

IMPLEMENTING PRODUCT LINE ARCHITECTURE
WITH CODE GENERATION AND SEPARATION

A THESIS IN
Computer Science

Presented to the Faculty of the University
of Missouri-Kansas City in partial fulfillment of
the requirements for the degree

MASTER OF SCIENCE

by
GHARIB GHARIBI

B.S., Jazan University, 2013

Kansas City, Missouri

2016

© 2016

GHARIB GHARIBI

ALL RIGHTS RESERVED

IMPLEMENTING PRODUCT LINE ARCHITECTURE
WITH CODE GENERATION AND SEPARATION

Gharib Gharibi, Candidate for the Master of Science Degree
University of Missouri-Kansas City, 2016

ABSTRACT

Software product line engineering (SPLE) emphasizes high level of reuse and mass customization of the core assets shared by a family of software products. Product line architecture (PLA) is a promising application of architecture-centric development in SPLE. However, unfaithful implementation of the PLA and manual implementation of its variation points remain two difficult challenges that need to be addressed in this area. While many PLA implementation approaches exist, they either focus on certain types of variability or require manual implementation of variation points.

In this thesis, I present a novel code generation and separation approach that can faithfully implement the PLA with a goal of reducing the inconsistency between the PLA and its implementation. Moreover, the approach can automatically implement the variation points modeled in the PLA and convert them to code entities using different techniques based on the variation point's type.

I have implemented the approach in ArchFeature, an Eclipse-based PLA development environment, and evaluated it in a case study with a chat application. The purpose of the evaluation was to validate the approach and to assess its feasibility, performance, and affordability.

APPROVAL PAGE

The faculty listed below have examined a thesis titled “Implementing Product Line Architecture with Code Generation and Separation,” presented by Gharib Gharibi. Student, candidate for the Master of Science degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Yongjie Zheng, Ph.D., Committee Chair

School of Computing and Engineering

Baek-Young Choi, Ph.D., Committee Member,

School of Computing and Engineering

Yugyung Lee, Ph.D., Committee Member

School of Computing and Engineering

CONTENTS

ABSTRACT.....	iii
LIST OF ILLUSTRATIONS.....	vii
LIST OF TABLES.....	viii
ACKNOWLEDGMENTS.....	viii
Chapter	
1. INTRODUCTION.....	1
1.1 Overview.....	1
1.2 Research Problem.....	2
1.3 Objectives.....	4
1.4 Contribution.....	4
2. BACKGROUND AND RELATED WORK.....	6
2.1 Background.....	6
2.1.1 Architecture-Centric Development in Software Product Line.....	6
2.1.2 Software Product Line (SPL).....	8
2.1.3 Code Generation.....	9
2.1.4 ArchFeature: A PLA Modeling Environment.....	10
2.2 Related Work.....	12
3. APPROACH.....	15
3.1 Overview.....	15
3.2 Code Generation and Separation.....	16
3.2.1 Implementing Core and Optional Elements of the PLA.....	19
3.2.2 Implementing Alternative Elements of the PLA.....	20
3.2 Example of the PLA Implementation.....	22
4. IMPLEMENTATION.....	26
4.1 Implementation Environment.....	26
4.1.1 Eclipse.....	26
4.1.2 Java Emitter Templates.....	28
4.1.3 ArchStudio.....	30
4.2 Implementation Tasks.....	33

4.2.1 Implementing Optional and Core Components	33
4.2.2 Implementing Alternative Components.....	34
4.2.3 Implementing Alternative Interfaces	36
5. EVALUATION.....	39
5.1 Objectives.....	40
5.2 Methodology	41
5.3 Results	42
6. CONCLUSION AND FUTURE WORK	44
REFERENCES	45
VITA.....	50

LIST OF ILLUSTRATIONS

Figure	Page
1. ArchFeature modeling environment.....	11
2. Implementation structure of a PLA component	17
3. Implementation structure of alternative components	21
4. Implementation structure of a component including an alternative interface	21
5. PLA example of a chat application implemented in ArchFeature	23
6. Example of a PLA component implementation	25
7. Eclipse platform structure	28
8. Screenshot of ArchStudio main interface.....	32
9. PLA of chat application.....	40

LIST OF TABLES

Table	Page
1. Comparision between existing variability implementation appraoches	12
2. Features of the chat application and their types	39

ACKNOWLEDGMENTS

This work would have never seen the sunlight without the help of many people.
Thank you all!

Mom, thank you for teaching me to be who I am today. Dad, I appreciate every single word you have ever taught me. Thank you for your continuous support, you are my hero. Brother, I am very happy to have you in my life. Keep up the good work. Grandma and Grandpa, I am grateful for raising me the way you did. I am blessed to have you. I wish you all could be here to celebrate this with me.

Dr. Zheng, thank you for teaching me how to be a better student and a better person. Thank you for your continuous motivation and trust. I appreciate all the time and effort you have invested in me. It's an honor to continue my graduate study with you.

Prof. Ghulam Chaudhry, thank you for the great support you have given to me. Because of you, I had the opportunity to teach college and get involved on campus.

Sandy Gualt, I am thankful for your great efforts in making UMKC a better place for international students. It was an honor to be part of your team.

Dr. Choi and Dr. Lee, thank you for your time and effort being my supervisory committee. I really appreciate all of your work.

Cuong Cu, my friend, thank you for your continuous help. I am grateful for your technical support throughout the project. I wish you a happy and successful life.

Finally, my sincere thanks goes to the School of Computing and Engineering faculty, staff, and friends. Thank you all.

CHAPTER 1

INTRODUCTION

In this chapter, I briefly introduce the area of my research, the research problem that I address in this area, the objectives of the work, and its contribution.

1.1 Overview

Software product line engineering (SPLE) [30] advocates a high level of platform reuse and mass customization in the development of a family of software products that share substantial commonalities, known as software product line [8]. Product line architecture (PLA) [7, 34] plays an increasingly important role in SPLE. Specifically, architecture-centric development advocates that the architecture must be the focus of the development rather than the code. In general, a software system's architecture is defined as the set of principal design decisions about the system [34]. Similarly, the PLA captures the principal design decisions of all products of a product line simultaneously. It is characterized as a monolithic architecture that captures both commonalities and variation points of a product line. Variation points [8] (e.g. optional components) are places in the PLA and the product line code that identify where a certain product line feature is modeled and implemented respectively. Product line code includes the implementation of all product line variants. A product line feature (or simply feature) is defined as a distinctive user-visible aspect, quality, or characteristic of the system. Some features are common among all the products, while others features only belong to a subset of the products.

While many PLA implementation approaches exist [1, 11, 13, 16, 35], the process of automatically mapping the PLA to its implementation, especially variability implementation, remains a difficult task. Specifically, the PLA and implementation of a product line are usually developed in two separate artifacts. As they evolve, it is essential to ensure that the variation point's model and its implementation are constant in the way they vary. In this project, we present a new code generation and separation technique that can automatically implement the PLA and convert it to product line code. The approach can automatically generate different code entities for different types of variation points included in the PLA. The approach aims at reducing the PLA-implementation inconsistency and reducing human efforts that should be focused on developing and maintaining the PLA. We have implemented and integrated the approach in ArchFeature [20]. Moreover, we have evaluated the approach on a real chat application to validate it and assess its feasibility, performance, and affordability. The evaluation methodology and results are presented in Chapter 5.

1.2 Research Problem

The PLA is a promising application of SPLE that enables reuse and customization when deriving new products, resulting in a lower cost and higher quality [7, 33]. In particular, PLA attacks product line complexity at a higher abstraction level than source code. However, as a software product line evolves, the PLA and its implementation soon become inconsistent in terms of defined variability resulting in **unfaithful implementation of the PLA**. Moreover, current PLA implementation approaches **require manual implementation of variation points in the PLA**.

Mapping the PLA with its variation point to implementation with a goal of reducing their inconsistency are challenging tasks. While many existing approaches [1, 11, 13, 16, 35] address variability implementation in PLA, they all face the challenge of automatically mapping the PLA variation points to implementation. With the absence of a novel approach that provides variability architecture-implementation mapping, software reuse is compromised and therefore an extensive amount of manual coding is required to introduce variability in software product families. Moreover, variation points may have negative impact on the software reusability if not implemented appropriately [16]. Specifically, it is difficult to maintain separation of concerns in product line implementation corresponding to the PLA components. Several approaches address this issue such as framework plug-ins [15] and feature-oriented software development [5]. However, they are limited to certain types of variability and require special programming paradigms. Moreover, it is well-known that the PLA frequently change and evolve, which requires to reduce the inconsistency between the PLA and its implementation. Currently, this requirement can only be manually achieved since none of the existing architecture-implementation mapping approaches addresses this issue.

In this project, I address the following research question: *How can we faithfully implement the PLA with all involved variation points into source code without losing their consistency?* Answering this question with a novel approach not only addresses automatic PLA and variability implementation, but also helps to widely adopt the PLA as centric approach in the development of software product lines.

1.3 Objectives

The primary objectives of this research project is to develop an approach equipped with a tool support that can be used to automatically map the PLA to implementation. Specifically, mapping the specification of each variation point (e.g. alternative component) present in the PLA to its corresponding implementation in the source code. Moreover, the approach aims to reduce the inconsistency between the PLA and its implementation. These objectives will ultimately help adopting PLA as a central role in the development of software product lines in SPLE.

1.4 Contribution

The main contributions of this research project are:

- **A novel code generation and separation technique** that can automatically implement the PLA. It can convert the PLA specifications into product line code (implementation) with a goal of keeping them consistent.
- **An automatic variability implementation technique** that implements variability specification based on its type. It can automatically convert optional and alternative components/interfaces to code entities. This will reduce human efforts required to implement the PLA variability, resulting in a better quality and lower cost.

The main difference between my approach and other existing PLA implementation approaches lies in the artifacts that developers can manually change. My approach guarantees high level of comprehensive code protection; therefore, it allows developers to

initiate changes only in the PLA and in specific isolated portions of the code (i.e. user-define code). Substantially, the PLA implementation approach consists of a code generation and separation technique that separates the code of each architecture component into three isolated language elements (i.e. classes). The first isolated class is called *Architecture-Prescribed Code*. It includes specifications of the component's structural elements and it is fully generated by the code generator and does not permit manual code changes. The second isolated class is called *User-Defined Code*. It requires the programmer to populate it with the functions provided by the corresponding component. The third isolated class is called *Variability-Specific Code*. It is generated for alternative variation points only that can accept several variants (implementations). It requires the programmer to manually provide the code for each variant related to the corresponding component. Those three isolated classes are integrated through explicit program mechanism calls (e.g. method calls). We further explain the implementation process in Chapter 4.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Background

This section presents a brief introduction to architecture-centric SPLE, software product line, and code generation. Then, I discuss the related work in this area including the existing approaches that focus on variability implementation in PLA.

2.1.1 Architecture-Centric Development in Software Product Line

Software architecture is defined as the set of principal design decisions made about the system [34]. *Design decisions* refer to all aspects of the system under development, including structural, behavioral, and non-functional properties. *Principal* refers to a degree of importance that promotes a design decision to be included in the architecture. Usually, stakeholders are the ones who decide what design decisions are principle based on the system's goals.

Architecture-centric development has emerged as successful alternative to traditional development (based on code) and replaced the focus to the architecture as the essential role in the lifecycle of software development. Substantially, the architecture-centric development attacks software complexity at a higher level of abstraction (i.e. architecture) then source code. Architecture-centric development requires that all development changes must start from the architecture and then be automatically mapped to code through an architecture-implementation mapping tool. This goal is even more challenging in SPLE since each architecture includes not only core elements but variation

points related to multiple products. To fully achieve this goal, it is necessary to develop a novel approach that can automatically implement variation points in the PLA to implementation at code level. While there exist many approaches and practices in this area, automatically mapping variation points in the PLA to implementation is still a challenging task.

One of the early approaches in this area was Koala [28]. Koala had its own architecture-description language consisted of special constructs to capture and implement variabilities in the architecture. It used code generation in implementing the underlying architecture model. However, there exist no sufficient literature work to fully understand how Koala handles variation points in the mapping process.

Another early work was Feature Oriented Model Driven Development (FOMDD) [5]. It based of feature models that can be divided into model refinements. Implementing model refinements is also based on code generation. Generative software development (GSD) [10] is another architecture-centric approach. It can automatically select and assemble based on abstract functional descriptions. GSD reduces the product line development to composition of adoptable domain components and uses C++ templates in implementing variability inside each component. Generally, the existing approaches all face the challenge of implementing the PLA into code entities with and maintaining their consistency. Moreover, they all lack the ability to automatically implement the variation points exist in the PLA. Therefore, I have focused on addressing these two challenges in this research project, as further explained in Chapter 3.

2.1.2 Software Product Line (SPL)

Software product Line Engineering (SPLE) has emerged as a successful approach to develop a family of software products that share substantial commonalities while differing in some variabilities. This kind of a software family is usually developed within the same organization and is known as software product line (SPLs), or simply a product line. SPL advocates a high level of software reuse and mass customization. These capabilities are important in both academic and industrial fields. Substantially, the benefits of SPL can be seen in improving products quality, time-to-market, and reducing development cost. According to Software Engineering Institute [33], SPL brought outstanding benefits to software development in many ways including: improved productivity by ten times, increased quality by ten times, decreased cost by sixty percent, and decreased time to market by ninety-eight percent.

A standard software product line consists of a feature model, product line architecture (PLA), and product line code. Feature model captures both common and variable features in the problem space [23]. Each feature is defined as a distinctive user-visible aspect, quality, or characteristic of the system. PLA [7, 34] captures both commonalities and variation points of a SPL in a single monolithic architecture in the solution space. A feature can be optional, alternative, or optional alternative. An optional feature may or may not be included in a product instance. An Alternative feature is always included in the products; however, it can have different implementations (variants). An optional alternative is an alternative feature that can be included in some of the product instances. The code base contains the actual implementation of the SPL at code level.

Implementing SPL is still a mapping issue between the PLA and its implementation at code level. Therefore, the main objective of this research work is to address mapping PLA to implementation. In contrast, feature models has been previously addressed in many research projects including a previous work called ArchFeature [20], which we have used in this project to model the PLA.

2.1.3 Code Generation

Architecture-centric development requires a non-trivial amount of code generation [13, 42]. In addition, adopting architecture-centric development in implementing SPLs brings new challenges that have not been well addressed by any other related work that I am aware of, by the time of writing this thesis. Therefore, my work focuses on implementing the PLA with code generation and separation, where the major tasks are to generate, separate, and integrate code to map the PLA to its implementation and to maintain their conformance.

Existing code generation techniques treat source code differently. Code can be treated as a model, program, or plain text. Code generators that treat code as model, such as Eclipse ATL, require a definition of meta-model which becomes very expensive in large and complex systems, not mentioning SPLs. This interferes with the fact that SPLs are meant to reduce development cost. Therefore, this approach of code generation is challenging in the context of generating code for the PLA. The code generators that treat code as program can only generate structural constructs such as classes and methods and therefore cannot be fully used in generating code when implementing the PLA. In contrast, code generators that treats code as plain text (template-based code generation) are independent of the target

language [14]. Therefore, they simplify the process of generating code regardless of the target code or infrastructure.

My research approach is based on the third type of code generation technique, template-based code generation. Specifically, I have used a model-to-text code generation engine that is built as a plug-in into Eclipse, called Java Emitter Templates (JET2). Substantially, JET2 generates files from a model using templates. Each template consists of a collection of target text (e.g. source code to be generated) and control tags (i.e. commands). The generated code can be Java, C, HTML, or even un-executable documents based on what is defined in the templates. The main template in any JET project is “*main.jet*”, which serves as a main entry point that invokes other templates using control tags. Tags are used to extract information from a model (e.g. XML model) and have the capability to navigate those using XPath expressions. These tags are used in JET templates to control the code generation process. Tags in JET are categorized in four main libraries as follows. Control tags: used to access input models and control templates execution. Format tags: used to alter the format of text in templates. Java tags: special tags used for generating Java code. Workspace tags: used for creating projects, folder, files, etc. I will further explain the code generation and separation technique developed in this research work in Chapter 3.

2.1.4 ArchFeature: A PLA Modeling Environment

In this project, I have used a PLA modeling tool called ArchFeature [20] to develop the PLA model. ArchFeature models the PLA as components connected via explicitly defined interfaces. It provides a graphical development environment that combines product line

features, PLA, and product line code in a single tool. A major benefit of ArchFeature that it provides automatic support for managing and visualizing variation points in the PLA.

ArchFeature models the PLA and its related features in a single xADL model. Figure 1 shows an overview of ArchFeature and its underlying PLA model. The top half of the figure shows the ArchFeature main interface, where a monolithic PLA and features are developed side-by-side in the same environment. The bottom half of the figure captures part of the PLA model, which includes specifications of one feature and a related PLA component. I have used this tool in my project to help me focus on achieving the main objectives of the project rather than wasting time on modeling the PLA and managing its variation points.

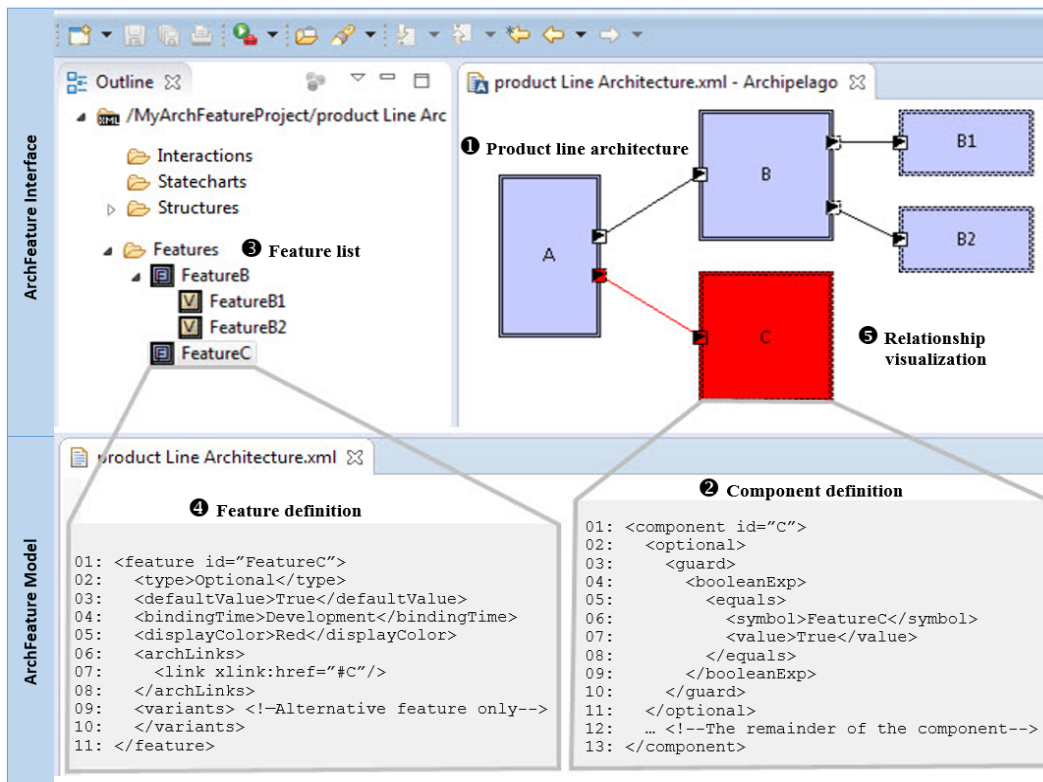


Figure 1. ArchFeature modeling environment

2.2 Related Work

In this subsection, I review and discuss the related work in this area to explain why there are no sufficient work to support faithful implementation of the PLA. Moreover, the current approaches require manual implementation of variation points in the PLA.

A number of variability implementation techniques exist. They have been characterized in specific ways in the existing literature [4, 10, 12, 17, 21, 32]. Table 1 classifies the existing variability implementation techniques based on their focal levels of reuse. I can conclude from the table that only several techniques support variability at the architecture component level and each can only be used in some specific situations as analyzed below.

Table 1. Comparison between existing variability implementation approaches

Reused Element	Introduced Variability	Implementation Techniques
Component	optional interface, alternative components	Component frameworks [6, 37], code generation [16, 36]
Group of classes (sub-component)	optional operations for all classes, new participants (i.e. classes)	Composition-based techniques: design patterns [18], feature-oriented software development [5], aspect-oriented programming [25]
Class	New fields, new methods, redefined methods	Programming language techniques: inheritance, overloading, generic programming, nested inheritance [27], open classes [9]
Method	Parameters of different types	
Line of code	optional statement, method signature change, expression change	Annotation-based techniques: metaprogramming [31], architecture-based code annotations [39]

Annotation-based techniques use annotations or metadata to represent software variability into source code, which can be processed by an annotation processor to create a single product. A novel technique in this area is metaprogramming. The annotations of

metaprogramming are special words (i.e. domain-specific languages) associated with a meta-program. Annotation processing in this case involves program transformation, such as inserting additional statements in the middle of a method. A main issue in this technique is that it can introduce type errors to the resulting program. In addition, automatically maintaining and processing those annotation is a difficult task. A novel architecture-based code annotation technique is introduced in [39]. It has the ability to automatically map variation points the PLA to code entities (e.g. a single line of code). A number of programming language techniques have been developed (e.g. nested inheritance, open classes) to address software reuse. In general, most can only support reuse at the level of a program element (e.g. method, class). However, those are all general purpose techniques and cannot address different types of variability in the PLA.

Composition-based techniques introduce variability in a group of classes or a subcomponent. Design pattern focuses on object composition over class inheritance. A number of patterns exist, such as visitor, factory, and observer. Each can support a specific type of variability. For example, the visitor pattern is usually used to add a new operation to a group of classes, whereas the observer pattern makes it easy to include a new observer (i.e. class). Feature-oriented software development (FOSD) is a novel technique that emphasizes composition of features. Each feature is implemented in an independent module that contains related fragments of a group of classes. Different feature modules are then superimposed to generate a single product. FOSD enforces feature modularity and superimposition on all features, and this may cause the issues related to granularity and crosscutting features. Aspect-oriented programming (AOP) is similar to FOSD in the sense

that it also emphasizes modularization of crosscutting concerns. AOP relies on some special constructs to support the composition process. Overall, AOP, FOSD, and the related approaches face the challenge of module composition, which can be difficult when the number of involved classes and their interdependencies significantly increase.

Component frameworks, code generation, and adaptor/wrapper are three techniques that support variability at the architecture component level. A software framework is a piece of predefined code that can be specialized with application-specific code to produce various applications. When applied in SPLE, a framework is defined as a set of classes embodying an abstract design for solutions to a family of related problems. A component framework implements default services that are relevant to all members of a product family, and can be extended to introduce variability in two different ways: subclasses (i.e. white-box frameworks) and plug-ins (black-box frameworks). A limitation of frameworks is that they mainly support functional differences among the products that can be implemented as plug-ins only. Code generation addresses variability typically by generating different code. The programmer's manual implementation is usually needed to complete application-specific logic. However, the existing code generation approaches offer little support or guidance at this point. Adaptor/wrapper can only be used to implement interface-related variations. They do not support variations that affect the inside structure or implementation of a component.

CHAPTER 3

APPROACH

This chapter presents a novel code generation and separation approach that focuses on implementing the PLA with a goal of reducing the PLA-implementation inconsistency. In particular, the chapter discusses in details how the approach can implement the PLA and how it can automatically implement the different types of variation points using different techniques. Before discussing the approach details, we present an overview to briefly discuss the basics of the approach.

3.1 Overview

I have developed a novel code generation and separation approach that can automatically implement the PLA and its variation points with a goal of reducing the PLA-implementation inconsistency. Specifically, the technique generates a separate set of code entities for each variation point in the PLA based on its type. The approach focuses on variation points modeled as alternative components and alternative interfaces. In general, the technique maps every element in the PLA (e.g. component) to two code entities (i.e. classes): an *architecture-prescribed code* and a *user-defined code*. A third class will be also generated in case of alternative variation points, *Variability-specific code*. The architecture-prescribed code is fully generated and it includes code that describes the corresponding architecture element. Manual changes are not allowed in this class and it can be updated via code regeneration only. We focus on this class to maintain the PLA-implementation conformance as explained in this chapter.

xADL is used as the architecture description language to model the PLA. Java is used as the programming language in this project. The modeling and implementation processes took place in ArchFeature, an open-source Eclipse-based PLA development environment. The developed approach focuses on variation points that represents optional and alternative components and interfaces that capture the differences among different product instances. However, it does not support mutual feature dependencies, other variability types, architecture properties, or other architecture elements (i.e. connectors). They are not directly related to the PLA implementation mapping and have been addressed by other implementation techniques. Moreover, our approach does not consider behavioral variation points, because they are usually not complete enough to be used in code generation.

3.2 Code Generation and Separation

The technique generates and decouples different code entities based on the PLA element's type. The approach parses the underlying *xADL* model of the PLA to identify the structural elements and their types. A PLA element (i.e. component, interface) can either be a core element or a variation point. Each variation point is defined over at least on product line feature. A variation point can be optional or alternative. Core and optional elements in the PLA are implemented in different ways than alternative components. Figure 1 illustrates the code structure of a PLA component implemented using this approach. (1) represents architecture-prescribed code, (2) the user-defined code, and (3) variability-specific code. The parallel lines represent program interfaces as further explained below.

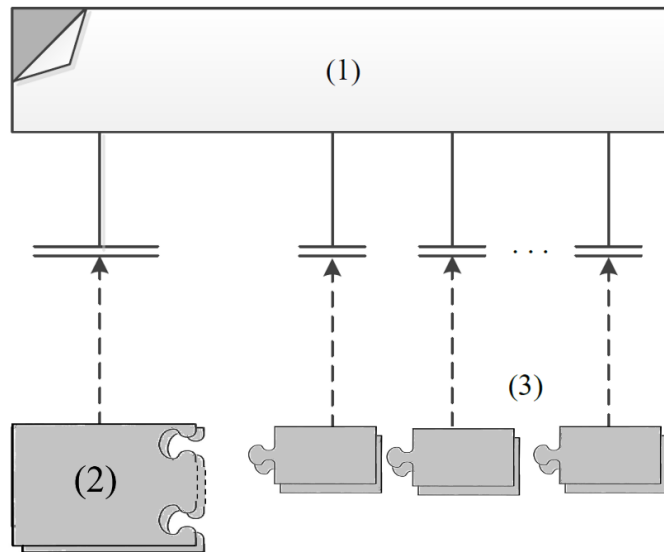


Figure 2. Implementation structure of a PLA component

- (1) *Architecture-prescribed code*: a separate class generated from the architecture specification of both core elements and variation points including optional and alternative types.
- (2) *User-defined code*: a separate class written by the programmer that implements the component's core and optional elements and functions. User-defined code may have multiple variants of implementation in case of alternative complements.
- (3) *Variability-specific code*: a number of separate classes written by the programmer that contains manual implementations of alternative variants defined as provided alternative interfaces in the PLA. This variability-specific code can exist to support alternatives of function-specific quality goals or implementations requirements (e.g. specific libraries). This increases the flexibility, code reusability, and customization.

The parallel bars in the figure represent the program interfaces that are also automatically generated. Each of them contains operations related to either core, optional, or alternative functions that are implemented by the corresponding user-defined class. The overlapping boxes means that alternatives of manual implementations can exist. The puzzle-like shape in (2) means that a component can include one or more alternative provided interfaces. The implementation of each alternative interface is isolated into a separate class, variability-specific code shown in (3). It allows the programmer to manually write its implementation. Each alternative interface may have at least one variant.

Note that this structure varies based on the way the variation point is defined in the PLA. Specifically, when an alternative function with different variants is defined as a separate alternative component (see alternative components in figure 2), then only an architecture-prescribed code and multiple user-defined code classes will be generated. In this case, we do not need to generate variability-specific code classes since each user-defined code class will represent a different variant implementation.

The code structure used in our approach is different from other approaches in the way they achieve code separation. Our approach separates each element in a complete isolated class integrated through regular program composition techniques such as interface implementation. The folded corner means that different code generation templates can be used to address product differences in the implementation platform. This is one of our future work. Another future work is to support *optional alternative* variation points. We believe that the presented code structure, if modeled correctly, with the use of ArchFeature capabilities, we can easily address the optional alternative variation points.

3.2.1 Implementing Core and Optional Elements of the PLA

The code generation and separations separates the implementation of each core/optional architecture component into two independent classes: architecture-prescribed code (aka generated code) and user-defined code (aka non-generated code). Program composition mechanism such as method calls are used to explicitly integrate those separated code entities. In particular, the user-defined code implements the application-specific operations of the product line. The available architecture resources are defined the architecture-prescribed code.

The architecture-prescribed code is automatically generated from the component's specification in the architecture. It includes the operations declared in the component's provided interfaces and variables referring to its required interfaces. All the operations in the generated code are defined by redirecting the request to the user-defined code. The generated code cannot be manually edited by the programmer and can only be updated via code regeneration. This prevents the mistaken changes of the architecture-prescribed code and increases consistency between the architecture and code. Moreover, a program interface is automatically generated for each core/optional component. The interface contains all the operations that need to be manually implemented, including the operations that are optional and correspond to different product line features.

The user-defined code implements the program interface and contains the implementation details manually developed by the programmer for the operations included in the interface. It can be seen as the internal implementation of the component that contains implementation details that are not specified in the PLA such as application-specific

functions, algorithms, libraries, web-services, etc. Unlike the architecture-prescribed code that defines external visible characteristic of the component, the user-defined code represents the internal implementation of the component. When a component includes a required alternative interface, the code generation technique will handle the implementation in a different way, as explained in the next subsection.

3.2.2 Implementing Alternative Elements of the PLA

The code generation and separation technique handles alternative variation points differently from other core and optional elements in the PLA. An alternative variation point means that it can have multiple implementations (variants). An alternative variation point can be modeled in the PLA as an alternative component or an alternative interface.

Specifically, when the variation point is modeled as a component, the code generation will generate a single architecture-prescribed code with multiple user-defined code entities based on the number of variants defined by the developer (see Figure 3). Each user-defined code implements a single variant. Only one variant of an alternative component will be included in any product instance at a time. When a specific variant is selected, the architecture-prescribed code will be updated to refer to the selected user-defined code based on program method calls, as explained earlier.

When the variation point is modeled as an alternative provided interface, contained in a core component, the code generation and separation technique will separate the implementation of the alternative interface from the involved components' user-defined code and create multiple variation-specific code entities (see Figure 4). In addition, a separate interface includes the operations of the alternative interface will be automatically

generated. Each variation-specific code entity will implement a different function-specific quality or implementations requirements (e.g. specific libraries) related to the alternative variation point. The number of alternative is specified by the developer. Only one alternative will be included in one alternative provided interface at a time. This increases the flexibility, code reusability, and customization support.

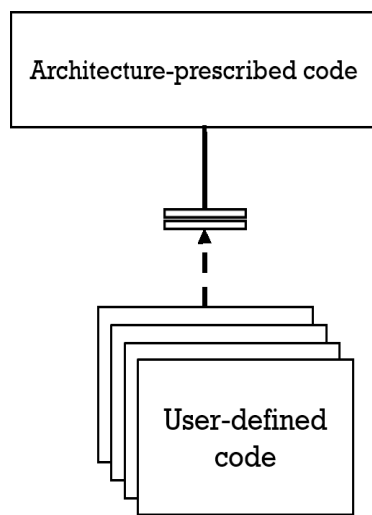


Figure 3. Implementation structure of alternative components

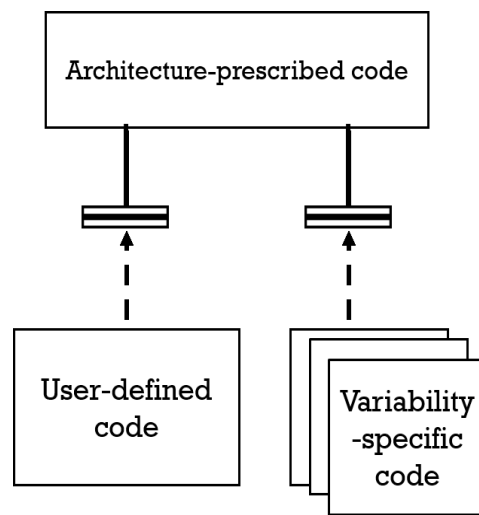


Figure 4. Implementation structure of a component including an alternative interface

A primary advantage of the code generation and separation technique is that the generated code can be automatically updated when the PLA is changed to maintain the PLA-implementation conformance. In addition, the generated code cannot be manually modified, which increases consistency between the PLA and its implementation. Meanwhile, the programmer's manual work will not be overwritten during code regeneration.

3.2 Example of the PLA Implementation

In the following two figures (Figure 5 and Figure 6), we present a PLA example of a text-based chat application (Figure 5) along with its corresponding code implementation (Figure 6) implemented using our approach to further explain it.

Figure 5 illustrates the PLA of the chatting application that we modeled in ArchFeature. The PLA is configured as components connected via interfaces. The interfaces pointing outwards the components are *required* interfaces that contain operations required by the component they belong to. Those required interfaces are connected to *provided* interfaces pointing inwards the component. They provide the operations implemented by their components. The component in Figure 5 that has two inner components represent an alternative component containing two variants. In this example, the alternative components provide an alternative functionality to save chatting history either as plain text (first variant) or multimedia (second variant). This component is drawn in dashed lines to distinguish it from other core components. All elements drawn by dashed lines represent variation points (e.g. *PlayGame* component).

The PLA includes core, optional, and alternative variation points. For example, one optional variation point includes the functionality of playing a game (i.e. *PlayGame* component). Another alternative component represents the functionality of saving history. This component has two variants (i.e. *plain text*, and *multimedia*) as we explained earlier. Also an alternative interface to share emojis (i.e. *SndEmoji*). This alternative interface includes two variants each implements different set of emojis (e.g. plain text and sad emojis) and only one set is included in the chat application at a time.

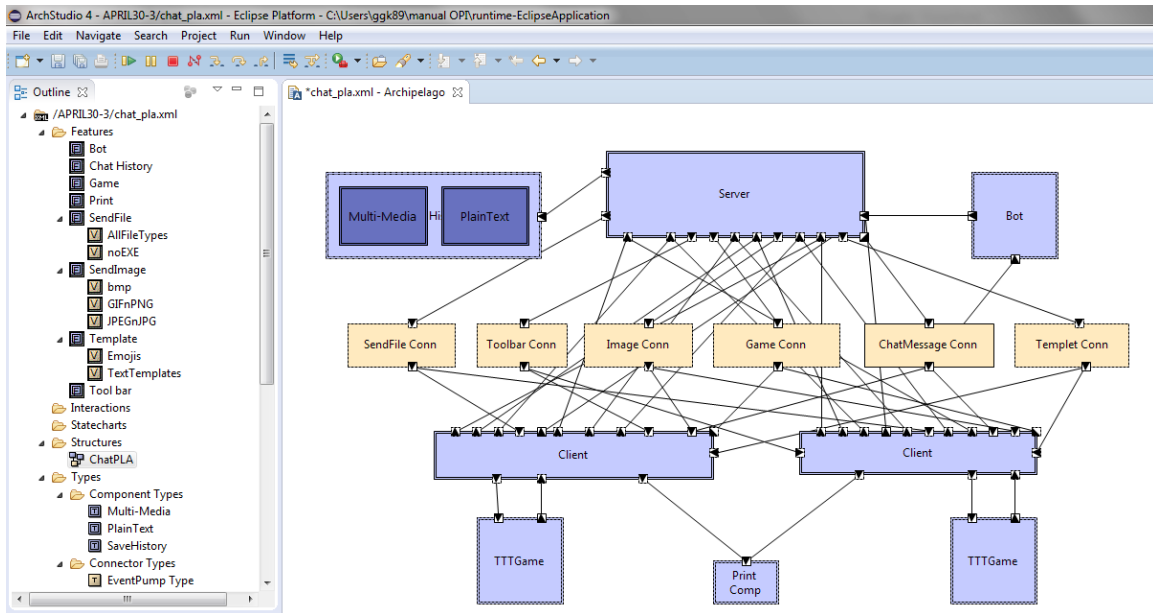


Figure 5. PLA example of a chat application implemented in ArchFeature

Figure 6 illustrates a code implementation example of the component *Server* shown in Figure 5. Class *ServerArch* in Figure 6.a is an automatically generated class from the component's architecture specification, it is the architecture-prescribed code of the component and it includes:

- Declaration and initialization of references to its manual implementations (Lines 02-03) and references to connected components (Lines 04-05)
- Implementation of operations defined in its provided interfaces (Lines 12-20), including core, optional, and alternative provided interfaces.
- Myx architecture framework life-cycle methods (Lines 06-11)
- More importantly, it prevents users from accidentally manipulating architecture code, which ultimately results in maintaining architecture implantation conformance.

Class *ServerImp* in Figure 6.c is manually written by the programmer and it implements both core operations (*SndMsg*) and optional operations (*DisplayDate*). *Iserver* is the program interface and is automatically generated. The interface *ISndEmoji* in Figure 6.d is an alternative interface that implements alternative operations (*sendEmoji*). *SendEmoji* can have one of two implementations (variants) to send two different sets of emojis (e.g. happy and sad emojis), see figures 6.e and 6.f. Each variant is isolated in a complete separated class. In the code regeneration process, only the architecture-prescribed code will be generated to ensure and keep the architecture implementation conformance, while keeping user-defined code unchanged.

The only challenge we can observe from Figure 2 is that optional and alternative interfaces are currently modeled as optional interfaces. This is because ArchFeature uses xADL as modeling language, which does not currently support modeling alternative interfaces. This challenge can be easily addressed by extending xADL language and implementing new constructs to support modeling alternative interfaces. This challenge is not considered in this research project because it is not directly related to architecture-implementation mapping process.

Our focus domain is structural PLA representing both core components and variation points. Variations in a software family can be one of three types: optional variation, alternative variants, or optional alternative variants. Our approach will support both optional and alternative variants in the PLA whether they are represented as components or interfaces.

```

01 class ServerArch extends MyxComponent {
02   Iserver _imp = getCoreImp();
03   ISndEmoji emojiImp = getEmjoilmp();
04   IFwdMsg out1;
05   IChatHistory out2;
06   public void init() {
07     out1 = MyUtils.getService("FwdMsg");
08     out2 = MyUtils.getServivce("ChatHistory");
09   }
10   public void destroy () {
11     .. //other life cycle methods
12   public void sndMsg (String msg){
13     _imp.sndMsg();
14   }
15   public void sndEmoji () {
16     emojiImp.sendEmoji();
17   }
18   Public void DisplayDate (){
19     _imp.DisplayDate();
20   }
21 }

```

Figure 6.a Architecture prescribed code of "Server"

```

01 interface Iserver {
02   public void sndMsg(String msg);
03   Public void DisplayDate(Date d);
04 }

```

Figure 6.b Server interface

```

01 class ServerImp implements Iserver {
02   ServerArch _arch;
03   public void sndMsg(String msg) {
04     _arch.out1.fwdMsg(String msg);
05     // Alternative function Save History as text
06     _arch.out2.saveMsg(msg);
07   }
08   Public void DisplayDate(Date d){
09     //Display date code implementation..
10   }
11 }

```

Figure 6.c User-defined code of "Server"

```

01 Interface ISndEmoji {
02   Public void sndEmoji(emoji e);
03 }

```

Figure 6.d Alternative function interface

```

// Variant 1 that implements sending emojis
01 class SndEmojilmp implements ISndEmoji {
02   ServerArch _arch;
03   Public void sndEmoji(emoji e) {
04     // implementation of sad emojis
05   }

```

Figure 6.e. Alternative variant 1 of sndEmojis

```

// Variant 2 that implements sending emojis
01 class SndEmojilmp implements ISndEmoji {
02   ServerArch _arch;
03   Public void sndEmoji(emoji e) {
04     // implementation of happy emojis
05   }

```

Figure 6.f Alternative variant 2 of SndEmojis

Figure 6. Example of implementing a core component containing variation points

CHAPTER 4

IMPLEMENTATION

The PLA implementation approach is equipped with tool support built in ArchFeature, which is itself a built and integrated in ArchStudio 4 [3]. ArchStudio is an open-source Eclipse-based software and systems architecture development platform. In this chapter, we first briefly introduce Eclipse and ArchStudio, the main environments that are used to implement our approach. Then, we discuss our core implementation tasks and address the main challenges faced us during the implementation phase.

4.1 Implementation Environment

The approach runs in an integrated development environment (IDE), which facilitates the communication and information exchange between different tools at different abstraction levels. Specifically, we use ArchStudio 4 to implement our tool. ArchStudio is implemented as a plug-in on top of Eclipse. Implementing our tool in such environments will help integrating and distributing our novel approach. The rest of this section focuses on Eclipse, ArchStudio, and their main features that are used and reused to implement our tool.

4.1.1 Eclipse

Eclipse is an open-source platform used for building and managing applications and web tools. The key value of Eclipse lies in encouraging rapid development based on extensible plug-in system. Plug-ins can be developed and delivered separately and can run unchanged on any supported operating system. Plug-ins are the smallest units of Eclipse.

Some small tools can be developed as a single Eclipse plug-in, while other tool's implementation is separated across several plug-ins. Common examples of Eclipse plug-ins include Eclipse Java Development tools (JDT) for java, the JUnit testing framework, Subclipse, and ArchStudio 4, that we also used in our implementation and it is further explained in the next section.

The Eclipse platform is widely used and adopted by many programmers. The main reason is the plug-ins and their integration and interaction mechanism. Simply, each plug-in can interact with other plug-ins through extensions and extension points. Moreover, plug-ins declare their interaction in a special manifest file. This mechanism advocates plug-ins reuse and extension. Even Eclipse itself is built as several plug-ins integrated together to build up the Eclipse platform. Figure 7 shows Eclipse platform structure. All the components shown in the figure are implemented as one or more plug-ins. Following are the major run-time components of eclipse:

- *Platform runtime*: it is implemented using OSGI framework. It dynamically discovers all plug-ins and maintains their information in a platform registry. All Eclipse basic functionalities are plug-ins built on top of the platform runtime (kernel).
- *Workspace (resource management)*: it's the plug-in responsible for creating and managing resources such as projects, folder, files, etc.
- *Team*: a set of plug-ins responsible for providing code control management system and version control.

- *Workbench*: it implements the user-interface to navigate the platform in addition to the non-UI specific tasks related to Eclipse itself including editors, views, actions menu actions. Moreover, it provides additional toolkits such as SWT and JFace for building user interfaces.
- *Help*: it is the plug-in that provides help documentation related to Eclipse.

Various utility-specific plug-ins also exist including debug support, searching and comparing resources and other plug-ins created for specific tasks such as Java Emitter Templates (JET) and ArchStudio that we used in implementing our approach.

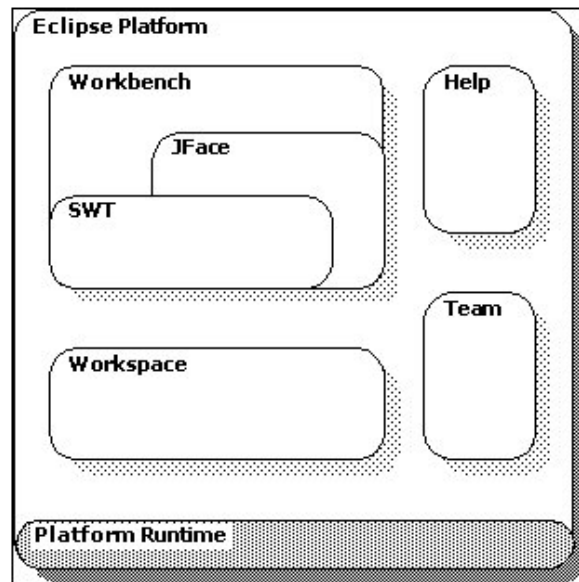


Figure 7. Eclipse platform structure

4.1.2 Java Emitter Templates

Java Emitter Templates (JET) [14] is a model-to-text code generation engine that is built as a plug-in into Eclipse. JET2 were used in xMapper to reliably map software architecture to implementation; however, the previous templates used in xMapper were

meant to be used in developing software architecture, not PLAs. Therefore, we extended and developed new JET templates that support mapping PLAs to implementation.

Substantially, JET generates files from models using templates. Each template consists of a collection of target text (e.g. source code to be generated) and control tags (i.e. commands). The generated code can be Java, C, HTML, or even un-executable documents based on what is defined in the templates. The main template in any JET project is “main.jet”, which serves as a main entry point that invokes other templates using control tags. Tags are used to extract information from models (e.g. XML) and have the capability to navigate those using XPath expressions. Tags in JET are categorized in four main libraries. Each of them has specific usage as follows:

- *Control tags*: used to access input models and control templates execution
- *Format tags*: used to alter the format of text in templates
- *Java tags*: special tags used for generating Java code
- *Workspace tags*: used for creating projects, folder, files, etc.

Examples of these tags include <c:get> (to write out the result of an XPath expression into the generated code), <c:iterate> (to traverse a set of elements in the input model and execute the tag body once for each iteration), and <ws: folder> (to create a workspace folder). It is important to mention here that JET does not permit embedding one XML tag in an attribute value of another. However, it permits the usage of XPath to navigate input models. For example, the full tag of creating file is given as follows:

```
<ws:file template="..." path="{XPath expression}"/>
```

In addition, JET can be extended to include user-defined libraries to add new code-specific generation requirements. All the aforementioned features of JET makes it a reliable code generator that we can use in implementing 1.X-Liner Mapper v2.0. We will further discuss our contributions in JET later in this chapter.

4.1.3 ArchStudio

ArchStudio is an open-source architecture development environment integrated into Eclipse as a plug-in. It supports developing, visualizing, and analyzing software and systems architecture models using xADL. ArchStudio provides a number of tools that support essential activates of architecture-centric development. For example, *Archipelago*, *ArchEdit*, and *AIM Launcher*. These tools can be extended to address other architecture development concerns. Moreover, we can build new tools and easily integrate them in ArchStudio to address new concerns. In addition, ArchStudio is equipped with a tool that allows and facilitates extending (defining and re-defining) new xADL constructs for stakeholders to suite their own needs. These extensibility capabilities make ArchStudio an ideal open-source platform for implementing our approach without the need to “re-invent the wheel” and give us more time to focus on PLA-implementation mapping.

Before this research project, ArchStudio did not have tool support for implementing PLA. With our tool built and integrated, ArchStudio is now upgraded in terms of support for PLA implementation.

Archipelago is the graphical editor in ArchStudio (Archipelago’s main editor interface is displayed in Figure 8 surrounded by dashed rectangle). It provides a user-friendly interface based on “boxes-and-arrows” style and it focuses on structural

architecture modeling. Archipelago supports visualizing architecture elements such as components, interfaces, connectors, and links that are described in the underlying xADL documents. Unlike other similar editors, Archipelago is cognizant of the underlying modeling language, xADL, and supports live synchronization between diagrams and their corresponding xADL documents, which means that any changes made in the diagrams are immediately reflected in the xADL document and vice-versa.

With regarding to modeling product line architectures, the current version of Archipelago has the capability to visualize alternative variation points as a single component that combines together the corresponding variants (see *SaveHistory* component in Figure 5). This feature is very helpful for this project to represent alternative variation points in PLA. Unfortunately, neither modeling nor visualizing alternative interfaces is supported in the current version of Archipelago, which is necessary for our project. However, thank to features support present in ArchFeature, we were able to address this problem. Simply, we modeled alternative interfaces as optional interfaces connected to alternative features. In this way, alternative interfaces can be visually distinguished from optional ones and have different implementation criteria that we explained earlier. Extending Archipelago to model and visualize alternative interface has been made a future work.

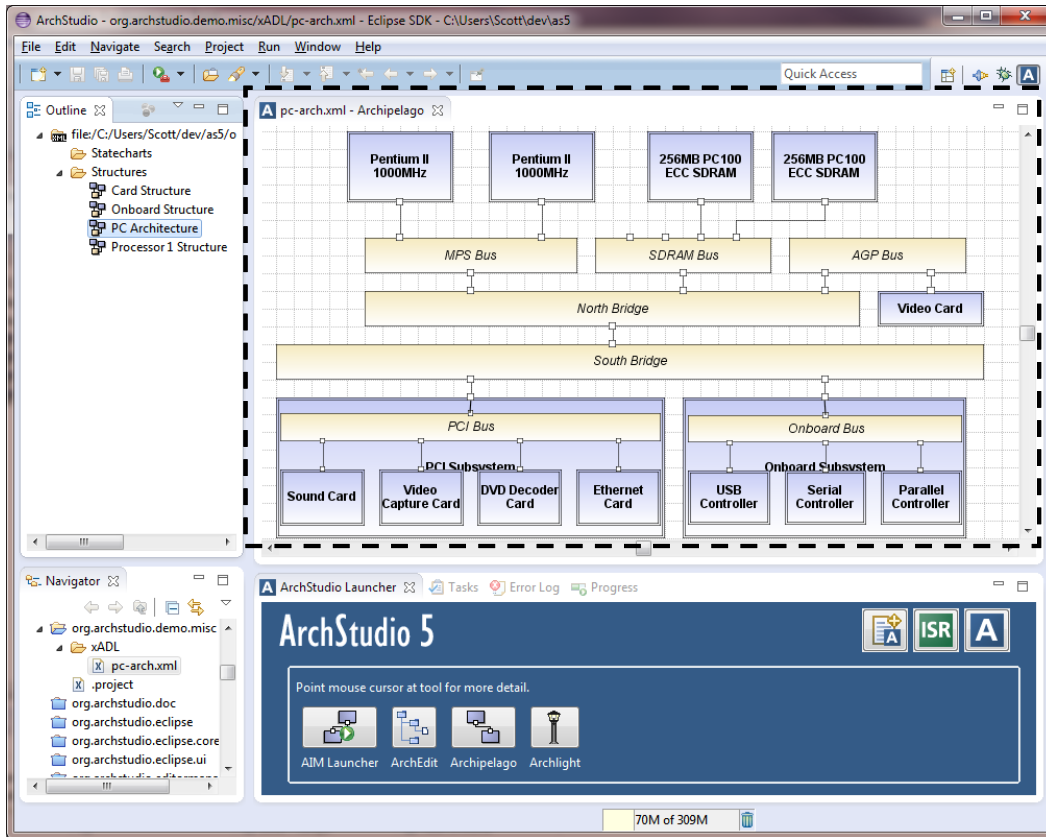


Figure 8. Screenshot of ArchStudio main interface [3]

ArchEdit is a graphical tool used to visualize and edit architecture specification written in xADL documents. One of the main features of this tool is that it is automatically populated with the architecture description in a tree format, where each node can be edited in a user-friendly interface. Another important feature of ArchEdit is that it can automatically adapt itself to new xADL schemas when the language is extended without the need to any changes in ArchEdit. We used ArchEdit to parse the PLA architecture looking for alternative components and interfaces that will be later used in generating the implementation code.

4.2 Implementation Tasks

In this section we introduce and discuss the implementation of our tool. Specifically, extending the previous code generation templates used in xMapper [40, 41] and introducing new templates that are focused on addressing the concerns of mapping PLA specification to implementation.

In this project, we focus on mapping specification of variation points (e.g. alternative components) to their corresponding implementation at code level. The current code generation templates used in xMapper cannot parse variation points in the architecture and map them into corresponding implementation at code level.

Implementation tasks included implementing three different groups of templates that varies based on the type of architecture elements. Our project focuses on architecture components and interfaces. Each one of them can either be a core, optional, or alternative element. We developed new templates to address the issues of mapping alternative variation points (i.e. components and interfaces) to their corresponding implementation. In addition, we extended the existing templates to solve the issues of mapping core and optional components. We further explain our implementation tasks as follows.

4.2.1 Implementing Optional and Core Components

Implementing core components included generating architecture-prescribed code, user-defined code, and a java interface for each separate architecture element. We reused and extended some of the templates developed in xMapper for this task. This implementation task was divided into two parts based on the type of architecture elements. Particularly, we extended those templates that were focused on generating code for core

architecture elements. Since core elements have been already addressed by xMapper, we were able to reuse some of those templates in addressing the process of implementing optional elements in a similar way.

4.2.2 Implementing Alternative Components

Alternative components define a set of possible variants (implementation). Each alternative component is modeled as a component consisting of inner components each one representing a variant. Figure 5 includes an example of a variant component (*Save History*) that is responsible to save the history in the chat application. This save history function can have two possible variants including saving the history as plain text (i.e. Plain Text) or saving the chat history including shared pictures, emoji, and videos (i.e. Multi-Media). Alternative components are surrounded by dashed lines to distinguish them from other optional and core components. They are accompanied by guard conditions that are mutually exclusive, which means that an alternative component can only resolve (implement) one variant at a time. When the condition of one variant type is met, the implementation of that variant will be considered as a default implementation of the alternative component.

Generating code for variant components required developing a new set of templates to handle mapping variant types of components to implementation. Specifically, we developed three new templates to map each alternative component to its implementation. Those templates are explained as follows:

- *AlternaiveCompArch.jet*: responsible for automatically generating architecture specific code for the alternative component and its interfaces (i.e. Architecture-prescribed code class). The name of the class is provided by the user.
- *AlternativeCompImp.jet*: responsible for generation a User-defined class for each alternative component. This template iterates one alternative component at a time looking for variants to create one separate implementation for each variant during one iteration. After creating the alternative implementation class, the template will populate it with the methods existing in the provided interfaces to be completed by the programmer. However, only one implementation can be resolved (executed) in any instance of the application at a time based on the user selection. The name of each user-defined class is derived from the alternative type provided by the user.
- *IAlternativeComp.jet*: responsible for generating an interface between architecture-prescribed code and the default user-defined code. It contains a list of specific operations that architecture-prescribed code expects user-defined code to provide. The name of this interface is provided by the user when he creates the interface type.

Before executing any of the previous templates, the code generator engine will parse the architecture looking for alternative components. Once it finds an alternative component, the code generator will run the then *AlternaiveCompArch.jet* to generate architecture prescribed code, then it will execute the *IAlternativeComp.jet* template to generate the corresponding interface. Then, the code generator engine will iterate that

component looking for its variants. For each variant, the code generator will run the *AlternativeCompImp.jet* template to generate user-defined code for that specific alternative. When the engine iterate all the alternatives it moves to the next component and repeats the previous steps.

4.2.3 Implementing Alternative Interfaces

Based on the way the architecture was modeled, some alternative functions could be implemented as alternative interfaces. The current version of ArchFeature does not support mapping those alternative interfaces to implementation. Practically, the current version treats all alternatives as optional variation points. This requires extensive time and coding efforts to introduce alternative interfaces with multiple possible variants within the same implementation (user-defined code). Therefore, we developed a novel approach that separates each provided alternative interface from the core implementation into a separate class with a separate interface to address the issue of having different function-specific goals or implementation requirements (e.g. specific library) for the same function. This solution will facilitates the process of developing PLA and enhances the process of mapping variation specification to implementation.

As we explained earlier, the current version of Archipelago does not support modeling alternative interfaces. Therefore, we had to think of a practical way to present them in the architecture. We used the power of features functionality that were previously developed in ArchFeature to define alternative interfaces with multiple variants. Therefore, we were able to create alternative interface with as many variants as required by our interface. Since the feature represents an alternative function, only one variant will be implemented at a

time. Again, using ArchFeature functionality we can assign any of the variants as the default variant. In this way, our code generator can extract all alternative interfaces and learn about their different implementations (variants) and then implement the default ones.

Implementing alternative interfaces required developing new JET templates to automatically generate code from the alternative variation points in the architecture and map them to implementation at code level. Generating code for alternative interfaces not only requires parsing the architecture elements, but also parsing the feature list looking for alternative features since each alternative interface must be related to an alternative feature. Specifically, our code generator will iterate all the interfaces of each component in the architecture searching for provided alternative interfaces. Once an alternative interface is found, the code generator will parse the related alternative feature in the feature list. Then, it will learn the names (i.e. description) of each variant feature to create a separate user-defined class and interface for each one of the variants. After creating a separate implementation for each variant, the code generator will search for the default implementation and set it up to substitute the alternative interface value in the architecture-prescribed code. In this way, only one variant (the default one) will be implemented when executing an instance of the software product line. To complete the aforementioned operations we developed the following templates:

- *AlternativeInterfaceArch.jet*: generates the architecture-prescribed code of the component that holds the implementation provided by the alternative interface. A unique implementation reference is automatically generated for each alternative interface and used to implement the alternative functions

provided by that interface. Only one architecture-prescribed code is generated for each component. The name of the generated class is provided by the user using a configuration panel.

- *AlternativeInterfaceImp.jet*: the code generator uses this template to create a separate user-defined class for each variant of the alternative interface. Each user-defined class is automatically populated with the functions that are provided by the corresponding interfaces and are left empty for programmers to write the function-specific code. The name of each user-defined class is derived from the alternative feature.
- *iAlternativeInterface.jet*: generates a separate interface file for each alternative interface. The name of the file is derived from the interface type name and suffixed with capital "I".

CHAPTER 5

EVALUATION

I have evaluated the developed approach by building a real PLA of a chatting application. The PLA presents a chat application that consists of core elements (e.g. client, server) and other optional and alternative variants. Specifically, the chat applications includes the functionalities shown in the following table.

Table 2. Features of the chat application and their types

Feature	type	Variants –if any
Send message	Core	
Save chat history	Alternative	multi-media
		plain text
Send image	Alternative	send JPEG and JPG only
		send gif and png only
		send bmp only
Send templates	Alternative	text templates
		Emojis
Send files	Alternative	accept all file types
		do not accept .exe files
Chat with robot	Optional	
Tool bar	Optional	
Print saved history	Optional	
Share location	Optional	
Play a game	Optional	

We built a full featured model of the chatting application from scratch using our approach techniques to exercise the PLA implementation functions supported by our approach. A screenshot of the architecture model, developed in ArchFeature, is illustrated

in Figure 10. We can observe in the architecture that the *SaveHistory* feature is modeled as an alternative component that has two variants, as mentioned in Table 2. Other features are represented as optional and alternative components and interfaces.

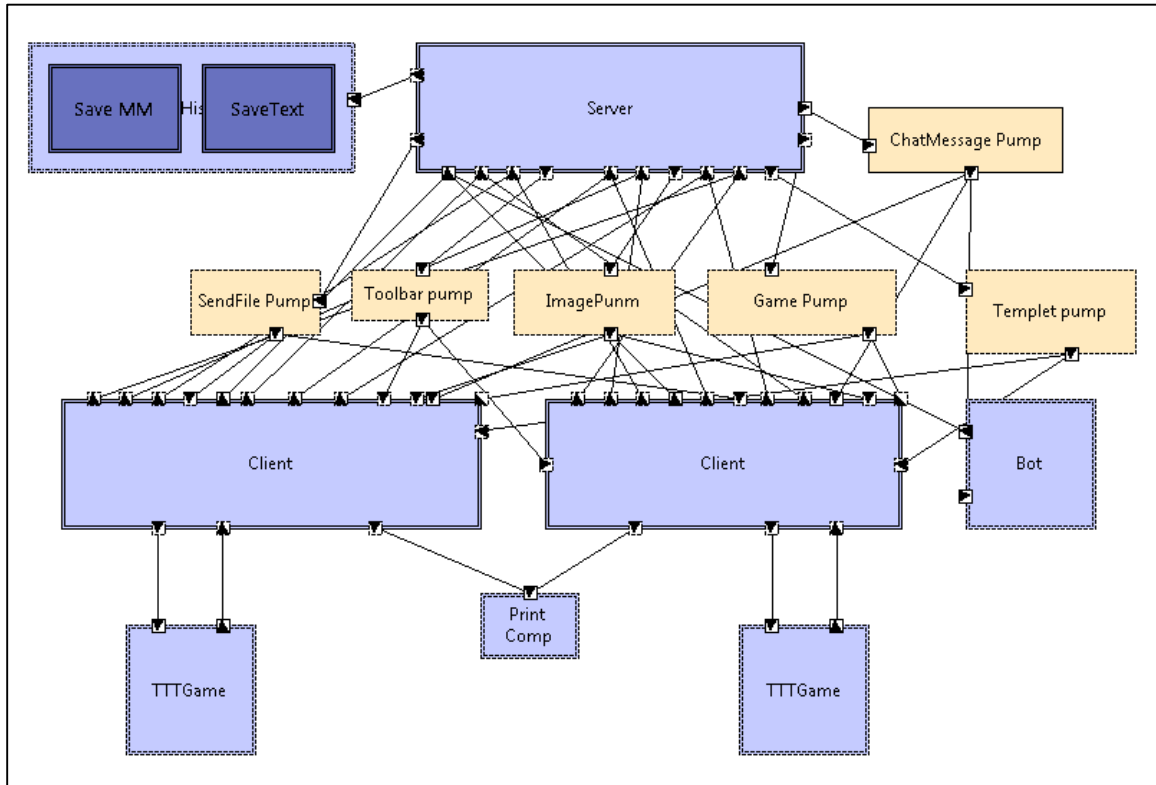


Figure 9. PLA of chatting application

5.1 Objectives

The primary goals of the evaluation are to validate the developed approach and to assess its feasibility, performance, and affordability. First, I wanted to evaluate if the presented code generation and separation technique is applicable and can be used to automatically implement the PLA including all involved types or variability. In addition, I wanted to evaluate if the technique is feasible and can be applied to the source code of a real system. Second, I wanted to assess the performance of the included techniques on a

real system to see if the performance of the related operations (e.g. implementing variation points) are acceptable. For example, the time that it takes to separate and implement a new variation point must be proportional to the size of the changes made to the PLA. Finally, I wanted to evaluate the affordability of the approach. That is, the cost of applying the approach to a system. Simply, to ensure that the approach will not impose significant changes to a system developed using traditional programming techniques.

5.2 Methodology

The evaluation process consisted of four primary steps: preparation, architecture development, functions exercising, and result analysis. Each step is further described below. At the end of the evaluation, I was able to run the chat application from its developed architecture model in the AIM Launcher tool of ArchStudio.

- i. ***Preparation.*** The first task of the evaluation was to set up a local development workspace in Eclipse for the chat application. Then, prepared the applications' code and dependencies and were able to run it locally in Eclipse.
- ii. ***Architecture development and code generation.*** I first developed an architecture model for the chat application that consisted of fifteen structural elements. After that, I developed interface types and linked them to their implementation using ArchEdit. Now, classes refer to other classes through the new interface with the same set of methods. Then, I created an architecture components that includes the interface in the architecture editor, and generate code for that component. I had to move the code from

the class into the user-defined class of the new architecture component. Similarly, I had to separate variation specific code into the new generated classes.

- iii. ***Functions exercising.*** Specifically, I exercised how implementing our approach may affect the PLA implementation in source code level. I introduced new variation points in the PLA, generated variation-specific code for them, and ran the application. Similarly, removed other variation points and observed the changes.
- iv. ***Results and analysis.*** I have examined the refactored code and the PLA of the chat application to analyze and observe the approach results. The main focus was the changes made to the application after implementing the approach and observing the amount of generated code.

5.3 Results

The architecture model that we developed for the chat application is illustrated in Figure 10. Table 2 lists the core, alternative, and optional features that were included and modeled in the application. For clarity, I divide the refactored code into following categories and analyzed it: (1) the original code from the chatting application without any changes made, (2) the code generated from the architecture, (3) the code manually developed. Note that the manual code was not developed from scratch. Instead, the majority of the manual code was already developed and integrated into the chat application existing code with some minor modifications made, for example, code refactoring for separated alternative interfaces.

After evaluating the approach on a chat application, we conclude the following:

- ✓ The developed code generation and separation technique can automatically implement the PLA into code entities, which helped in reducing the inconsistency between the PLA and its implementation.
- ✓ The developed technique can automatically map different variation points in the PLA to implementation, including optional and alternative variation points.
- ✓ The approach is **affordable** for developing PLA and converting it to implementation. Only a small portion of the code needs to be refactored and modified.
- ✓ The **feasibility** of the approach was validated by the fact that I have successfully finished the PLA development, implemented the PLA, and refactored some of the code and the application still functions correctly.
- ✓ In terms of **performance**, all the major functions we have exercised were able to complete in acceptable amount of time which was considered efficient.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this research project, I introduced a novel code generation and separation technique that can automatically implement the PLA and convert it to source code, including core and variation points in the PLA. The technique is based on a text-based code generation and separation engine that isolates the code of each element in the PLA into two classes: architecture-prescribed code and user-defined code. Moreover, the technique implements different variation points differently based on their type. In addition, the approach was implemented as a PLA-centric development implementation tool. The evaluation proved that the approach is feasible, effective, and can be adopted in implementing the PLA.

In the future, I plan to extend the current modeling environment of ArchFeature to support modeling alternative interfaces. In addition, I plan to develop a new code generation templates to involve implementation platforms and quality goals. For example, implementing the entire PLA in a different platform.

REFERENCES

- [1] Aldrich, J., Chambers, C., and Notkin, D. ArchJava: Connecting software architecture to implementation. in *Proceedings of the 24th International Conference on Software Engineering*, (Orlando, FL, 2002), ACM, 187-197.
- [2] Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J.-C., Rummler, A., and Sousa, A. A model-driven traceability framework for software product lines. *Software and Systems Modeling*, 9(4). 427-451.
- [3] ArchStudio. <http://www.isr.uci.edu/projects/archstudio/>. Retrieved August, 2015.
- [4] Bachmann, F. and Bass, L. Managing variability in software architectures. in *Proceedings of the ACM SIGSOFT Software Engineering Notes*, 26(3). 126-132.
- [5] Batory, D., Sarvela, J.N., and Rauschmayer, A. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6). 355-371.
- [6] Bosch, J. *Product-line architectures in industry: a case study*. in *Proceedings of the 21st International Conference on Software Engineering*, (Los Angeles, California, USA, 1999), IEEE Computer Society Press, 544-554.
- [7] Bosch, J. *Design and use of software architectures: adopting and evolving a product-line approach*. Addison-Wesley Professional, Massachusetts, 2000.
- [8] Clements, P. and Northrop, L. *Software product lines: Practices and patterns*. Addison-Wesley Professional, New York, 2002.
- [9] Clifton, C., Leavens, G.T., Chambers, C., and Millstein, T. MultiJava: modular open classes and symmetric multiple dispatch for Java. in *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (Minneapolis, Minnesota, USA, 2000), ACM , 130-145.
- [10] Czarnecki, K. and Eisenecker, U. *Generative programming: methods, tools, and applications*. Addison-Wesley Professional, Massachusetts, 2000.

- [11] Deelstra, S., Sinnema, M., and Bosch, J. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2). 173-194.
- [12] Dhungana, D., Grünbacher, P., & Rabiser, R. The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering*, 18(1). 77-114.
- [13] Diaz-Pace, J.A., Carlino, J.P., Blech, M.S., A., and Campo, M.R. Assisting the synchronization of UCM-based architectural documentation with implementation. in *the Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, (Cambridge, UK,2009), IEEE, 151-160.
- [14] Eclipse JET Project. <http://www.eclipse.org/modeling/m2t/?project=jet>. Retrieved March, 2015.
- [15] Fayad, M. and Schmidt, D.C. Object-oriented application frameworks. *Communications of the ACM*. 40(10). 32-38.
- [16] Forster, T. and Muthig, D. Optimizing model-driven development by deriving code generation patterns from product line architectures. in *Net.Object Days Conference*, (Erfurt, Germany, 2005). Conference Proceedings, 425-437.
- [17] Gacek, C. and Anastasopoulos, M. Implementing product line variabilities. in *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*, (Toronto, Ontario, Canada, 2001), ACM, 109-117.
- [18] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series, Massachusetts, 1995.
- [19] Garlan, D., Allen, R., and Ockerbloom, J. Architectural mismatch: why reuse is so hard. *IEEE Software*. 12(6). 17-26.
- [20] Gharibi, G. and Zheng, Y. ArchFeature: Integrating Features into Product Line Architecture. in *31st ACM Symposium on Applied Computing, Track on Software Architecture: Theory, Technology, and Applications (SA-TTA)*, (Pisa, Italy, 2016).

- [21] Hendrickson, S.A. and van der Hoek, A. Modeling product line architectures through change sets and relationships. in *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, (Minneapolis, USA, 2007), 189-198.
- [22] Heymans, P., Boucher, Q., Classen, A., Bourdoux, A., and Demonceau, L. A code tagging approach to software product line development. *International Journal on Software Tools for Technology Transfer*. 14(5). 553-566.
- [23] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., and Peterson, A.S. Feature-oriented domain analysis (FODA) feasibility study. *Software Engineering Institute, Technical Report, CMU/SEI-90-TR-21*.
- [24] Kästner, C., Apel S., Thüm, T., and Saake, G.. 2012. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology*, 21(3). Article 14.
- [25] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., and Irwin, J. Aspect-oriented programming. in *Proceedings of the 11th European Conference on Object-Oriented Programming*, (Jyväskylä, Finland, 1997), Springer-Verlag, 220-42.
- [26] Krueger, C.W. BigLever software gears unified software product line engineering framework. in *Proceedings of the 12th International Software Product Line Conference*, (limerick, Ireland, 2008), IEEE Computer Society, 353.
- [27] Nystrom, N., Chong, S., and Myers, A.C. Scalable extensibility via nested inheritance. in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (Vancouver, Canada, 2004), ACM, 99-115.
- [28] Ommering, R.v., Linden, F.v.d., Kramer, J., and Magee, J. The koala component model for consumer electronics software. *IEEE Computer*, 33(3). 78-85.
- [29] Parnas, D.L. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5(2). 128-137.

- [30] Pohl, K., Böckle, G., and van der Linden, F.J. *Software product line engineering: foundations, principles and techniques*. Springer-Verlag Berlin Heidelberg, New York, 2005.
- [31] Reppy, J. and Turon, A. Metaprogramming with Traits. in Editors *ECOOP 2007 - Object-Oriented Programming*, Springer Berlin Heidelberg, 2007, 373-398.
- [32] Rubin, J. and Chechik, M. Combining related products into product lines. in *Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering*, (Tallinn, Estonia, 2012), Springer-Verlag, 285-300.
- [33] Software product lines. <http://www.sei.cmu.edu/productlines/>. Retrieved January, 2016
- [34] Taylor, R.N., Medvidovic, N., and Dashofy, E.M. *Software architecture: foundations, theory, and practice*. John Wiley & Sons, 2010.
- [35] Ubayashi, N., Nomura, J., and Tamai, T. Archface: a contract place where architectural design and code meet together. in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, (Cape Town, South Africa, 2010), ACM, 75-84.
- [36] Weiss, D.M., Li, J.J., Slye, H., Dinh-Trong, T., and Hongyu, S. Decision-Model-Based Code Generation for SPLE. in *Proceedings of the 12th International Software Product Line Conference*, (Limerick, Ireland, 2008), IEEE Computer Society, 129-138.
- [37] Wijnstra, J.G. Supporting diversity with component frameworks as architectural elements. in *Proceedings of the 22nd international conference on Software engineering*, (Limerick, Ireland, 2000), ACM, 51-60.
- [38] Yan, H., Garlan, D., Schmerl, B., Aldrich, J., and Kazman, R. DiscoTect: a system for discovering architectures from running systems. in *Proceedings of the International Conference on Software Engineering*, (Edinburgh, Scotland, 2004), IEEE, 470-479.
- [39] Zheng, Y., and Cu, C. Towards implementing product line architecture. in *1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities*, (Austin, USA, 2016), ACM, 5-10.

- [40] Zheng, Y. and Taylor, R.N. Enhancing architecture implementation conformance with change management and support for behavioral mapping. in *Proceedings of the 2012 International Conference on Software Engineering*, (Zurich, Switzerland, 2012), IEEE Press, 628-638.
- [41] Zheng, Y. and Taylor, R.N. xMapper: an architecture implementation mapping tool. in *Proceedings of the 2012 International Conference on Software Engineering*, (Zurich, Switzerland, 2012), IEEE Press, 1461-1462.
- [42] Zheng, Y. and Taylor, R.N. A classification and rationalization of model-based software development. *Software & Systems Modeling*, 12(4). 669-678.

VITA

Gharib Gharibi

Gharibi joined UMKC in 2014 as a graduate student to pursue a Master's degree in Computer Science. His research interest includes *product line architecture and implementation*.