Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 9: State Design Pattern

Contents

9.1 Introduction

In this lecture you will learn all about the State design pattern. It is a good example of the application of the Protected Variation Principle that is commonly applied to alleviate problems associated with having to change systems during or after development. Therefore, this principle is briefly discussed.

9.2 Protected Variations GRASP principle

General Responsibility Assignment Software Principles (GRASP) is a series of 9 principles proposed by Larman [3] that should be used to govern software design. It is a learning aid to help you understand essential object design and apply design reasoning in a methodical, rational, explainable way. When we discuss the use of the different design patterns, we will refer to the GRASP principles applied by these patterns and introduce these principles as needed.

One of the problems that has been identified as a reason why projects fail is the impact of change during development. Change is inevitable. When mismatches in the expectations and perceptions of different stakeholders are discovered, adjustments are needed to resolve these mismatches in order to deliver a satisfactory system.

The principle we introduce here is called **Protected Variations** (PV). It addresses the problem of the introduction of bugs when code is changed by designing to minimise the impact of code change on other parts of the system. PV suggests that points of predicted variation or instability be identified, and that responsibilities be assigned in such a way that a stable interface is created around them ([3]:427). This is generally done by adding a level of indirection, an interface, and using polymorphism to deal with the identified points of predicted variation.

9.3 State Design Pattern

9.3.1 Identification

Name	**Classification**	**Strategy**
State	Behavioural	Delegation
Intent		
Allow an object to alter its behaviour when its internal state changes.The object will appear to change its class. ([1]:305)		

9.3.2 Structure

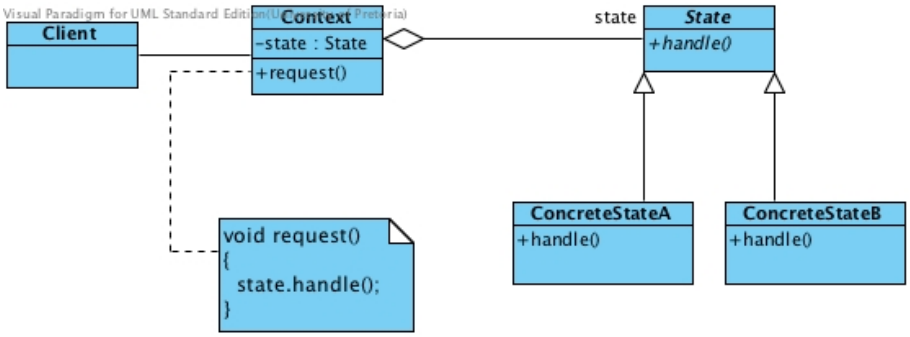

Figure 1: The structure of the State Design Pattern

9.3.3 Problem

The two major problem areas [2] that exist in which the state pattern can make a contribution are:

- when an object becomes large; and
- when there is an extensive number of state changes an object can go through.

The behaviour an object exhibits is dependent on the state of the object. Changing the state of an object will therefore influence the behaviour of the object at run-time. When objects become large, changing their state can become difficult. In order to control the complexity of changing state, the state of the object is managed externally to the object itself.

An object may be required to change state many times and into many different states. When these states become numerous and the flow is controlled by choice-statements (such as `if` or `switch` in C++), the ability to manage the statement flow diminishes. In order to control this complexity, the state of an object can be managed externally to the object itself by modelling the states as objects in their own right.

9.3.4 Participants

State

- Defines an interface for encapsulating the behaviour associated with a particular state of the Context.

ConcreteState

- Implements a behaviour associated with a state of the Context.

Context

- Maintains an instance of a ConcreteState subclass that defines the current state.
- Defines the interface of interest to clients.

9.4 State Pattern Explained

The application of the State Design Pattern is an example of the application of the PV principle. If the system is required to change its behaviour depending on its state, the different behaviours are a candidate for a point of predicted variation which should be wrapped in a stable interface. The `State` participant in the State Design Pattern serves as the stable interface required by this principle while the concrete state classes are isolated in the system to accommodate changes that will not impact on the rest of the system.

The State Pattern applies polymorphism to define different behaviour for different states. Thus, behaviour is altered by executing the code that is implemented in different subclasses. However, the strategy used by this pattern is mainly delegation. The Context class delegates work to be done to these polymorphic classes instead of doing it itself. This way no complicated decision structures are needed in the Context class to cater for different state dependent behaviours.

Client programs should not interact directly with the states. All calls to methods in the concrete state classes are redirections from methods in the context class. Thus, clients will only call methods in the context class. Clients are also not allowed to change the state of the context without the context's knowledge.

9.4.1 Improvements achieved

- **Increased maintainability**
  The State pattern puts all behaviour associated with a particular state into one object. Because all state-specific code lives in a State subclass the code associated with behaviour in a given state is localised and hence easier to maintain. It is also easy to define more states and transitions by defining new subclasses.

- **Eliminate large conditional statements**
  Like long procedures, large conditional statements are undesirable.They're monolithic and tend to make the code less explicit, which in turn makes them difficult to modify and extend. The State pattern offers a better way to structure state-specific code. The logic that determines the state transitions doesn't reside in monolithic `if` or `switch` statements but instead is partitioned between the State subclasses. That imposes structure on the code and makes its intent clearer.

- **Makes state transitions explicit**
  When an object defines its current state solely in terms of internal data values, its state transitions are implicit and have no explicit representation; they only show up as assignments to some variables. Introducing separate objects for different states makes the transitions more explicit. Also, State objects can protect the Context from inconsistent internal states, because state transitions are atomic from the Context's perspective – they happen by rebinding one variable (the Context's State object variable), not several.

9.4.2 Disadvantages

- **Higher coupling**
  The State pattern introduce high coupling. The pattern distributes behaviour for different states across several State subclasses. This increases the number of classes and is less compact than a single class and will require more communication between classes that would be the case if all was implemented in a single class.

9.4.3 Implementation Issues

9.4.3.1 Changing State

When implementing the state pattern one has to decide which class will be responsible to implement the change of state. There are three possible methods that can be applied:

a) **Context applying fixed Criteria**
If the criteria are fixed, then they can be implemented entirely in the Context in the normal flow of events. If this can not be done without conditional statements this approach is probably not appropriate. The disadvantage of this method is that it is not flexible and it is likely that the context code has to change when more states are added to the system. It is generally more flexible and appropriate, however, to let the State sub classes themselves specify their successor state and when to make the transition as described in the following two methods.

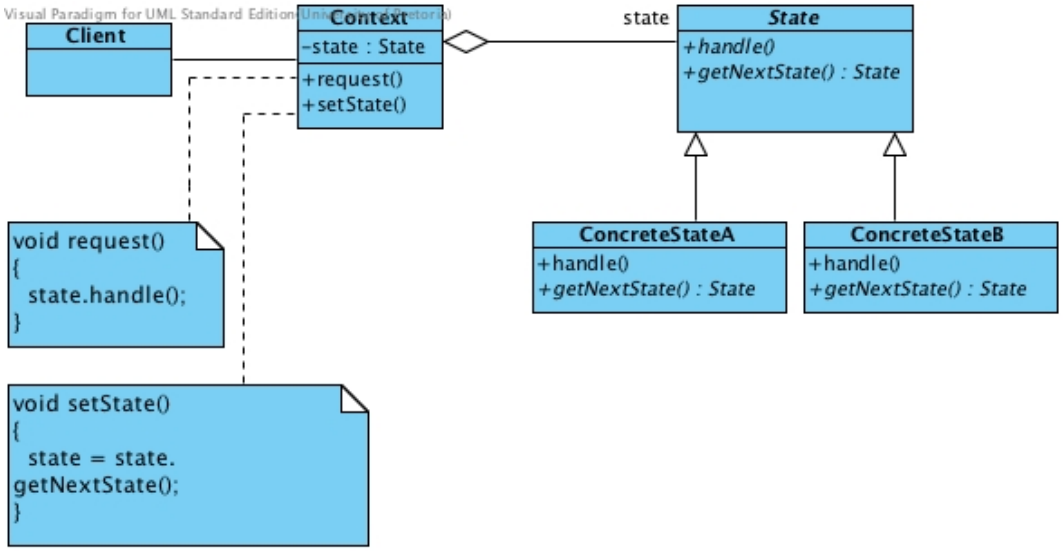

Figure 2: The context is responsible for changing state

b) **Context applying variable criteria**
When the context is responsible for changing the state it will be done in terms of an implementation as shown in Figure 2. Decentralising the transition logic in this way makes it easy to modify or extend the logic by defining new State subclasses. In this case coupling is lower than in the following method since the State is unaware of the Contexts. However, memory management in the context might be difficult. The new

state needs to be requested from the current state and thereafter the current state value has to be updated.

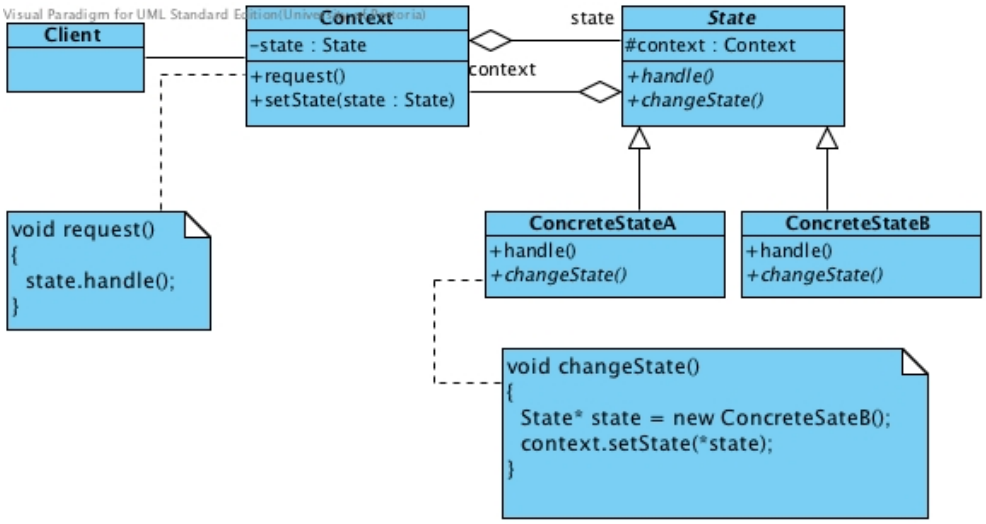

Figure 3: The state is responsible for changing state

c) **Concrete States applying variable criteria**
When the concrete states are responsible for changing the state it will be done in terms of an implementation as shown in Figure 3. This requires adding an interface to the Context that lets State objects set the Context's current state explicitly. In this case the coupling is higher since the State has to be aware of the Context. This can be implemented either by maintaining a reference to its context as shown in this figure, or by passing a pointer to the context as a parameter to the state (as we have done in the example in Section 9.5).

Note that in the last two of the above methods the concrete states are responsible for indicating the new state the context should assume when changing state. They adhere to the Protected Variations Principle while the first method does not. Any changes in the code related to changes in existing states as well as the addition or removal of states when applying the last two methods will not require any changes to the code in the Context class.

9.4.3.2 Sharing State Objects

State objects can be shared. If State objects have no instance variables – that is, the state they represent is encoded entirely in their type – then contexts can share a State object. When states are shared in this way, they are essentially flyweights with non-intrinsic state, only behaviour.

9.4.3.3 Creating and destroying State objects.

A common implementation trade-off worth considering is whether (1) to create State objects only when they are needed and destroy them thereafter versus (2) creating them

ahead of time and never destroying them. The first choice is preferable when the states that will be entered aren't known at run-time, and contexts change state infrequently. This approach avoids creating objects that won't be used, which is important if the State objects store a lot of information. The second approach is better when state changes occur rapidly, in which case you want to avoid destroying states, because they may be needed again shortly. Instantiation costs are paid once up-front, and there are no destruction costs at all. This approach might be inconvenient, though, because the Context must keep references to all states that might be entered.

9.4.4 Related Patterns

Strategy

The Strategy and State patterns have the same structure and both apply the PV Principle to achieve their goals. However, they differ in intent. The Strategy pattern is about having different implementations that accomplishes the same result, so that one implementation can replace the other as the Strategy requires while the State pattern is about doing different things based on the state, while relieving the caller from the burden to accommodate every possible state.

Singleton or Prototype

When implementing the state pattern, the programmer has to decide on how the state objects will be created. Often the application of the Prototype pattern will be ideal. State objects are also often Singletons.

Flyweight

State objects can be shared by applying Flyweight.

9.5 Example

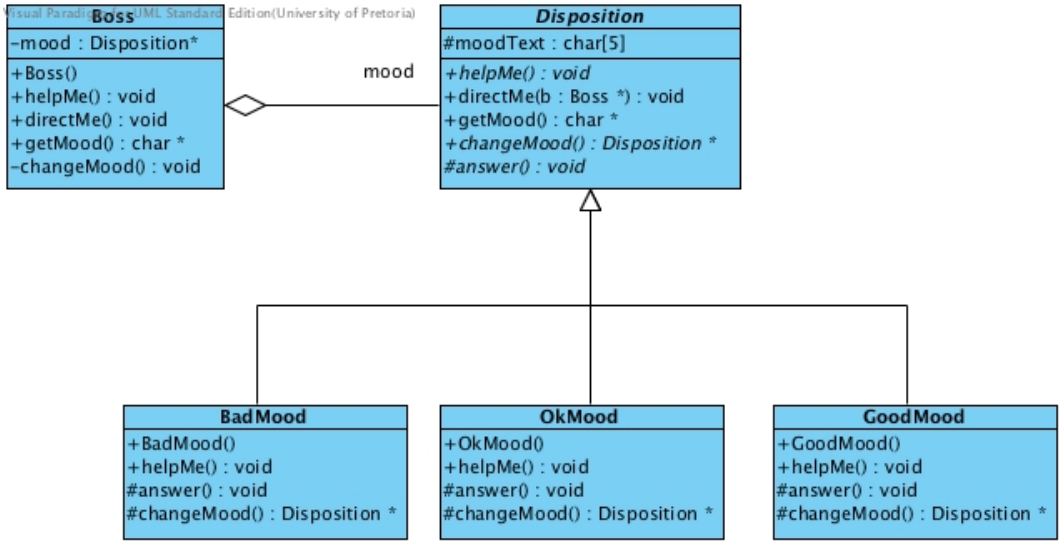

Figure 4: Class Diagram of a moody Boss simulation

Figure 4 is a class diagram of an application that implements the State design pattern. It is a nonsense program that implements three different states a Boss object can assume. The different behaviours of the Boss object is simulated in terms of different messages describing actions in the different states. These `cout` statements are mere placeholders where code that implements different behaviours can be inserted.

Participant	**Entity in application**
Context	Boss
State	Disposition
Concrete States	BadMood, OKMood and GoodMood
request()	helpMe() and directMe()
handle()	helpMe() and answer()

Context

- The `Boss` class act as the context.
- The implementation of both the `helpMe()` method and the `directMe()` method are redirections to the methods with the same names in the `Disposition` interface. The handle to these methods are provided by the `mood` instance variable of `Boss`.
- the `getMood()` method is provided to be able to display the current mood in order to verify the program state during execution.
- The `changeMood()` method is provided to enable a Boss object to change its own disposition. It is defined privately to prevent other objects to be able to change the Boss's state.

State

- `Disposition` is an abstract class that implements an interface containing the methods that implement variations in behaviour that is state dependent.
- `helpMe()` is a pure virtual method. When the `helpMe()` method in the `Boss` class is executed, execution is redirected to the method with the same name in the appropriate concrete state.
- `directMe()` is a template method. When the `directMe()` method in the `Boss` class is executed, execution is redirected to this method. In turn it calls the `answer()` method in the appropriate concrete state and also executes the `changeMood()` method of the appropriate concrete state and use the value returned by this method to manipulate the `mood` variable of the `Boss` class. This is an illustration of how the state is maintained when the concrete states are responsible for changing the state.

Concrete States

- The classes `BadMood, OKMood` and `GoodMood` act as concrete states.
- Each class provides its own implementation of the state dependent actions as defined by the virtual methods that are defined in the `Disposition` interface.
- To be able to distinguish between the execution of these methods, they display different messages.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[2] Vince Huston. Design patterns. `http://www.cs.huji.ac.il/labs/parallel/Docs/C++/DesignPatterns/`, n.d. [Online: Accessed 29 June 2011].

[3] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice-Hall, New York, 3^{rd} edition, 2004.