Achieving Product Line Architectures with Java Annotations

James Clark, Timothy Franzke, Dr. Yongjie Zheng
With help from Varun Narisetty
University of Missouri - Kansas City
December 7, 2014

Abstract

The purpose of this project is to study implementation of software product line architectures. In domain specific software engineering, software product lines may be exploited to take advantage of software architecture and code reusability. Extracting from software architecture a set of commonalities and desired features, a single software product may be reused to implement many subset products. This paper presents the findings from a case study where a single code base is used to produce a subset of product line architectures.

Keywords: Software Architecture, Product Line Architecture, Software Development, Java Annotations, Annotation Processing, Code Generation, Domain Specific Software Engineering

1: Introduction

A software product line is a family of related software products that have substantial commonality [Zhe14]. Our objective is to provide a means by which the end user can select application features from a software product line and correspondingly publish a derived version of the source code that matches their selections. Software products can be exploited to achieve economies of production [KM08]. We explore what is possible given one software product line and a survey of various software architecture and development tools [Arc,Obj]. From a single code base we are able to identify software features and derive subset applications. We generate code based on the original source, reusing only those parts that belong to the subset. This method applies to entire classes and can be executed on a granular level down to a single method or property. We even consider properties inside methods. We were provided a Java chatting application code base and the ArchStudio plugin for Eclipse IDE [Nar14].

We believe the best approach is to write Java annotations, use them to annotate the source code, and process them with a custom annotation processor. We considered the alternative approach of parsing javadoc comments and/or the Java class files themselves. The advantage annotations carry over comment or file parsing is that they are defined in the classpath and can be made available at run-time. Though an implementation of the AbstractProcessor class can be registered with the Java compiler for run-time processing, our processor is integrated with the ArchStudio feature selector. ArchStudio is a plugin for modeling and executing code from architecture but it is not possible in ArchStudio to generate architecture from source code. By integrating our annotation processor with the ArchStudio feature selector, ArchStudio will be able to automatically generate the derived product line architecture from source code and execute it.

In this paper we outline the challenges, tasks, and possible outcomes for designing a solution to the chatting application annotation processor and code generator. We first discuss related work, then we introduce our approach. We conclude with sections on team management and our project schedule.

2: Related Work

Differencing and Merging within an Evolving Product Line Architecture [CCG+14]

This paper addresses a problem that results when two different product line architectures are delivered to a customer. Changes developed separately in one product line may be deemed beneficial to the other product line but it is not a trivial task to merge the change. The authors use a method of tagging source code which resembles our approach with annotations. Their main contribution to this area of research is the development of both a merging algorithm and a differencing algorithm used together to detect changes in code and bring them together.

Code Generation using Annotation Processors in the Java Language [Hil11]

This article by Jorge Hildago is a three part series and tutorial. In the first part of the article, the author gives an overview of the java annotation. He describes common uses for already existing annotations and gives several examples. In the second section the author discusses how an annotation processor works. He shows how to build a simple annotation processor using Maven to build the project. Finally the author shows how to install the processor within javac and generate source code using Maven.

A Code Tagging Approach to Software Product Line Development [HBC+12]

The author discusses using tags to separate the different features in a product line architecture. The authors discuss the use of a feature models to help understand the variability of the architecture. With the feature model they are able to annotate the code at a granular level. This is a very similar approach we will be attempting with this project.

A Framework for Software Product Line Practice, Version 5.0 [NC+09]

For seemingly all things related to software product lines, the Carnegie Melon Software Engineering Institute maintains an industrial grade resource. Every visit to this website reveals some new facet of knowledge regarding the best practices and pitfalls of developing software product lines.

3: Developing the Processor

This is a fundamental task list for the primary objective of our project. Following this list is a detailed breakdown of each item.

- Install Archstudio
- Import Chatting Application
- Open Chatting Architecture
- Run Chatting Application
- Identify features in running application
- Extract block and class diagrams
- Identify within which block, class, method and properties each feature belongs to
- Create sample annotation code base and annotation processor
- Create namespace for application annotations and apply to each feature
- Build annotation processor and test several builds

3.1 Install ArchStudio, Import Chatting Application, Run Chatting Application

The first task required to complete the annotation processor was to install ArchStudio in the latest version of Eclipse (Luna at the time of this paper). Next we retrieved the chatting application from GitHub and imported the project into ArchStudio. At this point we were able to see what the code base consists of and run the application. The running application presents two client windows as seen in Figure 1.

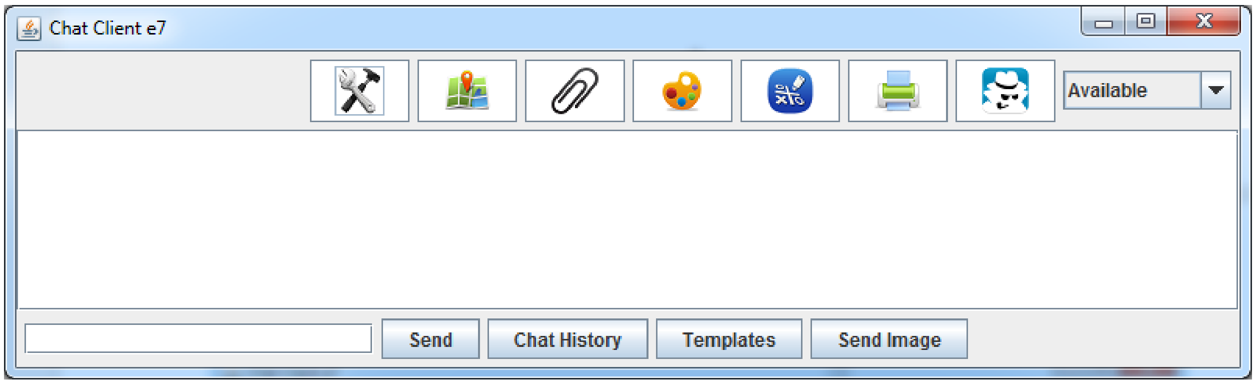


Figure 1: A running application from which we are able to visibly discern features of the product line

By visibly inspecting the running application, we were able to identify the obvious features that we believe represent individually distinguishable parts of a product line. In order to understand how to isolate these features for product line development, it is necessary to identify where features are represented in the architecture of this application. We begin with the list in Figure 2.

Feature	Visual	Feature	Visual
Settings	🛠	Incognito	🕵
Map	🗺	Availability	Available
Attachment	📎	History	Chat History
BGColor	🎨	Template	Templates
Game	🎮	Image	Send Image
Print	🖨	Bot	ChatBot (not visible)

Figure 2: The visibly discernable set of features, categorized

3.2 Extract Block and Class Diagrams

Because this is an ArchStudio project, the visual architecture diagram is presumably given to us. That architecture is as follows:

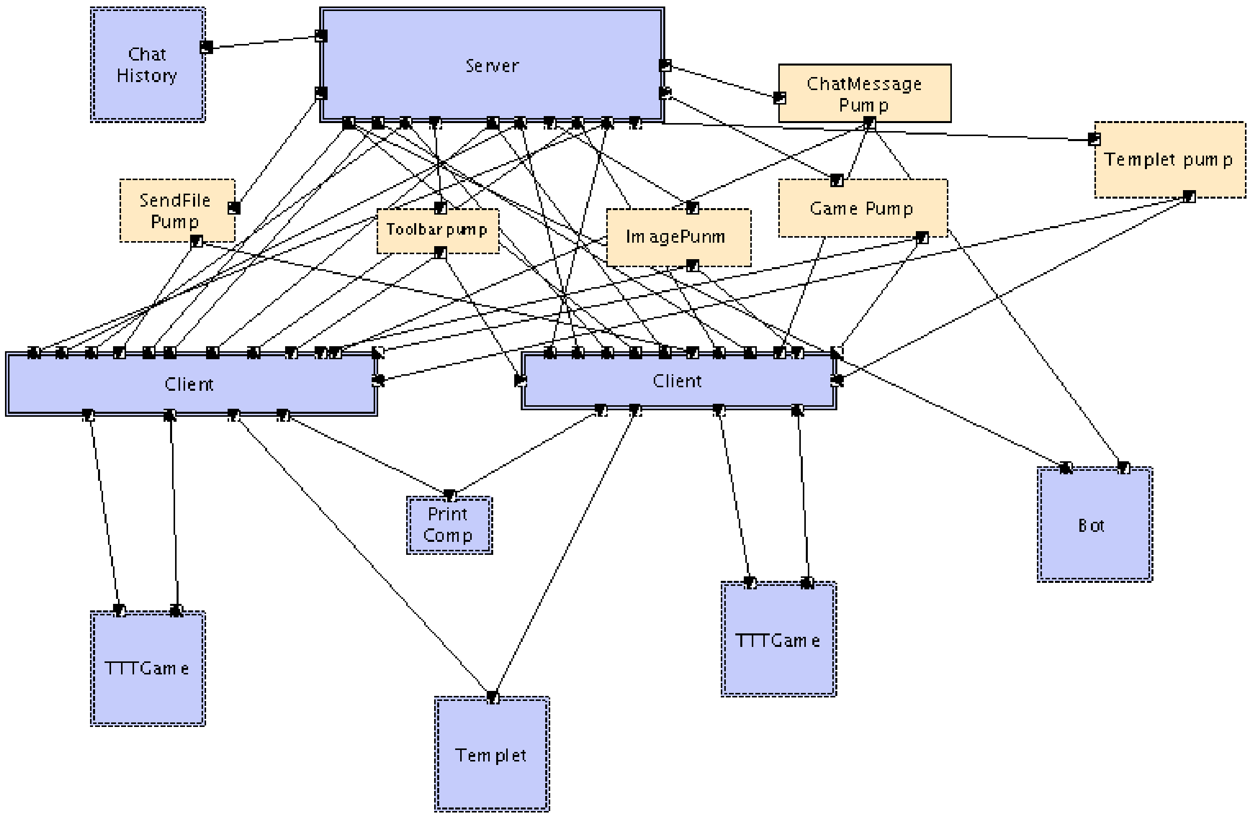


Figure: The chatting application architecture as represented in ArchStudio

Using ObjectAid UML Explorer for Eclipse [Obj] we were able to produce a class diagram representative of the entire application. In truth, the class diagram was no more useful than the Eclipse package explorer for identifying features. The class diagram is just large enough to make navigating it inconvenient. Beyond that fact, a list of properties, methods and relationships as seen in the class diagram did not tell us anything obvious about the architecture. We included this illegible view of the class diagram to give the reader a sense of its scope - reasonably small yet inconveniently large.

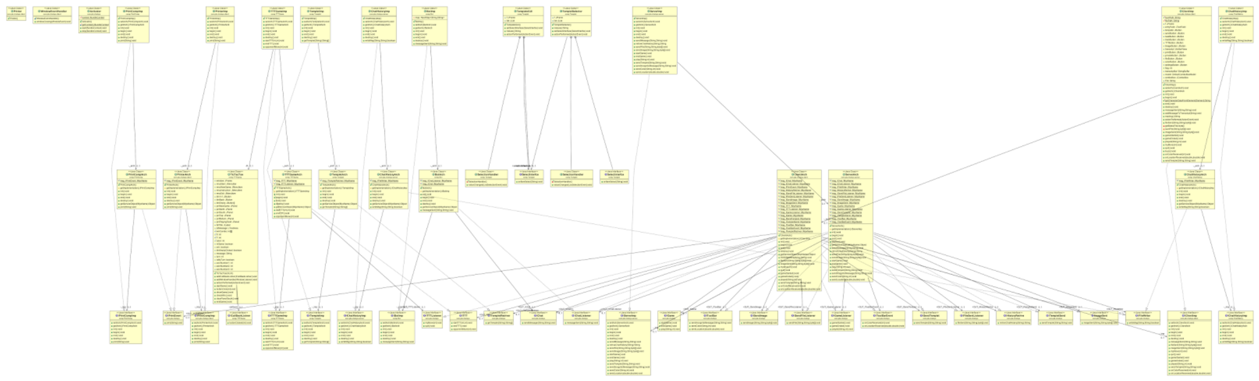


Figure: An illegible view of the class diagram extracted with ObjectAid plugin for Eclipse

3.3 Identify which features within the block/class/method/variables

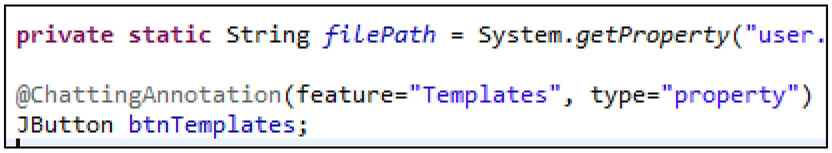
Given the available resources we have developed at this point, we have some idea of how to approach a product line derivation from the original chatting application. We are interested in inspecting the source code at a granular level to see what is obvious and what questions we may have. We will first label each block and class, inside this level of abstraction, with the potential feature associated. From there we will granulate further into the code, identifying methods and properties associated with the features chosen for that class. This will potentially require revisiting of the labeling for blocks and classes based on the information found in the methods.

3.4 Proof-Of-Concept

Before our team started any work on the production version of the annotation processor, we developed a proof-of-concept (POC). The initial idea was to read in each line of code through the processor and make decisions about the current line based on the previously retrieved lines of code. In order to demonstrate this was possible, we developed the POC. The job of the POC was to read in a single annotated class/file of java code and remove only those lines of code which we did not want in our export. The sample code consisted of three variables, a single method and a single class. The POC was able to determine, based on the annotation and the arguments enter in the command line, whether the annotated item was a variable, method or class and whether it was an included feature. Our team successfully demonstrated its capabilities to our fellow classmates.

3.5 Consistent Annotation Namespace

Consistency is key in being successful in our implementation. The POC allowed for a raw form of annotation. Our team did not care whether the naming of each annotation was consistent with each other, or even if the annotation was recognized as a proper annotation within Eclipse. The POC's job was take java code as the input, regardless of errors, and produce java code as the output. Once the POC was determined to be a success, our team quickly worked on defining the namespace for the annotation. The annotation we developed takes two parameters. The first defines the feature the block of code belongs to and is appropriately named "feature". The second parameter defines of what type the following block of code is and is named :"type". In future iterations, it is possible for the processor to be smart enough to recognize the type without the annotation parameter. Our team was able to write this annotation within a new package in the chatting application. We were then able to annotate the code base without any compilation errors.



```
private static String filePath = System.getProperty("user.

@ChattingAnnotation(feature="Templates", type="property")
JButton btnTemplates;
```

3.6 Annotation Processor

The annotation processor design revolved around the ChattingAnnotation namespace. Our team started with the POC as a guide, but quickly realized the logic was not smart enough to handle the code of a third party. With multiple implementations of a variable, constructor calls and method calls making assignments, we had to step back and look at the different ways each feature was used. The processor was given three parameters, first a list of features, second the source directory of the current chatting application, and third, a target directory for the generated code. Because the processor does not have a requirement for the accusatory user to provide each individual java file for processing, it must know how to traverse through a file structure when given the root. We decided that recursion was the best way to handle this challenge. Once the processor has the root of the application directory it will recursively move through the file structure visiting every piece of source code, annotated or not. When the processor determines that it has found an annotation, it sends the annotated line to the annotation engine of the application. It should be noted that multiple designs and architectures were examined when developing the annotation engine. As is common in today's coding practices, Object Oriented was the default pick, however, we quickly learned OO would not benefit functional flow of the engine. The key word is "flow." Ultimately, the processor is a data transform, so the benefits of OO (abstracting a class of usable methods to pick from) does not aid well for a process flow. The engine was built using the Functional Oriented architecture and a state-machine design pattern. The Functional Oriented design pattern allows the engine to move through sequenced, data transforming components. The engine takes a line of code as input and outputs the transform to a new file. The state machine design was necessary to process the annotation. The engine needs to know whether it is processing a variable, method or class and whether it is processing a desired feature. We used a global enumeration to maintain type and an annotated features list, generated by the users arguments, persists through the runtime.

4: Approach

Concluding our thorough survey of the chatting application, we began annotating the original source code. We placed annotations throughout the source code, annotating at the property, method, and class levels. We then used our processor to iterate the source and accordingly process our annotations, generating a separate set of source code representing the desired product line architecture. Where necessary, we refactored the code base to resolve any logical conflicts such as adjoining features that do not match the parent annotation. The result of our work is arguably a horizontally reusable product line architecture management tool. Because we use a single annotation namespace called ChattingAnnotation, our processor could easily be rewritten to use another namespace such as SimpleAnnotation or just Annotation. In planning, developing and implementing our solution, we identified a number of key challenges, tasks, and expected outcomes.

4.1 Challenges

Our team worked to discover any and all challenges processing this particular piece of source code might present before we attempted writing the processor. Any programmer who has been through a requirements phase of a project knows that challenges will always present themselves during the development process.

The initial roadblocks we identified were based around the unknown. For example, We did not know whether the chatting application code required refactoring to accommodate our processor. We had very limited experience using ArchStudio, a problem mitigated by accepting the assistance of Varun Narisetty who is an active contributor to the ArchStudio project.. We had also never built an application specific annotation prior to this project. Each of these challenges were solved easily. By identifying the features in the code early in the project, we quickly realized the only code addition that would be needed would be our custom annotation. The lack of knowledge around ArchStudio was not as big of a detriment as we had initially identified. Our processor worked on the code base without needing to import it into ArchStudio each time. When the time came to import the code, we enlisted the Varun’s assistance. When it came to not having experience with custom annotations the solution was simple, we had to learn. With the aid of tutorials [Hil11] we were able to gain the knowledge needed to build our custom annotation.

Once the development of the processor was underway, more challenges were identified.

4.1.1 Processing Turns Off

The processor was developed in such a way that when annotating variables, the developer only needs to annotate the declaration of the variable. Once a variable is annotated, it is collected into a list of annotated variables during the processor’s runtime. While the processor moves through each line of code, it identifies whether or not an implementation of a variable is actually an annotated variabile or not and then determines if the implementation needs to be removed or not.

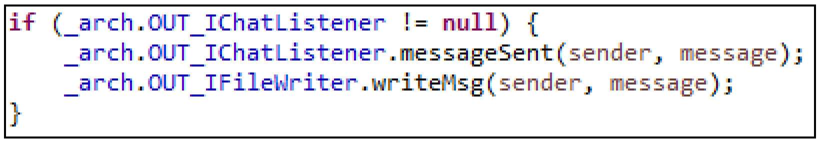
This logic saved our team time while annotating, however, it introduced a new set of problems. If an annotated method (a method which is considered a feature) contains an annotated variable of a feature that is not the same as the method and the method is an included feature, but the variable is not, the processor will finish processing the code once this method is exited. After the processor exits the method, it will exclude all other lines of code from the export. This bug results in an incomplete class and ultimately incomplete code.

To solve this problem would require a large amount of refactoring within the processor. Given that the bug was not identified until the end of the project, the solution became more of a hack, similar to “clearfix” [Wal13] offered in HTML to solve leftover floats. We introduced a new feature called “None.” This annotation is only intended to be used on the methods without a feature, and only on methods that come directly after one of the “special case” methods.

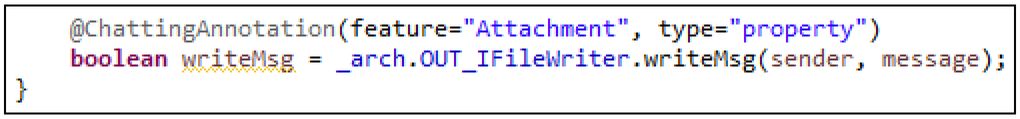
In the future, it would be ideal to refactor the code of the processor to handle these special cases. Our team has identified the root of the problem and knows where the solution can be implemented. Given the time this issue was discovered, making any kind of change to the logic of the processor has a large potential risk and may break the program entirely.

4.1.2 Limitations in Java Annotations

Another issue our team identified towards the end of the project was removing a method call to an annotated/removed method. The question we found ourselves asking was “How do you annotate an object using a removed method inside another class?” In our case, the ServerImp class was using an object which contained a method call to a removed feature/method. The object _arch.OUT_IFileWriter calls method “writeMsg” which had been removed. At first we believed the solution was as simple as annotating above this method call. Java annotations do not allow you to call a method such as _arch.OUT_IFileWriter.writeMsg(sender, message) [Ern13], because they are not assigned to a new object. This is a



```
if (_arch.OUT_IChatListener != null) {
    _arch.OUT_IChatListener.messageSent(sender, message);
    _arch.OUT_IFileWriter.writeMsg(sender, message);
}
```



```
    @ChattingAnnotation(feature="Attachment", type="property")
    boolean writeMsg = _arch.OUT_IFileWriter.writeMsg(sender, message);
}
```

limitation within in Java, so the solution required refactoring of the ServerImpl class. By assigning this method call to a new variable, we were then able to annotate this method call and have it removed when the feature is not desired.

5: Constraints

Since the chatting application is run from ArchStudio architecture, it is generated as an Eclipse runtime plugin. Therefore, the source code generated by our processor is intended to only work within the ArchStudio environment. Importing the source code generated by this project into Eclipse, without the Arch Studio plugin, will likely generate compilation errors. This also implies we are constrained by Eclipse.

The processor is Java language specific. While It's algorithm may be adapted to suit another language (we believe C# is the next most likely candidate), it would require a rewrite and certainly new obstacles may reveal themselves that are more language specific. Though our processor was not built for another application, we suspect it is generic enough in nature to be useful in another code base. Specific knowledge of the code base and package names is hard coded into the class. The fillCollection() method iterates over Eclipse specific project files and uses a feature to file map data structure. The feature names are a pivotal part of what makes the processor run correctly. Changing or adding features to the Client Application will require an edit to the processor.

6: Dependencies

The "Availabilty" feature has multiple dependencies in code that check the status of its associated user interface component. For example, sending an image, sending a text message, or sending a file each result in a call to the status of the recipient's availability. For this reason, we deem the relevant method is a mandatory part of architecture and multiple dependency is not handled by our processor. Many features rely on availability and methods associated with the feature such as messageSent() become mandatory. However, if the Image, Attachment, and Template features are not included in a build, the messageSent() method will still be in the code base, although never used. In future iterations of this project, this should be reconsidered in the processor algorithm to further reduce the need for unnecessary code in a project.

Though the application has many features, some should not be modifiable. For example, if it were possible for an end user to exclude the server or limit the number of clients to one, then the resulting generated application would be useless. This means the chat server should not be a selectable feature, nor should limiting the number of clients to one.

7: Team Management

Of the significant contributions to the project, items of note include the following: Franzke wrote the vast majority of the annotation processor (95% or more with minor logical contributions by Clark). Clark annotated the majority of the java class files and defined the final feature list to be used by annotations. Narisetty integrated the processor into the ArchStudio selector. A division of labor is further categorized in detail in the project schedule of section 8.

Due to the size of the team, we both filled multiple roles. We used the agile development process [BBB+01, BIP09] throughout the design and implementation of this processor. As we stated previously, defining the requirements at the beginning was not enough to identify all of the challenges within this project. To successfully produce a working processor, we were required to revisit the requirements throughout the development process and perform tests on the processor each week.

We did not formalize a single team leadership role. Both of us acted as the developer, the coordinator, and the team leader, exchanging in a healthy dialog and competitive environment for achieving progress toward completing the project. We had the unique opportunity to share the same schedule this semester. We met on Tuesdays and Thursdays, with the occasional Monday and weekend. The need for regular communication ebbed and flowed with the current state of the project and the remaining time to completion. Following the agile process, stand-ups were in the form of regular emails in which we outlined the completed tasks and road-blocks with the project.

The development tasks were divided into stories and assigned in the team meetings, keeping the development tasks distributed as evenly as possible. While we could have formalized a team lead role every meeting for the sake of keeping all work divided, the meetings will most likely be run in a discussion based format.

8: Future Work

Developing the processor as we did produces an algorithm that may apply to other languages. C# in particular uses an attribute mechanism. Attributes are like annotations and in the same way can be queried at runtime using reflection [M05].

9: Project Schedule

Deadline	**Task**	**Description**	**Labor**
October 7	Kickoff meeting	Establish a global perspective of the project	Clark 50% Franzke 50%
October 9	Requirements planning	Develop a clear list of objectives to satisfy	Clark 50% Franzke 50%
October 10-12	Code sprint	See how much can be done before the proposal presentation. Expose any technical constraints	Clark 10% Franzke 90%
October 13	Proposal presentation		Clark 50% Franzke 50%
October 15-17	Architecture analysis	Completed all diagrams necessary to analyze the architecture	Clark 100%
October 17	Feature extraction	Identified all features and which classes, methods and variables belong to each	Clark 50% Franzke 50%
October 27	Annotation processor	Develop annotation processor based on the algorithm used in the sample processor from code sprint	Clark 5% Franzke 95%
November 10	Annotate code	Annotate each feature in the code base	Clark 70% Franzke 30%
November 10	Code generation	Generate source files from annotated code base	Franzke 100%
November 22	Testing in ArchStudio	Bring processor and generated code into arch studio	Narisetty 100%
December 7	Code freeze	Stop development	
December 7	Analysis and reflection		
December 7	Write reports		
December 10	Final presentation		

References

[Obj] *The ObjectAid UML Explorer for Eclipse*
http://www.objectaid.com/

[Nar14] Varun Narisetty, ChattingApp & ArchStudioJars, 2014, GitHub Repository,
https://github.com/varunnarisetty

[Arc] *ArchStudio 5: Software and Systems Architecture Development Environment*
http://isr.uci.edu/projects/archstudio/

[Ern13] *Type Annotations Specification* (JSR 308)
http://types.cs.washington.edu/jsr308/specification/java-annotation-design.html#type-names

[Hil11] *Code Generation using Annotation Processors in the Java language*
Jorge Hidalgo
http://deors.wordpress.com/2011/09/26/annotation-types/

[Wal13] *CSS Clear Fix*
D. Walsh
http://davidwalsh.name/css-clear-fix

[BBB+01] *Agile Manifesto*
K Beck, M Beedle, A van Bennekum, A Cockburn, W Cunningham, M Fowler, J Grenning, J Highsmith, A Hunt, R Jeffries, J Kern, B Marick, R Martin, S Mellor, K Schwaber, J Sutherland, D Thomas, February 11-13, 2001
http://agilemanifesto.org/

[CCG+14] *Differencing and Merging within an Evolving Product Line Architecture*
P. Chen, M. Critchlow, A. Garg, C. Van der Westhuizen, A. van der Hoek
http://www.ics.uci.edu/~andre/papers/C32.pdf

[HBC+12] *A Code Tagging Approach to Software Product Line Development*
P. Heymans, Q. Boucher, A. Classen, A. Bourdoux, L. Demonceau
http://link.springer.com/article/10.1007%2Fs10009-012-0242-1

[XYW+04] *Function Block Oriented Architecture for Open Distributed Automation*
F Xia, H Yin, Z Wang, Y Sun
http://ieeexplore.ieee.org.proxy.library.umkc.edu/xpl/articleDetails.jsp?tp=&arnumber=1342090&queryText%3DFunction+Oriented+architecture

[KM08] *Granularity in Software Product Lines*
C Kästner, O Magdeburg

http://dl.acm.org/citation.cfm?id=1368131

[BIP09] *An industrial case of exploiting product line architectures in agile software development*
M.A. Babar, T Ihme, M Pikkarainen
http://dl.acm.org/citation.cfm?id=1753259

[NC+09] *A Framework for Software Product Line Practice, Version 5.0*
L Northrop, P Clements, F Bachmann, J Bergey, G Chastek, S Cohen, P Donohoe, L Jones, R Krut, R Little, J McGregor, L O'Brien
http://www.sei.cmu.edu/productlines/frame_report/index.html

[M05] *Attributes (C# Programming Guide)*
Microsoft Developer Network
http://msdn.microsoft.com/en-us/library/z0w1kczw%28v=vs.80%29.aspx

[Zhe14] *Variability Management and Product Line Architectures*
Y Zheng
http://y.web.umkc.edu/yzheng/classes/5590SA/SA13.pdf