A third-party extension support framework using patterns

Yiyang Hao
School of Fundamental Science and Engineering
Waseda University
Tokyo, Japan
Email: felixhao@moegi.waseda.jp

Hironori Washizaki
School of Fundamental Science and Engineering
Waseda University
Tokyo, Japan
Email: washizaki@waseda.jp

Yoshiaki Fukazawa
School of Fundamental Science and Engineering
Waseda University
Tokyo, Japan
Email: fukazawa@waseda.jp

***Abstract*—Software extension is a fundamental challenge in software engineering which involves extending the functionalities of a software module without modifying it. Many modern software developers choose to adapt third-party extension platform to further improve customizability. As the project evolves, the requirements may change to include third-party extension support. However to design and to implement such platform is no trivial task, and should happen at the beginning of the project. In this paper, we have shown the four types of extensions that are often made to object-oriented software, namely Member Access Extension, Subclass Extension, Event-based Extension and Data Extension. And proposed a language-independent platform design that can be applied to an existing software project to support such third-party extensions. The platform exercises design patterns to implement its features. We also developed an Eclipse plugin that helps developers introduce the platform to existing Java software via semi-automatic code manipulation. We further conducted a comparative experiment to test our tool with volunteers from Waseda University and noticed a significant decrease of required effort.**

I. Introduction

Software extension is a fundamental challenge in software engineering [21]. Software extension involves extending the functionalities of a software module without modifying it [17]. This can be achieved during the software evolution to adapt to the changing environment and requirements [15] or after the release of the software by third-parties via third-party software components, e.g. [12] and [11]. Software extension by third party is a common practice in modern software. A well-known example is Eclipse Rich Client Platform [18]. By the end of January 2015, there have been over 12 million copies of near two thousand extensions downloaded from Eclipse Marketplace [1]. Other notable examples are web browsers [5] [2] and video games [3], where such behavior of making or using extensions are often informally referred as "modding".

Unlike traditional software extension developed by original developers, third-party extensions bring huge potential for software user customization. And user customization is a well-needed feature for commercial software [19] [9]. Beside from the obvious benefits for users, a customizable software also has such benefits for developers that the software can satisfy conflicting needs from different user groups and potential reduction of the development and maintenance cost, given that certain features are developed by third parties. Supporting extensions developed by third parties usually requires separating the core part from potential hotspots at the beginning of a project. However it is hard to take every requirement for customization into consideration at first place. So an extension support framework must be extensible itself. Those extensions developed and distributed by third parties are also referred as plugins or mods. In the rest of the paper, we use the term extension to refer to software components developed by third parties that aim to extend the functionalities of original software.

An extension support framework usually provides APIs (application programming interface) to allow extensions to interact with the software. In this paper, we analyzed and categorized the APIs used in practice, and suggest implementations for the four types of APIs using software patterns. To evaluate our approach through user study, we implemented an Eclipse plugin for Java programs for the experiments. Through our approach, we aim to ease the introduction of extension support to existing inextensible software.

The contribution of this paper includes:

- Having conducted a case study on the API design and implementation a popular video game called Minecraft, which is a typical example of implementing API upon an existing software.
- Categorizing types of the extensions made by third-parties into four categories, namely *Member Access Extension*, *Subclass Extension*, *Event-based Extension* and *Data Extension*.
- Suggesting a way to implement APIs for third-party extensions of the four types using existing software patterns and other techniques including code clone detection and reflection.
- Evaluating the approach through a comparison on the efficiency and performance of developers who use our approach or not to achieve the same objectives.

II. Third Party Extensions

We are going through this topic by answering the following two questions:

1) RQ1 What kinds of functionalities are extended?

```
public final static Block genericDirt = new GenericBlock(Material.ground)
        .setHardness(0.5F).setStepSound(Block.soundTypeGravel)
        .setBlockName("genericDirt").setCreativeTab(CreativeTabs.tabBlock);
```

Fig. 1: Object initialization using member functions.

```
@EventHandler
public void load(FMLInitializationEvent event) {
        LanguageRegistry.addName(genericItem, "Generic Item");
        LanguageRegistry.addName(genericIngot, "Generic Ingot");
}
```

Fig. 2: Register names upon loading.

2) RQ2 How are third party extensions supported?

To answer these questions we have to narrow them down. So we did a case study on a video game called Minecraft and its open sourced API implementation, Forge [4]. Forge is the implementation of the API set for extension developers. It is written in Java and consists of 199 source files and 28320 lines of code. Forge is separate from Minecraft and serves as the implementation of APIs in the form of Java source code, and requires Java binary of Minecraft to work.

We choose Minecraft as the target project of the cast study because it is a well-developed and commercially successful video game, currently owned by Microsoft, and most importantly, it has a huge extension community including more than 50,000 members on the official forum and 1,500 released extensions.

A. What functionalities are extended?

Forge supports various extensions to the game. We categorize them in to 4 groups. Forge includes decompiled obfuscated Minecraft source.

Member access: Firstly, Forge allows extensions to access some member fields and some member functions of classes from Minecraft. For example, in Figure 1, $Block$ is a class from Minecraft and $GenericBlock$ is a subclass of $Block$ defined by third party. Third party can access member functions inherited from super class $Block$ to set the properties of a new $GenericBlock$ instance.

Subclass: Secondly, as we can also see from Figure 1, Forge makes some classes like $Block$ visible and inheritable. The produced subclasses can be directly used as if they are declared in original software. Such inheritable classes share the same characteristic that (1) they are all related to domain knowledge and (2) they tend to have many subclasses in original software.

Event-based: Thirdly, there are many time-restricted processes required from or provided for extensions. For example every extension needs to register its information into a global registry before the initialization of the game itself Figure 2. An extension sometimes works as a passive content provider instead of actively interfering the execution of software.

Data extension: Finally, Forge provides manipulation of some data collections. For example, in Figure 2 we can see an extension add the name of genericItem and genericIngot into the LanguageRegistry using addName interface provided by Forge.

B. How are third-party extensions supported?

Annotations: Forge extensively uses custom annotations, a Java language feature, to ease the development of extensions. Typically, extension developers use "Mod" annotation to specify the main class of an extension, and "EventHandler" annotation to register an event listener method. Using annotations by frameworks to apply behaviors to user-defined classes and methods is common practice. But they have the disadvantage that they are only supported by Java and can be hard to find an alternative for another programming language.

Event bus: Event bus in Forge is implemented using publish-subscribe pattern. Many events in Minecraft will notify every subscriber in every loaded extension.

Subclass: Minecraft is not an open source software. But Forge includes decompiled obfuscated Minecraft source, thus making deriving subclasses possible at language level. However, it is not always feasible to include source in API implementations and open them to third parties.

Registries: Registries are the solution to adding data into the game. These registries, e.g. EntityRegistry and GameRegistry, provide interface to add new instances to the data containers in Minecraft. Those data containers are abstract data type classes.

Hooks: Forge contains four hooks, namely $ChestGenHooks$, $DungeonHooks$, $FishingHooks$ and $ForgeHooks$. Those hooks play as middle-men between data providers and data users. Typically, $ChestGenHooks$ maintains a private data container for possible loot for chests. When either or tries to generate loot for a chest, the private data container will be checked if it should be used based on the rules set up by extensions. If not, the method invocation will be passed on to original Minecraft to use the default data container. Extensions can interact with the private data container and therefore can override the default behavior of the data container in the software. Those data containers are arrays with fixed length.

Forge Mod Loader (FML) is a module in Forge that handles connecting extensions that are in the form of compiled Java jar files with Minecraft and serves as an extension container. The implementation of FML extensively uses low-level Java bytecode operations like de-obfuscating, so that framework is hardly reusable for general purposes.

III. Approach

The previous subsection described the implementation of Forge for supporting third-party extensions. Their methods will have some issues if we want to use directly in a language-independent platform design. First, annotation is a language-specific feature. Second, hooks serve as a specific use case of event dispatching system and may not be necessary. Third, Forge decompiles Minecraft source to make classes visible to extensions, which can be illegal for many commercial software products. Lastly, replacing original data containers with registry logic requires a decent amount of code modification, especially when the original data container is a simply data type. That means every access to the container needs to be reworked.

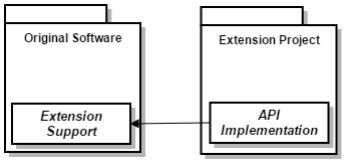


Fig. 3: Platform Infrastructure.

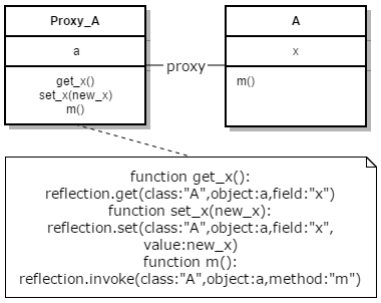


Fig. 4: Member Access through Proxy.

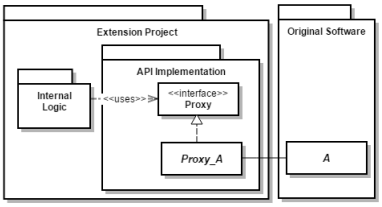


Fig. 5: Proxy Pattern.

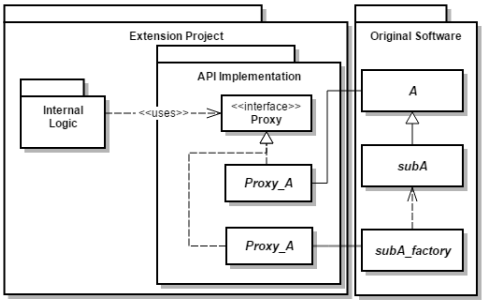


Fig. 6: Factory Pattern.

This section presents a third party extension supporting solution. Our solution covers the four types of extensions while retain minimal modification to original software and language independence. Our approach also does not require the source of original project open to third party.

A. Platform infrastructure

The basic infrastructure (Figure 3) contains extension support module embedded into original software and an API implementation package public for third party to use. Extension support module communicates with extensions using reflection. API Implementation handles the communication and makes it encapsulated and separate from the domain knowledge inside extensions.

B. Making Hotspots Extensible

As described in the previous section, there are four types of hotspots to deploy extension support framework. Our approach applies software patterns to original software to make the hotspots extensible.

C. Member Access Extension

To access member fields and methods from a class which is not visible for extensions, we suggest using proxy pattern to create proxy classes of the classes that are to be accessed. For example, In Figure 5 we have a class A in the software, then we create a proxy class for A that delegates some methods in A, thus allowing the extension uses those method via interface (Figure 4).

If we look at the case of $Block$ class from Minecraft (Figure 1), $GenericBlock$ needs to have a proxy class for it to be accessible from extensions. And since there is no guarantee that the proxy class of a class A, noted as A_proxy, remains to be the child class of the proxy class of A's father, noted as $SuperA\_proxy$, the resulting $genericDirt$ object cannot be referred as a $Block$ object directly. A explicit class casting from a proxy class to another proxy class should be used.

D. Subclass Extension

Many languages do not support creating new classes dynamically. So in our approach, we use a subclass factory to produce the instances of subclasses of a certain class. For example, In Figure 6 class $subA$ is a subclass of class A. Class $subA$ maintains a method overriding table and it overrides some methods in A in a way that it first checks if the invoked method is overridden in the extension via querying the method overriding table, if the method is overridden, then the execution will be handed over to the overriding implementation of the method, otherwise, it calls the method from its super class, A. Class subA works as if it is a decorator of A except that it is an actual subclass of A. The pseudocode of this process is shown in Algorithm 1. The interface to register overriding method in method overriding table is available in a factory class. Proxy pattern is then used to provide proxy class for the factory to allow accessing the interface from the extension.

Algorithm 1 Overriding with decorator

```
function m(parameters)
    if m is registered in overriding_table then
        m_new ← overriding_table.get(m)
        m_new(parameters)
    else
        superclass.m(parameters)
    end if
end function
```

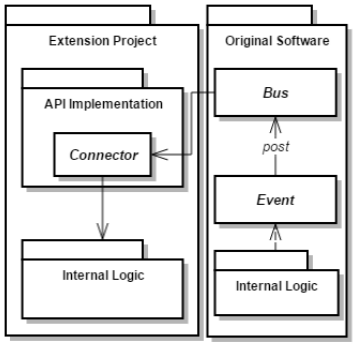


Fig. 7: Publisher-Subscriber Pattern.

All classes are managed through proxy classes in our platform. So the proxy for super class will also be the factory for its subclasses. In another word, all factory related methods are defined in the proxy class of the super class. However many languages do not allow dynamic class declaration and factories can only produce objects. So the generated subclass object should be an abstraction of the subclass and a factory for producing the instances of that subclass. For example, the code from Algorithm 2 will be used to create a subclass from class A and overriding method m of A with new_method.

Algorithm 2 Creating a new subclass with a proxy class

$Proxy\_subA \leftarrow Proxy\_A.subclass()$
$method \leftarrow MethodReflection('m')$
$Proxy\_subA.override(method, new\_method)$
$subA\_instance \leftarrow Proxy\_subA.instance()$

1) Event-based Extension: The framework will maintain a publish-subscribe model for events. When a certain event is fired, a new event instance will be pushed to the bus and then dispatched to subscriber extension. Figure 7

In the case of Figure 2, $load$ method should be registered as a subscriber for a "game load" event and that event should be fired upon the very start of the program.

2) Data Extension: As stated in the previous section, there are two types of data containers. The first one is ADT and the second one is language native data structure, e.g. arrays. After observation, we found that such data containers are used in a special pattern where successive lines of code share very similar code structure. We suggest an automatic detection algorithm for such kind of usage based on the abstract syntax tree (AST) similarity between the adjacent lines.

Our algorithm differs from code clone detection in the following ways:

- Our algorithm can have a much finer granularity than code clone detection which usually compares several lines of code. In the algorithm below, we show the process of detecting similar data container operations in consecutive statements. A statement can be easily replace with finer syntax elements like array initializer elements to detect data extension in other forms.
- Code clone detection aims to find code clone among the whole software source code, while our algorithm aims to find a unified pattern of usage of active data accessing and manipulation within a short distance span, whether in terms of time or offset in file.
- Our algorithm not only detects similar code structures, but also generates a syntax tree representation of the detected code structure with concrete information of identifiers and types that are shared among the code. The different parts are marked as blanks in the generated syntax tree so that third-parties can easily fill in the blanks and reuse the code structure we detected.

IV. Implementation

We made a tool that uses automatic code manipulation to facilitate the implementing of the discussed software patterns. Users of the tool, meanwhile authors of the software, are able to take advantage of our approach within a few clicks and a few lines of code.

Java allows reflection and Java Archive (JAR) and we use them to implement most of the features of the tool. Native reflection and JAR management require too much focus on such low level operations, so we need properly encapsulations and make them hidden from end users.

A. Platform infrastructure

Each extension is in a form of a JAR file and there is a unified entry point of an extension, which is a $Addon$ class. The tool will first generate an extension manager which loads all classes from extension JAR files into memory at the start of the software and offers basic supporting functions for the four types of extensions. The supporting functions will be explained in the following subsections.

B. Member Access Extension

The tool performs actions on existing software and provide interfaces for extensions to access internal fields and methods via reflection. As stated before, we will practice proxy pattern to make it happen. For each class that is allowed to be visible and accessed by extensions, a corresponding proxy class will be generated. And for each such field, a getter and a setter will be generated in the corresponding proxy class. The getter and setter use reflection to get and set the value of the field of corresponding object. And for each method visible and accessible for extensions, a dummy method will be generated. And it will use reflection to invoke the original method.

Due to security concerns, the user has to explicitly specify which methods and fields should be exposed to extensions, so that extension authors will interfere with internal logic less likely.

1) *Input*: original classes, fields to be exposed, method to be exposed
2) *Output*: proxy classes, getters and setters, dummy methods

C. Subclass Extension

Java does not allow dynamic class declaration during runtime. And we found a way to mimic inheritance by maintaining a method overriding registry using Algorithm 1. The interfaces provided by our tool are pretty much shown in Algorithm 2. For the ease of explanation, we are going to use the same names for the classes in Figure 6. During Step 1, $Proxy\_A$ will look for the $subA\_factory$ class with reflection. If it is found, then $Proxy\_subA$ will be created as the proxy class of $subA\_factory$. $subA\_factory$ is a class generated by our tool when the user specifies A as an inheritable class. During Step 2, $MethodReflection$ class will create a method signature called m. Later in Step 3, the signature will be used to find the m method in class A and then register new_method as the overriding method of m. Lastly in Step 4, $subA\_factory$ will produce an instance of a dummy subclass of A and pass the overriding registry to it. So that $subA\_instance$ is capable of invoking registered new_method.

However, this approach is not compatible with native Java language features. So users have to use predefined methods rather than Java syntax as the interfaces to create and to use new subclasses.

1) *Input*: classes to be inheritable
2) *Output*: dummy subclasses, subclass factories, proxy for subclass factories, method overriding support

D. Event-based Extension

The extension manager generated by our tool maintains an event bus where all events are fired and dispatched.. When the user wants to create a notification or to get a value from the extensions, an event should be created with a proper name and then the event will be dispatched to extensions.

E. Data Extension

We created a GUI for displaying and editing detected similar adjacent lines. When the tool is launched, the user will choose inheritable classes from a list suggested by the tool based on the number of the children (Figure 8). After that, a window for managing patterns of similar adjacent lines is shown (Figure 9). The first area of Figure 9 is a list of detected patterns. When a user choose a pattern from the list, area 2 will show the related source code. And area 3 will show the AST of the pattern. The differences between the AST of adjacent lines of code are marked as "mismatch" in area 2. Users can assign a "variable" to a "mismatch" with a name and a type, and it will be displayed in area 4. After all patterns are examined and proper "variable"s are assigned to every "mismatch", a "join point" will be inserted after the related code for each pattern. When the "join point" is met during execution, the corresponding pattern will be loaded into a pattern executor for execution. The "variable" is used during the execution when a "mismatch" is found. The extensions will be asked for the value of the "variable" and for each value extensions return, the pattern will be executed once with that value to replace the "mismatch".

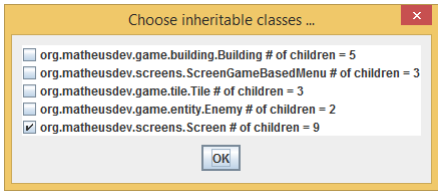


Fig. 8: GUI: Choosing classes to be inheritable

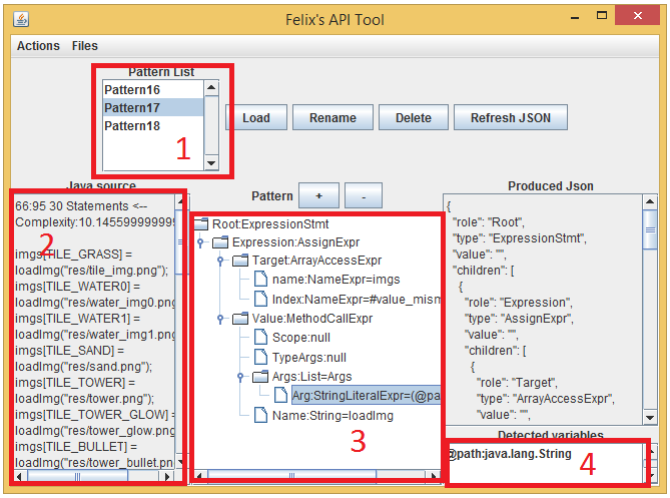


Fig. 9: GUI: Managing patterns of data extension

V. Evaluation

Our approach is evaluated based on these three questions.

1) RQ3 Is it easier to implement the four types of extensions APIs with our approach?
2) RQ4 Does this approach help lower the cost of development?
3) RQ5 Does this approach lower the complexity of code?

We implemented our approach as a tool in the form of an Eclipse plugin for evaluation purpose. The tool is designed for the developers of the original software. And it provides automatic source code generation and manipulation to implement the solutions we described in the previous section.

The main objective of this experiment was to tell whether implementing extension API for existing software that was not designed for extension at first place easier. And then, we evaluated other aspects of software development like cost and code quality.

A. Is it easier to implement the four types of extensions APIs with our approach?

We collected 5 participants from graduate and undergraduate students who major in computer science and have decent knowledge and experience about Java programming. They are asked to implement a set of APIs that involve all four types of extension for the target project TinyWorld [6]. TinyWorld is a relatively simple open source video game written in Java. Some of the APIs are not required to be implemented by undergraduate students due to the difficulty. Before the

experiment, they are provided with necessary information about the project and a document of the APIs they are asked to implement.

They will implement the APIs twice by using our approach and without using our approach. And after the experiment, we examine how many APIs are implemented correctly.

B. Does this approach help lower the cost of development?

We use number of files and lines of code (LOC) on each How produced attribute [20] to measure the cost of development. How produced consists of 5 attributes, including programmed, generated with source code generators, copied or reused without change, modified and removed. The assumption is that with lower number of files and lower LOC, the cost should be lower. Only the participants who implemented all requested APIs are included in the statistical analysis.

We also recorded the time parcitipants spent on completing each task. The time is estimated by the parcitipants themselves rather than measured directly since they spent time on getting familiar with both the target project and our approach when they are in the experiment group. The time was asked after they completed the experiment about how long they thought they spent on implementing the requested APIs.

C. Does this approach lower the complexity of code?

We compared the average cyclomatic complexity and the average lines of code per method of the implementation made by control group and experiment group. The average cyclomatic complexity of the control group is 1.7075, and the experiment group 1.36. The average lines of code per method of the control group is 8.02875 and experiment group 5.27. Both metrics dropped when students take our approach. Cyclomatic complexity dropped to 79.6% for experiment group comparing to control group and the average lines of code per method dropped to 65.6%.

After the experiment, we handed out a simple questionnaire to each participant on the objective opinion on developing experience. The questionnaire included two questions:

- Do you think our approach is helpful when implementing APIs? If so, how is it helpful?
- Do you prefer implementing APIs with our approach or without our approach?

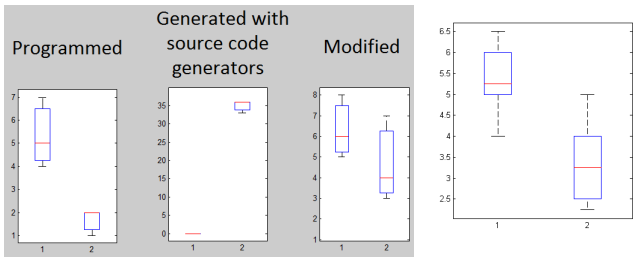
Every one of the 5 participants stated our tool were somewhat helpful. And every one of the 5 participants said they preferred implementing the APIs with our approach.

VI. Result

We collected the implementations of the APIs from all five participants, including total 47 files and 1,092 LOC from control group and 187 files and 9,425 LOC from experiment group in terms of the change metrics.

A. Is it easier to implement the four types of extensions APIs with our approach?

Without our approach, two undergraduates of our participants only correctly implemented 4 out of 6 requested APIs,



(a) Number of Files. (b) Time Spent.

Fig. 10: Code Metrics

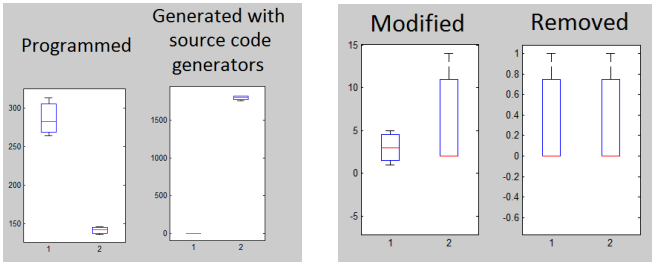


Fig. 11: Lines of Code.

while the rest (3 graduate students) implemented all 10 requested APIs correctly. With the Eclipse plugin we provide, all of them implemented all requested APIs (6 for undergraduates and 10 for graduate students) without problem. Our approach makes implementing the APIs easier for undergraduate students.

B. Does this approach help lower the cost of development?

In our experiment, among the five attributes of How produced, only programmed, generated with source code generators and modified happened in the number of files. In Figure 10a, Figure 11, group 1 is the control group where our tool is not provided, and group 2 is the experiment group where the participants are requested to use our tool. From the figures we can easily see that the implementations from experiment group have a significantly lower number of files and lines of code programmed and modified. The experiment group produced more files and LOC generated with source code generators because our tool has source code generation feature. The average modified LOC of the implementations from the experiment group is actually a little bit higher than that from control group. But the difference of programmed LOC is so large that the difference of modified LOC can be negligible. Our approach reduces the cost of developing APIs significantly.

In Figure 10b we see that control group spent more time on finishing the experiment than the experiment group. On average, 3.38 hours were spent for each task for experiment group and 5.33 hours for control group. So 1.96 hours were saved using our approach for each task on average. Thirty-minutes long tutorial on our approach and our tool is not included in Figure 10b.

Metrics	Control Group	Experiment Group	Reduction
# of files (programmed and modified)	8.75	4.88	44.3%
LOC (programmed and modified)	228.33	102.5	55.1%
Time spent	5.33	3.38	36.8%
Cyclomatic complexity	1.71	1.36	20.5%
Avg. LOC per method	8.03	5.27	34.4%

TABLE I: Comparing Control Group with Experiment Group

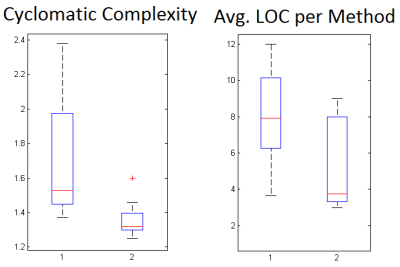


Fig. 12: Complexity Metrics.

C. Does this approach lower the complexity of code?

We compared the average cyclomatic complexity and the average lines of code per method of the implementation made by control group and experiment group. The average cyclomatic complexity of the control group is 1.71, and the experiment group 1.36. The average lines of code per method of the control group is 8.03 and experiment group 5.27. Both metrics dropped when students take our approach. Cyclomatic complexity dropped to 79.6% for experiment group comparing to control group and the average lines of code per method dropped to 65.6%. In Figure 12 we can see that medians and overall distribution of both complexity metrics from experiment group (group 2) is significantly lower than it of control group (group 1). In addition, our approach provided a unified solution to third-party extension support and greatly diminished the variance of code structure so the average cyclomatic complexity from experiment group has a much lower standard deviation than it from control group.

Compiling all collected data together and we have Table I, which brings the answers to the three research questions about evaluation.

VII. Threats to Validity

The experiment was conducted with limitations. We list some of the threats to validity in this section.

A. Threats to Internal Validity

Instrumentation: For the experiment group, they are offered with a tool in the form of an Eclipse plugin. And we taught and encouraged them to use it. The tool will automatically implement most parts of the platform design in our approach. The participants could possibly treat the tool and the interfaces it provided as requirements rather than supplements. So that they would prefer generated code untouched than make adjustments to it. This may decrease their performance. On the other hand, for the control group, reflection was the main tool they used to develop the APIs.

Compensatory rivalry: All participants were aware of the two groups and they knew which group they were in. This may cause participants in control group to try to compensate the lack of tool by putting extra efforts.

Diffusion of treatment: Every participant attended both groups due to the lack of participants. Starting with one group will surely improve the knowledge of the participant before starting with the other one. Since starting with one group first may influence on the performance when the participants was in the other group, we purposely chose randomly half of the participants to start with control group and the other half with experiment group. The duration of the experiment we used in analysis was estimated by participants themselves on how long do they need after getting familiar with the project.

B. Threats to External Validity

Population validity: All participants are students volunteers majored in software engineering from Waseda University. The 6 participants cover 3 different nationalities, aging from 23 to 29. Two of them had industry development experience with Java and others had written a few small Java projects. Despite that, they are after all students and might still not represent the average level of developers.

The target project is TinyWorld, a open-source Java video game. I chose TinyWorld because of its simplicity and all four types of extensions applicable. However a more complex project or another type of software or being written in another programming language may prove different. Experiments on other types of software and other programming languages may be our future work.

VIII. Related Work

Aspect Oriented Programming (AOP) is a programming paradigm that aims to boost isolation, composition and reuse of the code [14]. Several researches have been done on utilizing AOP for better extensibility. Kulesza et al. Alves et al. [7] and Calheiros et al. [8] use AOP to make product line evolvable to develop software of same domain. They suggest that to extract product-specific code and encapsulate it into aspects. Then, they use AOP to provide different implementations for those aspects and thus achieving evolvability. In the paper, they show an example of porting a J2ME game on different platforms with their approach. However AOP-based approaches fall short in these areas where either (1) AOP requires the development of the software done with AOP or (2) a potential massive refactoring must be done to make the software AOP-compatible.

Refactoring, or more generally, source code transformation, is another widely used means of improving source code quality including extensibility. Such technique usually starts with detecting “bad smells” [10] and then proposes source code modifications to eliminate those bad smells. Ratzinger

et al. [22] suggest that we can improve evolvability through refactoring. Tahvildari et al. [24] introduced a quality-driven OO software transformation framework which includes an instructive detection and transformation guide for primitive design patterns. These works are similar to our idea in that a bad smell detection is used to locate target code snippets and the goal is to improve some quality attributes. However, improving extensibility requires more specialized bad smell detection criteria and code transformation practice, moreover our goal reaches further than improving some quality attributes, because we aim to implement the support for third-party extensions which is an addition to the functionalities of the software.

Code clone detection is similar to one of the technique we use to automatically detect *Data Extension* type of extension. Code clone is the high identicalness of two separate code fragments. CCFinder [13] presents a typical design of code clone detection system which transforms and tokenizes code, and then performs a text-based comparison on the tokens. However, code fragment as the basic unit of code clone, is usually treated as a sequence of code lines. Even just a few lines of code can sometimes be too coarse for *Data Extension* detection. As described previously, Figure 2 shows an example of data extension happening between two consecutive lines. In some situations, developers hard-coded items in a primitive data structure like "Array" type in Java language in one single logistic line of code. Another reason for existing code clone detection techniques to fail in our use case is that code clone techniques focuses on finding remote clones rather than a dense set of operations on one data container [23]. So discovering code similarity within such fine-grained code structures cannot be achieved by conventional code clone detection tools.

IX. Conclusions

In this paper, we have shown the four types of extensions that are often made to object-oriented software, namely Member Access Extension, Subclass Extension, Event-based Extension and Data Extension. And proposed a language-independent platform design that supports such extensions. The platform exercises design patterns to implement its features. We also developed an Eclipse plugin that helps developers introduce the platform to existing Java software via semi-automatic code manipulation.

To confirm the effectiveness of this platform design and our tool, we conducted an experiment with 6 participants as the developers of TinyWorld, an open-source video game written in Java. In the experiment, we asked them to implement 10 APIs with and without our approach. It turned out that the amount of effort had been significantly reduced with our approach.

To conclude, such an approach can help in improving the extensibility of an existing software. Our future work will be further investigating into discovering more possible extension types and testing our approach on a broader selection of software.

References

[1] Eclipse marketplace. http://marketplace.eclipse.org/.

[2] Google chrome. https://chrome.google.com/webstore/.

[3] Minecraft. https://minecraft.net/.

[4] Minecraftforge. https://github.com/MinecraftForge/MinecraftForge.

[5] Mozilla firefox. https://addons.mozilla.org/.

[6] Tinyworld. https://github.com/matheus23/TinyWorld.

[7] V. Alves, P. Matos Jr, L. Cole, P. Borba, and G. Ramalho. Extracting and evolving mobile games product lines. In *Software Product Lines*, pages 70–81. Springer, 2005.

[8] F. Calheiros, V. Nepomuceno, P. Borba, S. Soares, and V. Alves. Product line variability refactoring tool. In *Proceedings of European Conference on Object-Oriented Programming*, pages 32–33, 2007.

[9] P. Dourish. Developing a reflective model of collaborative systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 2(1):40–63, 1995.

[10] M. Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 2002.

[11] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, et al. Quantum espresso: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21(39):395502, 2009.

[12] R. Joehanes and J. C. Nelson. Qgene 4.0, an extensible java qtl-analysis platform. *Bioinformatics*, 24(23):2788–2789, 2008.

[13] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97Object-oriented programming*, pages 220–242. Springer, 1997.

[15] A. C. Kouskouras, Konstantinos G. and G. Stephanides. Facilitating software extension with design patterns and aspect-oriented programming. *Journal of Systems and Software*, 81(10):1725–1737, 2008.

[16] U. Kulesza, V. Alves, A. Garcia, C. J. De Lucena, and P. Borba. Improving extensibility of object-oriented frameworks with aspect-oriented programming. In *Reuse of Off-the-Shelf Components*, pages 231–245. Springer, 2006.

[17] R. Lämmel and K. Ostermann. Software extension and integration with type classes. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 161–170. ACM, 2006.

[18] J. McAffer, J.-M. Lemieux, and C. Aniszczyk. *Eclipse rich client platform*. Addison-Wesley Professional, 2010.

[19] S. R. Page, T. J. Johnsgard, U. Albert, and C. D. Allen. User customization of a word processor. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 340–346. ACM, 1996.

[20] R. E. Park. Software size measurement: A framework for counting source statements. Technical report, DTIC Document, 1992.

[21] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, (2):128–138, 1979.

[22] J. Ratzinger, M. Fischer, and H. Gall. *Improving evolvability through refactoring*, volume 30. ACM, 2005.

[23] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[24] L. Tahvildari and K. Kontogiannis. A software transformation framework for quality-driven object-oriented re-engineering. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 596–605. IEEE, 2002.