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.

CCS Concepts • **Software and its engineering** → **Abstraction, modeling and modularity**; *Language features*; Very high level languages; Design patterns; • **Applied computing** → Enterprise architectures

Keywords Abstraction, Modularity, Design

1. Introduction

Increasing number and complexity of feature[1] 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[2]. To support this property well, enhancing modularity of features is crucial[3]. One of the major approach of language-driven variability implementation[4], Aspect-oriented Programming(AOP)[5] 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 andSymmetric modularisation[6] technique including Hyper/J[7] and Delta-oriented Progreamming[8] 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] In this paper, the term feature used for 'prominent and distinctive user-visible aspect, quality or characteristic of a software system or system'followed by Kang et al. [1] And the term behavior used for 'software aspect ofoperations for implement of feature', andmultiple behavior could establish single feature

1.1 Variabilities of Modern Software

We faced many kinds of variabilities while creating modern software which contains many high-level operations. We reason moduarisation 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 *simiar 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 likeService-oriented Architecture(SOA)[9] pushes more relying on software that operated in the other part of network[3] 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[10] 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 is 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, Hyper/J 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 1.1 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 method for perform method-based explicit refinement to applying variability, present more advanced implicit refinement like trait[15] or mixin[17] and custom refinement. Finally we analyse evaluation result of Self in both empirical and predictive studies.

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. For example, a behavior of *sending a message* in messaging program is consist of commonalities and variabilities. 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 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

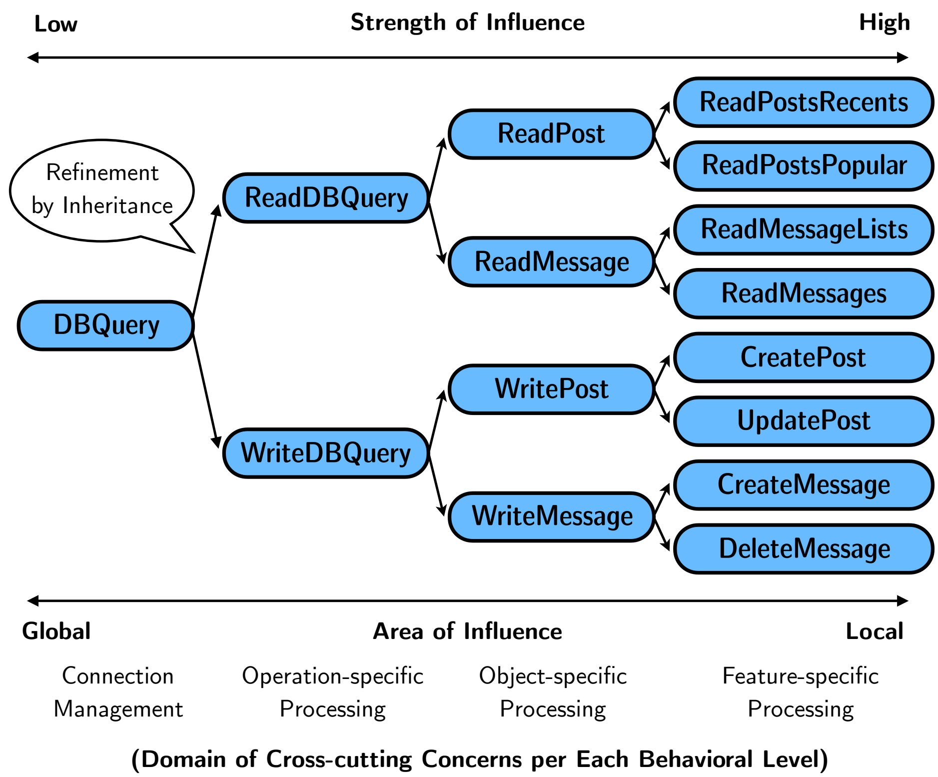


Figure 1: Multi-level Inheritance on Behavior of Database Module

refinement, multi-level inheritance is use for creating the refinable 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-

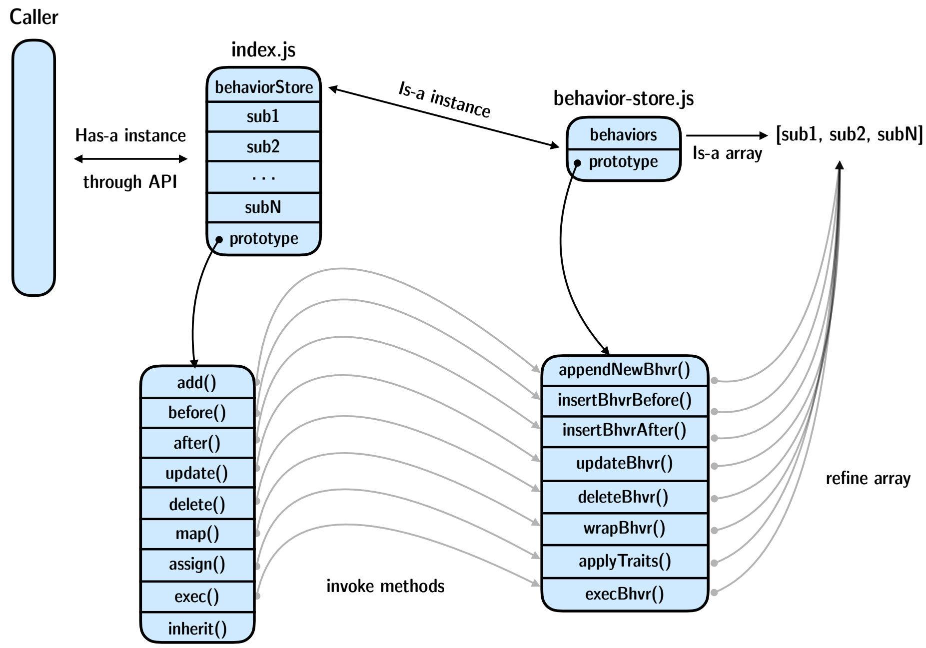


Figure 2: Architecture of Self

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.

Table 1: Method List of Self

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

[a] All method takes single native Function Object or Behavior Object created by Self.

5.2 Behavior Construction

```
var Behavior = require('self');

var DBQuery = new Behavior();

DBQuery.add(auth);
DBQuery.add(validate);
DBQuery.add(monit);
```

```
/* CONSTRUCTION PART */

// define self
var Behavior = require('self');

// initialising behavior
var DBQuery = new Behavior();

// adds some sub-behaviors
DBQuery.add(auth); // authentication checker
DBQuery.add(validate); // data validation
DBQuery.add(monit); // monitoring


/* REFINEMENT PART */

// inherit DBQuery to operation-specific, WriteDBQuery
var WriteDBQuery = new DBQuery();

// add some sub-behaviors (refinements)
WriteDBQuery.add(writeBack);

// update specified sub-behavior to new sub-behavior
WriteDBQuery.monitoring.update(cacheMonit);

// add sub-behavior in specified location
WriteDBQuery.validate.before(beforeValidate);
WriteDBQuery.validate.after(afterValidate);

// manipulating sub-behavior
WriteDBQuery.validate.map(() => {
return (validate) => {
  validateWrapper(validate);
}
});
//delete sub-behavior
WriteDBQuery.beforeValidate.delete();


/* ADDITIONAL REFINEMENT */

// inherit WriteDBQuery to object-specific query
var CreatePost = new WriteDBQuery();
var CreateMessage = new WriteDBQuery();

CreatePost.add(createUserSQLExec);
CreateMessage(createMsgSQLExec);

//additional modification
CreatePost.auth.update(2factorAuth);
CreateMessage.auth.before(geographicalBlock);
```

Listing 1: Self in a Nutshell

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

```
/* Operation-specific Processing */
var ReadDBQuery = new DBQuery();
var WriteDBQuery = new DBQuery();

// ...some refinement


/* Object-specific Processing */
var ReadPost = new ReadDBQuery();
var ReadMessage = new ReadDBQuery();
var WritePosts = new WriteDBQuery();
var WriteMessage = new WriteDBQuery();

// ...some refinement


/* Feature-specific Processing */
var ReadPostsRecents = new ReadPosts();
var ReadPostsPopular = new ReadPosts();
var ReadMessageLists = new ReadMessage();
var ReadMessages = new ReadMessage();
var CreatePost = new WritePost();
var UpdatePost = new WritePost();
var CreateMessage = new WriteMessage();
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

```
var WriteDBQuery = new DBQuery();

WriteDBQuery.add(writeBack);
WriteDBQuery.monitoring.update(cacheMonit);
WriteDBQuery.validate.before(beforeValidate);
WriteDBQuery.validate.after(afterValidate);
WriteDBQuery.validate.map(() => {
return (validate) => {
  validateWrapper(validate);
}
```

```
});
WriteDBQuery.beforeValidate.delete();

var CreatePost = new WriteDBQuery();
var CreateMessage = new WriteDBQuery();

CreatePost.add(createUserSQLExec);
CreateMessage(createMsgSQLExec);
CreatePost.auth.update(2factorAuth);
CreateMessage.auth.before(geographicalBlock);
```

Listing 4: Explicit Refinement of Self-composable behavior

In Listing 4, we refine DBQuery created in Listing 3 to create WriteDBQuery, and refine again for creating CreatePost and CreateMessage. Refinement is performed directly by various method of Self. Additional append of behavior can be performed by .add method and behavior insertion of relative location can be done by, .before and .after for custom pre and post processing. .map method is used to manipulating sub behavior in the context of given function. In this case, a validateWrapper function is used for manipulation. For the convenience of refinement Self exposes sub-behavior as property of object and each method is invoked from on each exposed sub-behavior. By specifying a name of behavior, we could specify location of behavior to perform the internals array manipulation operation.

5.5 Implicit Behavior Refinement

As Self relies on the OO system, by using well-established OO composition technique can be possible, while the disadvantage of explicit refinement is, because they are powerful, by performing many manipulations possibly occur break of correctness and decreases comprehension of code which result occur unexpected behavior or programmer to grasp system behavior directly. For using software composition technique like traits which are set of object independent behavior[15] made possible for high-level implicit refinement. Listing 5 shows implicit refinement for making public web API by using publicApiTraits traits. publicApiTraits is a trait of behavior that does not require authentication which represented in auth : null in trait object. By using .assign method, publicApiTraits can be applied to WriteDBQuery behavior. assign method which works similarly in native composition function Object.assign, a set of sub behavior could update or deleted by assigning new behavior or null object.

```
var publicApiTraits = {
    auth: null
};

WriteDBQuery.assign(publicApiTraits);
```

Listing 5: Implicit Refinement of Self-composable behavior

5.6 Custom Behavior Refinement

```
Behavior.defineMethod('deleteAddition',
    function () {
  var self = this;
  this.behaviorStore.behaviors.forEach(
      function (behavior) {
    if (behavior.name.slice(0, 3) === 'add') {
      self.delete.apply({name: behavior.name,
          behaviorStore: self.behaviorStore})
          ;
      // or by using private API
      //self.behaviorStore.deleteBehavior(
          behavior.name);
    }
  });
});
```

Listing 6: Custom Refinement of Self-composable behavior

By using defineMethod, user can create custom refinement method by accessing behavior array in the function. The following example is removing sub-behavior which start with add by doing simple pattern matching. In the definition, usage of standard API is possible by apply method with a custom scope.

5.7 Execution

```
CreateMessage.exec([arguments], Handler);
```

Listing 7: Ecxecution of Self-composable behavior

Execution of a function can be done by calling .exec method. Since all behavior object is plain JavaScript object it has the flexibility to operated in a various way including exportation of as a module. When .exec method invokes, first sub-behavior will execute with the given initial argument.

6. Analysis

6.1 Objectives

Despite one of the original contribution of Self is native, OO-based variability management, to further, In this section, we analyse and evaluate modularity improvement of programming and architecture using Self both conceptually and empirically. The programming technique is hard to evaluate because in effects not only optimising SLOC(source lines of code) but comprehension and architecture of program, as a result, without deep empirical studies across various domain, it is hard to proof its efficiency. In this analysis, we focus on identifying which factors of Self are benefiting modularity. In this evaluation, we implement some of the core modules of web service base on Self and jsAspect - an AOP library for JavaScript. To perform predictive analysis, regression analysis used to predict SLOC comparison of both techniques to analyses flexible reusability along with software growth. The goal of this evaluation is, analysis of how OO-based higher-order reusing provides better modularity then aspect from AOP.

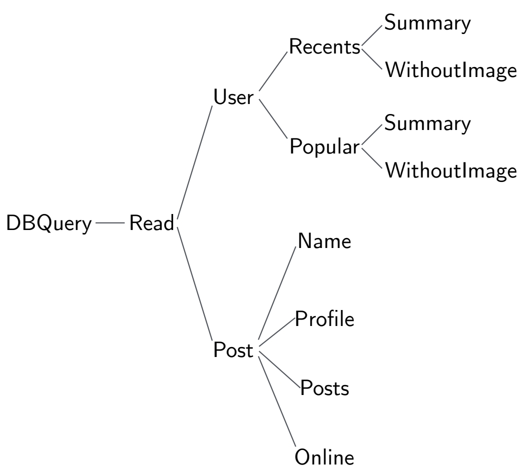
6.2 First Evaluation : Required SLOC per New Feature

In the first analysis, we implemented a feature for database read operation for Userand Post to derive required SLOC. It refines 8 final feature through passing four-level of inheritance. The feature has variability on authentication check and has commonality on all the other parts. As fugure 3 we performs self-composable domain analysis for making four-level of cross-cutting concerns - operationa-specific, object-specific, feature-specific and type specific (belongs to Post only). We have implemented each feature with AOP and Self as shown in the table 2, and the source code of each program with its helper function for JavaScript AOP based on jsAspect[22], a partial JavaScript implementation of AspectJ. The source code for

Table 2: Feature list of first analysis software

Method Name	Description
User.getName	Get specified user name with authentication
User.getProfile	Get specified user profile with authentication
User.getPosts	Get specified user posts with authentication
User.getOnline	Get specified user status with authentication
Post.getRecentSummary	Get recent post summary with authentication
Post.getRecentsWithoutImage	Get recent post text with authentication
Post.getPopularSummary	Get recent post summary without authentication
Post.getPopularWithoutImage	Get recent post text without authentication

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 retrived as concatenating left side to right side destination node

evaluation is available in Appendix A, B, C. 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(variabilities) 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 toral (b in Table 3). In the code for creating the object, we could confirm their sub-behavior are - logging, auth, cacheLookup, userIdValidation is scattered in similar pattern and AOP-based implementation made duplicated declaration and result could not perform reuse caused by variability from auth.

Table 3: Average SLOC of per Single Feature Implementation

	AOP	Self
SLOC of Whole Coordination Module (a)	26	14
SLOC of Cross-cutting Concern (b)	18	6
Number of Feture	8	
Average SLOC of per Single Feature Implementation (b/8)	2.25	0.75

Table 4: Self : SLOC per Each Level of Inheritance

Lev. of Inheritance	Num. of Parents	Num. of Child	Refinement SLOC	Total SLOC
First	1	2	5	10
Second	2	5	5	50
Third	5	10	5	250
Forth		*(projected)*		1250
Fifth		*(projected)*		6249

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

Table 5: AOP : SLOC per Each Level of Inheritance

Lev. of Inheritance	Num. of Parents	Num. of Child	Refinement SLOC	Total SLOC
First	1	2	10	20
Second	2	5	15	150
Third	5	10	20	1000
Forth		*(projected)*		7211
Fifth		*(projected)*		50988

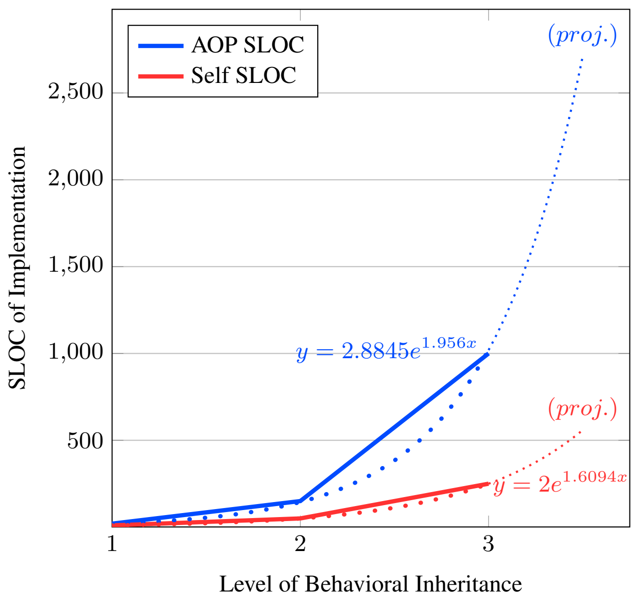


Figure 4: Prediction of Growing SLOC by Level of Inheritance

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: A. As a result, the amount of SLOC increases like the table 5. In the case of Self of the C, 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 5 is kept on 5 line.

6.4 Result and Improvement

As a result of deriving the trend function based on the table 4 and 5 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 on software product line engineering line like Delta-oriented Programming[8], a bit old researches including 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 concerns. The main difference between Self is a perspective of variability management. These approaches look the problem still OO perspective, while Self looks at behavior-oriented as the primitive citizen in programming. Although Self showed better performance compared to existing methods, such as the result of 4 in modularity, but this preliminary evaluation still requires more deep empirical studies. The test aimed to It was for confirmation at the level. The other part of Self differs from the existing method, which provides a way to flexibly model the behavior of the behavior using OOP ideas at the code level without the aid of code generators, IDEs, and tools. Self provides a framework of thinking that can create an action-only hierarchy independent of an object, by providing a perspective that sees the behavior of software in isolation from objects. We propose to programmers how they can model the features of the software they develop through self-composable domain analysis. The ultimate goal of Self is to make it easy to create modules with the variability that operate as built-in functions of a library or 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 designing the software apart from the use of AOP implementations, [30], 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 support statement computation, new object system and operations for a statement have to be researched.

8.2 Future Research Directions

The major limitation of Self is that it is also in line with the limitations of OOP. It has been possible to localise commonalities through multi-level inheritance and multiple application of refinement, but there is a problem that refinement and inheritance can be scattered. As a result, the end user may have difficulty grasping 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[23], for solidly separating various refinement by types and context space is needed to be adapted. As the previous experimental 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 wrapper 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 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 technuque by integrating the Self with the package management system such as npm[2] or pip[3]. In order to facilitate the composition of behaviors, the feature interaction problem has to be resolved[32]. 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

```
var ReadUser = createAspect(function () {
  logging(data);
  auth(data);
  cacheLookup(data);
  userIdValidation(data);
});

var ReadUserWithoutAuth = createAspect(function () {
  logging(data);
  cacheLookup(data);
  userIdValidation(data);
});

var User = {
   getName: applyAspect(ReadUser, readUserNameQuery),
   getProfile: applyAspect(ReadUser, readUserProfileQuery),
   getPosts: applyAspect(ReadUser, readUserPosts),
   getOnline: applyAspect(ReadUserWithoutAuth, readUserOnline)
};

ReadPost = createAspect(function () {
  logging(data);
  cacheLookup(data);
  postNumberValidation(data);
  rangeValidation(data);
  ReadRecentsSummaryQuery(data);
});

ReadPostWithoutAuth = createAspect(function () {
  logging(data);
  auth(data);
  cacheLookup(data);
  postNumberValidation(data);
  rangeValidation(data);
  ReadRecentsSummaryQuery(data);
});

var Post = {
  getRecentSummary: applyAspect(ReadPost, readPostRecentsSummary)
  getRecentsWithoutImage: applyAspect(ReadPost, readPostRecentsWithoutImage)
  getPopularSummary: applyAspect(ReadPostWithoutAuth, readPostPopularSummary)
```

[2] Package manager for JavaScript npmjs.com

[3] Package manager for Python pip.pypa.io

```
  getPopularWithoutImage: applyAspect(ReadPostWithoutAuth, readPostPopularSummary)
}
```

B. Source Code of AOP Helper

```
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

```
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

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," tech. rep., Technical Report CMU/SEI-90- TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[2] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[3] M. Shaw, "Modularity for the modern world: summary of invited keynote," in *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD*, pp. 1–6, 2011.

[4] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-oriented software product lines: concepts and implementation*. Springer Science & Business Media, 2013.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*, pp. 220–242. Springer Berlin Heidelberg, 1997.

[6] W. Harrison, H. Ossher, P. Tarr, and W. Harrison, "Asymmetrically vs. symmetrically organized paradigms for software composition," *IBM Rsch. Rpt. RC22685 (W0212-147)*, 2002.

[7] H. Ossher and P. Tarr, "Hyper/j: multi-dimensional separation of concerns for java," in *Proceedings of the 22nd international conference on Software engineering*, pp. 734–737, ACM, 2000.

[8] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-oriented programming of software product lines," in *International Conference on Software Product Lines*, pp. 77–91, Springer, 2010.

[9] T. Erl, *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 2005.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *European Conference on Object-Oriented Programming*, pp. 327–354, Springer, 2001.

[11] G. Kiczales, "Its not metaprogramming," *Software Development Magazine*, 2004.

[12] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, "Some features of the simula 67 language," in *Proceedings of the Second Conference on Applications of Simulations*, pp. 29–31, Winter Simulation Conference, 1968.

[13] A. P. Black, "Object-oriented programming: Some history, and challenges for the next fifty years," *Information and Computation*, vol. 231, pp. 3–20, 2013.

[14] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr, "N degrees of separation: multi-dimensional separation of concerns," in *Proceedings of the 21st international conference on Software engineering*, pp. 107–119, ACM, 1999.

[15] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, "Traits: Composable units of behaviour," in *European Conference on Object-Oriented Programming*, pp. 248–274, Springer, 2003.

[16] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *ECOOP'93 - Object-Oriented Programming, 7th European Conference*, pp. 406–431, 1993.

[17] G. Bracha and W. Cook, "Mixin-based inheritance," *ACM Sigplan Notices*, vol. 25, no. 10, pp. 303–311, 1990.

[18] M. D. McIlroy, E. N. Pinson, and B. A. Tague, "Unix time-sharing system: Foreword," *The Bell System Technical Journal*, vol. 57, pp. 1899–1904, July 1978.

[19] W. Teitelman, "Pilot: a step toward man-computer symbiosis," tech. rep., PhD thesis, September 1966.

[20] "Self - Refinable Function Constructor for Better Modularity Webpage." https://hiun.org/self.

[21] "Node.js Webpage." https://nodejs.org.

[22] "jsAspect Webpage." https://github.com/antivanov/jsAspect.

[23] T. Reenskaug and J. O. Coplien, "The DCI architecture: A new vision of object-oriented programming," *An article starting a new blog:(14pp) http://www. artima. com/articles/dci_vision. html*, 2009.

[24] S. Hanenberg and R. Unland, "Concerning aop and inheritance," in *Aspektorientierung-Workshop der GI-Fachgruppe*, vol. 2, pp. 3–4, 2001.

[25] S. Hanenberg and R. Unland, "Using and reusing aspects in aspectj," in *Workshop on Advanced Separation of Concerns, OOPSLA*, 2001.

[26] D. Batory and S. O'malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 4, pp. 355–398, 1992.

[27] W. Harrison and H. Ossher, *Subject-oriented programming: a critique of pure objects*, vol. 28. ACM, 1993.

[28] M. Mezini and K. Lieberherr, "Adaptive plug-and-play components for evolutionary software development," in *ACM Sigplan Notices*, vol. 33, pp. 97–116, ACM, 1998.

[29] M. VanHilst and D. Notkin, "Using role components in implement collaboration-based designs," *ACM SIGPLAN Notices*, vol. 31, no. 10, pp. 359–369, 1996.

[30] G. Kiczales, "Once more, from the top," *Software Development Magazine*, 2005.

[31] T. Reenskaug, "The model-view-controller (mvc) its past and present," *University of Oslo Draft*, 2003.

[32] S. Apel and C. Kästner, "An overview of feature-oriented software development.," *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.

[33] R. E. Lopez-Herrejon, D. Batory, and W. Cook, "Evaluating support for features in advanced modularization technologies," in *European Conference on Object-Oriented Programming*, pp. 169–194, Springer, 2005.

[34] D. Batory, "Feature-oriented programming and the ahead tool suite," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pp. 702–703, IEEE, 2004.

[35] R. E. Lopez-Herrejon and D. Batory, "A standard problem for evaluating product-line methodologies," in *International Symposium on Generative and Component-Based Software Engineering*, pp. 10–24, Springer, 2001.

[36] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.