Department of Computer Science
COS121 Lecture Notes: L29

Bridge Design Pattern

20 and 21 October 2014

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Contents

L29.1 Introduction

Separation of concerns is a general problem-solving idiom that enables us to break the complexity of a problem into loosely-coupled, easier to solve, subproblems [3]. When applied to software design it results in a design where the classes are cohesive and loosely-coupled.

The Bridge pattern is a good example of the application of the principle of separation of concerns in this context. It is motivated by the the desire to separate the interface, and subsequently the implementation of the interface, from an abstraction.

The design resulting from the application of the Bridge design pattern is two orthogonal class hierarchies that can vary independently.

L29.2 Bridge Design Pattern

L29.2.1 Identification

Name	**Classification**	**Strategy**
Bridge	Structural	Delegation
Intent		
Decouple an abstraction from its implementation so that the two can vary independently ([2]:151)		

L29.2.2 Structure

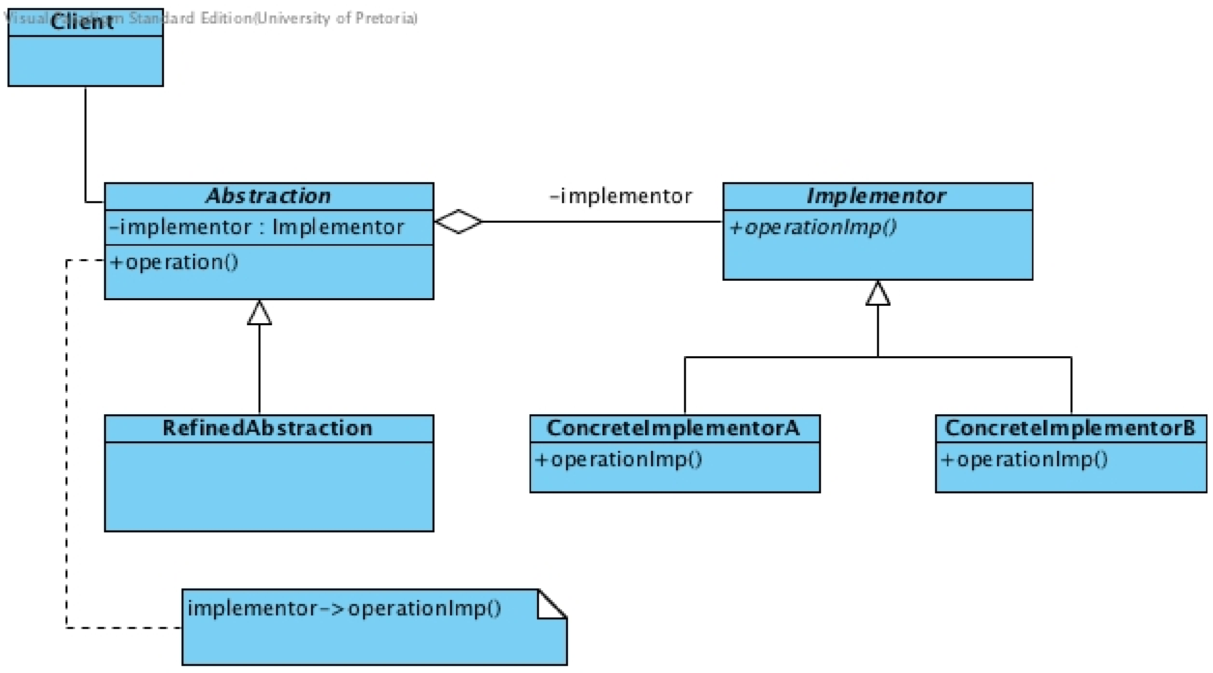

Figure 1: The structure of the Bridge Design Pattern

L29.2.3 Participants

Abstraction

- defines the abstraction's interface.
- maintains a reference to an object of type Implementor.

Refined Abstraction

- Extends the interface defined by Abstraction.

Implementor

- defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.

Concrete Implementor

- implements the Implementor interface and defines its concrete implementation.

L29.2.4 Problem

A system has a proliferation of classes resulting from a coupled interface and numerous implementations [4].

L29.3 Bridge pattern explained

L29.3.1 Motivation

Suppose you have a ThreadScheduler abstraction that starts off with implementations for Unix and for Microsoft Windows. For each type of scheduler (for example a time sliced scheduler, a preemptive scheduler, etc) you need to create two subclasses, one for each platform of implementation. In fact any subclass of the ThreadScheduler abstraction needs this further dual implementation. This problem is magnified if you want to make the ThreadScheduler abstraction work on additional platforms, such as the Macintosh or a JavaVM. For each existing subclass of ThreadScheduler, an implementation must be created and added to the inheritance hierarchy. This example from [4] illustrates the lack of flexibility of this simple inheritance structure shown in Figure 2. The Bridge pattern offers a solution to avoid this kind of proliferation of classes.

With the Bridge pattern, the ThreadScheduler abstraction becomes easier to implement. As shown in Figure 3 we have a ThreadScheduler hierarchy of all the abstract scheduler types on the abstraction side. The implementations of these thread schedulers are provided by a separate hierarchy of implementations. The link is provided by a reference to an implementation object, maintained in the ThreadScheduler hierarchy. Thus, any

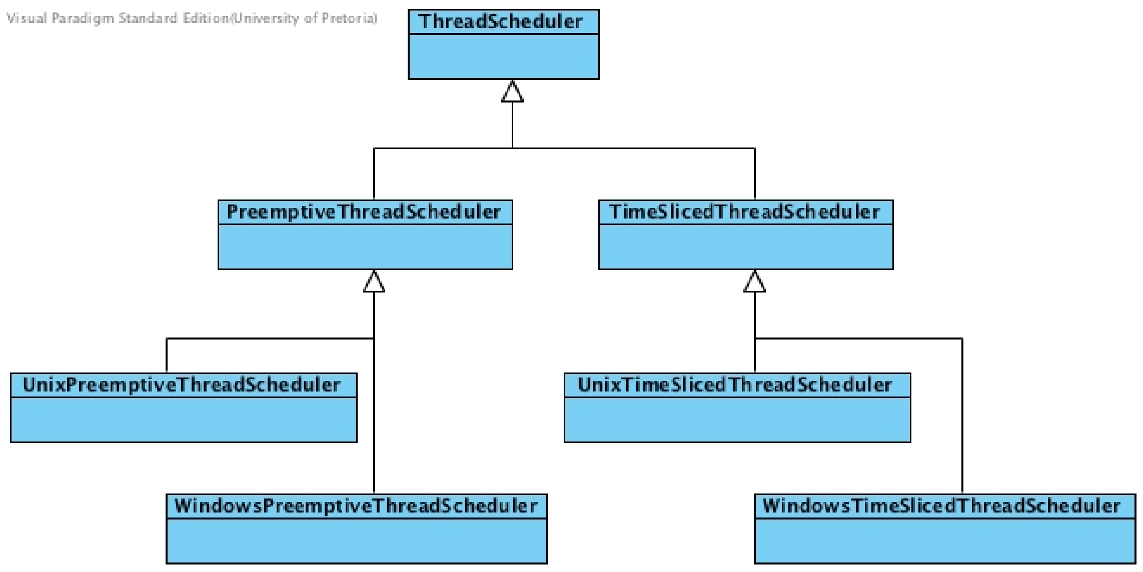
Visual Paradigm Standard Edition(University of Pretoria)
ThreadScheduler
PreemptiveThreadScheduler
TimeSlicedThreadScheduler
UnixPreemptiveThreadScheduler
UnixTimeSlicedThreadScheduler
WindowsPreemptiveThreadScheduler
WindowsTimeSlicedThreadScheduler

Figure 2: Proliferation of classes to accommodate different schedulers for different platforms

implementation, or platform specific methods are delegated to the implementation object, which is an instance of the specific implementation currently in use. This link and separation is the essence of the Bridge pattern.

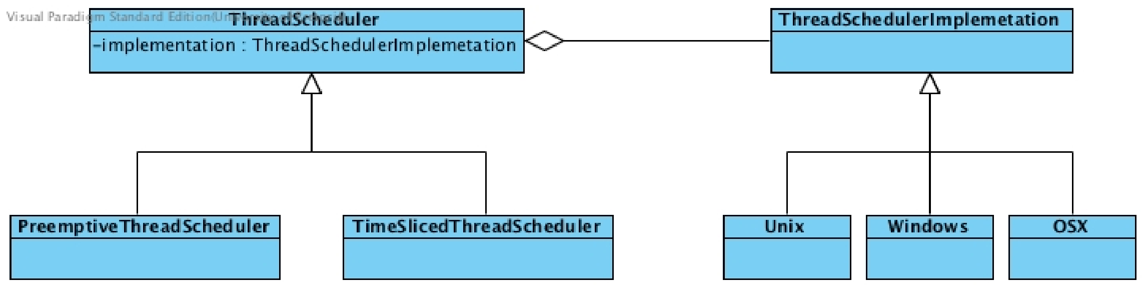
Visual Paradigm Standard Edition(University of Pretoria)
ThreadScheduler
-implementation : ThreadSchedulerImplemetation
ThreadSchedulerImplemetation
PreemptiveThreadScheduler
TimeSlicedThreadScheduler
Unix
Windows
OSX

Figure 3: A bridge to accommodate different schedulers for different platforms

L29.3.2 Non-software example

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches [1]

L29.3.3 Improvements achieved

Greater flexibility

The coupling between an interface and its implementation is no longer fixed in terms

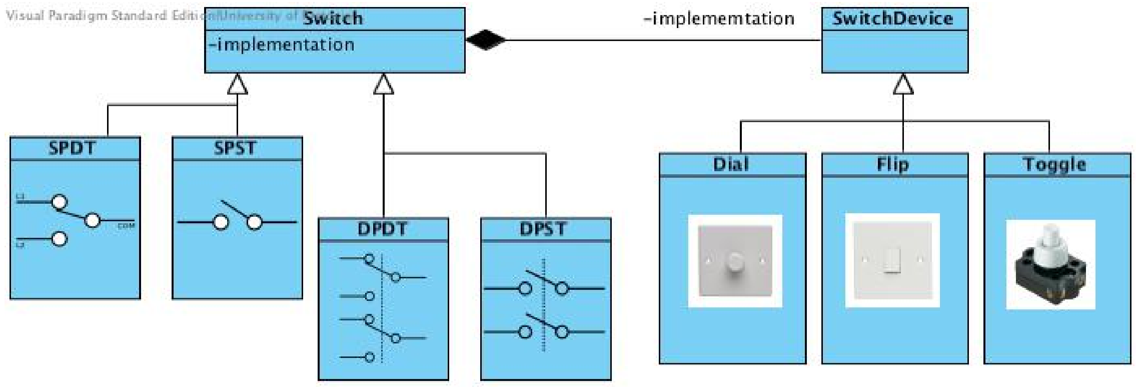

Figure 4: Household switch abstraction and implementation hierarchies

of an inheritance relation. Instead the relation is changed to delegation allowing the binding between the interface and its implementation to change at run-time. You can also extend the Abstraction and Implementor hierarchies independently.

Saving on compile-time
Decoupling Abstraction and Implementor eliminates compile-time dependencies on the implementation. Changing an implementation class doesn't require recompiling the Abstraction class and its clients. This property is essential when you must ensure binary compatibility between different versions of a class library.

Improved Structure
The decoupling between interface and implementation encourages layering that can lead to a better-structured system. The high-level part of a system only has to know about Abstraction and Implementor.

Hiding implementation details from clients
You can shield clients from implementation details. The definition of the Implementor class need only be visible to the Abstraction. It can be specified as private to the Abstraction class, hiding it completely from the clients using the Abstraction.

L29.3.4 Implementation issues

When implementing the Bridge pattern one has to decide how the concrete implementor objects will be instantiated.

If Abstraction knows about all ConcreteImplementor classes, then it can instantiate one of them in its constructor; it can decide between them based on parameters passed to its constructor. If, for example, a collection class supports multiple implementations, the decision can be based on the size of the collection. A linked list implementation can be used for small collections and a hash table for larger ones.

Another approach is to choose a default implementation initially and change it later according to usage. For example, if the collection grows bigger than a certain threshold, then it switches its implementation to one that's more appropriate for a large number of items.

It's also possible to delegate the decision to another object altogether. This can typically be achieved by implementing an abstract factory whose duty is to encapsulate detail related to the characteristics of the concrete implementors. The factory knows what kind of Concrete Implementor to create for each situation. An Abstraction simply asks the abstract factory for an Implementor, and it returns the right kind. A benefit of this approach is that Abstraction is not coupled directly to any of the Implementor classes.

L29.3.5 Related patterns

Adapter
Both Adapter and Bridge use delegation to implement cooperation between classes. However, the Adapter is more often implemented

Strategy
Both Strategy and Bridge use delegation through an abstract interface to concrete implementations performing operations. However, the operations performed by the strategy pattern are interchangeable algorithms while the operations performed by the bridge pattern are common operations acting on interchangeable implementations such as different data structures or different operating systems.

L29.4 Example

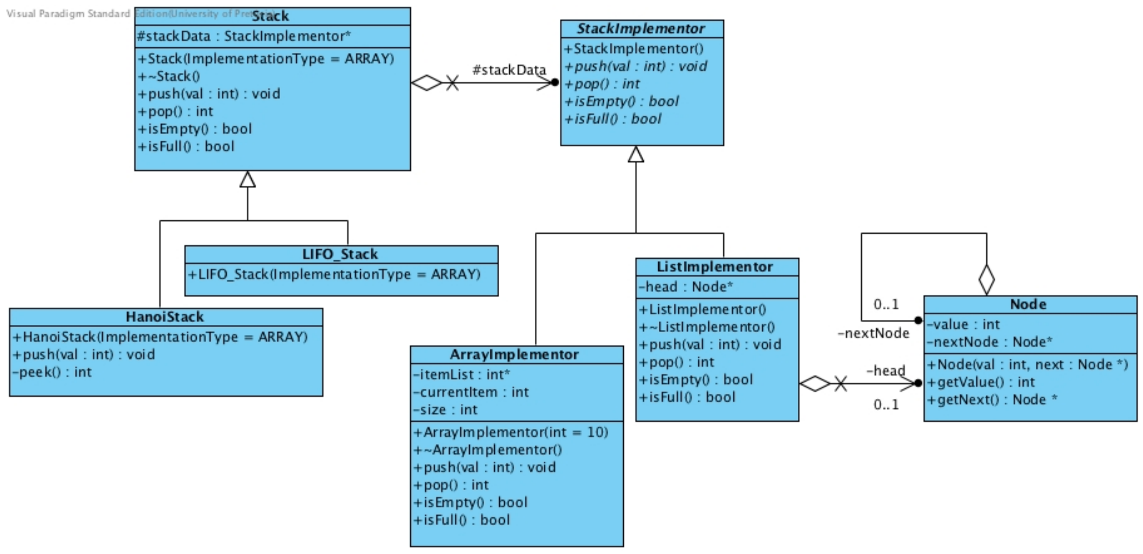

Figure 5: Different kinds of stacks implemented with different data structures

Figure 5 is a class diagram of our example implementation. It is a nonsense program that implements the bridge structure to illustrate how two orthogonal hierarchies can easily be used by a client to instantiate any type of refined abstraction implemented in terms of any concrete implementation. Given this structure it is also easy to add more types of stacks and/or more implementations.

Participant	Entity in application
Abstraction	Stack
Refined Abstraction	LIFO_Stack, HanoiStack
Implementation	StackImplementor
Concrete Implementation	ArrayImplementor, ListImplementor
operation()	push(), pop(), isEmpty(), isFull()
implementation()	push(), pop(), isEmpty(), isFull()

Abstraction

- The `Stack` class defines the abstraction's interface. It provides the definition of methods needed to implement any kind of stack. Although the pattern allows these methods to be concrete, this implementation defines these methods as virtual to allow the derived classes to override these methods if needed.
- `Stack` maintains a reference to an object of type Implementor. In this implementation the reference is established during construction and is not changed at runtime. To be able to change it at runtime this interface has to define a setter for `stackData`.

Refined Abstraction

- `HanoiStack` and `LIFO_Stack` are refined abstractions. They extend the interface defined by `Stack`. Since `Stack` implements the default actions on each of the operations, which is a simple redirection to the methods in the implementation with the same name, these derived classes only have to implement methods that deviate from the default for the specific kind of stack. In this case `LIFO_Stack` accepts all the default implementations

Implementor

- `StackImplementor` is the Implementor participant. It defines the interface for implementation classes. In this case this interface corresponds exactly to Abstraction's interface. This is not required for the pattern. In fact the two interfaces can be quite different. The `Stack` interface could for example have added the `peek()` operation that is defined in `HanoiStack` without having to change the `Implementor` or any of its derivatives.

Concrete Implementor

- `ListImplementor` and `ArrayImplementors` are concrete implementations of the `StackImplementor` interface and defines its concrete implementation.

L29.5 Exercise

1. Change the main program of the given system to create both implementations of both kinds of stacks.
2. Add another implementation to the given code that uses the `<stack>` from the C++ STL as an implementation to the given example system.

References

[1] Michael Duell. Non-Software Examples of Software Design Patterns. *Object Magazine*, 7(5):52 – 57, 1997.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1994.

[3] Hafedh Mili, Amel Elkharraz, and Hamid Mcheick. Understanding separation of concerns. In *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2004.

[4] Alexander Shvets. Design patterns simply. `http://sourcemaking.com/design\_patterns/`, n.d. [Online; Accessed 29-June-2011].