Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 5: Prototype design pattern

Contents

5.1 Introduction

In this lecture you will learn all about the Prototype design pattern. We discuss the principles it applies, the problems it addresses, and some implementation issues.

This pattern applies the basic concept of caching. Caching is a general technique that is applied in practice to improve the performance of a system at various levels ranging from hardware implementation of cache memory up to its use in design and in application programs. We explain the use of caching at the hand of a few practical examples on program implementation and use level.

The implementation of the prototype pattern relies on the ability to copy existing objects. Thus, when implementing this pattern the programmer should be aware of the practical implications of shallow and deep copying of complex objects. We define and explain these concepts to equip the reader with the required insight. We also discuss the idea of a copy constructor. The C++ language automatically provides a default constructor, copy constructor and destructor for each defined class. However, in some cases these defaults does not behave as indented by the design. In these cases the programmer has to provide an implementation to override the inappropriate default. This lecture gives you the necessary background knowledge to be able to implement the prototype design pattern.

5.2 Programming preliminaries

5.2.1 Caching

The principle behind the application of the prototype pattern is the concept of caching. To cache something on implementation level is to save the result of an expensive calculation so that you don't have to perform the calculation the next time its result is needed. Sometimes caching results is more efficient than re-doing the calculations every time the result is needed. The tradeoff is the the use of more memory for storage of the results.

5.2.1.1 Programming example

Caching is often applied in game programming where it is extremely important to execute the calculations for frame rendering at high speed. Although the following function presumably has stored the value of π instead of recalculating it every time the function is called, it is still expensive:

```
int nextX(int x)
{
    if (x < 10)
        return x * x - x;
    else
        return sin(x % 360);
}
```

The execution speed of this function can be drastically improved by precomputing values and store the results in lookup tables. A lookup table for the top calculation can be generated by the executing following code once:

```
int value[10];
for(int i = 0; i < 10; i++)
{
    value[i] = i * i - i;
}
```

Similarly a lookup table for the values of `sin` can be created by executing the following loop once:

```
int sinTable[360];
for (int i=0; i<360; i++)
{
    sinTable[i] = sin(i);
}
```

After defining these lookup tables, the above function can then be altered to simply look up these values instead of calculating them, resulting in marked performance improvement:

```
int nextX(int x)
{
    if (x < 10)
    {
        return value[x];
    }
    else
    {
        return sinTable[x % 360];
    }
}
```

5.2.1.2 Web services example

Caching is often applied to dynamic web-pages. The page is cached when it is loaded the first time. When it it needed subsequent times, instead of redoing all the calculations that may involve database queries, expensive template rendering, and execution of business logic, the cached page is simply copied from the cache.

5.2.1.3 Spreadsheet example

Caching can often be used to speed up the execution of a spreadsheet containing expensive computations that are re-used. For example the following formula requires that the same `time_expensive_formula` needs to be calculated twice:

```
B1=IF(ISERROR(time_expensive_formula),0, time_expensive_formula)
```

The execution speed can be halved by storing (caching) the result of this formula and simply re-using it by reference the next time:

```
A1=time_expensive_formula
B1=IF(ISERROR(A1),0,A1)
```

5.2.2 Copying complex objects

The Prototype pattern is dependent on the ability to create copies of existing objects. The implementation of C++ code to achieve this requires understanding of concepts of deep and shallow copying of objects, as well as an understanding of copy constructors. In this section we explain these concepts and illustrate how it is implemented using the C++ programming language.

5.2.2.1 Shallow copying

If an object of a class is copied, the copy constructor is called to construct the copy. The default copy constructor will perform a shallow copy. This means that the *values* of the instance variables of the original object are assigned to the corresponding instance variables of the copied object. If these instance variables are primitives, this poses no problem. However, if the instance variables are pointers to primitives or objects, assigning their values to the pointer variables of the copied object would mean that these pointers will point to the same objects. The result in a situation where the instance variables of the copy of the object and the original object are shared is shown in Figure 1.

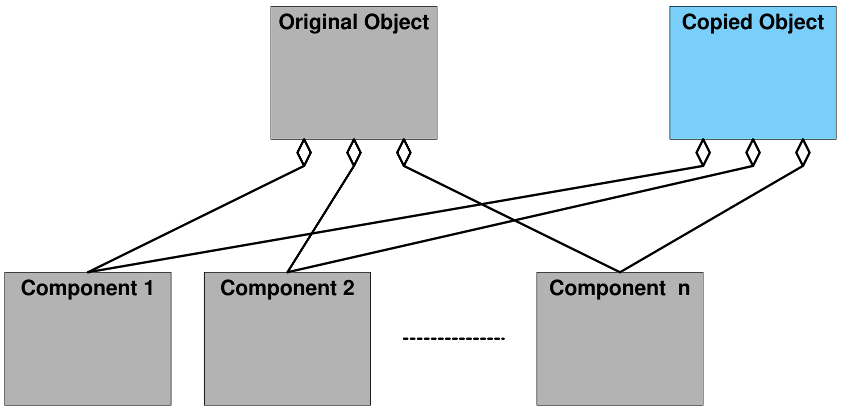


Figure 1: Copy of an object that shares the instance variables of the original

5.2.2.2 Deep copying

If the instance variables of a class are defined as pointers to dynamically allocated memory, it is better to define your own copy constructor that performs a deep copy. When performing a deep copy, a duplicate object of each instance variable of the original object is created. The result in a situation where the copy of the object has its own copies of the instance variables of the original object as shown in Figure 2. This way the copied object and the original object are totally independent.

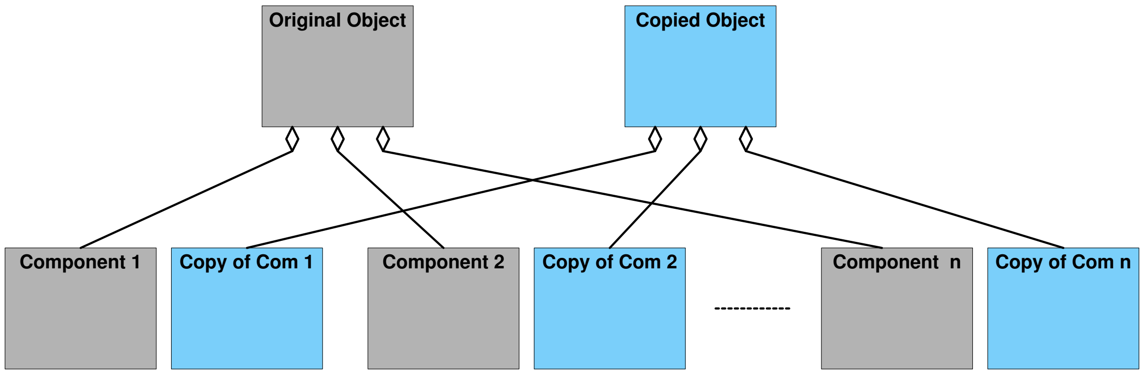


Figure 2: Copy of an object that has its own copies of the instance variables of the original

5.2.2.3 Copy constructor

In C++ every class has a copy constructor. It is a constructor that takes one parameter. This parameter is an object of the class. If the programmer does not implement a copy constructor, a default is provided. The default copy constructor makes a shallow copy.

When implementing the copy constructor one should use the default signature of the copy constructor in order to override it. It takes a reference to a const parameter. It is `const` to guarantee that the copy constructor doesn't change it, and it is a reference because if it was a value parameter the execution would require making a copy, which automatically invokes the copy constructor resulting in infinite recursive calls to the copy constructor.

The following is an example of the definition of a Person class containing explicit definitions of its default constructor, copy constructor, destructor and assignment operator.

```
class Person
{
    public:
        Person(); //default constructor
        Person(const Person & ); //copy constructor
        Person & operator=(const Person & ); //assignment operator
        ~Person(); //destructor
    private:
        char* name;
        Address address;
        int age;
};
```

Assume the class `Address` is properly defined and implemented. The following is an example of the implementation of the copy constructor of this `Person` class:

```
Person :: Person(const Person & p): age(p.age), address(p.address)
{
        char* temp = new char[strlen(p.name) + 1];
        strcpy(temp, p.name);
        name = temp;
}
```

Notice how it provides a deep copy of the character array containing the name of a person. Also notice how it uses a member-list to intialise the `age` and `address` instance variables. The first expression in this list (`age(p.age)`), initialises the instance variable called `age`, to the value of the corresponding instance variable of the `Person` object `p` that was passed as parameter to this copy constructor. Similarly, the second expression in this list (`address(p.address)`), initialises the instance variable called `address`, to the value of the corresponding instance variable of the `Person` object `p` that was passed as parameter to this copy constructor. However, in this case this instance variable is not a primitive data type. Therefore, the copy constructor of the `Address` class will be called to construct a copy of the address contained in `p.address`.

When a copy constructor is called a new object is created that is a clone of the object that was passed as a parameter. The following are examples of calls to the copy constructor of a class `Person`, assuming that `p` is an existing `Person` object.

```
Person* q = new Person(p);
Person r(p);
Person s = p;
```

In the first statement the copy constructor is explicitly called to create a new `Person q` that is a clone of `p`. The second statement implicitly calls copy constructor to build a Person object `r` to be a clone of `p`. The last statement initialises the variable `s` where it is declared. This statement also implicitly calls copy constructor to build a `Person`. In this case the object `s` is created to be a clone of `p`. Similar to how the copy constructor of `Person` calls the copy constructor of `Address`, the copy constructor of `Person` can be called whenever it appears as value parameter that is initialised with an argument.

If your design requires shallow copies, there is no need to implement a copy constructor. If the object has no pointers to dynamically allocated memory, there is no difference between a shallow and a deep copy. Therefore the default copy constructor is sufficient and you don't need to write your own. However, if you implement a copy constructor, it is good practice to also provide a destructor and an assignment operator.

5.2.2.4 Summary

Understanding the difference between shallow and deep copying enables you to apply the appropriate copying when implementing a system. Visit `http://www.youtube.com/watch?v=xCq3D9aFAyI` to see a video containing an explanation of the difference between deep and shallow copying. Understanding this difference combined with an awareness of the nature of the default copy constructor enables you to use or implement the most suitable copy constructor for your current design.

5.2.3 Associations revisited

Require an explanation regarding the relationship between classes and the effect that define an attribute of a class on the stack implies composition (part-of relationship) while defining it for heap allocation implies aggregation (has-a relationship). Refer to Figure 4 in L05 for an example of aggregation.

5.3 Prototype Pattern

5.3.1 Identification

Name	Classification	Strategy
Prototype	Creational	Delegation
Intent		
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype (Gamma et al. [2]:117)		

5.3.2 Structure

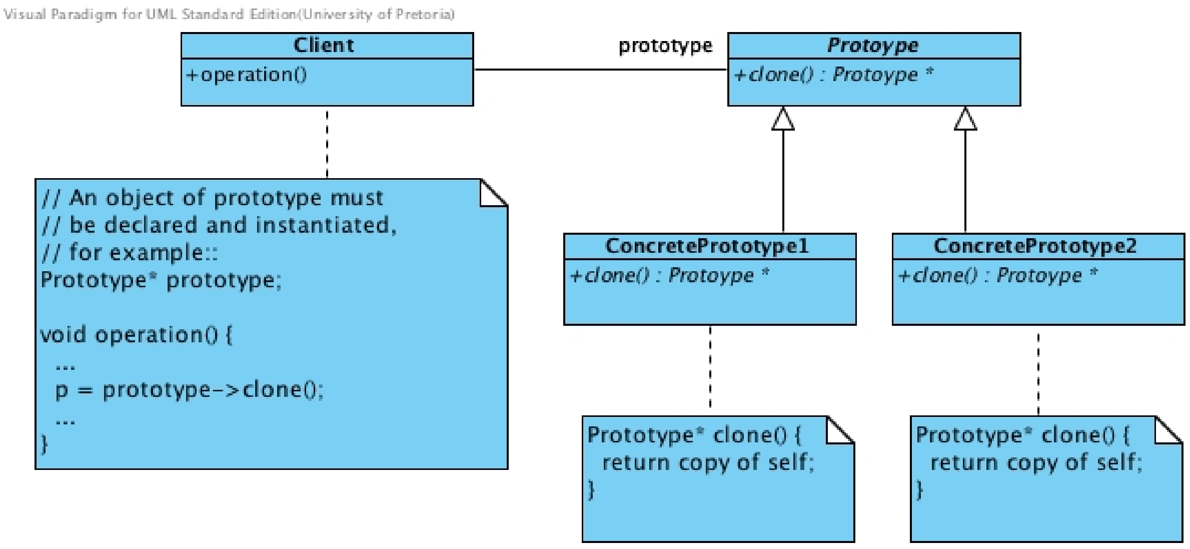


Figure 3: The structure of the Prototype Pattern

5.3.3 Problem

The constructor of a class contains computationally expensive or manually time consuming procedures. These can be avoided by creating a copy of the object rather than going through the entire creation process each time from the beginning.

5.3.4 Participants

Prototype

- Declares an interface for cloning itself.

ConcretePrototype

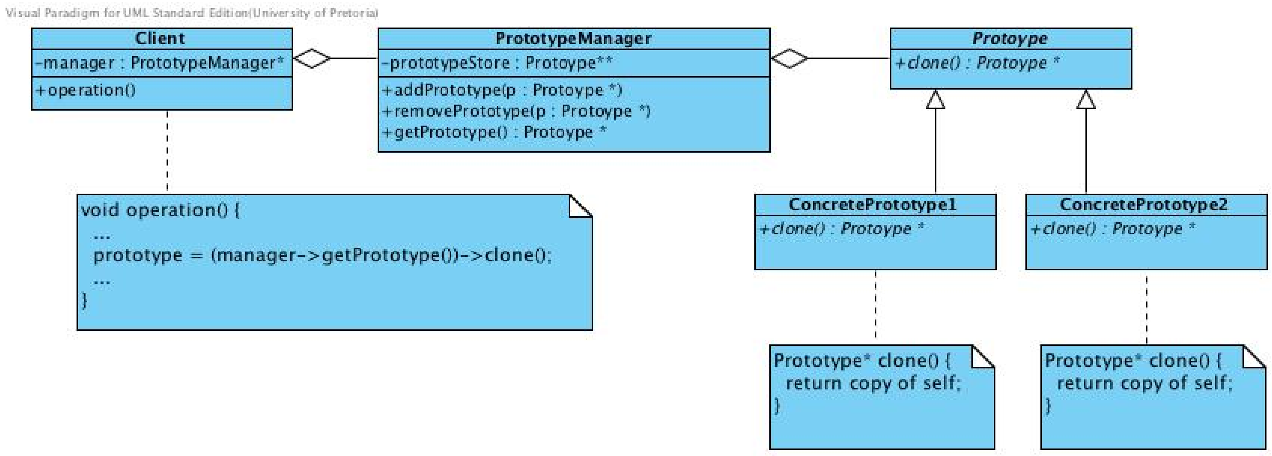
- Implements an operation for cloning itself.

Client

- Creates a new object by asking a prototype to clone itself

5.3.5 Alternate Structure

Although the prescribed structure of the prototype design pattern as provided by Gamma et al. [2] does not include a prototype manager, many writers Gamma et al. [2], Bishop [1], Huston [5], Malloy [6] advise to implement a prototype manager when using the prototype pattern. The manager maintains a list of existing objects that can be used as prototypes that are used when spawning new objects. Such a manager is needed when the application needs to add or delete prototypes dynamically. The existence of a prototype manager lets the clients extend and take inventory of the system without writing code. The structure of the prototype pattern when using a prototype manager is shown in Figure 4. The prototype manager has responsibilities similar to the caretaker participant of the Memento Pattern.



Visual Paradigm for UML Standard Edition(University of Pretoria)
Client
-manager : PrototypeManager*
+operation()
PrototypeManager
-prototypeStore : Protoype**
+addPrototype(p : Protoype *)
+removePrototype(p : Protoype *)
+getPrototype() : Protoype *
Protoype
+clone() : Protoype *
ConcretePrototype1
+clone() : Protoype *
ConcretePrototype2
+clone() : Protoype *
void operation() {
...
prototype = (manager->getPrototype())->clone();
...
}
Prototype* clone() {
return copy of self;
}
Prototype* clone() {
return copy of self;
}

Figure 4: The structure of the Prototype Pattern with a manager

5.3.6 Participants of the alternate structure

Prototype

- Declares an interface for cloning itself.

ConcretePrototype

- Implements an operation for cloning itself.

PrototypeManager

- Is responsible for keeping prototypes and provide functionality to add and remove prototypes.

- Act as mediator through which the client can ask a selected prototype to clone itself

Client

- Creates a new object by asking the prototype manager to ask a selected prototype to clone itself

5.4 Prototype Pattern Explained

5.4.1 Situations justifying the use of the pattern

The prototype pattern creates new objects by cloning designated existing objects. The following situations can benefit by the application of the prototype design pattern:

1. When it is known that the constructor of the class contains a computationally expensive or time consuming processes. In such cases processing time can be saved by the application of the prototype design pattern. Copying an object would be faster than creating a new object from scratch. A good example is when the constructor does file I/O.

2. When it is known that a relative small set of standard variations of the objects will be needed by users, the prototype design pattern can be applied to save user effort. In stead of expecting the user to specify a number of fine grained detailed information before an object is created, the user is presented with a set of prototypes to select from in which each of the prototypes the finer grained information has a certain combination of default values. Therefore, when instances of a class can have one of only a few combinations of state, these combinations can be encapsulated in prototype instances that can easily be recreated.

3. When an object undergoes a building process to be constructed, the prototype design pattern can be used to eliminate the process in subsequent constructions of the same object. Usually, the Builder pattern is applied to define such building process. However, when applying the builder design pattern all clients to have to understand or deal with this process. It is possible to relieve the clients from having to deal with the process by applying the prototype pattern [3]. When doing this, the builder can be applied to create a number of prototypes that are then later cloned by the clients rather than having to build each object from scratch.

5.4.2 Improvements achieved

- The client program will be more generic. Instead of containing code that invokes the `new` operator on a hard-coded class names, it will call the `clone()` method on the prototype. This level of indirection (calling a method that will create the object on your behalf) is how the pattern is providing the added flexibility and the ability to swap families of products without touching the client code.

- The system structure will be more streamlined. Without the application of the prototype pattern, the creation of a variety of object types is achieved by implementing derived classes to instantiate the different types resulting in an elaborate hierarchy, whereas the same variety can be achieved simply by creating prototypes instead of classes.
- If a prototype manager is implemented, it is possible to instantiate new types at runtime simply by adding more prototypes to the prototype manager.
- When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.
- Prototype reduces the necessity to subclass as what is done in the Factory Method. The Creator hierarchy of Factory Method is reduced to one class and the duty of creating a "copy" is left to the object itself.

5.4.3 Practical examples

There are many practical scenarios where this design pattern really helps. Here follows few of them mentioned by Gibson [3]

- Generating a GUI having many numbers of similar controls: This is a quite frequent scenario. One can have a form or GUI that hosts many similar controls. In order to maintain the consistency, one needs to initialise every object to the same standard state. This process of initialisation gets repeated again and again for each control increasing the number of lines of code. In order to optimise this part of the code, one can one have an object of a control initialised to a standard state and then replicate the same object to create as many controls needed.
- Building a game that often reuse different visual objects: The Prototype pattern is useful in this case because instead of creating the objects that get instantiated, various objects can easily be created during the game execution by cloning prototypical objects. An added benefit is that changes to the scene an objects in a scene can rapidly be changed by replacing the prototypical objects with different ones.
- Applying a variety of analyses on the same result set from a database: Database queries and the creation of result sets are computationally expensive operations. When an application does an analysis on a set of data from a database, normally the information from the database is encapsulated into an object and the analysis is performed on the object. If more analyses are needed on the same set of data, reading the database again and creating a new object for each analysis is expensive and can be avoided by using the Prototype pattern. The object used in the first analysis can be cloned and used for subsequent analyses.

5.4.4 Implementation Issues

The hardest part of the Prototype pattern is implementing the `clone()` operation correctly. In C++ an elegant way would be to use the copy constructor on `*self`. However, this is dependent on the correct implementation of the copy constructor. When cloning prototypes with complex structures, it is important to make deep copies to allow the copy to exist independent of the prototype [2].

In cases where there is a need to create variations of clones, one can parameterise the `clone()` operation. However, passing parameters in the `clone()` operation precludes a uniform cloning interface. One can also implement additional initialisation operations. However, in these situations the programmer should beware of deep-copying operations as the copies may have to be deleted (either explicitly or within Initialise) before you reinitialize them.

5.4.5 Common Misconceptions

- The use of the copy constructor provided in the C++ language does not necessary imply that the prototype pattern is applied. To be an application of the prototype pattern, the objects used for constructing new objects should be a finite set of stored objects.
- The cloning of objects by an application program does necessary imply that the prototype pattern is applied. An example that has been mentioned as a possible scenario for the application of the prototype pattern cloning the sessions from one server to another without disturbing the clients in an enterprise level application managing a server pool. It is unlikely that this would be implemented using the prototype design pattern since the reason for cloning these objects are not according to the intent of the pattern to clone multiple instances of an object that is hard to create from scratch. This scenario rather justify the application of the memento design pattern.

5.4.6 Related Patterns

Factory Method

Both Prototype and Factory Method are creational patterns. However, Prototype use delegation to create while Factory Method uses inheritance. They can also be used together. One can design a factory method that accept arguments and uses these arguments to find the correct prototype object, calls clone() on that object, and returns the result. The client replaces all references to the new operator with calls to the factory method.

Abstract Factory

Prototype and Abstract Factory are competing patterns in some ways. Prototype define new types simply by creating new prototypes while Abstract Factory requires the creation of new classes for defining new types. However, they can also be used together. An Abstract Factory might store a set of prototypes from which to clone and return product objects.

Abstract Factory and Builder
Similar to the prototype design pattern, these patterns hides the concrete product classes from the client, thereby reducing the number of names clients know about.

Composite and Decorator
Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype.

Singleton and Abstract Factory Prototype, Singleton and Abstract Factory are all creational patterns where you don't use the `new` keyword to create a product but you call a method that will create the specific product and return a pointer to it.

Singleton, Memento and Flyweight
These patterns administrate access to specific object instances similar to how Prototype administrates it. All of them offer factory methods to clients and share a create-on-demand strategy [4].

5.5 Example

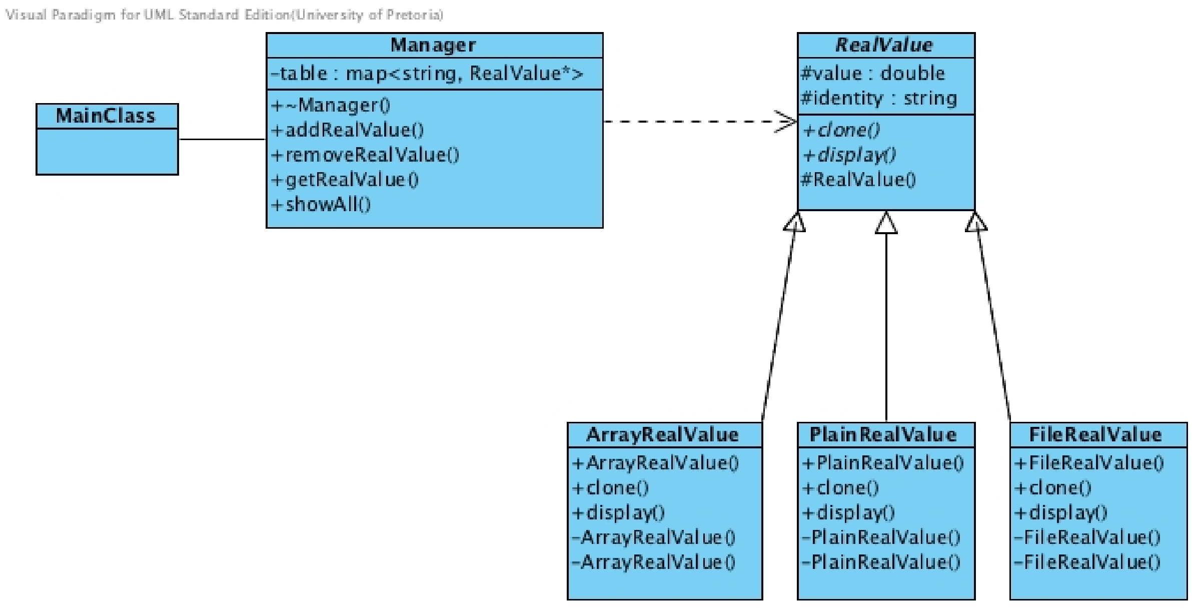


Figure 5: Class Diagram of a system illustrating the implementation of the prototype design pattern

Figure 5 is a class diagram of an application that implements the prototype design pattern. It is a nonsense program that allows the user to create a variety of objects. It also allows the user to nominate existing objects as prototypes which can be cloned. The instance variables of the objects that are cloned in this example are primitive data types. Therefore, there is no need to implement deep copies in their copy constructors or when cloning the objects. In this regard it is not a good example. However, the objects have time consuming constructors, which make them ideal candidates to benefit from being cloned rather than created from scratch when needed.

Participant	**Entity in application**
Prototype	RealValue
Concrete Prototypes	PlainRealValue, ArrayRealValue, FileRealValue
Prototype Manager	Manager
Client	main()

Prototype

- `RealValue` is an abstract class that implements an interface containing the pure virtual `clone()` method. For the purpose of this application a `display()` method is also defined. This method is used by our client to make the results of the execution visible. The default constructor of this class is protected to prevent client applications from instantiation objects of this abstract class type.

ConcretePrototype

- The classes `PlainRealValue`, `ArrayRealValue` and `FileRealValue` act as concrete prototypes.
- Both the default constructor and copy constructor of each of these classes are private to force client applications to use their specified parameterised constructors or their `clone()` methods to create an object of one of these kinds.
- The public constructors of these classes uses different procedures to create objects of their kind. The constructor of `PlainRealValue` uses a value that is passed through its parameter to instantiate an object. The constructor of `ArrayRealValue`, uses an array of real values that is passed through its parameters to instantiate an object. The average of the values in this array is calculated and the result is then used to instantiate an object. The constructor of `FileRealValue`, uses character string containing a file name ,that is passed through its parameters to instantiate an object. The file is read, and the average of the values in this file is calculated. The result is then used to instantiate an object.
- These classes implements `clone()` to clone itself. In each case this is done by using its private copy constructor with a command like the following:

```
FileRealValue* temp = new FileRealValue(*this);
```

- In this example the `identity` instance variable is changed when an object is cloned for the sake of being able to observe that a clone was made.

Prototype Manager

- The `Manager` class uses a `map` for keeping prototypes and provide functionality to add and remove prototypes.
- It acts as mediator through which clients can add and remove prototypes form the store and can also gain access to any of the stored prototypes.
- in this application the prototype store maintains pointers to objects owned by the client.

Client

- In this application the client has a `Manager` called `manager`. The user can specify at runtime which of the existing objects should be registered and used as prototypes, and uses `manager` to keep track of them.
- The client has a vector called `object`, of objects which are created on demand. The user can choose to create objects from scratch or to clone objects that has been registered as prototypes.
- When the user has chosen the appropriate option to clone an object, a new object is created by asking `manager` to ask a selected prototype to clone itself. In this example the user type the name of a prototype which is read into a variable named `name`. Thereafter a clone of the identified prototype is added to a vector of objects called `object` by executing the following statement:

```
object.push_back(manager.getRealValue(name)->clone());
```

5.6 Exercises

1. See `http://www.codeproject.com/KB/architecture/Prototype_Design_Pattern.aspx` for another nice example of an implementation of the prototype design pattern.
2. Write a simple system that manage presentations offered by Social Informer Pty Ltd (S-Inf). Identify attributes of each presentation should have – length, price, topic, etc. – and set them up as prototypes managed by a manager class. Create a test program that allows the administrator of S-Inf to select a presentation, get a copy of it, and fill in detail – date, venue, presenter, capacity, etc. – to create an instance for which members of public can register to attend.

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] David R. Gibson. Chapter 14 notes – prototype pattern. `http://mypages.valdosta.edu/dgibson/courses/cs4322/Lessons/Prototype/prototypeNotes.pdf`, 2011. [Online] accessed 2011/07/22.

[4] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Notes*, 37:161–173, November 2002. ISSN 0362-1340.

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

[6] Brian Malloy. Design patterns. `http://www.cs.clemson.edu/~malloy/courses/patterns/patterns.html`, 2000. [Online: Accessed 29 June 2011].