Extended Design Patterns in New Object-Oriented Programming Languages

Kazunori Sakamoto #1, Hironori Washizaki *2, Yoshiaki Fukazawa *3

# *National Institute of Informatics*

1 `exkazuu@nii.ac.jp`

* *Dept. Computer Science and Engineering, Waseda University*

2 `washizaki@waseda.jp` 3 `fukazawa@waseda.jp`

Abstract

Most of design patterns are implemented in major object-oriented programming languages such as C++ and Java. However, newer object-oriented programming languages than such languages has new language features which can improve implementations of design patterns.

In this paper, we propose two extended design patterns called customizable state pattern and deeply immutable pattern. We compares implementations of our design patterns in Java, C++ and eight new object-oriented programming languages through our motivating example. As a result, we confirmed new languages, in particular Scala, improved implementations of our design patterns.

1. Introduction

The Gang-of-Four (GoF) design patterns (DPs) make software quality better by providing reusable design solutions [2]. The authors' sample implementations of the GoF DPs are written in C++. Other samples also exist in different programming languages such as Java and C#.

Most of new object-oriented programming (OOP) languages have functional features. For example, Java7 [1] had planned to employ closure features. Although Java8 will employ closure features instead Java7, several new languages on Java Virtual Machine (JVM) such as Scala have already employed functional features including closures.

While DPs do not depend on programming languages, implementations of DPs depend on programming languages. The reason is that each programming language has different language features so that ways to implement instances of DPs are different with respect to each programming language. For example, C++ supports multiple inheritance, while Java lacks it. Thus, it is difficult to design a class to have multiple roles in Java.

The GoF DPs are currently designed mainly for C++ and Java. However, newer OOP languages than C++ and Java can improve implementations of the GoF DPs. For example, the strategy pattern is designed as an alternative way to implement functional values. OOP languages with functional features such as Scala can directly handle functional values without the strategy pattern. As another example, Scala supports an object class which represents a singleton object with the singleton pattern.

Many researchers improved implementations of DPs with other programming paradigms than the OOP [3–5, 9]. Moreover, a number of DPs are also proposed for other paradigms than the OOP [1, 6].

In this paper, we propose two extended DPs through our development experience of ACM JavaChallenge 2012, which is an artificial intelligence programming (AI) contest related to game software [2]. We discuss how we can improve implementations of our DPs in new OOP languages including Scala in comparison with C++ and Java. As a result, we confirmed new OOP languages can improve implementations of our DPs. We believe our result indicates new OOP languages promote improving and extending existing DPs.

The contributions of this paper are as follows.

- Two extended DPs called customizable state and deeply immutable DPs.
- Improvements on implementations of our DPs in new OOP languages such as Scala.
- What language features are required to improve our design patterns.

2. Motivating example

We explain a need of extended DPs through our development experience of JavaChallenge 2012.

2.1. Requirements of sample game software

We developed game software for JavaChallenge 2012 in Java and Scala. The contestants develop AI programs on our game software to compete with each other. As a result

[1] http://www.jcp.org/en/jsr/detail?id=335

[2] We will publish a paper on ICSE GAS 2013 which highlights whole development experience of JavaChallenge 2012 including our DPs. However, this paper highlights only extended DPs in new OOP languages.

of our requirement analysis, we found the game software have to satisfy five requirements as follows. Note that we consider the game software as sample software for our discussion in this paper.

Functional requirement (FR1) The software have to make progress based on a state machine. The software shows and executes various scenes such as a title, main and end scenes corresponding to a current state.

Functional requirement (FR2) The software have to allow users to switch various playing modes such as user manipulation, AI manipulation, graphical user interface (GUI) and console user interface (CUI) modes.

Non-functional requirement, security (NFR1) Game states of the software have to be protected from user-developed AI programs. AI programs is prohibited to modify the game states illegally.

Non-functional requirement, concurrency (NFR2) The software have to be concurrently work to speed up AI programs so that it aids them to find the best action.

Non-functional requirement, maintainability (NFR3) The source code of the software have to have no duplicated code and no redundant code because such code reduces maintainability such as changeability and understandability.

We had previously developed a framework called Game AI Arena (GAIA) which aids to develop game software where user-developed AI programs can be added [7]. GAIA provides a `Scene` interface and a `SceneManager` class as a feature of a state machine designed by the state pattern. The `SceneManager` class changes game scenes by switching an active `Scene` object. The `Scene` interface has an `advance` method which returns a next `Scene` object for the switch.

2.2. Design with the state pattern in Java

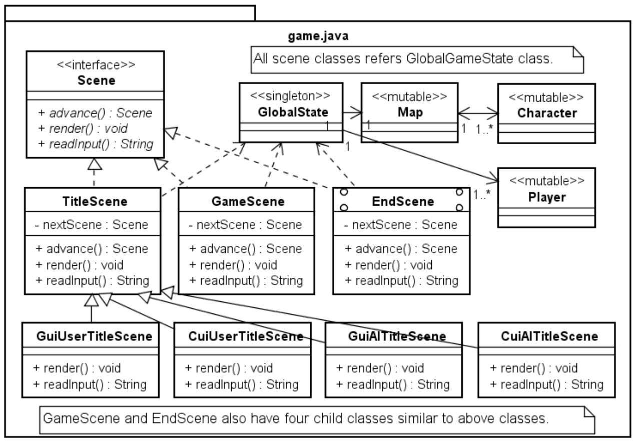

Figure 1. Class diagram of the software with the state pattern in Java

Figure 1 shows a class diagram of the software in Java. This diagram does not contain the `SceneManager` class and several unrelated classes for the simplicity. The `GlobalState`, `Map`, `Character` and `Player` classes are for representing a game state. Theses classes are mutable, and thus, their objects can be modified in playing games.

The game consists of three scenes to satisfy FR1: a title, a main, and an end scenes. The `TitleScene`, `MainScene`, `EndScene` abstract classes represent these scenes. The title scene initializes the game showing a title image, the main scene deals with the game logic showing a game screen and the end scene calculates the result of the game showing the result.

The software has four modes to satisfy FR2: a user manipulation, an AI manipulation, a GUI and a CUI modes. Although the user manipulation mode acquires game inputs from a keyboard, the AI manipulation mode acquires game inputs from AI programs. The user manipulation mode is suitable for debugging the software and AI programs. The GUI mode shows a graphical game screen, while the CUI mode shows a text-based game screen. The CUI mode is also suitable for debugging AI programs by speeding up the game. For example, the `GuiUserTitleScene` class is for the title scene with the GUI and the user manipulation modes.

This design has three problems as follows.

- The classes representing the game state are mutable. The mutable classes cause risks of illegally modifying the game states so that this design violates NFR1.
- The scene classes are strongly combined with the singleton class as a global variable of the game state. This dependence makes the concurrent execution difficult so that this design violates NFR2.
- Duplicated and redundant code exists between scene classes such as the `GuiUserTitleScene` class and the `GuiUserMainScene` class (abbreviated in the diagram) so that this design violates NFR3.

2.3. Design with the state pattern in C++

Although we can apply multiple inheritance which is applied only when defining classes into the scene classes to reduce duplicated code, applying such multiple inheritance increases classes. Figure 2 shows a class diagram of the software with the state pattern in C++, which supports the multiple inheritance. For example, duplicated code between the `GuiUserTitleScene` and `CuiUserTitleScene` classes are extracted into the `UserInputScene` class. The `GuiScene`, `CuiScene` and `AIInputScene` classes are also newly added to extract the duplicated code.

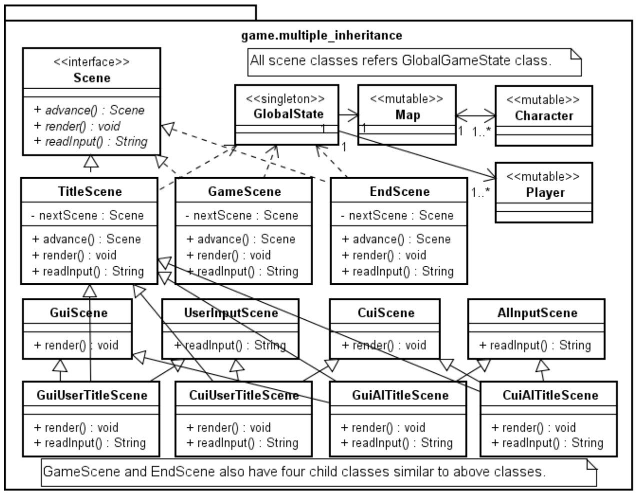

Figure 2. **Class diagram of the sample game software with the state pattern in C++**

2.4. Design with the state and decorator patterns

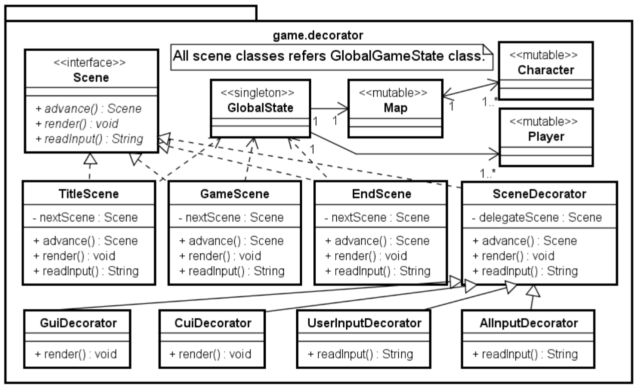

Figure 3. **Class diagram of the software with the state and decorator patterns**

```
Scene scene = new GuiDecorator(
    new UserInputDecorator(new TitleScene()));
```

Figure 4. **Java code constructing a similar object to a `GuiUserTitleScene` object**

We can also employ the decorator pattern instead of multiple inheritance. The decorator pattern allows to enhance objects with delegation. Figure 3 shows a class diagram of the software with the state and decorator patterns in Java. The `SceneDecorator` class is a base class of a decorator which has the `delegateScene` field to be enhanced. For example, Figure 4 shows Java code which constructs a similar object to a `GuiUserTitleScene` object. Although this design adds the five decorator classes such as the `SceneDecorator`, `GuiDecorator`, `CuiDecorator`, `UserInputDecorator` and `AIInputDecorator` classes, it removes 16 redundant classes such as the `GuiUserTitleScene` class in Figure 1 to satisfy NFR3. Moreover, this design extracts the duplicated code into the five decorator classes to satisfy NFR3. Thus, this design is better than the designs of Figures 1 and 2.

2.5. Design with the state pattern in Scala

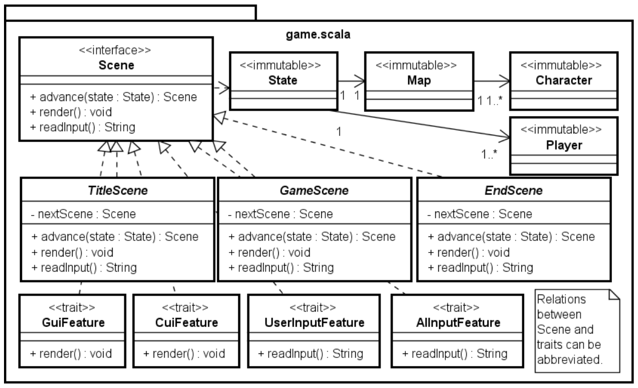

Figure 5. **Class diagram of the software with the state pattern in Scala**

```
val scene = new TitleScene()
            with UserInputFeature with GuiFeature
```

Figure 6. **Scala code constructing a similar object to a `GuiUserTitleScene` object with mixin**

Scala supports mixin instead of multiple inheritance so that Scala can merge classes with traits [8]. Scala also allows to construct an object with mixin which is applied when creating objects. Figure 5 shows a class diagram of the software with the state pattern in Scala. Figure 6 shows Scala code which constructs an similar object to a `GuiUserTitleScene` object with the mixin. Although this design adds the four traits such as the `GuiFeature`, `CuiFeature`, `UserInputFeature`, and `AIInputFeature` traits, it removes 16 classes such as the `GuiUserTitleScene` class in Figure 1 to satisfy NFR3. Moreover, this design extracts the duplicated code into the four traits to satisfy NFR3. This design is better than the designs of Figures 3 because this design does not require a similar class to the `SceneDecorator` class.

2.6. Design about immutability

Immutability is preferable because changes of variable values increase complexity of programs. Functional programming represents programs with mapping of function application while imperative programming represents programs with changes of variable values. Thus, functional programming can easily employ immutability in programs. The `GlobalState` class and the related classes such as the `Map`, `Character` and `Player` classes in Figures 1, 2 and 3 are mutable. Objects of these classes can be easily modified from other programs.

We can employ immutable objects to prevent illegal modification completely. The `State` class in Figures 5 is immutable. Moreover, the classes referred from the `State` class are also immutable. When the referred classes are mutable, the game state can be modified partially. Thus, the `State` class have to be deeply immutable to satisfy NFR1.

The `GlobalState` class is a singleton class which represents the game state. All scene classes refer the `GlobalState` class to make progress on games. This design strongly combines the scene classes with the `GlobalState` class. For example, AI programs sometimes require enormous time to search the best action. Although executing multiple games concurrently can reduce the time, this combination makes the concurrent execution difficult. Moreover, the singleton object cannot represent multiple game states. In contrast, the `State` class is used only as a parameter of methods in the scene classes. These scene classes can treat multiple game states for the concurrent execution to satisfy NFR2 because the scene classes do not have `State` objects in fields.

3. Customizable state pattern

We extract a new DP called customizable state pattern which extends the state patterns from Figures 3 and 5. In this section, we show the description of the customizable state pattern.

3.1. Context

Program behavior changes corresponding to a state based on a state machine. A program on each state deals with various operations such as doing logic and rendering an UI. The operations differ depending on options such as a CUI and a GUI modes.

3.2. Problem

Options cause combinatorial explosion so that it drastically increases conditional branches or classes. Options also cause duplicated code and redundant code so that it reduces maintainability.

3.3. Forces

- Combinatorial explosion owing to options increases program elements linearly with the number of options.
- Source code representing options does not contain duplicated code and redundant code.

3.4. Solution

Create modules representing behavior of states with respect to each option. Note that the number of modules should be equal to the number of options except for additional modules such as a base class. Combine state modules with option modules with one of the following ways.

3.5. Implementation

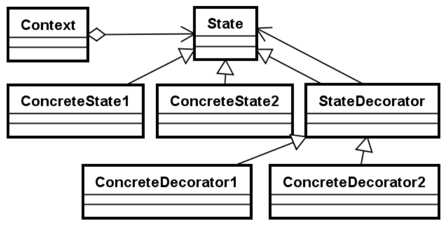

Figure 7. Class diagram of customizable state pattern with the mixin

Figure 7 shows a class diagram of the customizable state pattern with the state pattern and multiple inheritance or mixin which is applied when creating objects. The `StateFeature` modules are merged with `ConcreteState` classes corresponding to options. This implementation requires only modules corresponding to options without additional classes.

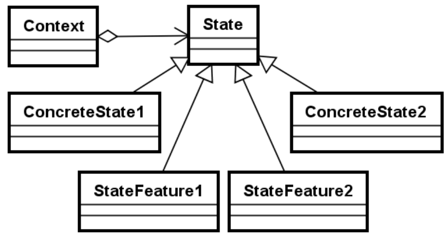

Figure 8. Class diagram of customizable state pattern with decorator pattern

Figure 8 shows a class diagram of the customizable state pattern with the state and decorator patterns. The `ConcreteDecorator` classes are corresponding to options and enhance the `ConcreteState` classes. This implementation requires additionally the `StateDecorator` class in comparison with the design of Figure 7.

3.6. Consequences

The customizable state pattern deals with the combinatorial explosion owing to options. Basically, the implementations of the options requires only the same number of the modules as the options. The modules modularize the implementations of the options well so that duplicated code and redundant code do not appear.

3.7. Related Patterns

State patter Although the state pattern does not consider how to modularize state classes which have various operations, the customizable state pattern aids to modularize state classes where options are added.

Decorator pattern and bridge pattern The decorator and bridge patterns can be utilized to modularize state classes with various operations. Although multiple inheritance or mixin which is applied when creating objects is a better way to modularize them, the decorator pattern is also one of ways to modularize them in OOP languages without the multiple inheritance and the mixin.

4. Deeply immutable pattern

We extract a new DP called deeply immutable pattern which extends the immutable patterns from Figure 5. In this section, we show the description of the deeply immutable pattern.

4.1. Context

A program consists of a set of classes for representing a program state (not related to state role for the state pattern). The program state must not be illegally changed by added user programs. The program allows to clone and restore program states to search other program states. The program can concurrently work using multiple program states to speed calculations up.

4.2. Problem

It is not clear how to modularize the program state as immutable modules in OOP languages because most of major OOP languages are designed for imperative programming. Moreover, the singleton pattern is frequently used as only global variables. However, a singleton class cannot be utilized to clone and restore program states and cannot treat multiple program states.

4.3. Forces

- Program states are protected from illegal modification by user programs.
- Program states are cloned and restored to search other program states.
- The program treats the multiple program states.

4.4. Solution

Modularize the set of classes for representing a program state with a tree structure making all the fields in the classes immutable. Determine one of program state classes as a root class and the other classes as node classes. Change each class so that they have one-way connections to child node classes and all node classes can be scanned from the root class. Copy the root and node classes from a modified node class recursively when generating a new program state.

4.5. Implementation

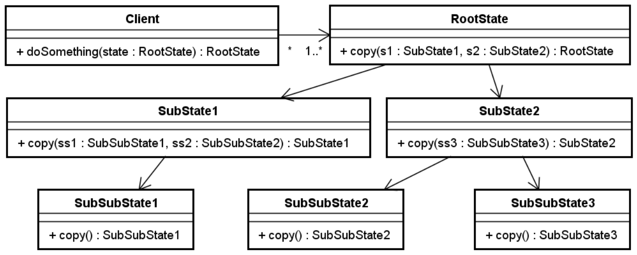
Client
+ doSomething(state : RootState) : RootState
*
1..*
RootState
+ copy(s1 : SubState1, s2 : SubState2) : RootState
SubState1
+ copy(ss1 : SubSubState1, ss2 : SubSubState2) : SubState1
SubState2
+ copy(ss3 : SubSubState3) : SubState2
SubSubState1
+ copy() : SubSubState1
SubSubState2
+ copy() : SubSubState2
SubSubState3
+ copy() : SubSubState3

Figure 9. Class diagram of deeply immutable pattern

Figure 9 shows a class diagram of the deeply immutable pattern. The `Client` class manipulates the program state locally (e.g. acquiring a `State` object as a parameter and passing a new modified object to other methods) because a global immutable object cannot be utilized to represent changeable program states.

```
case class State(SubState1 sub1, SubState2 sub2) {
  def newState() = {
    val newSub1 = createNewSub1()
    this.copy(sub1 = newSub1)
  }
}
case class SubState1(SubSubState1 subSub1,
                     SubSubState2 subSub2) {}
case class SubState2(SubSubState3 subSub3) {}
case class SubSubState1() {}
case class SubSubState2() {}

val state: State = initialize()
val newState = state.newState()
```

Figure 10. Scala code which defines the classes

Figure 10 shows Scala code which defines the classes in Figure 9. Several OOP languages with functional features such as Scala aids to implement immutable classes. Case class in Scala generates several useful methods including the `copy` method. The `copy` method copies the receiver's object accepting parameters for changing the specific fields. The Scala code in Figure 10 generates a new `State` object by modifying `sub1` field with a return value of the `createNewSub1` method.

4.6. Consequences

The deeply immutable pattern protects program states from being modified illegally. This pattern also allows to utilize immutable classes for representing changeable program state by generating and passing modified program states.

4.7. Related Pattern

Immutable pattern The immutable pattern treats only a target class without referring classes so that the target class can have relations to mutable classes. Thus, immutable pattern cannot guarantee a set of classes for representing program states are immutable.

5. Evaluation

We evaluate our extended DPs through the sample software. Table 1 shows the numbers of required classes and relations which indicate inheritances and references, and existence of duplicated code in the state and customizable state patterns, respectively. We consider six new OOP languages on the JVM and two new OOP languages on the .NET Framework: Scala, Kotlin, Xtend, Ceylon, Fantom, Gosu, F# and Nemerle.

Let S be the number of state classes and O be the number of combinations of options, and P be the number of option classes which a state depends on. Note that the numbers of required classes and relations are expressed as formulas with concrete numbers in the case of the sample software.

The implementations with the customizable state pattern have only $S+O+1$ or $S+O$ classes with $S+O+2$ or $S+O$ relations while them without the customizable state pattern has $S+S*O$ or $S+S*O+1$ classes with $S*O$ or $S*O*(P+1)$ relations. Scala, Kotlin, and Ceylon support mixin which is applied when creating objects using an anonymous class so that they remove classes and relations about decorators. The customizable state pattern reduces approximately 50% classes and 33% relations at least for the sample software.

Table 2 shows the numbers of required implementations for immutable classes and of required copy methods. Let F be the number of fields in classes for representing a program state and C be the number of classes for representing a program state.

Xtend and Fantom support annotations for marking immutable classes. Fields in Ceylon and Nemerle are immutable by default. Fantom can check whether an immutable class is deeply immutable at compile-time. Scala and F# automatically generate copy methods for updating immutable objects. Kotlin will support immutability with copy methods similarly to Scala. Thus, several new OOP languages, in particular Scala and F#, can reduce implementation costs for applying the deeply immutable pattern.

Table 1. Evaluation of customizable state pattern

Implementation	Class Relation	Duplication
State /wo multiple inheritance and mixin in Java, Xtend, Nemerle, F#	$S+S*O$ (15) $S*O$ (12)	Exist
State /w multiple inheritance and mixin in C++, Scala, Kotlin, Gosu Ceylon, Fantom	$S+S*O+1$ (16) $S*O*(P+1)$ (36)	None
Customizable state with decorator in Java, C++, Xtend, Fantom, Gosu, Nemerle, F#	$S+O+1$ (8) $S+O+2$ (9)	None
Customizable state with mixin in Scala, Kotlin, Ceylon	$S+O$ (7) $S+O$ (7)	None

Table 2. Evaluation of deeply immutable pattern

Implementation	Implementation for immutability	Copy method
C++, Java, Kotlin, Gosu	F	C
Xtend, Ceylon, Fantom, Nemerle	0	C
Scala, F#	0	0

6. Conclusions

We proposed the customizable state pattern and the deeply immutable pattern. We found that newer OOP languages such as Scala than traditional ones such as C++ and Java can improve implementations of our DPs.

We plan to improve existing DP such as the GoF DP in new OOP languages with functional features. Moreover, we will find which language features can improve DPs and how the features can improve DP with case study.

References

[1] S. Antoy and M. Hanus. New functional logic design patterns. In *Proceedings of the 20th international conference on Functional and constraint logic programming*, WFLP'11, pages 19–34, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[3] J. a. L. Gomes and M. P. Monteiro. Design pattern implementation in object teams. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2119–2120, New York, NY, USA, 2010. ACM.

[4] O. Hachani and D. Bardou. Using aspect-oriented programming for design patterns implementation. In *In Proc. Workshop Reuse in Object-Oriented Information Systems Design*, 2002.

[5] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37(11):161–173, Nov. 2002.

[6] R. Lämmel and J. Visser. Design patterns for functional strategic programming. In *Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, RULE '02, pages 1–14, New York, NY, USA, 2002. ACM.

[7] K. Sakamoto, A. Ohashi, H. Washizaki, and Y. Fukazawa. A framework for game software which users play through artificial intelligence programming (in japanese). *IEICE Transactions*, 95(3):412–424, mar 2012.

[8] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer Berlin Heidelberg, 2003.

[9] G. T. Sullivan. Advanced programming language features for executable design patterns "better pattern through reflection", 2002.