Introduction to Design Patterns

Prabhjeet Rekhi

Prekhi@answerthink.com

TABLE OF CONTENTS

1 Prerequisites

In order to have a good understanding of this paper, it is expected that the reader is familiar with Unified Modeling Language. Since these design patterns apply to the design phase of software life cycle, an understanding of the same along with object oriented programming will make it easier to understand the specific design pattern examples. A general understanding of software development life cycle will also be useful.

2 What is Design? (Design Phase)

Design:

Function**:** *noun*

Date**:** 1588

1: preliminary sketches or outlines showing the main features of something to be executed.

2: the arrangement of elements or details in a product or work of art

To create a piece of software so that it meets the required specifications and objectives, and performs efficiently. A particular method of implementing the requirements.

3 What is a Pattern?

Pattern:

Function**:** *noun*

Etymology**:** Middle English patron, from Middle French, from Medieval Latin patronus

Date**:** 14th century

1: a form or model proposed for imitation

2: something designed or used as a model for making things

A repeating phenomena with no or minimum changes. A particular method of implementing a given requirement such that it is repeated for a similar requirement in a different context. A pattern

is basically a three-part rule; it expresses a relation between a certain context, a problem and its solution. A common definition of a pattern is - solution to a problem in a context.

4 What are Design Patterns?

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

Architects and developers observe that while solving various types of business problems, they often come across similar issues that can be solved by using same design methods. Having done it in the past makes it easier for them to solve it the second, third time. Implementing the similar business problems using similar design in different context is to use a design pattern.

Identification of design patterns and documenting them takes experience of having solved similar problems. It helps new programmers in developing their skills faster. Further it helps re-usability of design which makes a system easy to maintain.

5 Why are they needed? (Benefits)

Design patterns are a very good way to document and communicate the design used to solve similar problems. Since there is a diverse set of applications that are developed, knowledge of design patterns can be a good point of reference when developers from different backgrounds come together.

Availability of documentation of patterns if used extensively can greatly reduce the design time. This can give a good time advantage in the design phase, especially for big projects. Since, by definition, a pattern becomes a pattern only if it is well repeated it can help reduce the disagreements that usually occur during design of any project, hence increased productivity.

6 How are they used?

Usage of design patterns is language specific. A pattern documented in UML can be used in any language as long as the language supports the basic features of an object-oriented language.

As an example,

Singleton is a design pattern that is used whenever only one instance of a particular class is needed. This is used to solve the problems where there is a need to maintain a piece of information that applies to the application as a whole and not to any particular instance of the application. For example if a very low-update configuration is needed by an application, one instance of it can be loaded and be used every time the configuration is needed.

7 Templates

Patterns are usually documented in a standard way so that a reader can easily understand the details. The information included when documenting a pattern is:

- Pattern Name
- A description of the problem with a concrete example and its context. It also describes a solution to the concrete example and when to use the pattern.
- Considerations that lead to the general solution.
- General solution. It describes the elements of design, their relationships, responsibilities and collaborations.
- Consequences, good and bad of using the given solution. It describes the results and trade-off of applying the pattern.
- Related patterns.

This paper will use the following headings to describe each pattern.

a. Pattern Name: Contains the name of the pattern being described.

b. Synopsis: A short description of the solution this pattern is trying to provide.

c. Context: The problem that the problem address, with a concrete example

d. Forces: Considerations that lead to a general solution that this pattern is proposing to present.

e. Solution: The general solution that this pattern presents for the general problem.

f. Consequences: Any implications, good or bad, of using this pattern.

g. Implementation: Discusses any implementation related issues while using this pattern.

While providing the classification of design patterns usually followed, this paper will give examples for specific patterns. It will give a summary of other types of patterns in each classification.

8 Types of design patterns

The types of problems commonly encountered during software development are classified in categories. The patterns used in each of these categories are hence organized together for a better understanding. The design patterns described here apply to these categories.

8.1 Fundamental Design Patterns

Fundamental Patterns are the basic patterns used in designing individual classes. They are usually used in conjunction with other patterns, like structural and behavior, in order to define the structure and behavior of the classes.

Some of the different types of Fundamental Patterns are:

- Delegation: It is a way to extend and reuse the functionality of a class by writing an additional class with added functionality that uses instances of the original class to provide the original functionality.
- Interface: Keep a class that uses data and services provided by instances of other classes independent of those classes by having it access those instances through an interface.
- Immutable: It increases the robustness of objects that share references to the same object by forbidding any of the object's state information to change after the object is constructed.
- Proxy: It is a way to call the method indirectly from the class that implements it. More details on this follow.

8.1.1 Proxy

8.1.1.1 Synopsis

The proxy pattern forces method calls to an object to occur indirectly through a proxy object that acts as a surrogate for the other object, delegating method calls to that object.

8.1.1.2 Context

A proxy object is an object that receives method calls on behalf of another object. Client objects call the proxy object's methods. The proxy object's methods do not directly provide the service that its clients expect. Instead, the proxy object's methods call the methods of the object that provides the actual service.

Proxy objects generally use share a common interface or super class with the service providing object. That makes it possible for client objects to be unaware that they are calling the methods of a proxy object rather than the methods of the actual service-providing object. Transparent management of another object's services is the basic reason for using a proxy.

Depending on the behavior of a proxy object, it can be a remote proxy, access proxy, virtual proxy etc.

8.1.1.3 Forces

- It is not possible for a service-providing object to provide service at a time or place that is convenient.
- There are situations in which client does not or can not reference an object directly, but wants to still interact with the object.
- Access to a service-providing object must be controlled without adding complexity to the service-providing object or coupling the service to the access control policy.

8.1.1.4 Solution

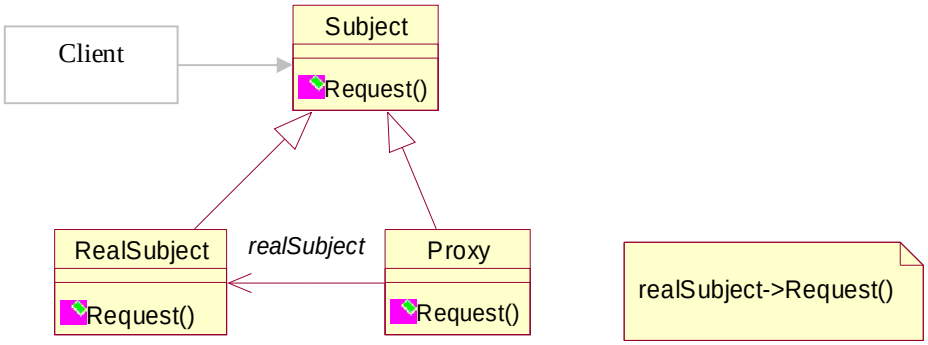
Transparent management of a service-providing object can be accomplished by forcing all access to the service-providing object through a proxy object. To achieve this, the proxy object and the service-providing object must either be instances of a common super class or implement a common interface.

The proxy pattern is not very useful unless it implements some particular service management policy.

Some of the common situations where proxy pattern is applicable are:

- Remote proxy: provides a local representative for an object in a different address space
- Virtual proxy: creates expensive objects on demand
- Protection proxy: controls access to the original or service providing object
- Cache proxy: provides temporary storage of the results of expensive target operations so that multiple clients can share the results
- Firewall proxy: protects target from bad clients
- Synchronization proxy: provides multiple accesses to a target object

8.1.1.5 Structure



8.1.1.6 Consequences

The service provided by a service-providing object is managed in a manner transparent to that object and its clients.

8.1.1.7 Implementation

Without any specific management policy, the implementation of the Proxy pattern simply involves creating a class that shares a common super class or interface with a service providing class and delegates operations to instances of the service providing class.

8.2 Creational Patterns

Creational patterns provide guidance as to how to create objects when their creation requires making decisions. These decisions will typically involve dynamically deciding which class to instantiate or which objects an object will delegate responsibility to.

The different types of creational patterns are:

- Abstract factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- Builder: Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Factory Method: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
- Prototype: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Singleton: To allow only one instance at most for a class. More details follow.

8.2.1 Singleton

8.2.1.1 Synopsis

Ensure a class only has one instance, and provide a global point of access to it. All objects that use an instance of that class use the same instance.

8.2.1.2 Context

It is important for some classes to have exactly one instance. These classes usually involve the central management of a resource. The resource may be external, as is the case with an object that manages the reuse of database connections. The resource may be internal, such as an object that keeps an error count and other statistics for a compiler.

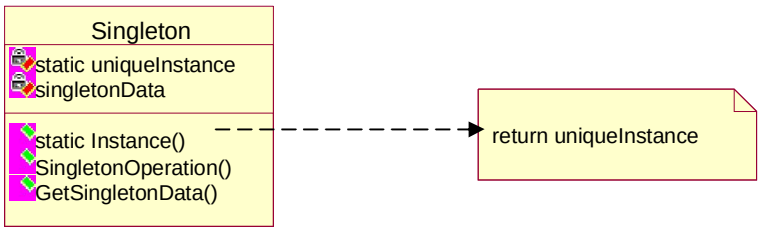
8.2.1.3 Forces

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.

8.2.1.4 Solution

A singleton class has a static variable that refers to the one instance of the class that you want to use. This instance is created when the class is loaded into memory. The class should be implemented in such a way that prevents other classes from creating any additional instances of a singleton class. To access to the instance of a singleton class, the class provides a static method.

8.2.1.5 Structure



8.2.1.6 Consequences

- Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.

- Sub-classing a singleton class is awkward and results in imperfectly encapsulated classes. To subclass a singleton class, its constructor must not be private. If the subclass needs to be a singleton also, it will be necessary to override static methods like getInstance() (that returns the static instance). Java does not allow overriding static methods.

8.2.1.7 Implementation

The class must be coded in a way that prevents other classes from directly creating instances of the class. All of class' constructors must be private. At least one private constructor must be declared, otherwise a default public constructor will be generated.

If a singleton class' object is garbage collected, the next it time it is used, a new instance of the class will be created and this will result in re-initialization. This may create inconsistency depending on the usage of the class. The other option is to always refer to the object, directly or indirectly through a live thread so that it is never garbage collected.

8.3 Structural Patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures.

A few different types of structural patterns are:

- Adapter: To map an available interface for a class to another interface clients expect. More details follow.
- Bridge: De-couple an abstraction from its implementation so that the two can vary independently.
- Composite: Compose objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.
- Decorator: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

8.3.1 Adapter

8.3.1.1 Synopsis

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. A change in the implementation at any level of the client interfaces is completely transparent to the clients. This interface is also called Wrapper.

8.3.1.2 Context

Suppose you are building an application that relies on third party libraries. You want to use these libraries for ease of use, so that you do not have to write access methods to the actual data representation. One of the objectives here should be that you want to keep your usage of the third party libraries de-coupled from your code. This gives you a flexibility of being able to change the third party libraries if you feel you have a better option, or simply because they do not perform as expected. Additionally, if a newer version of the libraries becomes available, having an adapter interface will allow you to migrate to the newer version very easily without having to change any of the client code.

8.3.1.3 Forces

- If there is a need to use an existing class, and its interface does not match the one that is needed by the client.
- If a reusable class needs to be created that cooperates with unrelated or unforeseen classes, i.e. classes that don't necessarily have compatible interfaces.

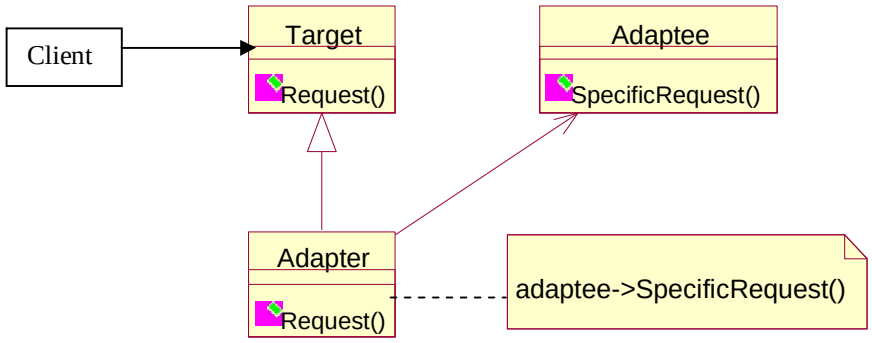
8.3.1.4 Solution

Suppose you have a client that uses a particular set of classes. You want to extend the classes that you use with the client, or replace the classes with the newer version. The classes that you use implement an interface that the client uses, but the newer version of classes do not use. You can

very easily write an adapter that will implement the interface that clients use. For the implementation of this adapter, you can use the methods available with the newer version of the classes. The roles that classes and interfaces play are:

- Client: The class that calls a method of another class through an interface.
- Target Interface: This interface declares the method that a Client class calls.
- Adapter: This class implements the Target Interface. It implements the method that the client classes by having it call a method of the Adaptee class, which does not implement the Target Interface.
- Adaptee: This class does not implement the Target Interface method but has a method that you want the Client class to call.

8.3.1.5 Structure



8.3.1.6 Consequences

- The Adapter adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work if we want to adapt a class and all its subclasses.
- Adapter can override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- The client and the adaptee classes remain independent of each other.

8.3.1.7 Implementation

The implementation of the adapter class is straightforward. An issue that needs top be considered when implementing the pattern is how the adapter objects know what instances of the adaptee class to call. There are two approaches to it:

- Pass a reference to the adaptee object as a parameter to the adapter object's constructor or one of its methods. This allows the adapter object to be used with any instance or possibly multiple instances of the adaptee class.
- Make the adapter class an inner class of the adaptee class. This simplifies the association between the adapter object and the adaptee object by making it automatic. It also makes the association flexible.

8.4 Behavioral Patterns

Behavioral patterns describe patterns of communication between objects. These patterns characterize complex control flow that's difficult to track at run-time. They shift the focus away from flow control to concentrate just on the way objects are interconnected.

Some of the behavioral patterns are:

- Iterator: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Observer: Also called Publish-Subscribe pattern. More details to follow.
- State: Allow an object to alter its behavior when it's internal state changes. The object will appear to change its class.
- Strategy: To have a set of algorithm implementations to chosen from at run-time. More details to follow.
- Visitor: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

8.4.1 Observer

8.4.1.1 Synopsis

Allows objects to dynamically register dependencies between objects, so that an object will notify those objects that are dependent on it when its state changes. Usually works with one-to-many dependency between objects. It is also called Publish-Subscribe pattern.

8.4.1.2 Context

Suppose you are building a client-server application for product sales. Since it is heavily dependent on the latest specials available, while a representative is on call with the customer, they should be able to offer the specials to the customer as soon as they are available. This type of model requires that as soon as the data changes, or promotions are added, it should be notified to the client that uses the data to be presented to the customer. This will enable the client to automatically update with the promotions, and hence the customer can get the benefits. Since there could be multiple clients accessing the same product at a time, all of them will need to be notified as soon as an update takes place.

8.4.1.3 Forces

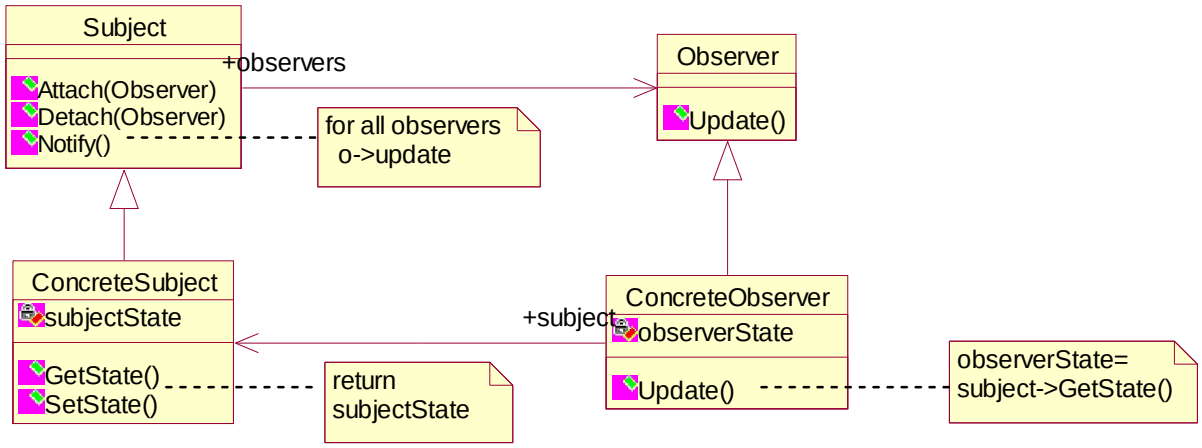
- We are implementing two independent classes. An instance of one will need to be able to notify other objects when its state changes. An instance of the other will need to be notified when an object it has a dependency on changes state. However, the two classes are not intended to work with each other and should not have direct knowledge of each other. These two objects should not be tightly coupled.
- There is a one-to-many relationship that may require an object to notify multiple objects that are dependent on it when it changes its state.

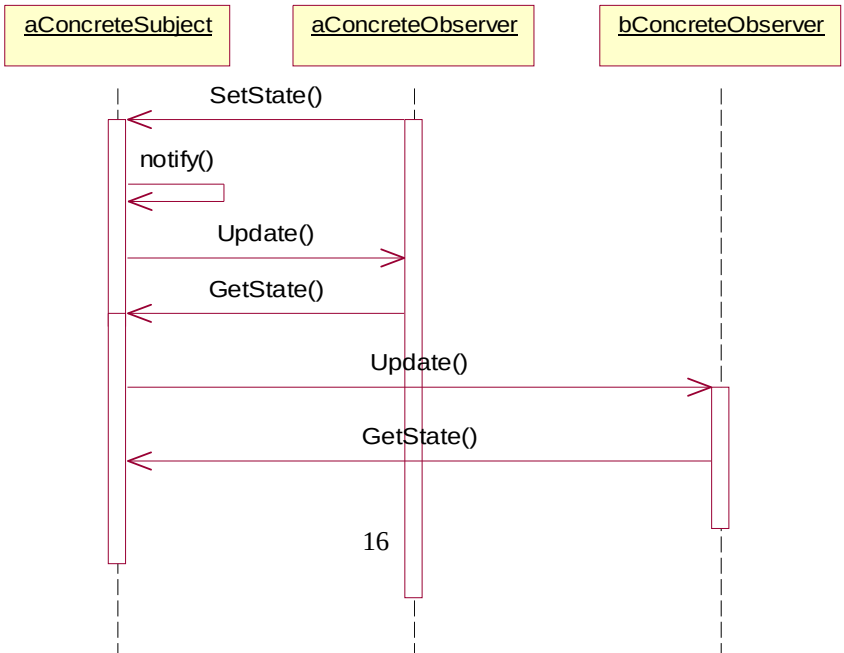
8.4.1.4 Solution

Here are the descriptions that classes and interfaces play in the Observer Pattern:

- Subject: It knows its observers. Any number of observer objects may observe a subject. It provides an interface for attaching and detaching Observer objects.
- Observer: Defines an updating interface for objects that should be notified of changes in a subject.
- Concrete Subject: Stores state of interest to Concrete Observer objects. Sends a notification to its observers when its state changes.
- Concrete Observer: Maintains a reference to a Concrete Subject object. Stores state that should stay consistent with the subject's state. Implements the Observer updating interface to keep its state consistent with the subject's state.

8.4.1.5 Structure





8.4.1.6 Consequences

- Delivering notifications can take a long time if an object has a large number of objects to deliver notifications to. This can happen because one object has many observers directly registered to receive its notifications. It can also happen because an object has many indirect observers because its notifications are cascaded by other objects.
- A more serious problem happens if there are cyclic dependencies. Objects call each other's notify methods until the stack fills up.
- The notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it. The subject doesn't care how many observers exist; it is only responsible to notify the observers. This gives a freedom to add and remove observers at any time. It is up to the observer to handle or ignore a notification.

8.4.1.7 Implementation

- The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject. Such storage may be too expensive when there are many subjects and few observers. One solution is to maintain an associative lookup for subject-to-observer mapping. Here, a subject with no observers does not incur storage overhead, but there is an increase in cost of accessing the observers.
- If an observer needs to be dependent on more than one subject, it may be needed to tell the observer which subject has changed. This can be achieved by passing the subject itself as a parameter in the Update operation.
- Deleting a subject should not produce dangling references in its observers. One way is to make the subject notify its observers as it is deleted so that they can reset their reference to it.

- The two models of notifications are Push and Pull model. In the Push model, the subject sends all the updated information as part of the notification itself, and the observer can use it to update it self. In the Pull model, the subject only sends minimum notification that it has changes, and the observer then asks for details of changes explicitly. In the push model, the subject has some knowledge of its observers, whereas in pull model, this knowledge is minimum. The push model may make observers less reusable as it assumes certain things about observers that may not be true for all observers. The pull model may be inefficient since the observers have to get the changes from the subject, with little help from subject.

8.4.2 Strategy

8.4.2.1 Synopsis

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. It is also known as Policy.

8.4.2.2 Context

Suppose you are service provider that evaluates credits for client credit card companies. Each client may have a different business rule to evaluate the credit that will result in an approval or rejections for a credit card application. Since you want to support multiple clients, different policies will need to be implemented for credit evaluation. Depending on the client that is submitting the request, the appropriate evaluation policy class will be called, and hence the appropriate algorithm will be used.

8.4.2.3 Forces

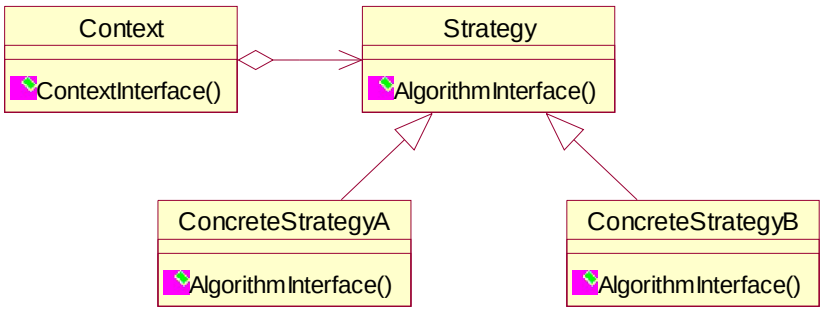
- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- The clients for these classes do not need to know anything about the implementations of their behavior. A common interface for all behaviors will make it completely transparent to the clients.

8.4.2.4 Solution

Here are the descriptions that classes and interfaces play in the Strategy Pattern:

- Client: The client class delegates an operation to an abstract class or interface. It does that without actually knowing the actual class of the object it delegates the operation to or how that class implements the operation.
- Abstract Strategy: This class provides a common way to access the operation encapsulated by its subclasses. An interface can also be used instead.
- Concrete Strategy: Classes in this role implement alternative implementations of the operation that the Client class delegates.
- Context: It is configured with concrete strategy object. It maintains a reference to the strategy object. It may define an interface that lets strategy access its data.

8.4.2.5 Structure



8.4.2.6 Consequences

The Strategy pattern allows the behavior of client objects to be dynamically determined on a per-object basis.

The Strategy pattern simplifies client objects by relieving them of any responsibility for selecting behavior or implementing alternate behaviors. It simplifies the code for client objects by eliminating if and switch statements.

8.4.2.7 Implementation

The strategy and context interfaces must give Concrete Strategy efficient access to any data it needs from a context, and vice versa. One approach is to have Context pass data in parameters to Strategy operation. This keeps Strategy and Context de-coupled. Another technique has a context pass itself as an argument, and the strategy requests data from the context explicitly. Alternatively, the strategy can store a reference to its context, eliminating the need to pass anything at all.

9 Bibliography

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Mass: Addison-Wesley, 1995.

Mark Grand. Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML. John Wiley & Sons, Inc, 1998