Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 3: Template Method design pattern and Public Inheritance

Contents

3.1 Introduction

In this chapter, you will learn all about the Template Method design pattern. It uses public inheritance as basic strategy for its implementation. Therefore, public inheritance as an object oriented programming (OOP) concept and its implementation using C++ is introduced. To explain the inner working of Template Method, it is contrasted with the structure of a system that evolved through the application of the *pull up method* refactoring, which is also introduced in this chapter. After explaining the Template Method through discussing its inner working, its consequences, some implementation issues, as well as some common misconceptions and related patterns, we provide example implementations and conclude with exercises.

3.2 Programming Preliminaries

The Template Method pattern applies public inheritance to provide polymorphic operations. In this section we explain the OOP concept of inheritance and illustrate how it can be implemented in C++. We also include an example of refactoring that is very useful to reduce code duplication that can often be applied in class hierarchies that are formed by using public inheritance.

3.2.1 Public Inheritance

In OOP code reuse is promoted by allowing programmers to create new classes that reuse the code of existing classes through public inheritance. The inheritance relationships (also known as an *is-a* relationship) of classes gives rise to a hierarchy. The new classes, known as subclasses (or derived classes), inherit all public and protected attributes and methods from their superclasses (or parent classes) i.e. a derived class may use these attributes and methods as if they are its own. Subclasses may extend their parent classes by adding attributes and methods. Sometimes derived classes may override (replace) some of the methods of their parent classes.

Public inheritance is often applied to avoid if-then-else and switch structures in code in order to enhance the readability and maintainability of the code. However, one should be cautious not to overuse this technique. The *is-a* relation should always be semantically sound. It should make sense that everything that applies to the parent class also applies to its derived classes. The requirement of semantic soundness in inheritance relationships between classes is explained by Item 32 in [4] that states:

> *Make sure public inheritance models "is-a"*

We talk about *false is-a relations* if this item is violated.

In OOP class hierarchies three types of methods can be distinguished:

non-virtual methods
The implementation of these methods are provided in the class that defines them.

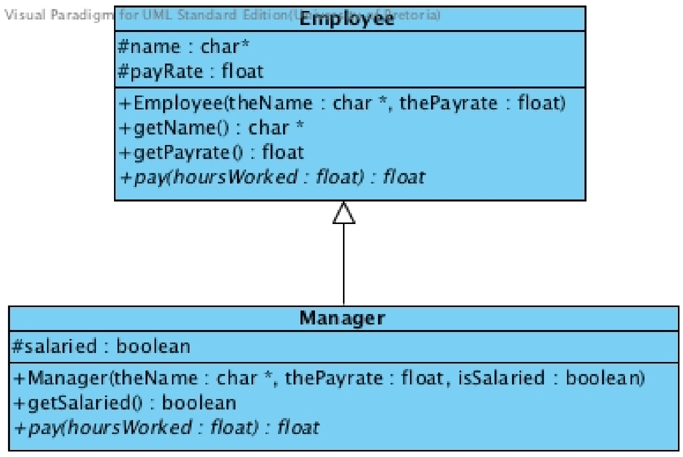

Visual Paradigm for UML Standard Edition(University of Pretoria)
Employee
#name : char*
#payRate : float
+Employee(theName : char *, thePayrate : float)
+getName() : char *
+getPayrate() : float
+pay(hoursWorked : float) : float
Manager
#salaried : boolean
+Manager(theName : char *, thePayrate : float, isSalaried : boolean)
+getSalaried() : boolean
+pay(hoursWorked : float) : float

Figure 1: Class diagram showing inheritance

virtual methods
These are methods for which a default operation is supplied by the class that defines them, but may be overridden in classes that are derived from the class that defines them. These methods are declared `virtual` in the class definition. A default implementation may be supplied, like the `pay(float)` implementation in the above example. The default operation may also be empty. An empty default implementation is indicated as such by adding {} after the method declaration in the class definition.

pure virtual methods
These are methods that are not implemented in the class that defines them. They have to be implemented in classes that are derived from this class. These methods are also declared `virtual` in the class definition. The fact that it is pure virtual is indicated by adding `=0` after the method declaration in the class definition. When doing so the compiler will check that all derived classes provide implementations for the method.

In UML virtual and pure virtual methods are written using an italic font.

3.2.2 C++ Example of public inheritance

The example we use here to illustrate the modelling and C++ implementation of public inheritance was adapted from [6]. Figure 1 is the class diagram of a base class `Employee` and a derived class `Manager` showing the inheritance relationship between them. Every manager is an employee, while not all employees are managers. Therefore this *is-a* relationship between `Manager` and `Employee` is semantically sound.

The following is the code to define the `Employee` class that will typically be stored in a file called `Employee.h`:

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
```

```
class Employee {
public:
  Employee(char* theName, float thePayRate);

  char* getName() const;
  float getPayRate() const;

  virtual float pay(float hoursWorked) const;

protected:
  char* name;
  float payRate;
};
#endif
```

You will notice that the instance variables are **protected** and not **private** as we are used to. This is to give the methods of any derived classes of this class access to these variables. The methods of this class will typically be defined in a file called `Employee.C`. The code in this file must be combined with the above mentioned definition when compiling the code. This is achieved by including the statement `#include "Employee.h"` at the beginning if this file. The `pay(float)` method is declared **virtual** to enable the derived class to override it. The following is the implementation of the methods in this class. Note how the constructor body is empty because all the instance variables are initialised in an initialiser list. Be careful when you do this with `char *` parameters.

```
#include "Employee.h"

using namespace std;

Employee::Employee(char* theName, float thePayRate) :
     name(theName),
     payRate(thePayRate) {}

char* Employee::getName()  const
{
    return name;
}

float Employee::getPayRate() const
{
    return payRate;
}

float Employee::pay(float hoursWorked) const
{
    return hoursWorked * payRate;
}
```

You will notice that `name` and `theName` are pointing to the same location in memory and therefore it is up to the program making use of the class to clear the memory when needed. In these cases, it would be best if the constructor allocates separate memory so that `name` and `theName` are not dependent on each other.

The following is the class definition of the `Manager` class:

```
#ifndef MANAGER_H
#define MANAGER_H

#include "Employee.h"

class Manager : public Employee
{
public:
    Manager(char* theName, float thePayRate, bool isSalaried);
    bool getSalaried() const;
    virtual float  pay(float hoursWorked) const;

protected:
    bool salaried;
};
#endif
```

The line `class Manager :  public Employee` causes `Manager` to inherit all the public and protected data and methods of `Employee`. Only the data and methods of `Manager` that are additional to those inherited from `Employee` are included in the class diagram of `Manager` and need to be written in code. When writing the code to implement these methods one should call the methods in the `Employee` rather than rewriting the code. It is important to avoid duplicating code. Duplicate code is undesirable because you face the risk that alteration to such code may lead to inconsistencies in the behaviour of the system when not all the instances of that code is updated.

The following is an example of how the constructor of the derived class can be implemented. Once again it uses an initialiser list to initiate all its instance variables. The first item in this list is a call to the constructor of its parent class.

```
Manager::Manager (char* theName,  float thePayRate,
                  bool isSalaried) :
    Employee( theName, thePayRate), salaried(isSalaried) {}
```

In this example the calculation depends on whether the manager is salaried or not. If the manager is salaried his/her flat pay rate is returned, otherwise, it makes the same calculation as a regular `Employee` simply by calling the existing method.

```
float Manager::pay(float hoursWorked) const
{
    if (salaried)  return payRate;
    return Employee::pay(hoursWorked);
}
```

Note how the class name of the parent class is needed to call the method in the parent class. Without this specification this statement will call the current method. Then it will be an infinite recursive call.

`Manager` objects can be used just like `Employee` objects in any code that uses these classes. However, it has additional methods that is usable only by `Manager` objects.

3.2.3 *Pull up method* Refactoring

Refactoring is the process of changing existing code in such a way that its behaviour is not altered, yet the internal structure is improved. It is accepted that no matter how well a system is designed before it is implemented, it is inevitable that its design needs alteration after implementation to cater for unforeseen changes and extensions. Fowler [1] has written a book in which he names and explains 72 refactoring methods that can be applied to methodically and systematically improve the design of existing code. We will introduce a number of basic refactoring methods in these lecture notes as we deem fit.

The refactoring we introduce here is called *Pull up method*. If it ever happens that two or more subclasses of a parent class have methods with identical results, this refactoring suggests that a generalised version of the method be created in the parent class to replace these methods in the subclasses. The result is a system where multiple methods in subclasses call this generalised method situated in the parent class ([1]:322).

3.3 Template Method Pattern

3.3.1 Identification

Name	**Classification**	**Strategy**
Template Method	Behavioural	Inheritance
Intent		
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. ([2]:325)		

3.3.2 Structure

The structure of the Template Method design pattern is given in Figure 2. The design pattern comprises of two classes, the abstract class which defines the interface and the template method function. This function defers implementation of primitive operations to its subclass.

3.3.3 Problem

Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended [3].

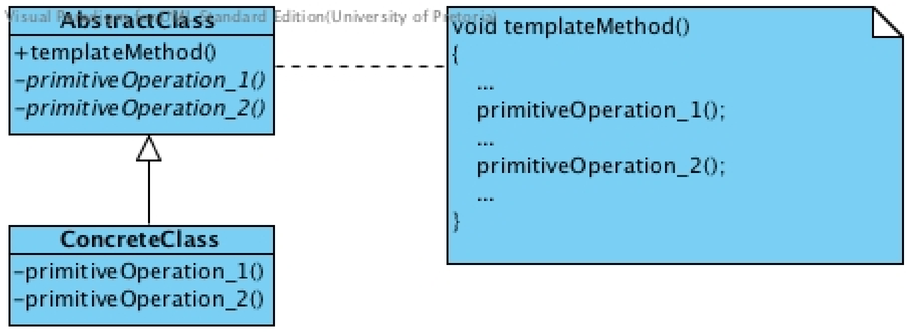


Figure 2: The structure of the Template Method pattern

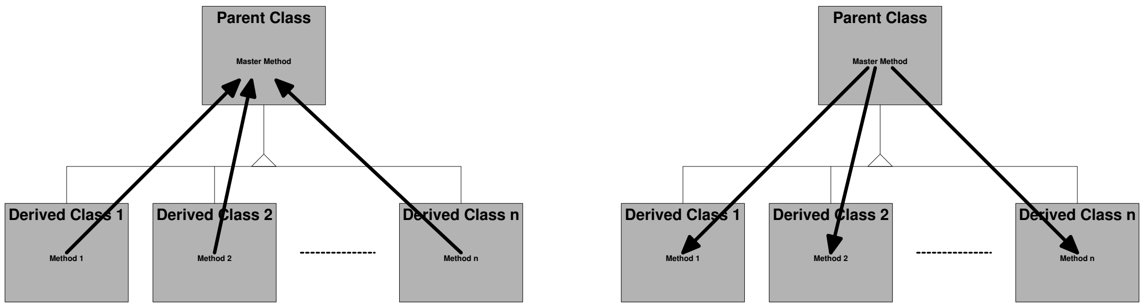


Figure 3: The Hollywood principle

3.3.4 Participants

AbstractClass

- Implements a template method defining the skeleton of an algorithm. Among others, this method calls a number of defined primitive operations.
- Defines abstract primitive operations that appear as steps in the template method.

ConcreteClass

- Implements the defined primitive operations to carry out the subclass-specific steps of the algorithm.

3.4 Template Method Pattern Explained

3.4.1 Clarification

The Template Method pattern prescribes the use of a template method. Usually a template method is described as an algorithm with pluggable steps. Some of the steps of the algorithm may be handled by the class defining the template method while others are pluggable. The pluggable steps are declared as abstract (virtual) methods and may therefore be overridden in subclasses.

The Template Method pattern does something similar to the outcome of *pull up method* refactoring mentioned in Section 3.2.3, just the other way around. Instead of having multiple methods calling a common method, the template method contains one or more statements calling abstract methods. These abstract methods have varying implementations situated in subclasses. Figure 3 illustrates the difference between the outcome of *pull up method* refactoring and the application of the Template Method pattern. After application of the *pull up method* refactoring, multiple methods in multiple subclasses call the master method that is implemented in the parent class. In the application of the Template Method pattern, the master method in the parent class calls methods that are implemented in subclasses. This is referred to as the Hollywood principle: *Don't call us, we'll call you* coined by Sweet [9].

Three types of operations can be defined in the body of a template method. These coincide with the three types of methods that are distinguished in OOP:

invariant operations
These are steps in the algorithm that are the same for all objects regardless to which derived class they may belong. These steps are implemented using non-virtual methods.

optional operations
These are steps that may be skipped under certain conditions. These methods are declared `virtual` in the class definition and provided with an empty default implementation. The designer of a subclass can decide whether it needs to be implemented or not. If a subclass does not implement such step it is skipped in the execution of the algorithm for objects of that subclass.

variant operations
These are steps that was identified as the pluggable parts of the algorithm. They have to be implemented in the derived classes. They are defined as pure virtual methods in the class definition.

3.4.2 Code improvements achieved

- Code duplication is reduced because the algorithm described in the template method appears only once in the base class as opposed to be duplicated in subclasses if the template method was not used.
- The code pertaining to the algorithm is easier to maintain because it is centralised in the template method for its invariant parts and also need not to worry about the variant parts because that is deferred to the subclasses.
- The system is easier to extend since a new class that has to use the algorithm described by the template method need not implement the whole algorithm, but only the parts that may vary.
- Coupling is reduced because classes that call the template method is separated from the concrete classes. There are less dependencies because they depend only on the abstract class and never communicate directly with the concrete classes.

3.4.3 Implementation Issues

Following the advice of [8], the virtual methods should be declared private. If a method is private in a parent class it cannot be called by methods in its subclasses. Therefore, these methods can only be invoked through calling the template method itself – which is exactly what is intended.

A problem with the Template Method pattern that was pointed out by [5] is that by its nature, it promotes false *is-a* relations. You are advised to be cautious about this and diligent in maintaining semantic integrity in your code.

Another problem that often arises when applying the Template Method Pattern is added complexity caused by the fragmentation of the code implementing the algorithm. The algorithm is implemented in a number of methods of which some are implemented in the base class and others are implemented in the derived classes. In order to minimise such fragmentation, you are advised to minimise the number of virtual and pure virtual methods defined for template methods. More virtual methods increases the flexibility but also increases the complexity and maintainability of the system.

3.4.4 Common Misconceptions

- The use of inheritance does not necessary imply that the template method pattern is applied. To be an application of the template method pattern, the abstract class needs to have a method that acts as an algorithm skeleton. This method must call a number of abstract methods that are defined in the parent class and implemented in the subclasses.

- Templates as defined in the C++ language are a feature of the C++ programming language is not related to the Template Method pattern. C++ define templates for data types whereas the Template Method pattern define a template for an algorithm. A method that applies a C++ template acts like an implementation of the Template Method to operate with generic types. This allows a method to work on many different data types without being rewritten for each one as long as all the operations used in the method is defined for the data types substituted in the template type. This feature can be applied as an alternate solution to solve many of the problems that can also be solved using the Template Method pattern. For example the problem in Exercise 3. However, the application of C++ templates can only vary operations that can be overloaded while the Template Method pattern allows for varying any kind of operation.

- We do **NOT** agree with [7] who states that Factory Method is a specialisation of the Template Method. For this statement to apply, the participants of Factory Method should match those of the Template Method which is not the case. Factory Method can only be viewed as an implementation of the Template Method pattern if one see the `AnOperation()` method of Factory Method as a template method.

3.4.5 Related Patterns

Strategy

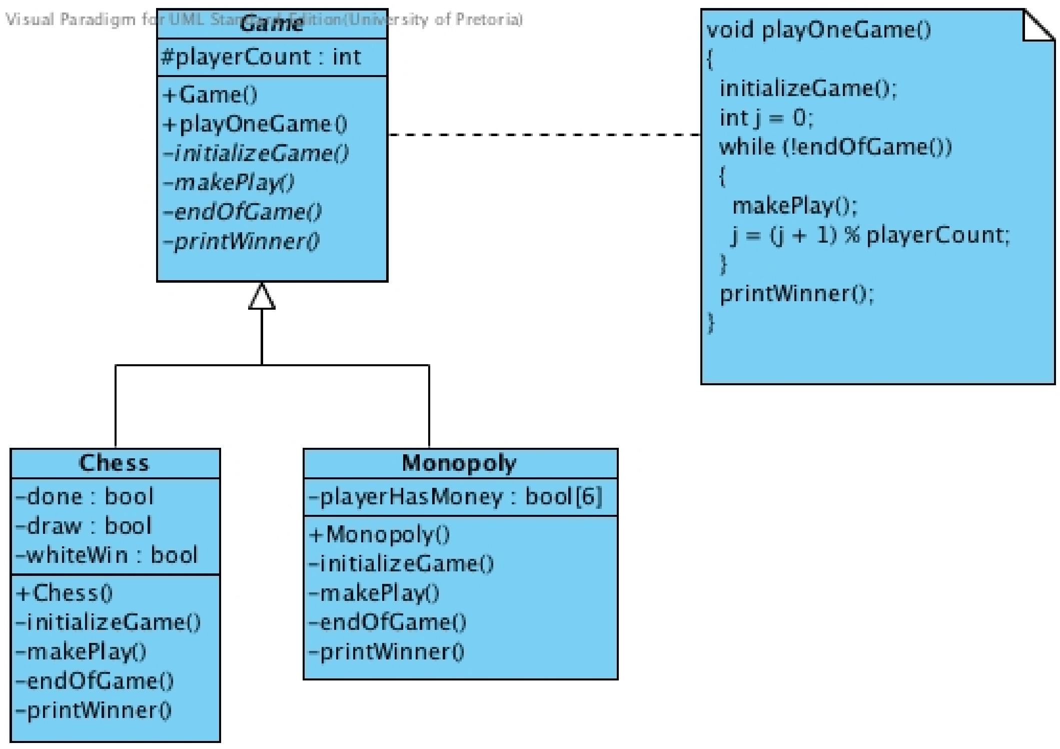
Both Strategy and the Template Method pattern defer implementation. However, Strategy uses delegation to defer the implementation of a complete algorithm while the Template Method pattern uses inheritance to defer only specific parts of an algorithm.

Factory Method
Although the Factory Method is not a specialisation of the Template Method pattern, it is related to the Template Method pattern. Many of the non-virtual methods that participate as `AnOperation()` in solutions that apply the Factory Method pattern are often template methods that, among others, call factory methods.

Adapter
Both the Adapter pattern and the Template Method pattern provides an interface through which operations that are implemented in other classes are called. The difference is that Adapter provides an interface to access non-complying operations that cannot be changed while the operations accessed through the template method is expected to comply and are most likely to change.

Builder
Both Builder and the Template Method pattern require a process to be encapsulated in a method. In fact the method that has to be implemented by the concrete builders to assemble a product is a template method.

3.5 Example



```
void playOneGame()
{
  initializeGame();
  int j = 0;
  while (!endOfGame())
  {
    makePlay();
    j = (j + 1) % playerCount;
  }
  printWinner();
}
```

Figure 4: Class diagram of a Game c4ass implementing a Template Method

Figure 4 is a class diagram of an application that implements the Template Method pattern. It is based on the example of a an application of the Template Method pattern by [10]. The participants are as follows:

Participant	**Entity in application**
AbstractClass	Game
ConcreteClass	Monopoly Chess
templateMethod()	playOneGame()
primitiveOperation()	initializeGame() makePlay(int) endOfGame() printWinner()

AbstractClass

- The `Game` plays the role of AbstractClass. It implements a template method named `playOneGame()` that defines the skeleton of an algorithm as shown in the note in the UML Class Diagram in Figure 4. This method is an algorithm that calls the defined primitive operations named `initializeGame()` `makePlay(int)` `endOfGame()` `printWinner()` as specified.
- The primitive operations are defined as abstract member functions of the `Game` class. They are declared pure virtual and hence are not implemented in the `Game` class.

ConcreteClass

- This example has two concrete classes named `Monopoly` and `Chess`. Each of these classes implements the defined primitive operations to carry out the subclass-specific steps of the algorithm which is different for each game.

3.6 Exercises

1. Figure 5 is a UML class diagram showing classes called `Cow`, `Pig`, `Animal` and `Goat`. These classes are part of the implementation of a strategy game in which the player is required to run a farm as profitable as possible. It is an implementation of the Template Method Pattern.
    - identify the participating classes.
    - which method is the template method?
    - identify a private non-virtual method in the Animal class.
    - identify a virtual method that is not pure virtual
2. Assume a class called `Secretary` that is derived from the `Employee` class have been added to the system described in Section 3.2.2. Further, assume the following methods are implemented respectively in the `Manager` class and the `Secretary` class:

```
void Manager :: workDay()
{
        cout << "arrive" << endl;
        cout << "drink␣coffee" << endl;
```

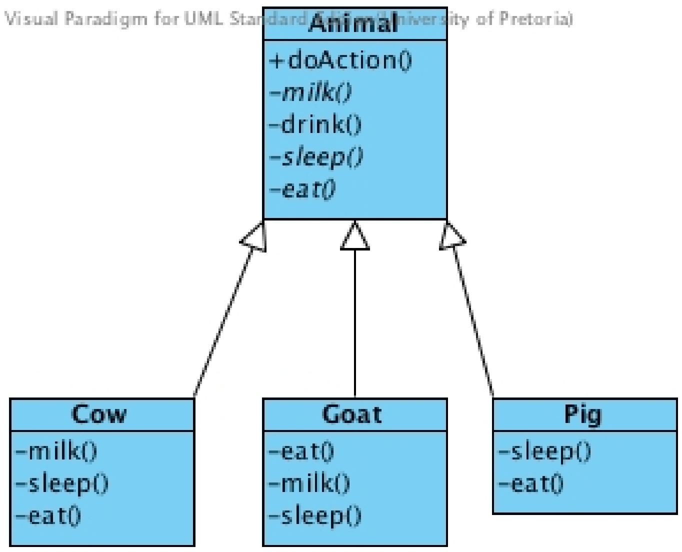


Figure 5: Animal class implementing a template method

```
        cout << "check calendar" << endl;
        cout << "chair meetings" << endl;
        cout << "take lunch break" << endl;
        cout << "design  business solutions" << endl;
        cout << "go home" << endl;
}

void Secratary :: workDay()
{
        cout << "arrive" << endl;
        cout << "drink coffee" << endl;
        cout << "check calendar" << endl;
        cout << "take minutes of  meetings" << endl;
        cout << "take lunch break" << endl;
        cout << "arrange venues and catering for meetings" << endl;
        cout << "go home" << endl;
}
```

- code a template method in the `Employee` class that describes the basic algorithm of a work day for both these classes in steps that describe the flow in steps separating the invariant sections from the variant sections
- define primitive operation methods for each of these steps and implement them in the appropriate classes.

3. Implement the selection sort algorithm to sort an array of objects of unknown type as a template method. Implement concrete classes for two different types of objects, for example integers and strings, with implementations for the abstract methods you had to defined and used in your template method. Add another concrete class that extends your abstract class for yet another type of object that can be sorted by the

algorithm in your template method, for example triangles which are compared in terms of their areas.

References

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Reading, Mass, 1999.

[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].

[4] Scott Meyers. *Effective C++: 55 specific ways to improve your programs and designs.* Pearson Education Inc, Upper Saddle River, NJ 074548, 3^{rd} edition, 2008.

[5] Alex Miller. Patterns I Hate #2: Template Method. `http://tech.puredanger.com/2007/07/03/pattern-hate-template/`, July 2007. [Online; accessed 29-June-2011].

[6] Robert I. Pitts. Introduction to inheritance in c++. `http://www.cs.bu.edu/teaching/cpp/inheritance/intro/`, March 2010. [Online; accessed 29-June-2011].

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

[8] Herb Sutter. Sutter's mill: virtuality. *C/C++ Users Journal - Graphics*, 19:53 – 58, September 2001. ISSN 1075-2838.

[9] Richard E. Sweet. The mesa programming environment. *ACM SIGPLAN Notices*, 20:216–229, June 1985. ISSN 0362-1340.

[10] Wikipedia. Template method pattern — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Template_method_pattern&oldid=436540809`, 2011. [Online; accessed 30-June-2011].