Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 25: Façade Design Pattern

Contents

25.1 Introduction

It is generally a good idea to build systems that are generic and reusable. The result of such practice is the development of systems that include a wide variety of versatile functions. Unfortunately, while improving the reusability of code in this manner, the complexity of the code increases and so does the ease of use. The Façade pattern can be applied to hide some of the complexity of such systems and to simplify the use of the system for commonly needed functions without compromising the usability of functions provided by subsystems that are not necessary commonly used.

The application of most design patterns result in more and smaller classes which are aimed at making life easy for people who need to change the code. A negative side effect of this practice is that the code becomes harder to use for clients that need not change the code. The Façade pattern provides an interface for clients who only need to use the code, making life easier for them by providing a simplified interface through which they can communicate with the subsystems in a more predictable manner.

25.2 Façade Design Pattern

25.2.1 Identification

Name	**Classification**	**Strategy**
Façade	Structural	Delegation
Intent		
Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. ([2]:185)		

25.2.2 Problem

A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem [3].

25.2.3 Structure

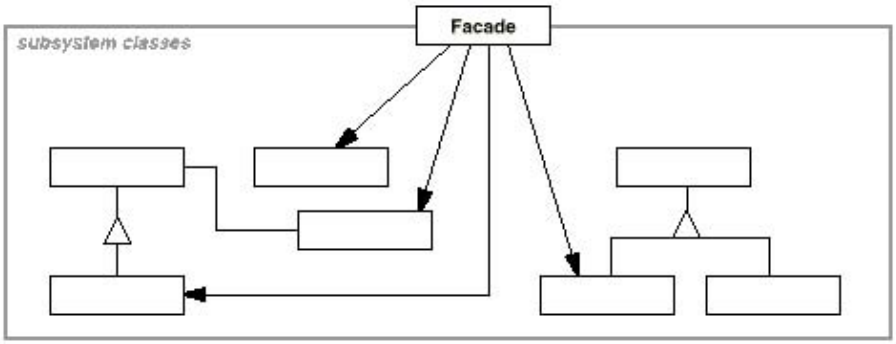


Figure 1: The structure of the Façade Design Pattern

25.2.4 Participants

Façade

- Knows which subsystem classes are responsible for a request.
- Delegates client requests to appropriate subsystem objects.

Subsystem classes

- Implements subsystem functionality
- Handle work assigned by the Façade object.
- Have no knowledge of the façade and perform operations independent of the façade.

25.3 Façade Pattern Explained

25.3.1 Improvements achieved

- **Reduce coupling between clients and the system**
  It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use
- **Promotes weak coupling between subsystems**
  Weak coupling lets you vary the components of the subsystem without affecting its clients. Facades help layering a system and the dependencies between objects thereby reducing compilation dependencies and promoting portability of the code.

25.3.2 Practical examples

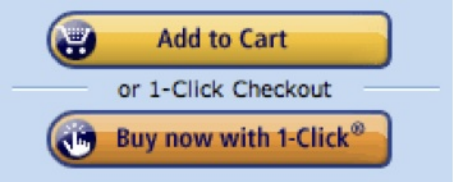
The Façade is applied to automate a process that consist for a sequence of steps where the different steps in the process may be executed by different subsystems in different classes.

If a variable process is executed in the same way every time in **certain identifiable situations** it justifies a Façade.

There are many situation where the Façade design pattern can be applied. The following are a few practical examples:

- Automate the compilation of programs to perform the steps of scanning, parsing, compiling and linking with a single instruction.
- Provide a vacation planner interface that links with subsystems for example accommodation planning, travel planning, site seeing planning, entertainment planning, etc.
- Provide an installation wizard to install a large system.
- Provide an automated procedure to make a backup of the data in a system.

- Provide a wizard in an application program such as a word-processor to semi-automatically perform a complicated procedure such as creating a table.
- Provide an automated online order procedure for identified customers [1].



- Provide an automated procedure to prepare the roll-over of a system at the end of a logical cycle. For example a financial system at the end of a financial year or a student registration system at the end of an academic year.

25.3.3 Common Misconceptions

- The Façade is not any automated process. To be an implementation of the Façade, the steps in the automated process should still be available as individual functions that can be performed without the aid of the Façade. It is important to notice that the façade should be implemented in such a way that It doesn't prevent applications from using subsystem classes if they need to. Thus the clients must have the freedom to choose between ease of use (using the façade) and generality (bypassing the façade).

25.3.4 Related Patterns

Adapter
: Façade is in a sense a huge object adapter that simultaneously adapts a number of classes with the intent to simplify communication with those classes. While both the façade and the adapter my wrap any number of classes, their intent is different. The adapter wraps to provide the *expected* interface, while the façade wraps to provide a *simplified* interface.

Template Method
: Façade is in a sense a huge template method that defines the skeleton of an algorithm for a process that is automated. However, instead of deferring some steps to subclasses, it delegates steps to the different classes comprising a system.

Mediator
: Mediator is similar to Façade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them. A mediator's colleagues are aware of and communicate with the mediator instead of communicating with each other directly. In contrast, a facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn't define new functionality, and subsystem classes don't know about it.

25.4 Implementation Issues

There are various enhancements that can be considered when implementing the façade design pattern. We mention two that were suggested by [2]:

25.4.1 An abstract façade

The coupling between clients and the subsystem can be reduced even further by making Façade an abstract class with concrete subclasses for different implementations of a subsystem. Then clients can communicate with the subsystem through the interface of the abstract Facade class. This abstract coupling keeps clients from knowing which implementation of a subsystem is used.

25.4.2 Configurable façade

To enhance the usability of a façade, it can be implemented in a way that allows the client to configure a Facade object with a selection of different subsystem objects. Then the façade, can be cutomised by replacing one or more of its subsystem objects.

25.5 Example

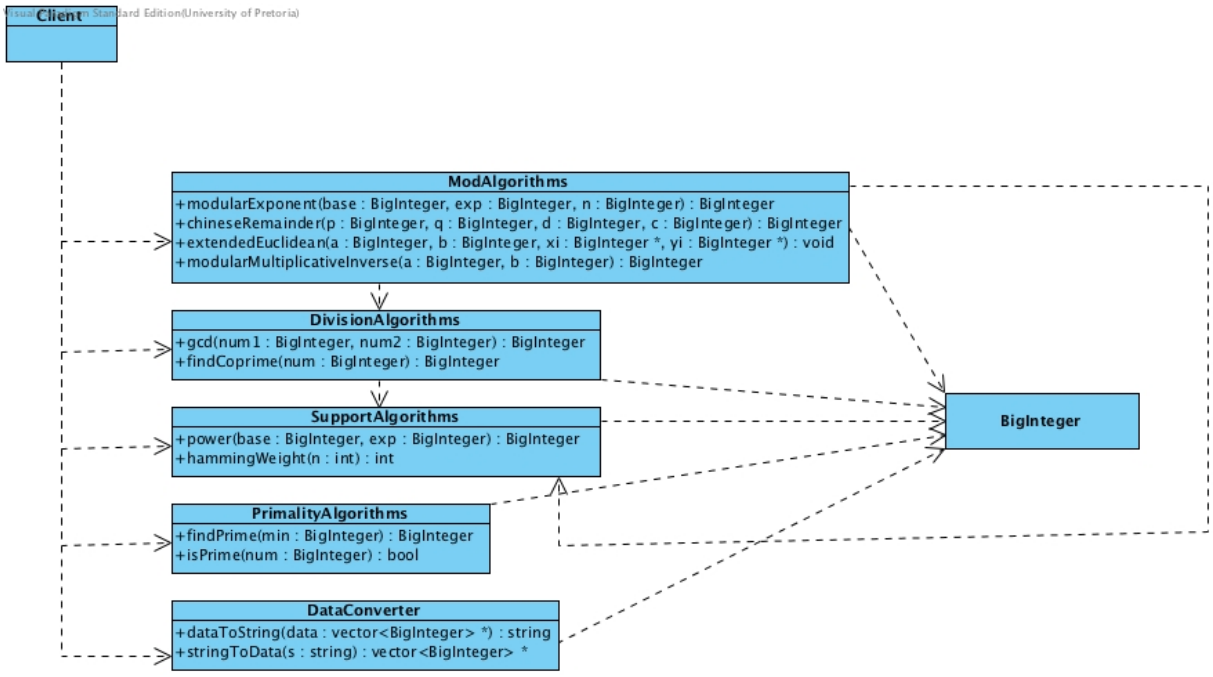


Figure 2: Class Diagram a system before implementing a façade

Figure 2 is a class diagram of a system implementing the RSA Encryption Algorithm. The client prompts the user for an input string. This string is encrypted and decrypted

using a number of methods provided in a number of inter-related classes. The following is the code of the client[1]:

```
#include <cstdlib>
#include <vector>
#include <iostream>
#include <string>

#include "BigInteger.h"
#include "BigIntegerUtils.h"
#include "DataConverter.h"
#include "PrimalityAlgorithms.h"
#include "ModAlgorithms.h"
#include "DivisionAlgorithms.h"

using namespace std;

void initializeKeys
    (BigInteger &p,  BigInteger &q, BigInteger &n,
     BigInteger &d, BigInteger &e);

vector<BigInteger>* encrypt
    (string m, BigInteger n, BigInteger d);

string decrypt
    (vector<BigInteger>* c, BigInteger p,
      BigInteger q, BigInteger d);

string getInput();

int main()
{
        srand ( (unsigned)time(NULL) );
        string input = getInput();

        BigInteger p, q, n, d, e;
        initializeKeys(p, q, n, d, e);

        vector<BigInteger>* cypherText = encrypt(input, n, e);
        cout << "\nEncrypted data: ";
        for( unsigned i = 0; i < cypherText->size(); ++i)
                cout << cypherText->at(i) << " ";
        cout << endl;

        string output = decrypt(cypherText, p, q, d);
        cout << "\nDecrypted string: " << output << endl;
}
```

[1]The detailed implementation of the methods declared in this client is omitted here

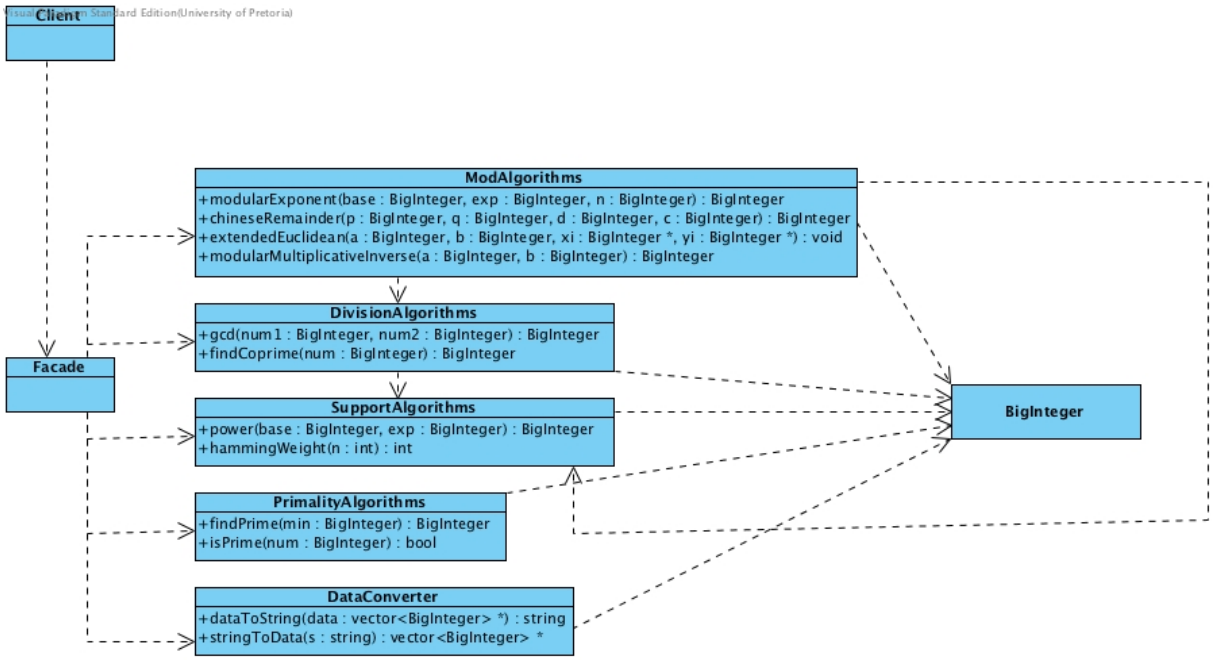


Figure 3: Class Diagram a system after implementing a façade

We will now illustrate the creation of a façade to handle encryption and decryption, and how it can be used by this client. Figure 3 is a class diagram of this system when using this façade.

The façade we will create here has the purpose of hiding the detail needed by the encrypt and decrypt procedures from the client. The details like the generation of the keys used in the encryption are irrelevant to the user of the encryption functions. Therefore it would be a good idea to include the keys used by these methods the façade class and to provide an interface with encrypt and decrypt methods requiring only the strings to be passed as parameters. The following is the definition (h-file) of a proposed façade class:

```
#ifndef FACADE
#define FACADE

#include <cstdlib>

#include "BigInteger.h"
#include "BigIntegerUtils.h"
#include "DataConverter.h"
#include "PrimalityAlgorithms.h"
#include "ModAlgorithms.h"
#include "DivisionAlgorithms.h"

class Facade
{
        public:
                Facade();
                vector<BigInteger>* encrypt(string m);
```

```
                string decrypt(vector<BigInteger>* c);
        private:
                BigInteger p, q, n, d, e;
};
#endif
```

Note that this file has `#include` statements for all classes. It also defines instance variables `BigInteger p, q, n, d` and `e`. The `encrypt` and `decrypt` methods are declared with less parameters and the `initializeKeys()` method is removed. The implementation of this class should now contain the detailed implementation of the mentioned methods.

The implementation of the `encrypt` and `decrypt` methods should be the same as it was in the original client. Except that the values of `p, q, n, d` and `e` need not be passed as parameters. They are now directly available as instance variables.

The constructor has to initialize the keys. Therefore, the body of the `initializeKeys()` method should become the body of the constructor. The statement to initialise the random number generator; `srand ((unsigned)time(NULL));` can also be executed here.

The client code can now be simplified by applying this this façade. The following is a listing of the simplified main.C

```
#include "Facade.h"

string getInput();

int main()
{
        Facade facade;
        string input = getInput();

        vector<BigInteger>* cypherText = facade.encrypt(input);
        cout << "\nEncrypted data: ";
        for( unsigned i = 0; i < cypherText->size(); ++i)
                cout << cypherText->at(i) << " ";
        cout << endl;

        string output = facade.decrypt(cypherText);
        cout << "\nDecrypted string:" << output << endl;
}
```

Note the following:

- The statement `srand ( (unsigned)time(NULL) );` is no longer part of this client.
- It is sufficient to include only the Façade.h file. The other files that needs to be included are implicitly included because they are included in this .h file.
- To be able to call the methods provided in the Façade, a variable of type `Facade` is instantiated and used to call the encrypt and decrypt methods.

References

[1] Judith Bishop. *C# 3.0 design patterns*. O'Reilly, Farnham, 2008.

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

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