Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 2: Memento design pattern

Contents

2.1 Introduction

This lecture note will introduce the Memento design pattern. In order to understand the pattern, modelling delegation in UML will be discussed before the pattern is introduced. C++ techniques required to implement the pattern will also be introduced before tackling the pattern. This will support the reader to understand the example implementation of this pattern.

2.2 UML Preliminaries

2.2.1 Class diagrams

Classes in UML are modelled by drawing a rectangle containing at least the class name that is being modelled. Class names should begin with a capital letter and be descriptive.

The rectangle should divided into three sections. The top section is used for the class name, the middle section for the attributes and the bottom section for its operations. Figure 1 shows the basic structure of a class in UML. If sections are empty, they may be omitted. It is, however, recommended that all three sections be shown even if they are empty. In some modelling software it is impossible not to draw the second and third sections of a class even if they are empty [1].

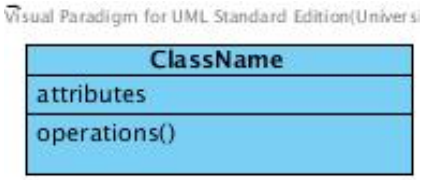

Figure 1: Structure of a UML Class

Attributes and operations are assigned visibility. The visibility relates to whether the feature (attribute or operation) is visible to the class that own the feature only, subclasses, outside the class either the package or globally. The classifications for the visibility are private, protected, package or public respectively. Private visibility is shown in UML using a minus (-), protected a hash (#), package a tilde(˜), and public a plus (+) before the feature. Omission of visibility means that it is either unknown or has not been shown.

Both attributes and operations can be further refined in order to be more descriptive.

Attribute refinements : There are three refinements that can be applied to attributes, namely: default values; derived attributes and multiplicity. Figure 2 illustrates how this are drawn in UML. `dateRegistered` is an example of an attribute with a default value. `age` is an example of a derived attribute. The multiplicity of `middleNames` states that an object of class Student may have 0 to 3 middle names representing the state of the object. Multiplicity may further be refined showing whether the values are ordered or unique, for example: `addressLine[2..4]{ordered}` indicates that each object should have at least two and maximum 4 addressLines and that the order in which these lines are used is important.

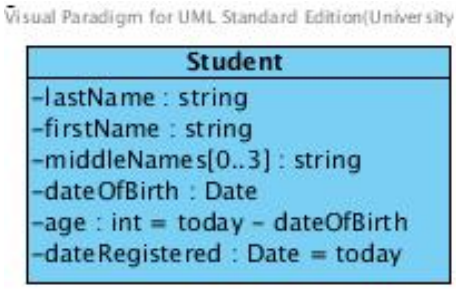

Figure 2: UML class attributes

Operation specification and refinement : Operations, refer to Figure 3, have the following form when specified in a class diagram: `operationName(parameter_list) : return_type`. The `parameter_list` is optional, but if included each parameter will have a basic form of: `parameter_name : parameter_type`, with the `parameter_name` being optional. Thus, `setName(name : String)` may also be expressed as `setName( : String)`. Omission of the return type assumes it to be void.

Parameters may further be assigned default values. The form to assign default values is by adding `= default_value` to the basic form of a parameter. An example would be assigning the current date to the date of registration for the operation for registration approval, that is: `registrationApproved(date : Date = today)`. Each parameter can further be specified as `in`, `out`, or `inout` in order to distinguish between *pass-by-value* and *pass-by-reference* in the implementation. The default is pass by value and is therefore `in`.

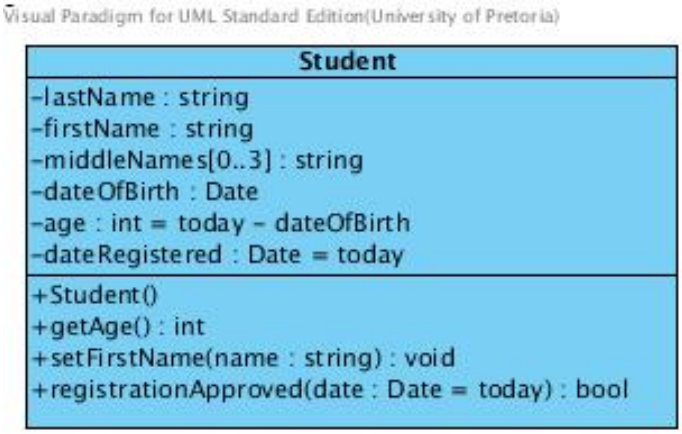

Figure 3: UML class operations

In some UML modelling tools it is possible to group operations using stereotypes which include `<<constructor>>`, `<<query>>`, `<<update>>` etc. Adding properties to features is also possible.

2.2.2 Modelling delegation

Modelling delegation relationships between UML classes is achieved by drawing a solid line between the classes involved in the relationship. Figure 4 shows two classes named ClassA and ClassB that are associated with one another.

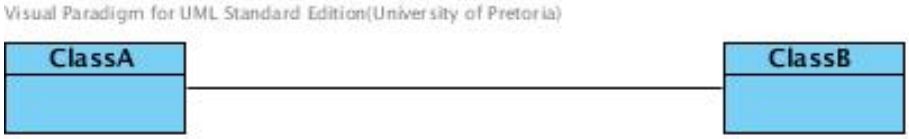

Figure 4: Binary relationship

As with attributes, relationships also have **multiplicity** indicating the number of object instances at the other end of the relationship for an instance of the current class. The omission of multiplicity assumes 1. In Figure 5, the relationship shows that a library may have many books, but that a book may belong to only one library.

Figure 5: Binary Directed Association showing multiplicity

Associated with multiplicity are **role names** and possibly their respective visibility. Figure 6 shows how this is achieved in UML. A student object will require to have exactly one Address object for the relationship of home address.

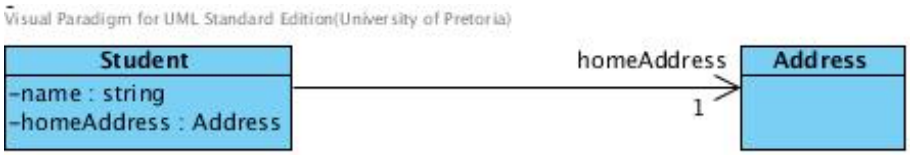

Figure 6: Relationships with role names

Relationships, modelling delegation can be refined into two types of relations namely dependencies and associations. Things such as naming a parameter type and creating an object in a temporary variable imply a dependency. Associations are tighter relationships than what dependencies are. An association is used to represent something like a field in a class. Associations also come in two levels of granularity, namely: aggregation and composition.

Each of these delegation-based relationships will be discussed further in the sections that follow. Code that can serve as examples of how to implement the different relationship types using C++ can be found in Section 2.3.2.

Dependency (*uses-a* relationship) indicates that there is a dependency between the two classes in that one class makes use of the other class. Making use of a class could either be as a parameter to an operation in the class or as a local variable in an operation of the class. This relationship is weak and is shown in UML by an arrow with a dotted line from the class that is using to the class that is being used. In the dependency relationship shown in Figure 7, CrazyPrinter's print operation accepts a pointer to an object of DoubleWrapper as a parameter.

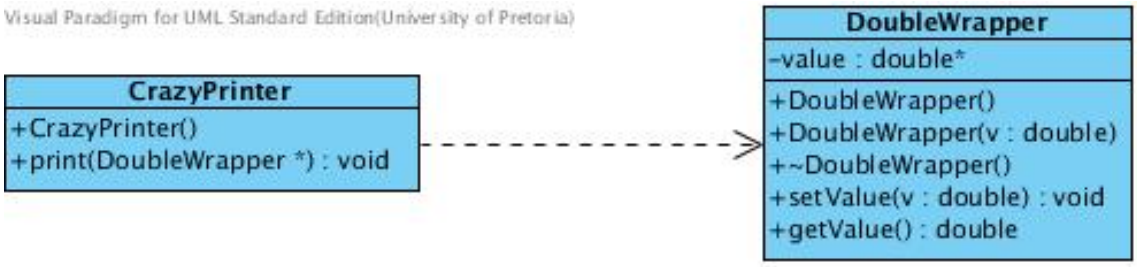

Figure 7: Dependency relationship

Association two types exist:

- **Aggregation** (*has-a* relationship) represents either a "part-whole" or a "part-of" association in which the life dependency of the objects involved are independent of each other. The association is drawn as a solid line with an open diamond on the side of the "whole"-side of the relationship. Figure 8 shows that class APrinter *has-a* association with class DoubleWrapper.

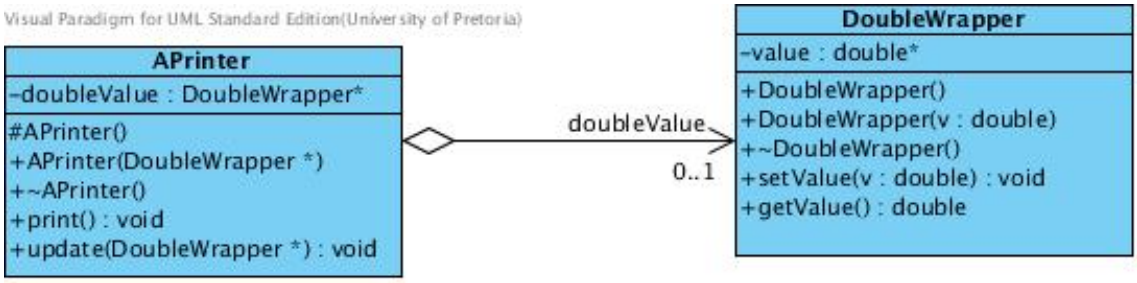

Figure 8: Aggregation association

- **Composition** (*owns-a* relationship) is stronger than aggregation in which the life-times of the objects are the same. It is important to note that the multiplicity of the the "whole" must either be `0..1` or `1`. The multiplicity of the "part" may be anything. The association is shown in Figure 9. When AnotherPrinter goes out of scope, the DoubleWrapper object associated with it will also automatically go out of scope and therefore the tight association between the two classes.

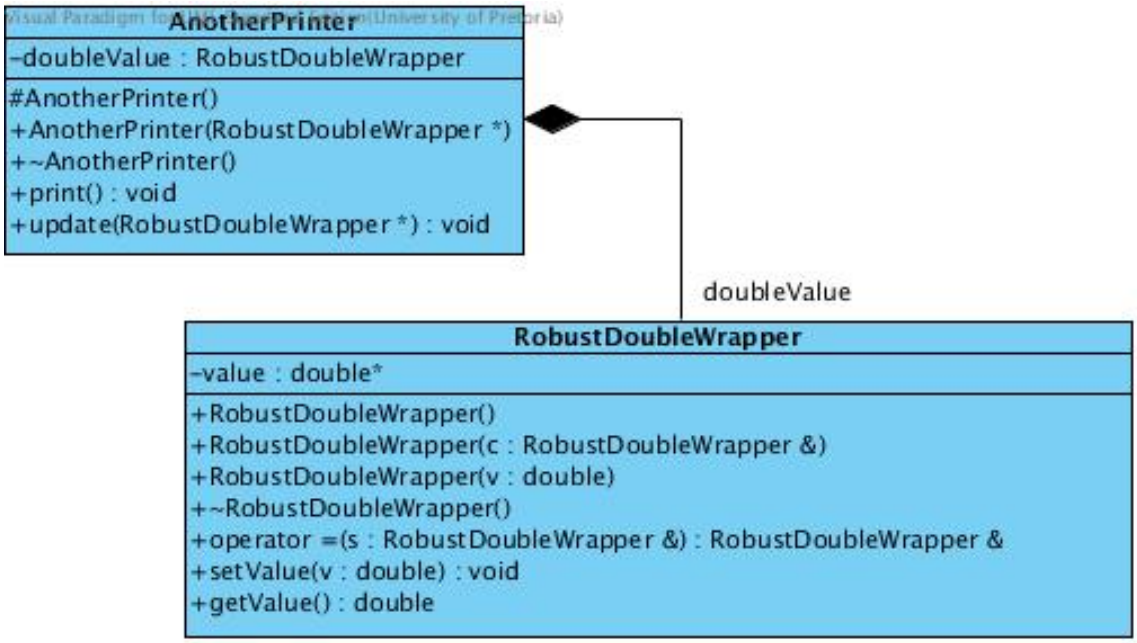

Figure 9: Composition association

2.3 Programming Preliminaries

2.3.1 Attribute defaults

Assigning attribute default values, such as `dateRegistered` in Figure 2, in C++ cannot be done in the class definition, but must be initialised in the constructor. The default constructor therefore should be implemented so that the compiler does not provide an implementation of its own. If possible one should use initialiser lists in the constructor header to assign the initial values to its attributes rather than having an assignment statement in the body of the constructor.

2.3.2 Delegation in C++

For each of the relationships described in Section 2.2.2, the corresponding implementation in C++ will be provided. As can be seen from the figures presented in Figures 7 and 8, `CrazyPriter` and `APrinter` are both associated with the same *DoubleWrapper* class implementation. To illustrate composition, as shown in Figure 9, `AnotherPrinter` is associated with a similar class called `StrongDoubleWrapper`. The reason that the same class could not be used in this example is because the object is created on stack memory and in order for it to be copied successfully (that is, for a deep copy to take place) the assignment operator must be implemented.

The header file and corresponding implementation for the DoubleWrapper is given by:

DoubleWrapper.h

```
#ifndef DoubleWrapper_H
#define DoubleWrapper_H

class DoubleWrapper
{
  public:
    DoubleWrapper();
    DoubleWrapper(double v);
    ~DoubleWrapper();
    void setValue(double v);
    double getValue();
  private:
    double* value;
};
#endif
```

DoubleWrapper.C

```
#include <iostream>
#include "DoubleWrapper.h"
using namespace std;

DoubleWrapper::DoubleWrapper(): value(0) {}

DoubleWrapper::DoubleWrapper(double v)
{
  value = new double(v);
}

void DoubleWrapper::setValue(double v)
{
   if (value != 0)
   {
     delete value;
   }
   value = new double(v);
}

double DoubleWrapper::getValue()
{
  if (value != 0)
  {
    return *value;
  }
  return -1;
}
```

```
DoubleWrapper::~DoubleWrapper()
{
  if (value != 0)
  {
    delete value;
    value = 0;
  }
}
```

Note that `value` can be initialied in the initialiser list when the default constructor is implemented. However this is not possible for the constructor taking an initial value as parameter because the memory dynamic allocation needed to initialise this value can not be done using the initialiser list. The initialiser list can only be used for heap allocation.

Dependency relationship requires the class to use `DoubleWrapper` either as a parameter to a operation or to be defined locally in an operation. In the example given in Figure 7 the relationship is as a parameter to an operation. The code is given in the following header and implementation files for `CrazyPrinter`.

CrazyPrinter.h

```
#ifndef CrazyPrinter_H
#define CrazyPrinter_H

#include "DoubleWrapper.h"

class CrazyPrinter
{
  public:
    CrazyPrinter();
    void print(DoubleWrapper*);
};

#endif
```

CrazyPrinter.C

```
#include <iostream>

#include "CrazyPrinter.h"

CrazyPrinter::CrazyPrinter(){}

void CrazyPrinter::print(DoubleWrapper* val)
{
  std::cout<< val->getValue() << std::endl;
}
```

Aggregation association relationship is a relationship in which the class `APrinter` has a handle to an object of `DoubleWrapper` as can be seen in Figure 8. The handle is a pointer to an instance of the wrapper object in heap memory.

APrinter.h

```
#ifndef APrinter_H
#define APrinter_H

#include "DoubleWrapper.h"

class APrinter
{
    public:
      APrinter (DoubleWrapper*);
      ~APrinter();
      void print();
      void update(DoubleWrapper*);
    protected:
      APrinter();
    private:
      DoubleWrapper* doubleValue;
};
#endif
```

APrinter.C

```
#include <iostream>
#include "APrinter.h"

using namespace std;

APrinter::APrinter(DoubleWrapper* value): doubleValue(value) {}

void APrinter::print()
{
  if (doubleValue != 0)
  {
    cout << doubleValue->getValue() << endl;
  }
  else
  {
    cout << "undefined" << endl;
  }
}

void APrinter::update(DoubleWrapper* doubleValue)
{
  this->doubleValue = doubleValue;
}
```

```
APrinter::~APrinter()
{
  doubleValue = 0;
}
```

Composition association relationship requires the life-times of the two objects to be closely dependent on each other. In order to achieve this in C++ it must either be coded in the destructor of the class that acts as the owner, or stack memory can be used to enforce the requirement of ownership. The UML class diagram in Figure 9 makes use of the stack to enforce ownership. Therefore the implementation of the class `DoubleWrapper` needs to include an implementation for at least the assignment operator to successfully implement deep copies when objects of `DoubleWrapper` as assigned to each other within `AnotherPrinter`.

RobustDoubleWrapper.h

```
#ifndef RobustDoubleWrapper_H
#define RobustDoubleWrapper_H

class RobustDoubleWrapper
{
  public:
    RobustDoubleWrapper();
    // copy constructor added
    RobustDoubleWrapper(const RobustDoubleWrapper&);
    RobustDoubleWrapper(double);
    ~RobustDoubleWrapper();
    // assignment operator added
    RobustDoubleWrapper& operator = ( const RobustDoubleWrapper&);
    void setValue(double);
    double getValue();
  private:
    double* value;
};
#endif
```

RobustDoubleWrapper.C

```
#include <iostream>
#include "RobustDoubleWrapper.h"

using namespace std;

RobustDoubleWrapper::RobustDoubleWrapper(): value(0){}
```

```
RobustDoubleWrapper::
    RobustDoubleWrapper(const RobustDoubleWrapper& c)
{
  if (value != 0)
  {
    delete value;
  }
  value = new double(*(c.value));
}

RobustDoubleWrapper::RobustDoubleWrapper(double v)
{
  value = new double(v);
}

RobustDoubleWrapper::~RobustDoubleWrapper()
{
  if (value != 0)
  {
    delete value;
    value = 0;
  }
}


RobustDoubleWrapper& RobustDoubleWrapper::
    operator = ( const RobustDoubleWrapper& s )
{
  if (value != 0)
  {
    delete value;
  }
  value = new double(*(s.value));
  return *this;
}


void RobustDoubleWrapper::setValue(double v)
{
    if (value != 0)
    {
      delete value;
    }
    value = new double(v);
}
```

```
double RobustDoubleWrapper::getValue()
{
  if (value != 0)
  {
    return *value;
  }
  return -1;
}
```

AnotherPrinter.h

```
#ifndef AnotherPrinter_H
#define AnotherPrinter_H

#include "RobustDoubleWrapper.h"

class AnotherPrinter
{
   public:
     AnotherPrinter (RobustDoubleWrapper* doubleValue);
     ~AnotherPrinter();
     void print();
     void update(RobustDoubleWrapper* doubleValue);
   protected:
     AnotherPrinter();
   private:
     RobustDoubleWrapper doubleValue;
};
#endif
```

AnotherPrinter.C

```
/*
 Note: the commented code describe the changes
 required compared to the class APrinter
*/

#include <iostream>

#include "AnotherPrinter.h"

using namespace std;

AnotherPrinter::AnotherPrinter(RobustDoubleWrapper* value)
{
  // cannot use initialiser list owing to dereferncing needed.
  doubleValue = *value;
}
```

```
void AnotherPrinter::print()
{
    // no condition needed -- doubleValue is on the stack
    // also note the use of . instead of -> to call the function
    cout << doubleValue.getValue() << endl;
}

void AnotherPrinter::update(DoubleWrapper* value)
{
  // dereference value before assigning
  doubleValue = *value;
}

AnotherPrinter::~AnotherPrinter() {}
  // doubleValue is on the stack and is automatically released.
```

2.3.3 Variable sized interfaces

In the memento pattern there is a need to create a class which has a narrow interface with one class while having a wider interface with another class. By default all classes that interface with a given class will share the same size interface. If a class is known to the class that has to provide variable sized interfaces, its interface can be widened in C++ by defining the known as a private friend class of the class allowing it to access members that are not public.

Friends in C++

In order for another class to be able to access features in a given class that are not public, the class must be assigned "friend"-status. Friend status can be protected or private. The visibility of the "friend"-status specifies to which category of members of the class the friend will be granted access. Refer to the example in Section 2.6 to see how the Originator is given private friend status of the Memento class to widen the interface of the Memento class with the Originator class while maintaining a narrow interface of the Memento class with the Caretaker class.

2.4 Memento Pattern

2.4.1 Identification

Name	**Classification**	**Strategy**
Memento	Behavioural	Delegation (Object)
Intent		
"Without violating encapsulation, capture and externalise an object's internal state so that the object can be restored to this state later." ([2]:283)		

2.4.2 Structure

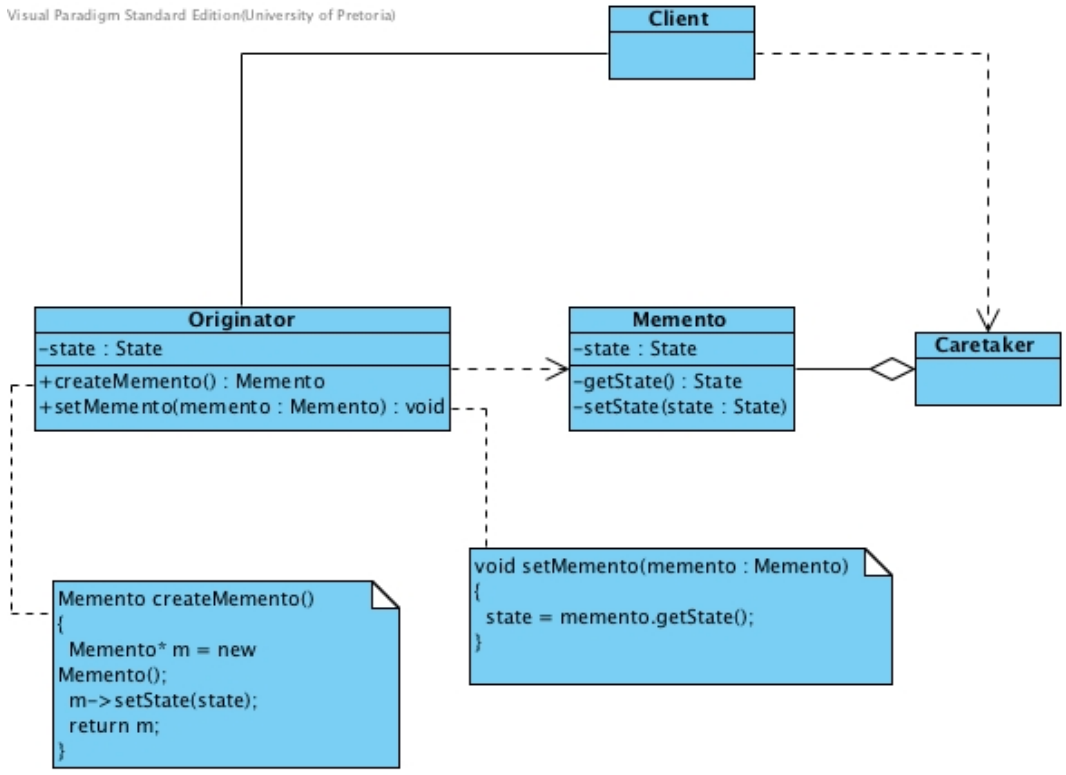

Figure 10: The structure of the Memento Pattern

2.4.3 Problem

The memento pattern enables an object to be restored to its previous state. Memento can be seen as a snapshot of the system at a particular point in time.

2.4.4 Participants

Originator

- an object with an internal state which it creates a snapshot of to store
- the snapshot is used to restore the state

Memento

- takes a snapshot of as much state as required by the originator
- only allows the originator access to the state

Caretaker

- keeps the memento safe
- is unaware of the structure of the memento

The two main participants on the pattern are the Originator and the Caretaker. A wide interface exists between the Originator and the Memento, while a narrow interface exists between the Caretaker and the Memento.

2.5 Memento Pattern Explained

2.5.1 Clarification

The memento pattern is handy when there is the need to keep information in tact for use at a later stage. The pattern is only useful when the time taken to store and later restore the state does not impact heavily on the functionality and performance of the system being developed.

2.5.2 Implementation Issues

A problem that may occur when using the memento is that the internal state of the originator object may be inadvertently be exposed.

2.5.3 Related Patterns

Command
: Command can make use of mementos to maintain the state of commands, in the order they were issued, in order to support undoable operations.

Iterator
: Mementos can be used for iteration to maintain the state of the iterator.

Bridge
: The bridge pattern can be applied in order to separate the interface from the implementation of the memento in order to provide the wide interface between the originator and the memento without using the friend technique which violates object-oriented encapsulation.

2.6 Example

Consider a calculator application for complex numbers. As all calculators have the functionality to store and recall number from memory so must this implementation of the complex calculator. The code presented shows the essence of such an implementation and can be extended in order to provide all operations and required calculator functionality. The UML class diagram for the application is given in Figure 11. Interesting aspects of the implementation, and in particular the implementation for the friend relationship is shown here.

In this example we give inline implementations of all definitions and assume that all code is written in a single .cpp file. When doing this the order in which the classes are presented is important. One class cannot use another class before it is declared. In

this case ComplexNumber requires StoredComplexNumber to be defined before itself and vice versa. In such a case one determine which of the two requires the most detail of the other to be exposed and declare the that requires the least information of the other first. As can be seen from the implementation of the class , it does not need much detail regarding the `ComplexNumber` class. It basically needs to know that it exists. Therefore class `StoredComplexNumber` can be defined before class `ComplexNumber` as long as the line `class ComplexNumber;` (a forward declaration similar to a function prototype) is inserted just before where class `StoredComplexNumber` is defined.

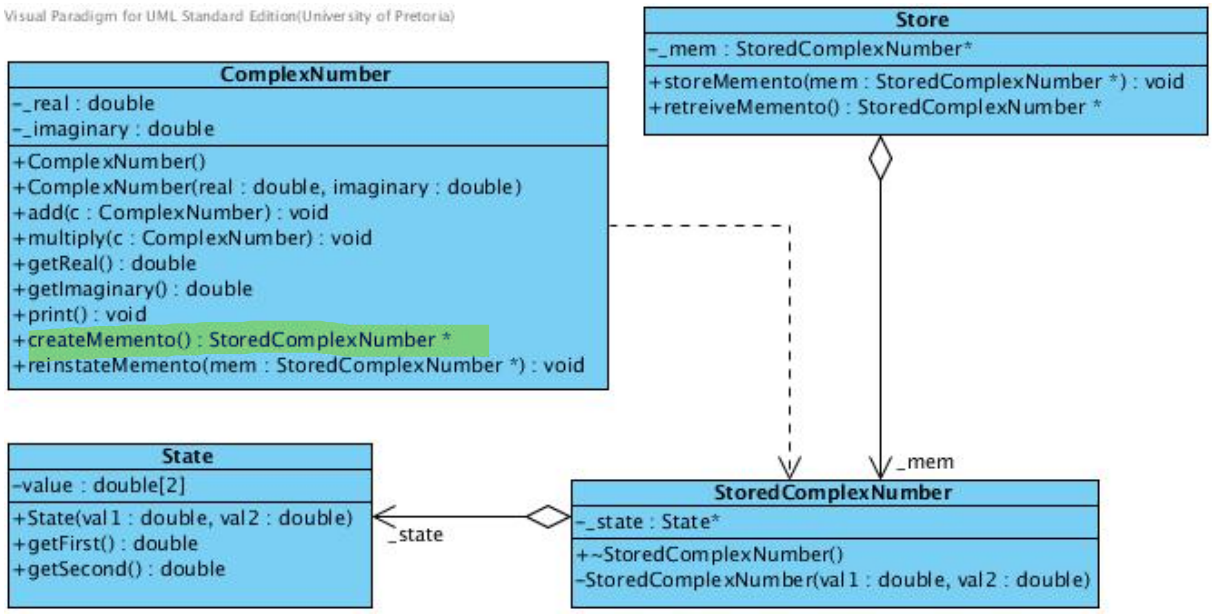

Figure 11: Class diagram for an implementation of a complex number calculator that illustrates the Memento design pattern.

Here is the definition with inline implementation of the StoredComplexNumber class. Note how the constructor is declared private to prevent other classes to be able to create objects of this kind. By giving the class ComplexNumber friend status it is granted access to the private variables and methods of the class. Therefore it will be able to create objects of this kind and also access its private variables. Also note that an initialiser list can not be applied in this constructor because memory for the `_state` variable is to be allocated on the heap.

```
class StoredComplexNumber
{
  public:
    virtual ~StoredComplexNumber()
    {
        delete _state;
    }
  private:
    friend class ComplexNumber;

    StoredComplexNumber(double val1, double val2)
    {
```

```
        _state = new State(val1,val2);
    }
    State* _state;
};
```

ComplexNumber excerpts

```
ComplexNumber :: ComplexNumber() : _real(0), _imaginary(0) {}

ComplexNumber :: ComplexNumber(double real, double imaginary)
        : _real(real), _imaginary(imaginary) {}

StoredComplexNumber* ComplexNumber::createMemento()
{
  return new StoredComplexNumber(_real,_imaginary);
}

void ComplexNumber::reinstateMemento(StoredComplexNumber* mem)
{
  State* s = mem->_state;
  _real = s->getFirst();
  _imaginary = s->getSecond();
}
```

Note how the instance variables of this class are initiated through the use of initialiser lists because they are allocated on the heap.

Implementation of the Caretaker

```
class Store
{
  public:
    void storeMemento(StoredComplexNumber* mem)
    {
        _mem = mem;
    };

    StoredComplexNumber* retreiveMemento()
    {
        return _mem;
    };

    ~Store()
    {
        delete _mem;
    };
  private:
    StoredComplexNumber* _mem;
};
```

The design includes the implementation of the `State` class. This class encapsulates all the instance variables of the `ComplexNumber` class. It is used by the `StoredComplexNumber`

class. While it is not required that the complete state of the originator be stored in a single object, it is recommended because it may simplify the combination of the memento pattern with other patterns.

2.7 Exercises

1. Change the example given for composition so that the object in `AnotherPrinter` is not on the stack but on the heap without changing the relationship to aggregation as in the `APrinter` example.

2. Extend the example in 2.6 to include more calculator-like functionality.

References

[1] Simon Bennett, John Skelton, and Ken Lunn. *Schaum's Outline of UML*. McGraw-Hill Professional, UK, 2001. ISBN 0077096738.

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