Object-orientation for Behavior Modeling and Composition

Hiun Kim
Department of Computer Science and Engineering
Sejong University, Seoul, Korea
hiun@divtag.sejong.edu

Abstract
Many variability management techniques rely on sophisticated language extension or tools to support it. While this can provide dedicated syntax and operational mechanism but it struggling practical adaptation for the cost of adapting new technology as part of development process. We present Self-composable Programming, a language-driven, composition-based variability implementation which takes an object-oriented approach to modeling and composing behaviors in software. Self-composable Programming introduces hierarchical relationship of behavior by providing concepts of abstract function, which modularise commonalities, and specific function which inherits from abstract function and be apply refinement to contain variabilities to fulfill desired functionality. Various object-oriented techniques can applicable in the refinement process including explicit method-based, and implicit traits-based refinement. In order to evaluate the potential of independence of behavior from the object by applying object-orientation to function, we compare it to Aspect-oriented Programming both conceptually and empirically.

Keywords: Abstraction, Modularity, Design

1. Introduction
Increasing number and complexity of features in modern software introduces large variability within single software. The property includes reusability, flexibility and comprehension are crucial to managing reliability and sustain the evolution of software. To support this property well, enhancing modularity of features is crucial. One of the major approach of language-driven variability implementation is the Aspect-oriented Programming(AOP) which improves reusability and comprehension of feature by modularising cross-cutting concern(a commonalities that are scattered and tangled across software), while AOP has limitations of higher-order reuse of aspect since aspect is not modular by construct, it decrease its usability when variabilities of cross-cutting concern is high, by enforcing redefinition whole aspect when small portion of cross-cutting concerns are changed. Later works on both Asymmetric and Symmetric modularisation technique including HyperJ and Delta-oriented Programming provides a more flexible approach of the compositional approach of variability management, yet they still require special language extension as part of development process. This is because special syntax is required for simple composition or definition of variability and commonalities within the level of source code, we can capture the core idea of variability management share ones from Object-oriented Programming(OOP) and by applying OOP we could get not only mitigate fundamental requirements of language extension but fully utilise the potential of previous research many advanced object manipulation to behavior manipulation. And another difference is that, previous researches are performing this refinement of behavior in object-oriented way. Self-composable Programming(Self), differ from these previous researches, we address variability problem without object, rather independent, behavior-oriented perspective to pursue flexible higher-order reusability through bringing hierarchical relationship of internal behavior of software.

1.1 Variabilities of Modern Software
We faced many kinds of variabilities while creating modern software which contains many high-level operations. We reason modularisation of modern software is difficult because of increased variability and reduced commonalities. We could able to realise this high-volume of variabilities and some commonalities have come from domain constraint that makes less shared procedure but more similar procedure by constraint such related to operational or safety concerns. We specified this property to called behavioral similarity. One of the most notable, well-adapted examples is of behavioral similarity is network related software, unlike system software(i.e. OS) this software is consist of a relatively large set of but a smaller complexity of modules(i.e API server). As a result, these modules consist large part of theses software and the commonalities for each module such as authentication management, caching or data validation has behaved similarity by showing similar patterns of invocation. Additionally, Network-dependent architecture like Service-oriented Architecture(SOA) pushes more relying on software that operated in the other part of network by their correct collaboration will result in accelerated scattering of commonalities. As a result, the increasing of network relationship, behavioral similarity will be a prominent attribute of variabilities in modern software. In the area of safety-critical systems, robotics or intelligent system has faced the same phenomenon which is inevitable for achieving advanced functionalities. The behavioral similarity is can be handled by well-established variability management technique such as AOP while its high-volume and dynamics require modular by the construct approach of modularisation of behavior.

1.2 Aspect-oriented Approach
One implementation of AOP, AspectJ decomposes single module into core concern and cross-cutting concern, modularise scattered and tangled cross-cutting concern into Aspect object which contains pointcut information, and uses to jointpoint in a location of pre and post processing. For example in web service, when core concern is writing a post or send a message, suitable cross-cutting
concerns would be authentication and validation which a form of internal operation to support a correct operation of core concerns. This fashion of modularisation is possible through metaprogramming or meta object protocol but the contribution that AOP provides a framework for easy, safe and manageable modularisation in direct semantical way[11].

1.3 Object-oriented versus Behavior-oriented

Object-orientation addressed by OOP[12] is new programming and architectural paradigm to modeling things in the real world, eventually to simulate things in the real world and their interaction[13]. Asymmetric AOP such as AspectJ captures the cross-cutting concerns in the behavior and provides high-modularity by localising it to the Aspect object. In symmetric modularisation technique, no concept of the base module. HyperJ uses multi-dimensional separation of concern[14] and compose feature from there. While building block of both approach is still in the context of software composition based on object, which means, we could reuse object while could not precisely reuse it’s behavior, each object can be reused only by replacing technique like overriding or traits[15]. In section[14], we show a single behavior can construct from multiple sub-behavior, instead of reusing a single portion of sub-behavior, we need to reuse a collection of behavior which they are used in the similar pattern of invocation. To achieve this goal, we need a framework to create behavior modular by construct. In rest of paper, we first briefly elaborate some of the key ideas of object-orientation which is the root of an object, a class is and its realisation instance and hierarchical relationship between the objects which enabled from refinement by inheritance. To apply object-orientation to the behavior we made the property which behavior must have called Self-composability, and as an implementation of this concept, we introduce Self-composable Programming(Self). Self creates abstract behavior which represent the class in OOP and specific behavior for inherited and refinable behavior. Self implements 2 property Self-composability and Multi-level inheritance to flexible support for construction, inheritance, and refinement of behavior at the level of programming.

1.4 Self-composable Approach

Similar to symmetric modularisation, Self takes an approach to bringing a hierarchical relationship to behavior for modularising commonalities which spread to the arbitrary structure in each module. Like OO language is taking the approach to modeling object in the real world, we take an approach to modeling behavior in the real world, by doing so we treat behavior is not dependent on a subset of an object rather an independent being. As a result, just like inheriting object, inheritance of behavior is possible. Self implements variability by allows creating specific child behavior from abstract parent behavior by inheritance and apply series of refinements. In the process of refinement, the commonalities are localised to parent behavior and child behavior will self-composed to achieve desired functionality. In other words, parent behavior works like the builder of feature-specific design pattern[16]. On the other part of this paper, we introduce the concept of self-composability and its implementation written in JavaScript Self with the introduction of self-composable domain analysis as a subset of the process of requirement engineering with an example of relationship modeling as in the case of web service. Additionally, We introduce, set of a quirement engineering with an example of relationship modeling self-composable domain analysis as a subset of the process of re-

2. Self-composable Programming

2.1 Abstract Function

Like Unix Philosophy[18], the term compose means behavior composition to perform a more high-level operation, inversely, a composed behavior could be recomposed are possible to support higher-order composability. If we compose feature get file and sending a message at once, it could be composed high-level behavior called file sharing. Modern software requires behaviors from various dimension so, by code-level higher-order composability is provides better modularity.

2.2 Terminology on Composition

Like Unix Philosophy[18], the term compose means behavior composition to perform a more high-level operation, inversely, a composed behavior could be recomposed are possible to support higher-order composability. For example, a behavior of sending a message in messaging program is consist of cross-cutting concerns and core concern. If we compose share file and automatic sending at once, it could be composed high-level behavior called file sharing. Modern software requires behaviors from various dimension so, by code-level higher-order composability is provides various level of granularity in a single solution.

2.3 Self-composability on Behavior

Self-composability is core concept of Self consists of following four aspects.

Self-addition of behavior. To compose behavior, programmer could construct behavior from a set of low-level behaviors by adding them. For example, followed above example of a web application, the send message behavior can be composed of sub-behavior like authentication, logging, validation, context management. By adding these sub-behavior, a programmer can able to construct the framework of abstract behavior. Self-addition also used after inheritance of behavior by adding core behavior into it.

Self-update of behavior. Again, a composed behavior is consist of low-level behavior after construction. An individual behavior can be updated. In other words, Self-update is a partial update for super behavior. For example, when some sub-behavior in web application requires new authentication mechanism in another module, the authentication sub-behavior can be replaced. The significance of Self-update is that authentication module itself can be partially updatable if it is consist of sub-sub-behavior.

Self-deletion of behavior. Deletion is important to remove specific low-level behavior to working behavior correctly. For example, when building public API that does not require authentication, by Self-deletion could partially delete authentication sub-behavior from API behavior.

Self-manipulation of behavior. Manipulation is a free mode of manipulating sub-behaviors, although above three example is provides directed usage for manipulating sub-behaviors, Self-manipulation provides restriction free manipulation. For example, self-manipulation can be used for repeating sub-behavior or manipulating arguments. Another property of self-composability has sophisticated usage of manipulation to low-level behavior, the Self-manipulation can partially manipulate the anything about behavior.

2.4 Multi-level inheritance on Behavior

Self-composed behavior is used in the lifecycle of composing, inheritance, refinement, and execution. Self-composability is used to
refinement, multi-level inheritance is use for creating the refirable instance. Like refinement is consulted for making object specific in OOP, refinement in Self is used for making behavior specific. For example in data access based on a database. As figure 1 shows, each behavior gets a relationship to other behaviors. The behavior could have a relationship at the same level while having a hierarchical relationship as its sub-behavior. In this example, we present, 4-level of domain cross-cutting concerns for processing our core concern - database operations. Based on fundamentality the level of domain starts from ‘connection management’, ‘operation-specific management’ and ‘object-specific processing’ for the object of operation and finally user-visible ‘feature-specific operation’. The most abstract parent behavior DBQuery localise of connection management, and next ReadDBQuery and WriteDBQuery performs localise of commonalities about operations. Next part the behavior called ReadPost and WritePost which localise an operation of object Post and lastly the behavior ReadPostRecents and CreatePost localise variabilities of each feature. As we can see, the abstract behavior has global influence while low strength to each specific child behavior. Conversely, specific behavior has high influence while its area is local. The arrow between each behavior represents each behavior is inherited from more abstract behavior and localised commonalities(or variabilities in whole system perspectives) by applying refinement to be transformed into more sophisticated specific behavior. (e.g. transformation of operation-specific DBQuery to from general DBQuery module) Previous mentioned hierarchical relationship of behavior is similar to how the organisation of people made a decision, which originated by C-level managers and their decision is realised by employees. The who takes fundamental responsibility for his organisation which is DBQuery behavior which influences as far as those from the edges of the organisation like ReadPostRecents but their influences are limited, while the influence of his direct manager ReadPost has limited are of influence but stronger to his directed employee then DBQuery. As result the content of final behavior is influenced by many advice[19] from layers of abstract behavior and this advice could affect small even or large changes of final behavior. By using this hierarchical relationship of behavior, we could model behavior more accurately in both architecture-level and programming-level for the SOA and other large-scale systems which in the environment of distribution collaboration.

3. Self-composable Domain Analysis
As a result architecture of hierarchy of Self-composable behaviors, we could able to perform domain analysis by defining the domain of cross-cutting concerns based on its fundamentality and based on its domain, to architect software in self-composable form.

4. Self : A Prototypal Implementation
4.1 Overview
Self[20] is a JavaScript implementation of Self by supporting self-composability and multi-level inheritance at code-level. We have chosen OOP for the implementation medium by its direct support of method notation and inheritance. We chose an implementation language for JavaScript by its lightweight support for OOP and functional programming.

4.2 Design and Implementation
Self is JavaScript library. The behavior object constructed by Self contains an array which has serialised sub-behaviors and set of method to perform manipulation to it as described in table[1] Each method gets an argument as a primitive function or another behav-

![Diagram](image-url)
ior object. When executing the behavior, Self uses given initial arguments and passes its result as an argument of next sub-behavior. By improving adaptability of function interface to used in variety of composition circumstances, programmer can non-invasively generalise the function using operation like .map, or .before method. This non-invasive manipulation allows to separate its commonalities and improve reusability to work sub-behavior as an atomic building block.

4.3 Architecture
Self has two major part index.js for provides user-visible API and behavior-store.js for internal operations. When index.js is loaded as a behavior constructor in program then user interact with standard API in prototype of behavior instance, the sub1, sub2 in behavior instance does not store actual behavior instance but it stores only name which designated to provides an anchor for invoke internal operating mechanism.

4.4 Operating Mechanism
The goal of most operations in Self, as a both conceptual and implementation perspective, is fulfilled variability of software feature by applying easy and sophisticated refinement to elements - a sub behavior in the array. To do this, every behavior instance has its own behavior-store instance which stores actual behaviors array and its method to perform manipulation. As a result, the caller user program indirectly manipulates behavior.

5. Using Self
In this section, we elaborate code-level overview of using Self in the context of construction, inheritance and refinement and as well as its internal mechanism.

5.1 Self in a Nutshell
Listing 1 shows complete code-level lifecycle of web service consist of connection management, operation-specific processing, object-specific processing and feature-specific processing.

5.2 Behavior Construction

```javascript
1 var Behavior = require('self');
2 DBQuery = new Behavior();
3 DBQuery.add(auth);
4 DBQuery.add(validate);
5 DBQuery.add(monit);
6 DBQuery.add(wrapBlvr);
7 DBQuery.add(map());
8 DBQuery.add(updateBhvr);
9 DBQuery.add(deleteBhvr);
10 DBQuery.add(applyTraits);
```

Table 1: Method List of Self

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior#add</td>
<td>Append given function or behavior into high-level behavior</td>
</tr>
<tr>
<td>Behavior#sub#before</td>
<td>Insert given function or behavior before specified behavior</td>
</tr>
<tr>
<td>Behavior#sub#after</td>
<td>Insert given function or behavior after specified behavior</td>
</tr>
<tr>
<td>Behavior#sub#update</td>
<td>Update specified behavior into given function or behavior</td>
</tr>
<tr>
<td>Behavior#sub#delete</td>
<td>Delete specified behavior</td>
</tr>
<tr>
<td>Behavior#sub#map</td>
<td>Manipulate specified behavior with new function or behavior that takes original behavior as an argument</td>
</tr>
<tr>
<td>Behavior#assign</td>
<td>Assigns traits to specific behavior with given traits object</td>
</tr>
<tr>
<td>Behavior#sub#assign</td>
<td>Define new method for behavior refinement which access directly behaviors array</td>
</tr>
</tbody>
</table>

*d All method takes single native Function Object or Behavior Object created by Self.
Listing 1: Self in a Nutshell

```javascript
1 /* CONSTRUCTION PART */
2 // define self
3 var Behavior = require('self');
4 // initialising behavior
5 var DBQuery = new Behavior();
6 // adds some sub-behaviors
7 DBQuery.add(auth); // authentication checker
8 DBQuery.add(validate); // data validation
9 DBQuery.add(monit); // monitoring
10
11 /* REFINEMENT PART */
12 // inherit DBQuery to operation-specific, WriteDBQuery
13 var WriteDBQuery = new DBQuery();
14 // add some sub-behaviors (refinements)
15 WriteDBQuery.add(writeBack);
16 // update specified sub-behavior to new sub-behavior
17 WriteDBQuery.monitoring.update(cacheMonit);
18 // add sub-behavior in specified location
19 WriteDBQuery.validate.before(beforeValidate);
20 WriteDBQuery.validate.after(afterValidate);
21 // manipulating sub-behavior
22 WriteDBQuery.validate.map(() => {
23 return (validate) => {
24 validateWrapper(validate);
25 };
26 });
27 // delete sub-behavior
28 WriteDBQuery.beforeValidate.delete();
29
30 /* ADDITIONAL REFINEMENT */
31 // inherit WriteDBQuery to object-specific query
32 var CreatePost = new WriteDBQuery();
33 var CreateMessage = new WriteDBQuery();
34 CreatePost.add(createUserSQLExec);
35 CreateMessage.add(createMsgSQLExec);
36 // additional modification
37 CreatePost.auth.update(2factorAuth);
38 CreateMessage.auth.before(geographicalBlock);
```

Listing 2: Construction of Self-composable behavior

Listing 5.2 shows Behavior construction using Self. Self is assigned to variable behavior through require statement which supported by general purpose JavaScript runtime Node.js [21]. As a constructor, Behavior create DBQuery instance consist of data and method. Data is an array that contains function or another behavior instance. In this example, shows addition of commonalities auth, validate and monit through .add method.

5.3 Behavior Inheritance

```javascript
1 /* Operation-specific Processing */
2 var ReadDBQuery = new DBQuery();
3 var WriteDBQuery = new DBQuery();
4 // ...some refinement
5
6 /* Object-specific Processing */
7 var ReadPost = new ReadDBQuery();
8 var ReadMessage = new ReadDBQuery();
9 var WritePosts = new WriteDBQuery();
10 var WriteMessage = new WriteDBQuery();
11 // ...some refinement
12
13 /* Feature-specific Processing */
14 var ReadPostsRecents = new ReadPosts();
15 var ReadPostsPopular = new ReadPosts();
16 var ReadMessageLists = new ReadMessage();
17 var ReadMessages = new ReadMessage();
18 var CreatePost = new WritePost();
19 var UpdatePost = new WritePost();
20 var CreateMessage = new WriteMessage();
21 var DeleteMessage = new WriteMessage();
```

Listing 3: Multi-level Inheritance of Self-composable behavior

Listing 3 shows an example of behavior inheritance using new keyword. In this listing, localisation of commonalities will be performed through refinement based on three-level of inheritance. At the internal of inheritance mechanism is that, it hard copying data and links prototype method of super behavior to newly created Behavior instance.

5.4 Explicit Behavior Refinement

```javascript
1 var WriteDBQuery = new DBQuery();
2 WriteDBQuery.add(writeBack);
3 WriteDBQuery.monitoring.update(cacheMonit);
4 WriteDBQuery.validate.before(beforeValidate);
5 WriteDBQuery.validate.after(afterValidate);
6 WriteDBQuery.validate.map(() => {
7 return (validate) => {
8 validateWrapper(validate);
9 })
10 }
```
can be applied to
resented in
trait of behavior that does not require authentication which rep-
Table 2: Feature list of first analysis software

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User.getName</td>
<td>Get specified user name with authentication</td>
</tr>
<tr>
<td>User.getProfile</td>
<td>Get specified user profile with authentication</td>
</tr>
<tr>
<td>User.getPosts</td>
<td>Get specified user posts with authentication</td>
</tr>
<tr>
<td>User.getOnline</td>
<td>Get specified user status with authentication</td>
</tr>
<tr>
<td>Post.getRecentSummary</td>
<td>Get recent post summary with authentication</td>
</tr>
<tr>
<td>Post.getRecentsWithoutImage</td>
<td>Get recent post text with authentication</td>
</tr>
<tr>
<td>Post.getPopularSummary</td>
<td>Get recent post summary without authentication</td>
</tr>
<tr>
<td>Post.getPopularWithoutImage</td>
<td>Get recent post text without authentication</td>
</tr>
</tbody>
</table>

Figure 3: Behavior Relationship for First Software Analysis

The relationship of behavior when takes 4 level of behavior analysis. The final name of feature can be retrieved as concatenating left side to right side destination node.

evaluation is available in Appendix A. In order to calculate the SLOC for each function, we classified the SLOC into two purposes. First, we count SLOC for integrating cross-cutting concerns to implement features. In AOP, this could be achieved by creating aspect object, and in Self, this part will achieve through construction and refinement of three-level of inheritance to behavior. Secondly, we count code that represents cross-cutting concerns (variables) to fulfill desired functionality.

6.2.1 Analysis of AOP-based Implementation

The code for creating an aspect object is line 1~5, 8~12, 21~27, 29~36 - 24 lines in total (a in Table 3) and code for their actual cross-cutting concern is line 2~5, 9~11, 22~26, 30~35 18 lines in total (b in Table 3). In the code for creating the object, we could confirm their sub-behavior are - logging, auth, cache-

Table 3: Average SLOC of per Single Feature Implementation

<table>
<thead>
<tr>
<th>SLOC of Whole Coordination Module (a)</th>
<th>AOP</th>
<th>Self</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLOC of Cross-cutting Concern (b)</td>
<td>26</td>
<td>14</td>
</tr>
<tr>
<td>Number of Feature</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Average SLOC of per Single Feature Implementation (b/8)</td>
<td>2.25</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Table 4: Self : SLOC per Each Level of Inheritance

<table>
<thead>
<tr>
<th>Lev. of Inheritance</th>
<th>Num. of Parents</th>
<th>Num. of Child</th>
<th>Refinement SLOC</th>
<th>Total SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Second</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>Third</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>250</td>
</tr>
<tr>
<td>Forth</td>
<td>(projected)</td>
<td></td>
<td></td>
<td>1290</td>
</tr>
<tr>
<td>Fifth</td>
<td>(projected)</td>
<td></td>
<td></td>
<td>6249</td>
</tr>
</tbody>
</table>

Lookup, userIdValidation is scattered in similar pattern and AOP-based implementation made duplicated declaration and result could not perform reuse caused by variability from auth.

6.2.2 Analysis of Self-based Implementation

Self-based implementation shows a hierarchical, higher-order combination, unlike AOP, does flattened combinations, to avoid reusability degradation of code due to redundant calls without affecting program comprehension. For 1~2 level construction and refinement for User are line 3, 5, 7, 14, total 4 line and for Post are line 15, 19 total 2 and overall in 6 line (b in Table 3). Additionally, the code used for final step refinement is line 9~12, 16~17, 20~21 total 8 and by combining those SLOC for all three step of refinement, the SLOC for declaring reusable medium is line 14(a in Table 3).

6.2.3 Result Analysis

As Table 3 shows an encouraging result, average SLOC for implementing the single feature is 2.25 for AOP and for Self, it is 0.75. In other words, when using AOP average 2.25 SLOC of cross-cutting concern will be used while Self requires only 0.75 SLOC. In addition, as OOP encapsulates data and performs information hiding, Self encapsulates information by encapsulating the sub-behavior which makes up the super-behavior. The advantages and disadvantages of hiding are also understandability, although it allows high-level usage, but it made difficult for the programmer to understand the internals of behavior when such activity of refinement scattered. In addition, current direct refinement may violate the correctness of the behavior, so additional research is required on how to indirectly improve it and how to localise it. Finally, in OOP, as the content and phase of collaborations increase in OO collaboration, programmers have no way to define or understand system behavior directly. In the OOP environment, a DCI architecture has been proposed to separate the data and the interaction and create a context to glue the two together [23]. A context environment for super-behavior and sub-behavior would help users to grasp the inside of the action and to be informed if they could be combined in such context.

6.3 Second Evaluation : Predicting Reusability per Software Growth

The second analysis is to measure how much SLOC is needed while using AOP and Self to advance the functionality with more variability. The purpose of this analysis is to simulate the first
It is possible to effectively suppress the increase in the number of
support lines of code needed for improvement on the table 5 is kept on 5
ultimately achieving variability at minimal cost. This proves that
directly, thus increasing the flexibility of program modification and
of source code used per activity. In this experiment, we create 2,
5, and 10 child behaviors respectively through inheritance, and add
5 lines of code for each inheritance to improve the requirement
and context space is needed to be adapted. As the previous experi-
architecture[23], for solidly separating various refinement by types
needs to be created. One of best example that shows separating the
process more expressive. In system-level, an architectural pattern
be scattered. As a result, the end user may have difficulty grasp-
through multi-level inheritance and multiple application of refine-
ations of OOP. It has been possible to localise commonalities
perspective that sees the behavior of software in isolation from ob-
ability management. These approaches look the problem still OO
perspective, while Self looks at behavior-oriented as the primitive
person in programming. Although Self showed better performance
compared to existing methods, such as the result of in variability
management. These approaches look the problem still OO perspec-
tial results show, Self is useful for large-scale variable software,
programming language and thus can be used interchangeably
with AOP as well as OOP as needed. Just as AOP has offered a
programmer an aspectual thinking and proposed a direction for de-
signing the software apart from the use of AOP implementations,
self-assembly programming also allows programmers to look at
the behavior of the software. The goal is to present a new point of
view.

8. Discussions
8.1 Implementation Types
Currently, Self is implemented in the form of a library but can be
processed more efficiently if embedded in the language in terms of
grammatical freedom and processing efficiency. To natively
support self-composability, a programming language that supports
statement as a primitive class citizen is essential. In order to sup-
port statement computation, new object system and operations for
statement have to be researched.

8.2 Future Research Directions
The major limitation of Self is that it is also in line with the lim-
itations of OOP. It has been possible to localise commonalities
through multi-level inheritance and multiple application of refine-
ment, but there is a problem that refinement and inheritance can
be scattered. As a result, the end user may have difficulty grasp-
ing the inside of the behavior accurately. In a code-level solution of
this problem, we support traits to make refinement and inheritance
process more expressive. In system-level, an architectural pattern
needs to be created. One of best example that shows separating the
mental model of a programmer from a data model of a computer is
MVC architecture[31], and more generalised example such as DCI
architecture[22], for solidly separating various refinement by types
and context space is needed to be adapted. As the previous experi-
mental results show, Self is useful for large-scale variable software,
but refactoring legacy systems into a self-composable way is costly.
Therefore, it is necessary to use a program transformation or wrap-
per technique that can make the existing system self-composable
is needed, possibly at runtime. In terms of syntactical perspective,
domain-specific behavior creation and notation (eg. .add instead

### Table 5: AOP : SLOC per Each Level of Inheritance

<table>
<thead>
<tr>
<th>Lev. of Inheritance</th>
<th>Num. of Parents</th>
<th>Num. of Child</th>
<th>Refinement SLOC</th>
<th>Total SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Second</td>
<td>2</td>
<td>5</td>
<td>15</td>
<td>150</td>
</tr>
<tr>
<td>Third</td>
<td>5</td>
<td>10</td>
<td>20</td>
<td>1000</td>
</tr>
<tr>
<td>Forth</td>
<td>(projected)</td>
<td></td>
<td>7211</td>
<td></td>
</tr>
<tr>
<td>Fifth</td>
<td>(projected)</td>
<td></td>
<td>50988</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Prediction of Growing SLOC by Level of Inheritance

analysis on a larger scale and measure how much SLOC is used
for each stage of refinement of the behavior. As the number of
functions of the variabilities on software increases as the size of
the first analysis increases. It is a method to estimate reusability by
predicting the amount of code. In AOP, there is a way to reuse an
aspect object through inheritance, but it does not directly support
high-level reuse [24] [25]. In this analysis, the behavior of the
virtual web service is specified through three-level of inheritance.
Therefore, the SLOC used in each step is the product of the number
of parent actions, the number of child activities, and the three items
of source code used per activity. In this experiment, we create 2,
5, and 10 child behaviors respectively through inheritance, and add
5 lines of code for each inheritance to improve the requirement
satisfaction. Since AOP can only be reused in the first order, it
creates a new aspect with an execution pattern of the same behavior
as the existing one, such as the appendix of the first analysis. As
a result, the amount of SLOC increases like the table. In the case
of Self of the the actor can manipulate only the actual changes
directly, thus increasing the flexibility of program modification and
ultimately achieving variability at minimal cost. This proves that
the source code needed for improvement on the table is kept on 5
line.

6.4 Result and Improvement
As a result of deriving the trend function based on the table and
It is possible to effectively suppress the increase in the number of
codes. In other words, it is confirmed that Self effectively manages
the variability of the software, so that it can support only the
necessary part of the SLOC that exponentially increases by the
existing methods such as AOP.

7. Related Works
There are related research in software product line engineering
line like Delta-oriented Programming[8], a bit old researches in-
cluding AOP[5], GenVoca[26], subject-oriented programming[27],
adaptive plug-and-play components[28] and role components[29]
to cope with the variability of software through separation of con-
cerns. The main difference between Self is a perspective of vari-
ability management. These approaches look the problem still OO
perspective, while Self looks at behavior-oriented as the primitive
person in programming. Although Self showed better performance
compared to existing methods, such as the result of in variability
management. These approaches look the problem still OO per-
spective, while Self looks at behavior-oriented as the primitive
person in programming. Although Self showed better performance
compared to existing methods, such as the result of in variability
management. These approaches look the problem still OO per-
spective, while Self looks at behavior-oriented as the primitive
person in programming. Although Self showed better performance
compared to existing methods, such as the result of in variability
management. These approaches look the problem still OO per-
spective, while Self looks at behavior-oriented as the primitive
person in programming. Although Self showed better performance
compared to existing methods, such as the result of in variability
management. These approaches look the problem still OO per-
spective, while Self looks at behavior-oriented as the primitive
person in programming. Although Self showed better performance
compared to existing methods, such as the result of in variability
management. These approaches look the problem still OO per-
spective, while Self looks at behavior-oriented as the primitive
person in programming. Although Self showed better performance
compared to existing methods, such as the result of in variability
management. These approaches look the problem still OO per-
spective, while Self looks at behavior-oriented as the primitive
person in programming. Although Self showed better performance
compared to existing methods, such as the result of in variability
management. These approaches look the problem still OO per-
spective, while Self looks at behavior-oriented as the primitive
person in programming. Although Self showed better performance
of .addValidation) can be useful to increase comprehension and accuracy of refinement against generic notation. In addition, it is possible to develop a system aiming at rapid prototyping of high-level functions based on transformation or wrapper technique by integrating the Self with the package management system such as npm or pip. In order to facilitate the composition of behaviors, the feature interaction problem has to be resolved. Also, we extend examination to more previously established evaluation metrics including Expression Product Line[33] and other works[34,35] compare to technique include feature-oriented programming[36] have to performed to examine substances of research. As mentioned earlier in the[1] chapter, the ultimate goal of this research is not to use the methods and paradigms of OOP for modeling and composing behaviors. The ultimate goal of this research is to emphasize the necessity and provides a practical approach for modeling behavior based on the fact that modularity of behavior can be improved by the independent from object-based product line implementation. Therefore, in the long-term, it is reasonable to develop the OOP-independent programming language that supports the concept of self-composability and multi-level inheritance. To develop such language, a new linguistic notation and to study on sequential nature of behavior and how to model the nature of behavior in the real world from various perspectives have to be established.

9. Conclusion

In this paper, we present Self-composable Programming, a language-driven, composition-based variability implementation that bringing core idea of object-orientation to improve modularity without needs of special language extension or tooling support. We propose the concept of the hierarchical relationship of behavior for modularisation of software by introducing the concept of abstract function and specific function and introduces the concept of self-composability and multi-level inheritance for behavior modeling and composition. To support these properties in language, we proposed Self-composable Programming using the favor of object orientation. We elaborated the limitations of various current symmetric and asymmetric modularisation techniques, both requires a special language extension in the development process and does not fully utilise previous research on software composition in compare to Self, a JavaScript-based implementation of Self-composable Programming. Self also supports higher-order reusability by creating behavior modular by construct as a result, the programmer would able to partially refine sub-behavior which only affects to variability to super-behavior by the perspective of behavior-oriented variability management instead of object-oriented. Self can construct a behavior that can be self-composed and introduce a self-composable domain analysis as a part of requirements engineering to architect behavior-first software engineering. We evaluated and analyzed the potential of Self to the web service in comparison with AOP, and we were able to confirm the efficiency provided by applying object-orientation to behavior to support flexible refinement compared with the existing method, AOP. We propose Self as a practical programming technique as well as emphasize the importance of behavior modeling by the independence of behavior from objects, and modularisation by bringing the hierarchical relationship to behaviors and provide new perspectives to researchers and practitioners. Thus, we emphasized the importance of research of new programming language that has dedicated notation to express the essence of the behavior without OOP. We present the importance and value of accurately modeling and simplifying control of the behavior in the real world, as OOP models things in the real

world in 50 years ago, and we shows Self-composable Programming can be used as a medium of models behaviors in the real world.

A. Source Code of AOP-based Implementation

```javascript
1 var ReadUser = createAspect(function () {
2   logging(data);
3   auth(data);
4   cacheLookup(data);
5   userIdValidation(data);
6 });
7
8 var ReadUserWithoutAuth = createAspect(
9   function () {
10   logging(data);
11   cacheLookup(data);
12   userIdValidation(data);
13 });
14
15 var User = {
16   getName: applyAspect(ReadUser, readUserNameQuery),
17   getProfile: applyAspect(ReadUser, readUserProfileQuery),
18   getPosts: applyAspect(ReadUser, readUserPosts),
19   getOnline: applyAspect(ReadUserWithoutAuth, readUserOnline)
20   };
21
22 ReadPost = createAspect(function () {
23   logging(data);
24   cacheLookup(data);
25   postNumberValidation(data);
26   rangeValidation(data);
27   ReadRecentsSummaryQuery(data);
28 });
29
30 ReadPostWithoutAuth = createAspect(function () {
31   logging(data);
32   auth(data);
33   cacheLookup(data);
34   postNumberValidation(data);
35   rangeValidation(data);
36   ReadRecentsSummaryQuery(data);
37 });
38
39 var Post = {
40   getRecentSummary: applyAspect(ReadPost, readPostRecentsSummary)
41   getRecentsWithoutImage: applyAspect(ReadPost , readPostRecentsWithoutImage)
42   getPopularSummary: applyAspect(ReadPostWithoutAuth, readPostPopularSummary)
```

2 Package manager for JavaScript npmjs.com
3 Package manager for Python pip.pyppa.io
B. Source Code of AOP Helper

```javascript
function createAspect (beforeFunc , afterFunc)
{
    return new jsAspect . Aspect(new jsAspect .
        Advice . Before(beforeFunc , afterFunc) ;
}

function applyAspect (aspect , func)
{
    //wrapper for applying aspect to function ,
    instead of object .
    var obj = {
        method: func
    };
    aspect . applyTo(obj) ;
    return obj .method ;
}
```

C. Source Code of Self-based Implementation

```javascript
var DBQuery = new Behavior().add(logging);

var DBQueryRead = new DBQuery().add(auth).add(
    cacheLookup);

var DBQueryReadUser = new DBQueryRead().add(
    userIdValidation);

var User = {
    getName: new DBQueryReadUser().add(
        readUserNameQuery),
    getProfile: new DBQueryReadUser().add(
        readUserProfileQuery),
    getPosts: new DBQueryReadUser().add(
        readUserNameQuery),
    getOnline: new DBQueryReadUser().add(
        readUserOnline).auth .delete ()
};

var DBQueryReadPost = new DBQueryRead().add(
    postNumberValidation).add(rangeValidation );

var DBQueryReadPostRecents = new
    DBQueryReadPost().add(ReadRecentsQuery);

var DBQueryReadPostPopular = new 
    DBQueryReadPost().add(ReadPopularQuery);

var Post = {
    getRecentSummary: new DBQueryReadPostRecents.
        ReadRecentsQuery . update(
            ReadRecentsSummaryQuery),
    getRecentsWithoutImage: new
        DBQueryReadPostRecents . ReadRecentsQuery .
        update(ReadRecentsSummaryWithoutImageQuery),
    getPopularSummary: new DBQueryReadPostPopular .
        ReadPopularQuery . update(ReadPopularSummaryQuery).auth . delete (),
    getPopularWithoutImage: new
        DBQueryReadPostPopular . ReadPopularQuery .
        update(ReadPopularWithoutImageQuery).auth . delete ()
};
```

Acknowledgments

I give special thanks to members of Divtag Computer Research Group who provides resources and experience of joyous of programming.

References


