file.

VII. Conclusion and Future Work

In this paper, we investigated the possibility of using SATD comments to resolve architectural divergences. We leveraged a data set of classified SATD comments and traced them to bad architectural implementations that we surfaced by contrasting the conceptual and concrete architectures of ArgoUML, using its available design documentation and source code. Our study revealed 29 architectural divergences, 7 in high-level layers, and 22 among subsystems. Our preliminary results show that merely 4 out of 29 divergences (14%) can be directly traced to SATD, and that looking at SATD comments can provide enough information to fix them. To validate this, we manually implemented the changes as suggested by debt comments in each of the 4 traced divergences and confirmed that addressing the comments directly leads to resolve the divergences. Although SATD can be used as an indicator for architectural divergences, it requires considerable time and effort, and will not result in a significant architectural improvement.

To generalize our findings, we plan on replicating this study on a broader scale. Looking at SATD comments that can be traced to architectural divergences resulted effective in resolving them, thus, we plan to further investigate this with better architectural recovery and more advanced SATD detection approaches, potentially in an automated manner.

References

[1] Argouml wiki: Design. http://argouml.tigris.org/wiki/Design, 2009. (Accessed on 10/22/2018).

[2] G. Bavota and B. Russo. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 315–326. ACM, 2016.

[3] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 555–563. ACM, 1999.

[4] W. Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993.

[5] T. R. de Castro, S. N. A. de Souza, and L. S. de Souza. Case tool for object-relational database designs. In *Proceedings of the 7th Iberian Conference on Information Systems and Technologies*, CISTI '99, pages 1–6. IEEE, 2012.

[6] A. E. Hassan and R. C. Holt. A reference architecture for web servers. In *Proceedings of the 7th Working Conference on Reverse Engineering*, WCRE '00, pages 150–159. IEEE, 2000.

[7] A. E. Hassan and R. C. Holt. Using development history sticky notes to understand software architecture. In *Proceedings of the 12th International Workshop on Program Comprehension*, ICPC '04, pages 183–192. IEEE, 2004.

[8] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, 23(1):418–451, 2018.

[9] Y. Kamei, E. d. S. Maldonado, E. Shihab, and N. Ubayashi. Using analytics to quantify interest of self-admitted technical debt. In *Proceedings of the 1st International Workshop on Technical Debt Analytics*, TDA '16, pages 68–71. CEUR-WS, 2016.

[10] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15.

[11] E. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik. An empirical study on the removal of self-admitted technical debt. pages 238–248, 2017.

[12] E. Maldonado and E. Shihab. Detecting and quantifying different types of self-admitted technical debt. In *Proceedings of the 7th International Workshop on Managing Technical Debt*, MTD '15, pages 9–15. IEEE, Oct 2015.

[13] E. Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, Nov 2017.

[14] S. Mensah, J. Keung, M. F. Bosu, and K. E. Bennin. Rework effort estimation of self-admitted technical debt. pages 72–75, 2016.

[15] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. *Software Engineering Notes*, 20(4):18–28, 1995.

[16] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *Transactions on Software Engineering*, 27(4):364–380, 2001.

[17] Oracle. Parsing an xml file using sax (the java tutorials - java api for xml processing (jaxp) - simple api for xml). https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html, 2017. (Accessed on 10/16/2018).

[18] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, ICSME, pages 91–100. IEEE, 2014.

[19] Sci Tools. Quickly comprehend legacy code — scitools.com. https://scitools.com/legacy-code-tool/, 2018. (Accessed on 10/17/2018).

[20] srcML Tools. srcml. http://www.srcml.org/, 2018. (Accessed on 10/16/2018).

[21] S. Wehaibi, E. Shihab, and L. Guerrouj. Examining the impact of self-admitted technical debt on software quality. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 1 of *SANER '16*, pages 179–188. IEEE, 2016.

[22] F. Zampetti, C. Noiseux, G. Antoniol, F. Khomh, and M. Di Penta. Recommending when design technical debt should be self-admitted. pages 216–226, 2017.