Contents

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 1: Introduction to UML and DP

Contents

1.1 Introduction

This lecture discusses the Unified Modelling Language (UML). It is a visual modelling language used for software modelling and design in the object oriented paradigm of software development. After providing a brief history of the language, an overview of the components of UML 2 is provided.

Software design methods have strived to provide a way to model "good" design in software. The notion and background to Design Patterns is given, which provides a foundation for design.

1.2 Unified Modelling Language (UML)

Prior to 1994, many people were busy developing modelling techniques and tools focussing on different aspects of object-orientation. In 1991, Rumbaugh *et al*, proposed the Object-modelling technique (OMT) which focussed on Object-oriented analysis (OOA). Grady Booch, while working at Rationale in the early 1990's developed the Booch method which focussed on both OOA and Object-oriented design (OOD). In 1992, Ivar Jacobson developed Object-oriented Software Engineering (OOSE). Rationale bought out the company for which Jacobson worked.

In 1995 Rumbaugh joined Rationale and started work on a modelling language along with Booch and Jacobson. The trio are fondly referred to as the *"Three Amigos"*. The modelling language they created unifies the respective languages they have independently created earlier. They named this language UML. UML 1.x was released in 1996 and included class diagrams, object modelling visualisations and use cases. In 2005, UML 1.4.2 was adopted as an ISO standard. 2005 also saw the Object Modelling Group (OMG)[1] adopting UML 2.x and taking responsibility for further developing UML [4].

1.2.1 UML 2.x

From the beginnings in UML 1 of supporting 3 diagram types, UML 2 supports 14. Diagrams are divided into two broad categories, structural and behavioural. Figure 1 shows the diagram types supported by UML 2.

Structural diagrams depict what must be present in the system, being modelled. It shows a static view of the software being modelled. There are five types of structural diagrams namely class, object, package, component, and deployment diagrams.

Behavioural diagrams depict what the system being modelled must do. It models dynamic aspects of the software over time. There are four types of behavioural diagrams namely use case, state, activity and interaction diagrams. Interaction diagrams are further subclassed into communication, sequence, timing and interaction overview diagrams.

The diagram types highlighted in blue are relevant to this series of Lecture Notes and will be explained in more detail when required.

[1] `http://www.omg.org/gettingstarted/gettingstartedindex.htm`

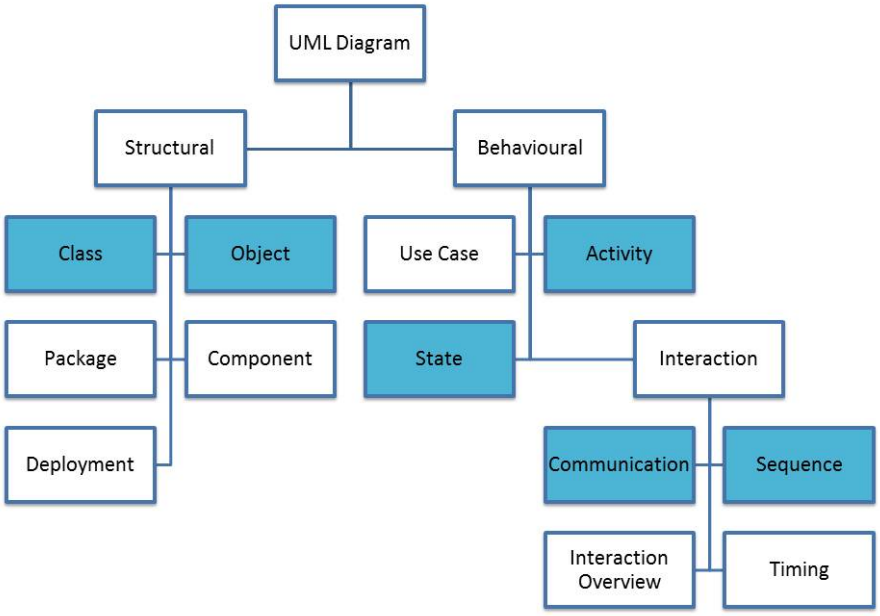


Figure 1: UML diagrams

1.2.2 Class diagrams

A class diagram show the classes that an object-oriented system comprises of as well as the relationships between these classes. The internal structure of the class is described in terms of attributes and operations. The relationships are of two types, those that are related to the structure of the class and those that show the messages to be passed between the classes [2]. An example of a class diagram is provided in figure 2.

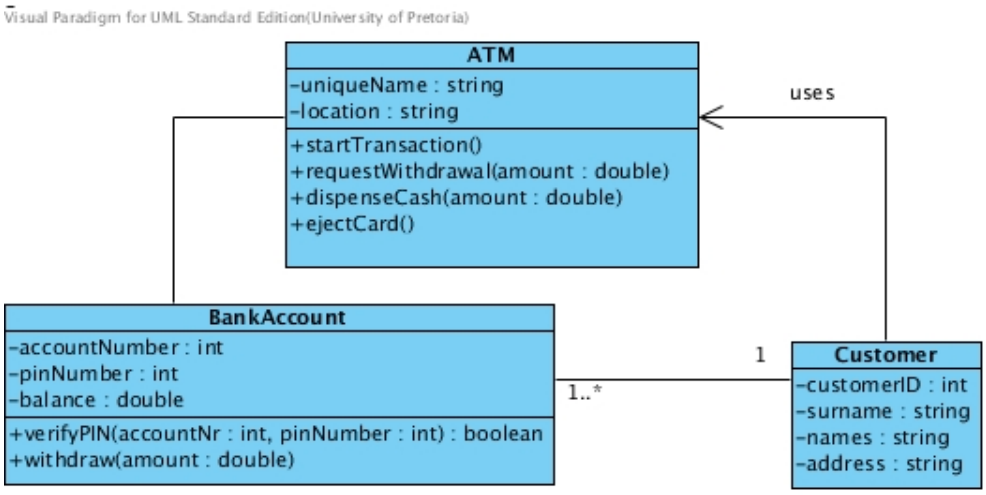


Figure 2: An example of a class diagram

1.2.3 Object diagrams

Object diagrams are a special type of class diagram. An object diagram depicts the state of a system at a particular point in time. Object diagrams preserve the relationships between objects and show the current "state" or values of the attributes of the particular instances of the classes at a specific point in time. Figure 3 shows the state of the objects in the class diagram that was given in Figure 2 at a specific point in time while a user called John Doe was doing a transaction on his savings account at a specific ATM in the Hatfield Plaza.

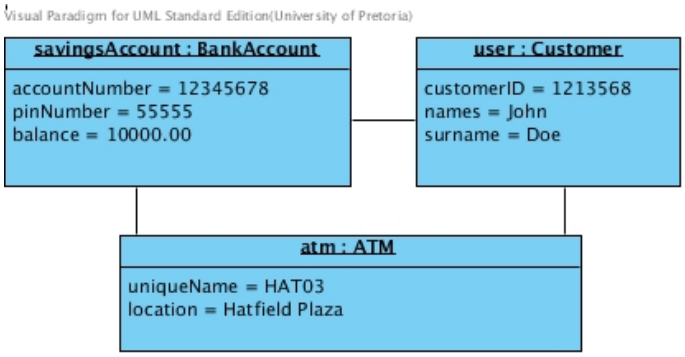


Figure 3: An example of an object diagram

1.2.4 State diagrams

State diagrams illustrate process in terms of state changes. A state diagram models dynamic behaviour of objects. It shows the changes in the state of an object. For example the states of an ATM can be 'Waiting', 'Connecting', 'Active', etc. Figure 4 shows that insertion of a card triggers the ATM to change from 'Waiting' to 'Connecting'. If successful it will change to the 'Active' state during which the user can perform various transactions, otherwise it will eject the card and return to the 'Waiting' state. When the user indicates that he/she wishes not to perform any more transactions, the ATM will eject the card and return to the 'Waiting' state.

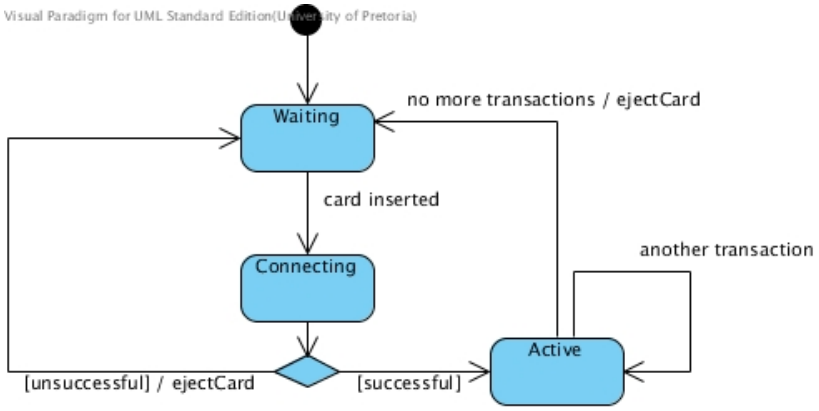


Figure 4: An example of a state diagram

1.2.5 Activity diagrams

Activity diagrams illustrate process in terms of activities. An activity diagram is a kind of flow chart which shows the workflow behaviour of an operation as a sequence of actions. For example the activities while an ATM withdrawal is processed can be 'insert card', 'enter PIN', 'check balance', 'eject money', etc. Figure 5 models the activities of a cash withdrawal process.

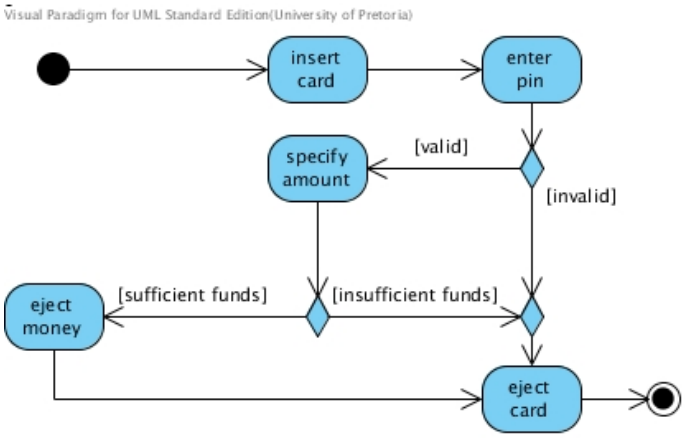


Figure 5: An example of an activity diagram

1.2.6 Sequence diagrams

A sequence diagram is a type of interaction diagrams. Interaction diagrams model the interaction between objects, also referred to the messages passed between objects. Compared to the sequence diagrams in UML 1.x, the expressive power of sequence diagrams has been increased in UML 2.x. Sequence diagrams show interaction between objects. The order in which the messages are passed is visualised in the order of the communication lines in the diagram while all objects participating in the interaction are placed at the top of the diagram. Figure 6 shows some interactions between the objects shown in Figure 3 for a cash withdrawal transaction.

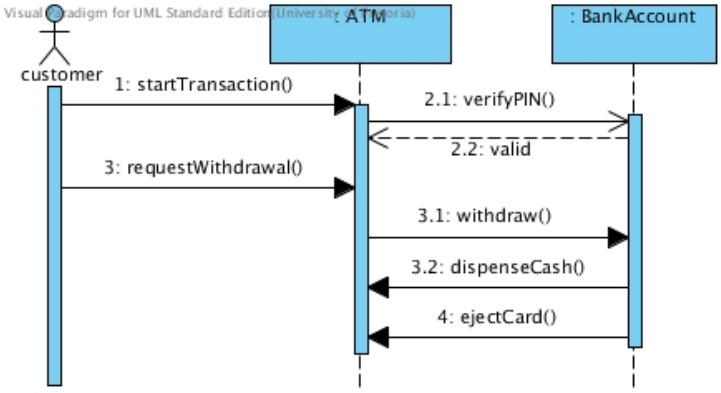


Figure 6: An example of a sequence diagram

1.2.7 Communication diagrams

A communication diagram, like a sequence diagram, is a type of interaction diagram. Communication digrams was referred to as collaboration diagrams in UML 1.x.

The main distinction between sequence and communication diagrams is that sequence diagrams show interaction between objects over time while communication diagrams show the depth of the interaction between objects.

In communication diagrams the objects participating in the interaction are placed in proximity with one another to better visualise which objects are communicating directly with one another irrespective of the order in which the messages are passed. Figure 7 shows the same interaction that is shown in Figure 6.

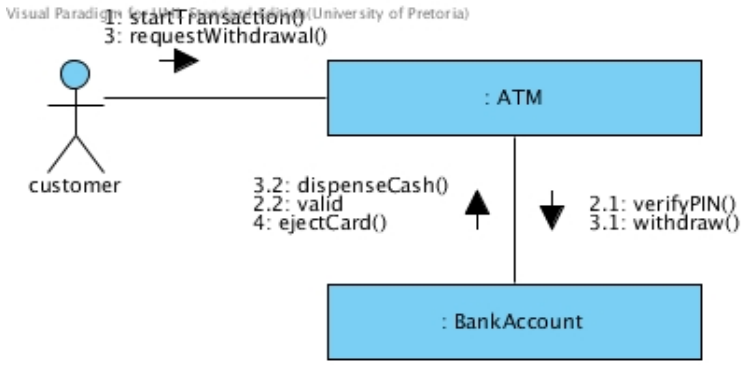


Figure 7: An example of a communication diagram

1.2.8 Internet resources

The following internet resources may be helpful and provide a very good overview of UML.

- Wikipedia:URL: `en.wikipedia.org/wiki/Unified_Modeling_Language`
- OMG: URL:`www.uml.org`, refer to the document that describes the superstructure of UML
- Sparx System: URL:`www.sparxsystems.com/resources/tutorial`, presents a good overview of UML in the form of a tutorlal
- Tutorials Point: URL:`http://www.tutorialspoint.com/uml/uml_overview.htm`, another good tutorial on all the UML diagrams.

1.3 Design Patterns

Patterns originated as an architectural concept when designing buildings. Christopher Alexander introduced this concept in his 1977 book on architecture, urban design, and community livability [1]. He proposed the following definition for a pattern.

> *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*

Design patterns in software design are no different. In software systems there are problems that occur over and over again for which design patterns can be identified to provide solutions to the problems. The idea of applying patterns to software was formalised by Kent Beck, of the Agile movement, and Ward Cunnigham and presented at OOPSLA in 1987.

The idea of software design patterns was originally not well received. It only started to gain popularity after 1994 when Gamma *et al* published their book of design patterns that describes simple and elegant solutions to specific problems in object-oriented software design [3].

The authors of the book is affectionately referred to as the "*Gang of Four*" or GoF. In the preface of the book, the GoF, states that the book neither introduces object-oriented programming and design, nor is it an advanced reference for object-oriented programming. They state that the book:

> *... describes simple and elegant solutions to specific problems in object-oriented software design. Design patterns capture solutions that have developed and evolved over time.*

The 23 classical design patterns, discussed by GoF, are categorised according to their purpose into Creational, Behavioural and Structural Patterns. A further level of categorisation applied has to do with the relationships between the classes. The object-oriented concept of delegation is classified as "object" while patterns with a predominantly inheritance relationship structure are referred to as "class" patterns.

When writing software it is important to address the quality attributes of the solution which includes user friendliness, effectiveness and efficiency. All patterns adresses one or more software quality requirements. Some patterns addresses specific quality requirements. For example Singleton and Flyweight addresses efficiency of memory usage while Prototype is aimed at reducing execution time.

Despite the fact that most patterns **increases** the execution time owing to indirection created through excessive use of inheritance and delegation, invariably adaptability and maintainability of the code is greatly enhanced. Software applying design patterns is usually more robust and reliable owing to the use of tried and tested techniques. Design patterns are very useful abstraction tools that software engineers have at their disposal moving design decisions to a higher level of abstraction. Programmers who are competent in using design patterns are likely to be more effective in their work.

1.3.1 Internet Resources

The following internet resources may be helpful and provide a very good overview of the classic design patterns.

- Wikipedia:
  - URL: `http://en.wikipedia.org/wiki/Software_design_pattern`, for an overview of design patterns
  - URL: `http://en.wikipedia.org/wiki/Design_Patterns`, for an overview of GoF [3]
- Huston: URL: `http://www.vincehuston.org/dp/`
- OODesign: URL: `http://www.oodesign.com`

References

[1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Cess Center for Environmental)*. Oxford University Press, later printing edition, August 1977. ISBN 0195019199.

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

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

[4] Wikipedia. Unified modeling language — wikipedia, the free encyclopedia, 2012. URL `https://en.wikipedia.org/wiki/Unified_Modeling_Language`. [Online; accessed 28-July-2012].

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].

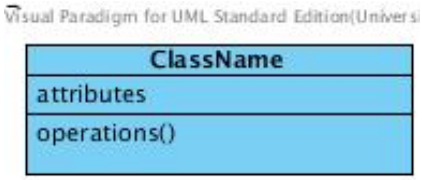


Figure 1: Structure of a UML Class

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.

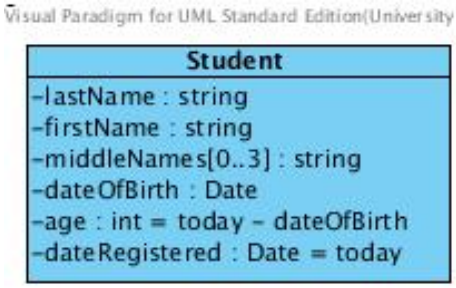


Figure 2: UML class attributes

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`.

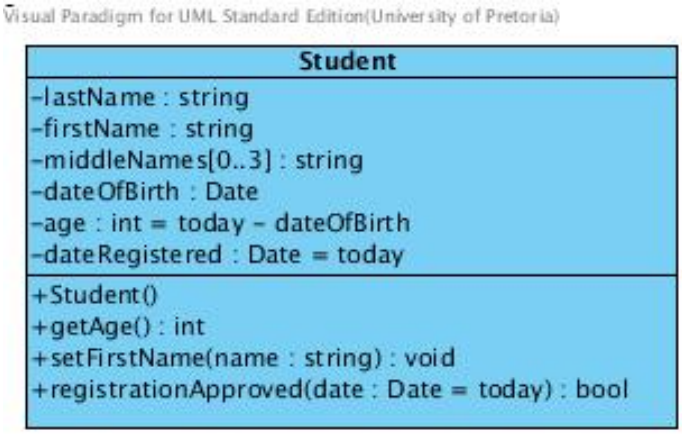


Figure 3: UML class operations

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.

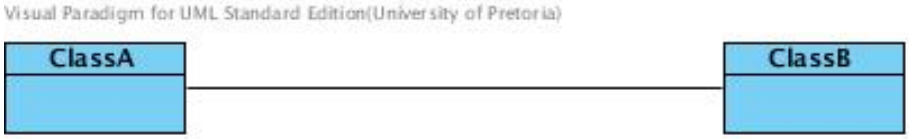


Figure 4: Binary relationship

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.

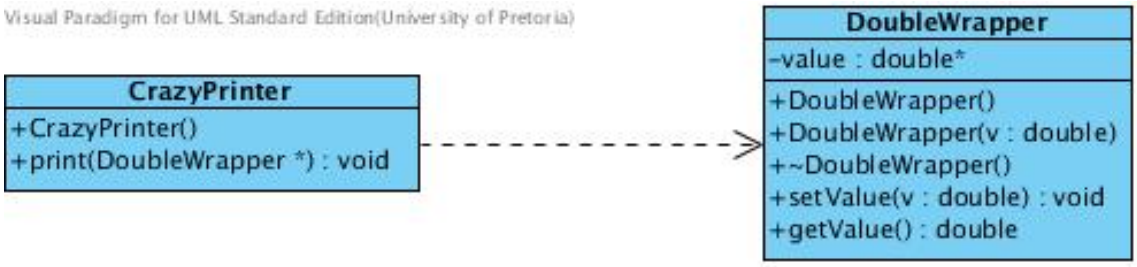


Figure 7: Dependency relationship

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.

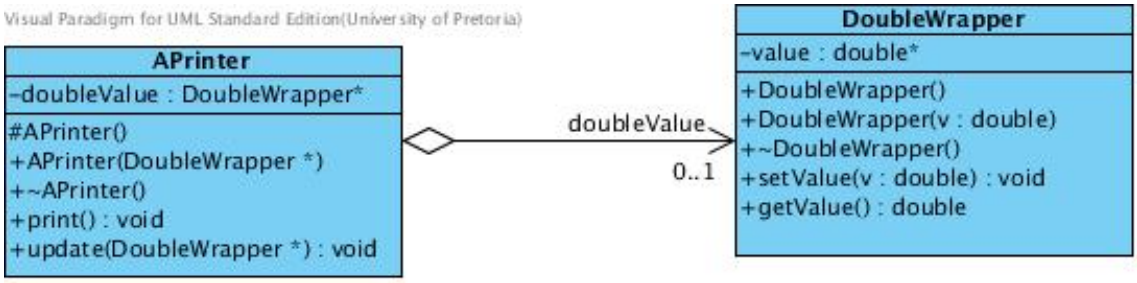


Figure 8: Aggregation association

- **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.

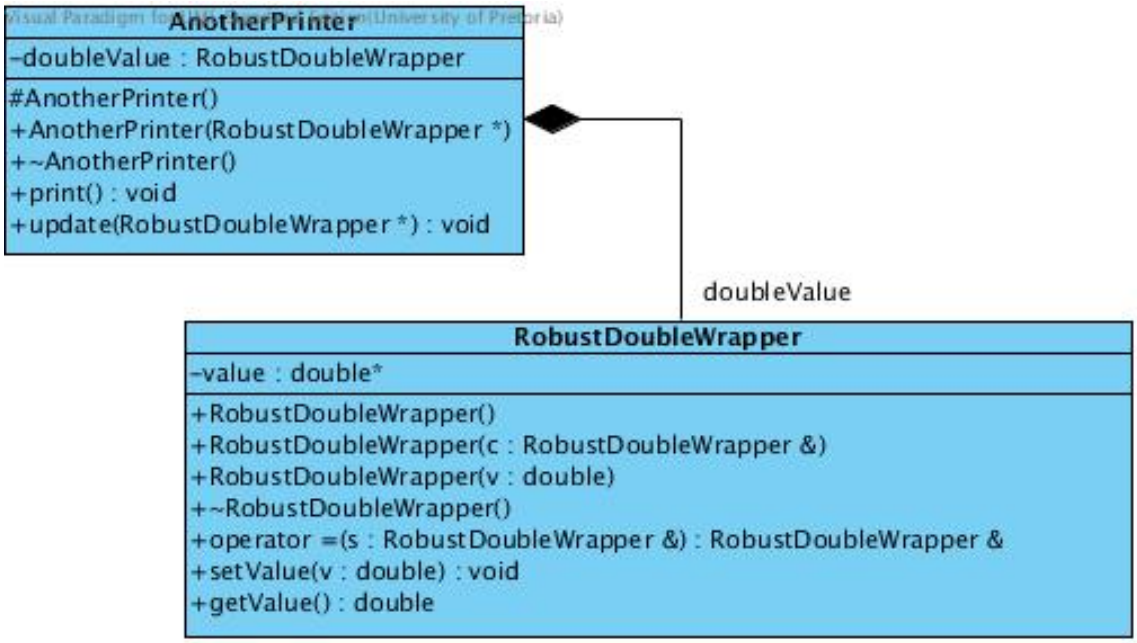


Figure 9: Composition association

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;
}
```

difference from dependency

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
```

has object of the other object

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

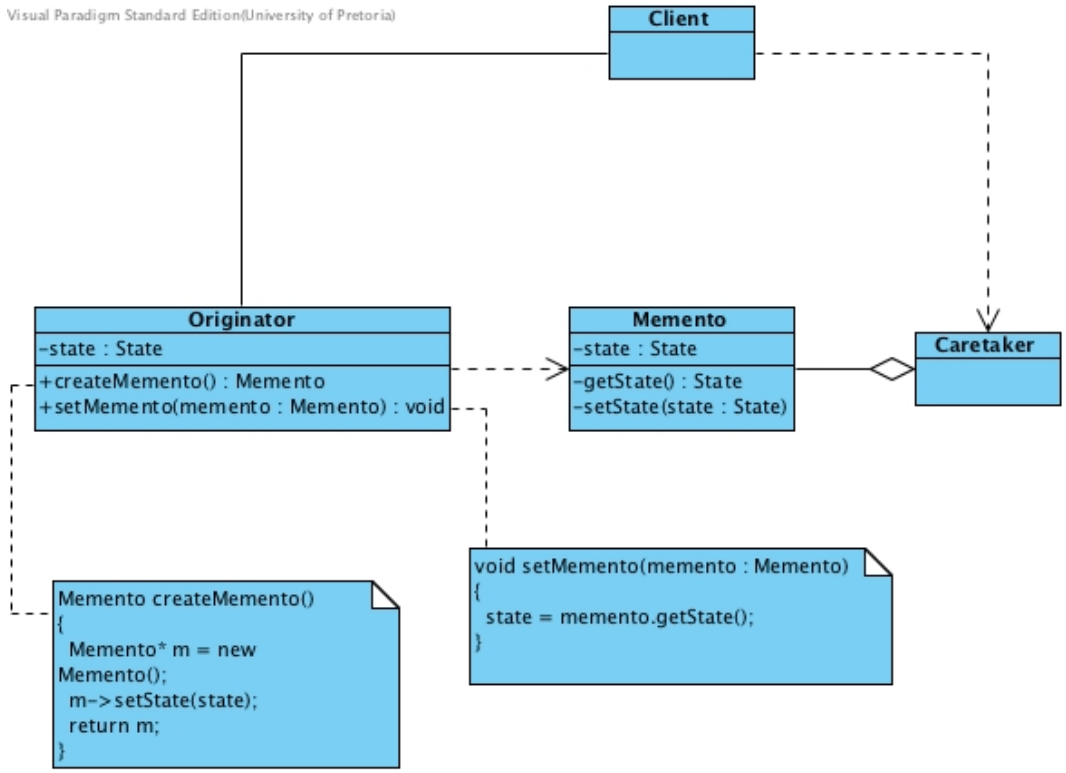


Figure 10: The structure of the Memento Pattern

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.

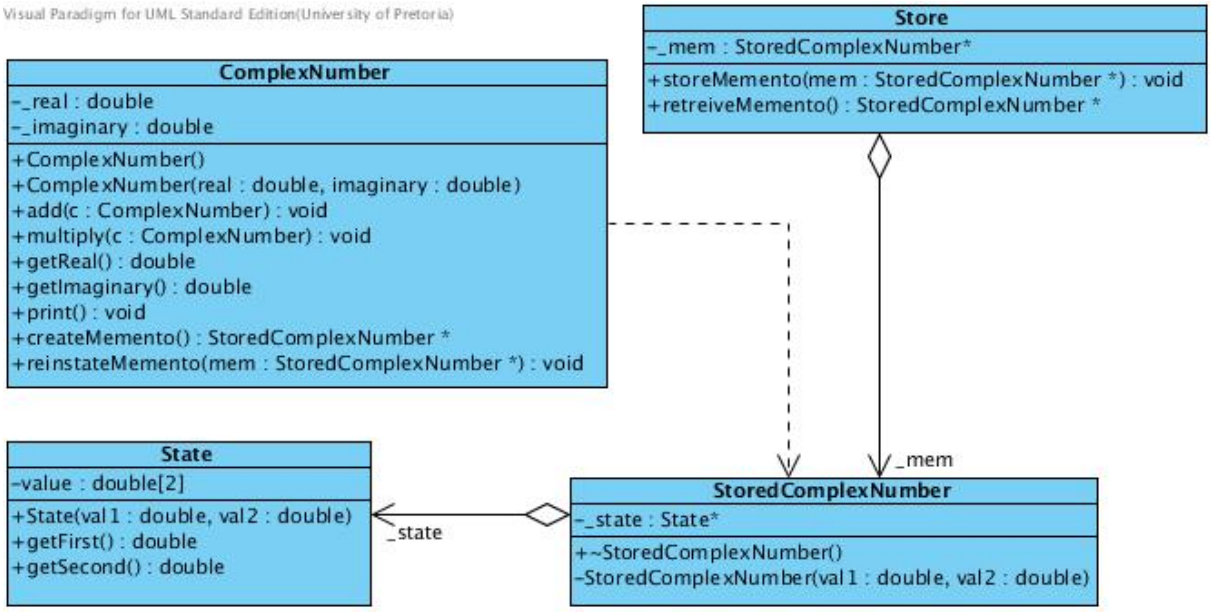


Figure 11: Class diagram for an implementation of a complex number calculator that illustrates the Memento design pattern.

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.

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 (redefined)
: The implementation of these methods are provided in the class that defines them.

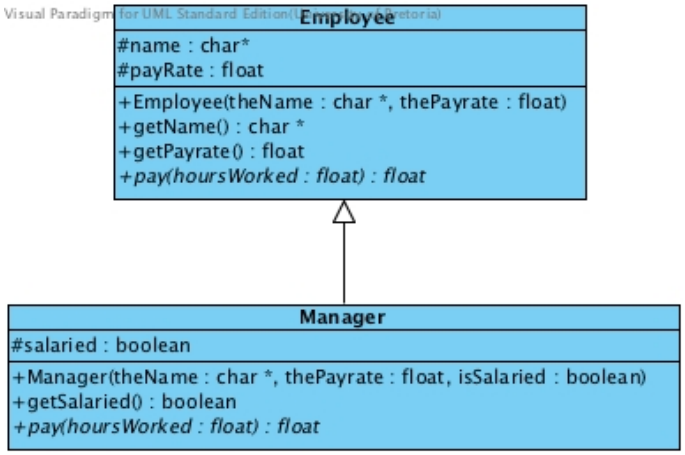


Figure 1: Class diagram showing inheritance

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 (primitive)

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].

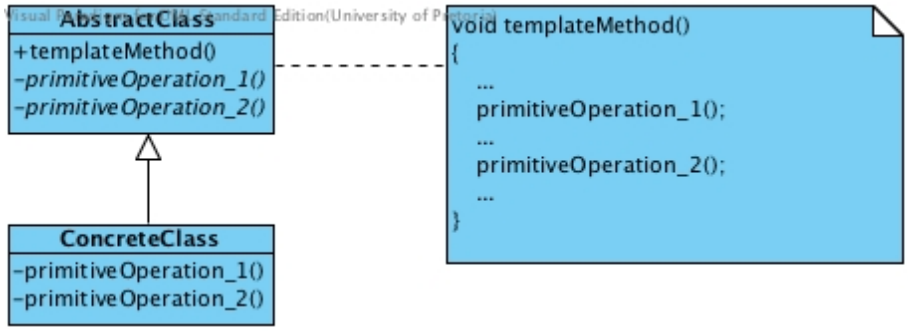


Figure 2: The structure of the Template Method pattern

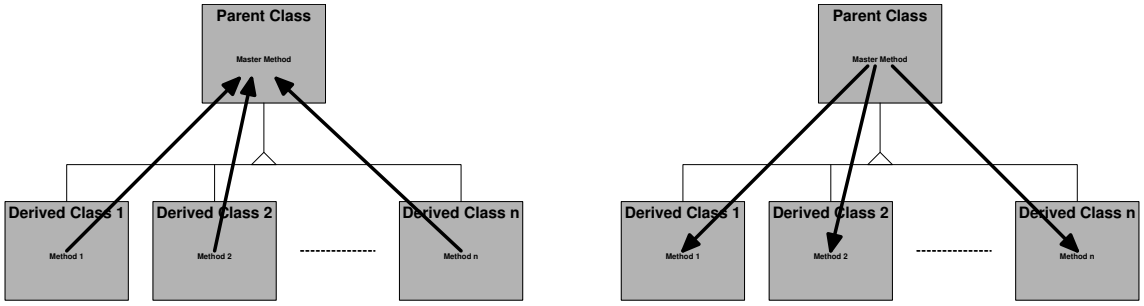


Figure 3: The Hollywood principle

3.3.4 Participants

pure virtual

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

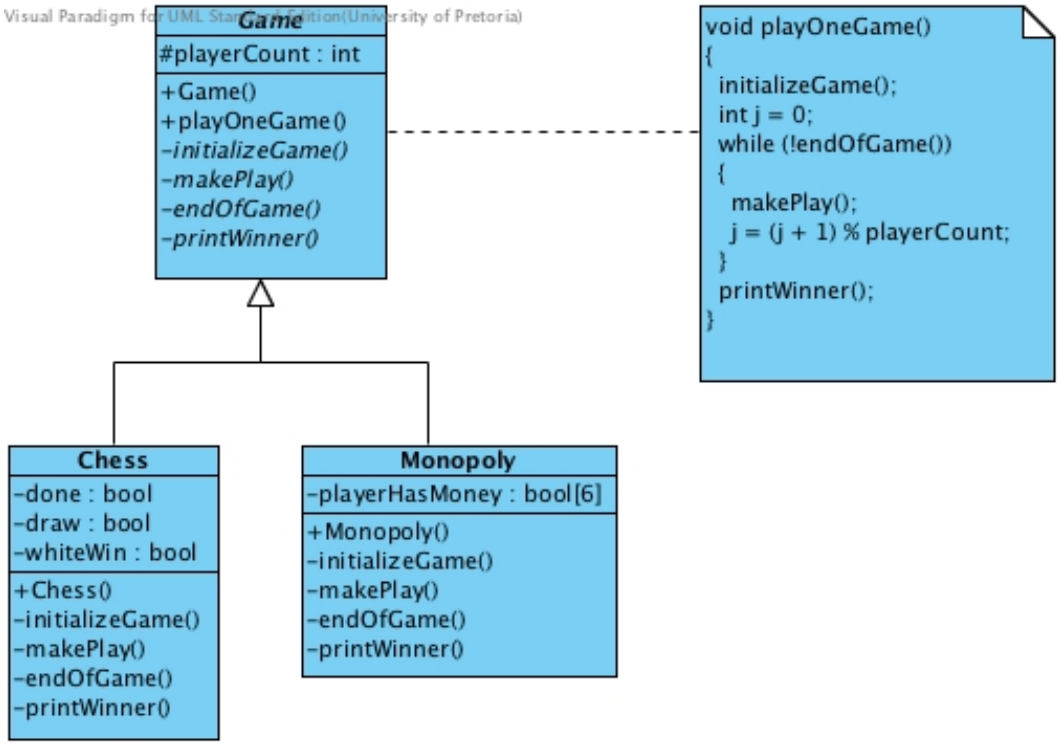


Figure 4: Class diagram of a Game c4ass implementing a Template Method

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;
```

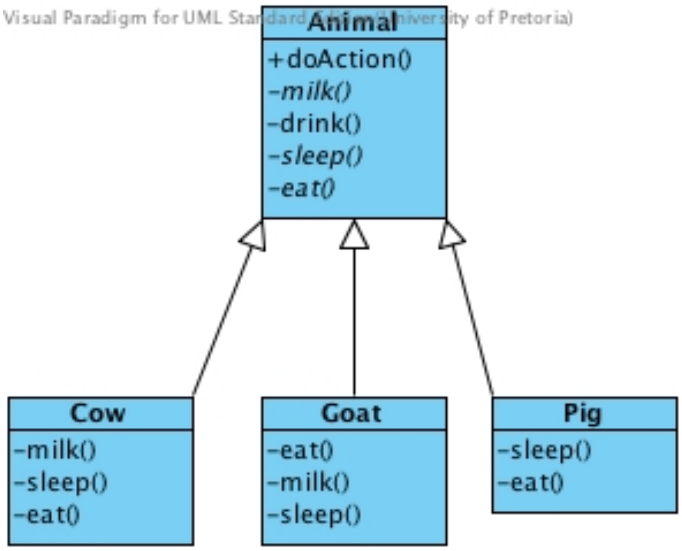


Figure 5: Animal class implementing a template method

```
        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].

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 4: Factory Method design pattern

Contents

4.1 Introduction

This chapter will introduce the Factory Method design pattern. The pattern provides a structure whereby the creation of objects is delegated to subclasses in such doing not needing to specify the class that the object belongs to.

4.2 Programming preliminaries

When an object is instantiated (created), a constructor is called. If a constructor, with the specific parameter list, is not defined for the class the default constructor is called. When an object goes out of scope (deleted, destroyed), the destructor is called. The behaviour exhibited by the constructor and destructor is to handle memory and attributes. The constructor allocates memory for attributes and assigns values to attributes, while the destructor must release any memory that has been assigned during the life-time of the object [1].

In Lecture Notes L04 - Template Method Pattern, constructors were used. The lecture note showed how to make use of member-list initialisation in order to assign values to the attributes of the class. It also illustrated the use of the member-list initialisation technique to call the constructor of the base class in an inheritance relationship. In this section, the concepts of object construction, destruction and initialisation will be explained in more detail.

4.2.1 Constructors

The constructor is a member function defined in the class. The name of the constructor is the same as that of the class. The constructor can take parameters, but does not have a return type. A constructor that does not take any parameters is called the default constructor. If a constructor, or the default constructor, has not been defined for the class, the compiler will automatically generate a default constructor so that objects of the class can be created.

Constructors are used to initialise class member variables (attributes) and other setup-type requirements for the object. Initialising of member variables can be done either in the body of the class or in the member-list initialisation of the constructor. The choice between body or member-list is simple, variables that do not require memory to be allocated on the heap can be initialised using the member-list, otherwise they should be initialised in the body. Superclass constructors must be called in the member-list in order for any superclass member variables to be initialised in a controlled manner.

4.2.2 Destructors

The job of a destructor is to release any memory that the object might have acquired during its lifetime. As with constructors, if a destructor has not been explicitly defined, the compiler will define a default destructor for the class. Unlike constructors, only one destructor is needed per class.

The name of the destructor is the same as the class and it takes no parameters. To distinguish the destructor from the default constructor, a destructor is define with a tilde (∼) before its member function name.

4.2.3 An example

Let us revisit the Employee example given in L04 - Template Method Pattern and apply the understanding we have gained with regards to constructors and destructors.

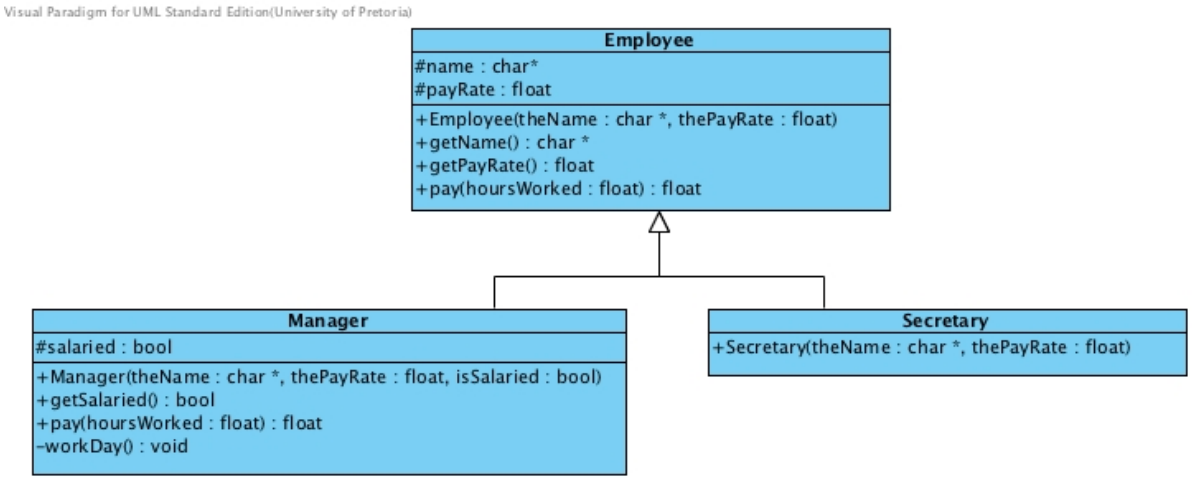


Figure 1: Employee class diagram as presented in L04

The constructor parameter that is going to give the most problems in `theName` in `Employee`. `theName` would have been allocated on the heap by the client. Merely assigning `theName` to `name` will result in `name` pointing to the same memory location as what `theName` does. Any changes made to the memory being pointed to will result in both variables, the one in the client and the one in the `Employee` hierarchy, changing value. In order to make sure that this does not occur, the class attribute `name` needs independent memory allocated to it. The following implementation of the constructor does just this.

```
Employee :: Employee(char* theName,
                     float thePayRate) : payRate(thePayRate)
{
    name = new char[strlen(theName)+1];
    strcpy(name, theName);
}
```

Having allocated the memory in the class, the class needs to take responsibility to delete the memory when it goes out of scope. It is therefore necessary to define the default constructor in the class and provide the implementation for it. The class definition will include the following as a public member:

```
~Employee();
```

The implementation of the destructor follows:

```
Employee::~Employee(){
    delete [] name;
}
```

4.3 Factory Method Pattern

4.3.1 Identification

Name	Classification	Strategy
Factory Method	Creational	Inheritance (Class)
Intent		
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. ([2]:107)		

4.3.2 Structure

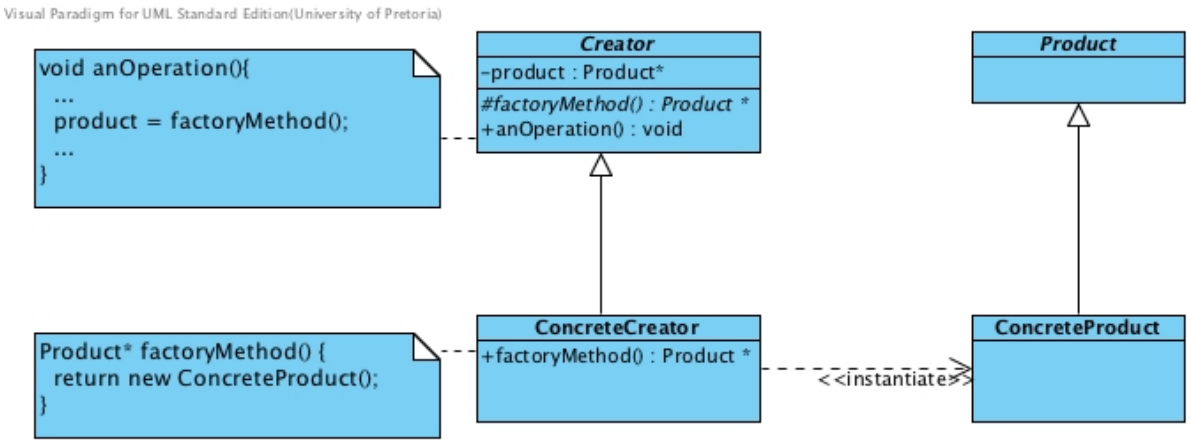


Figure 2: The structure of the Factory Method Pattern

4.3.3 Problem

The factory method essentially wraps the class construction into an operation with a descriptive name and requires the concrete creator to make the decision as to which product constructor is going to be called consequently resulting in the creation of the product. The pattern solves the problem of having a tight coupling between objects that create product and the product objects themselves.

4.3.4 Participants

Creator

- declares the factory method which returns a product object
- default factory method implementations may return a default concrete product

ConcreteCreator

- overrides the factory method to return an instance of the product

Product

- defines the product interface for the factory method to create

ConcreteProduct

- implements the interface for the product

4.4 Factory Method Pattern Explained

4.4.1 Clarification

The creator is not sure what class of product is to be created and delegates this responsibility to it subclasses. It is the responsibility of the the concrete creator classes to create specific product. This results in the parallel hierarchies of Creator and Product with the dependencies between the hierarchies on the concrete level, ConcreteCreator uses a ConcreteProduct.

4.4.2 Code improvements achieved

Objects are created in an orderly fashion. Central management of the object creation process exists. This could be used with great effect to control the life-time of the object and ensure that as with creation, deletion is also conducted in an orderly way.

4.4.3 Implementation Issues

The factory method can be implemented either by defining the creator class as abstract or as concrete. In the case of an abstract creator class, a concrete creator per product must be defined or the concrete creator needs to be parameterised to produce the correct concrete product. When the creator class is defined as a concrete class it must provide default implementations for all operations it defines.

4.4.4 Common Misconceptions

Using only a wrapper with a descriptive name for the construction process [3], does not mean that a Factory Method design pattern has been used. Consider the `ComplexNumber` class in Figure 3 that participated as the originator in the Memento pattern described in L03. There is no distinction in this class with regards to cartesian or polar coordinates and implementing a constructor that can distinguish between these is not feasible as an extra parameter will be required to make the distinction. The best would be to provide a public operation, with a descriptive name, that indicates the co-ordinate system being used as parameters for the creation of the object which returns an instance of `ComplexNumber`.

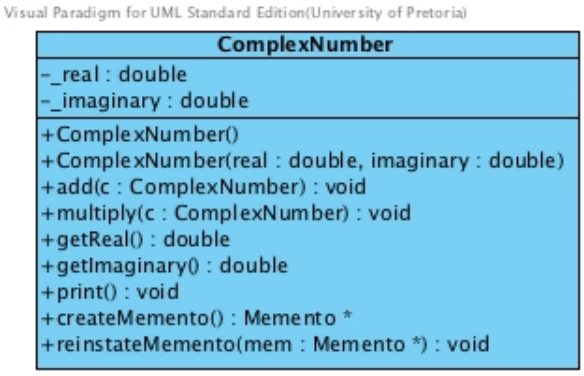


Figure 3: UML class diagram of ComplexNumber

A suggestion for the implementation of these co-ordinate specific operations is given below. The visibility of the constructor of the class that accepts two doubles as parameters can be changed to protected in order to ensure that it is not inadvertently called with an incorrect co-ordinate system.

```
ComplexNumber* ComplexNumber::fromCartesian(double real, double imaginary)
{
  return new ComplexNumber(real,imaginary);
}

ComplexNumber* ComplexNumber::fromPolar(double modulus, double angle)
{
  return new ComplexNumber(modulus*cos(angle),modulus*sin(angle));
}
```

In order for this to be considered as an implementation of a Factory Method design pattern, the `ComplexNumber` class needs to inherit from an abstract class, say `Number`. The determining of the type of co-ordinate system should be left to the concrete creator which forms part of the parallel factory hierarchy that needs to be defined. Refer to the example given in section 4.5 for a suggestion to implement the `ComplexNumber` class as a product of the Factory Method design pattern.

4.4.5 Related Patterns

Template Method
The Factory Method may make use of Template Method in both the Product and the Creator hierarchies.

Abstract Factory
The Factory Method may be used in the implementation of the Abstract Factory design pattern.

Prototype
Factory Methods can be used to initialise prototypical objects. The prototype also can be used instead of the factory method to avoid large parallel hierarchies.

Singleton
In only one instance of a concrete factory is required, the concrete factory can be made a Singleton.

4.5 Example

This example can be combined with the Memento example given in Lecture Note 03. For clarity, all references to the Memento have been removed in order to illustrate only the Factory Method design pattern. Figure 4 shows the relationships between the classes and the structure of each of the classes participating in the Factory Method design pattern.

The corresponding pattern participants for this example are:

- Creator: `NumberGenerator` - an abstract class defining the factory method `generateNumber`. `nextNumber` is a template method operation that forms part of the Template Method design pattern in the example.
- ConcreteCreator: `ComplexNumberGenerator` - produces a `ComplexNumber` product object. The instantiation of the product object is dependent on the co-ordinate system encapsulated in the creator hierarchy.
- Product: `Number` - provides the interface for numbers.
- ConcreteProduct: `ComplexNumber` - defined exactly as it was for the Memento example, except for the removal of the Memento specific operations and the inheritance relationship with the `Number` class.

4.5.1 Implementation notes

To successfully implement the design pattern for the given example, the following should be noted:

Virtual destructor
`NumberGenerator` defines the interface to generate different number types, specifically a complex number in this example. Instantiating an object of `ComplexNumberGenerator`

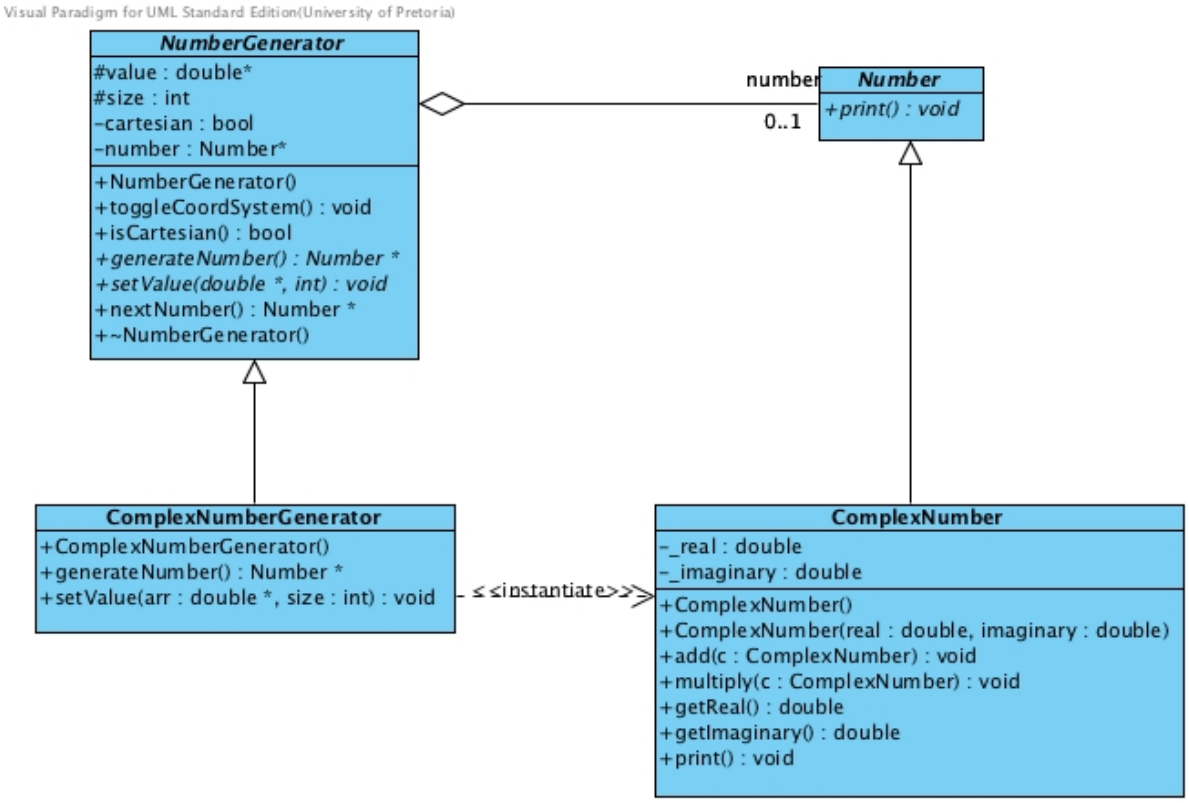


Figure 4: UML class diagram for the example of the Factory Method design pattern

reserves heap memory that has been defined in the corresponding base class. In order to successfully clear the memory when an object of `ComplexNumberGenerator` goes out of scope, the destructor of the base class `NumberGenerator` must be defined as virtual. This destructor must then deallocate the heap memory defined by it that instantiating classes in the hierarchy would have allocated. The definition and implementation of `NumberGenerator` is given in the listing that follows.

```
class NumberGenerator
{
  public:

    NumberGenerator()
    {
      number = 0;
      cartesian = true;
      value = 0;
      size = 0;
    };

    void toggleCoordSystem()
    {
      cartesian = !cartesian;
    };
```

```
    bool isCartesian() {
      return cartesian;
    };

    virtual Number* generateNumber() = 0;
    virtual void setValue(double*,int) = 0;

    Number* nextNumber() {
      number = generateNumber();
      return number;
    };

    virtual ~NumberGenerator()
    {
      if (number != 0) {
        number = 0;
      }
      if (size != 0) {
        delete [] value;
        value = 0;
      }
    }
  protected:
    double* value;
    int size;
  private:
    bool    cartesian;
    Number* number;
};
```

Calling the constructor of the base class

In order to initialise the attributes of `ComplexNumberGenerator` that are defined in the base class, the constructor of the base class must be called by the constructor of the derived class. Implementing this is trivial and can be accomplished by using member-list initialisation. The implementation of the constructor is given by:

```
ComplexNumberGenerator::ComplexNumberGenerator() : NumberGenerator()
{
}
```

Primitive operation implementation

Notice the memory management applied to the array of values derived from the base class in the code for the primitive operations of the template method for the `ComplexNumberGenerator` given below. As complex number is implemented in terms of the cartesian co-ordinate system, it is necessary that the generator for complex number class does the conversion of polar to cartesian.

```
Number* ComplexNumberGenerator::generateNumber()
{
```

```
    if (size == 0) {
      value = new double[2];
      value[0] = 0;
      value[1] = 0;
      size = 2;
    }
    if (isCartesian())
      return new ComplexNumber(value[0], value[1]);
    else
      return new ComplexNumber(value[0]*cos(value[1]),
                               value[0]*sin(value[1]));
};

void ComplexNumberGenerator::setValue(double* arr, int size) {
    if (this->size != 0) {
      delete [] value;
      this->size = 0;
    }
    value = new double[size];
    value[0] = arr[0];
    value[1] = arr[1];
    this->size = size;
};
```

4.5.2 Main program

An example of a test program is given. Notice that the responsibility of deleting objects of `Number` is left to the client of the factory method. It is also important to note that the client never directly instantiates an object of `ComplexNumber`, it is the job of the corresponding generator to do so. It is a good habit to ensure that all heap memory is deallocated in the reverse order of allocation.

```
int main()
{
  double* valueList;

  valueList = new double[2];
  valueList[0] = 3;
  valueList[1] = 8;

  NumberGenerator* factory = new ComplexNumberGenerator();

  Number* one = 0;
  Number* two = 0;

  one = factory->nextNumber();
  one->print();
```

```
    factory->toggleCoordSystem();

    factory->setValue(valueList,2);
    two = factory->nextNumber();

    one->print();
    two->print();

    delete two;
    delete one;

    delete factory;
    delete [] valueList;

    return 0;
}
```

4.6 Exercises

1. Consider the class diagram presented in Figure 5 and answer the questions that follow:

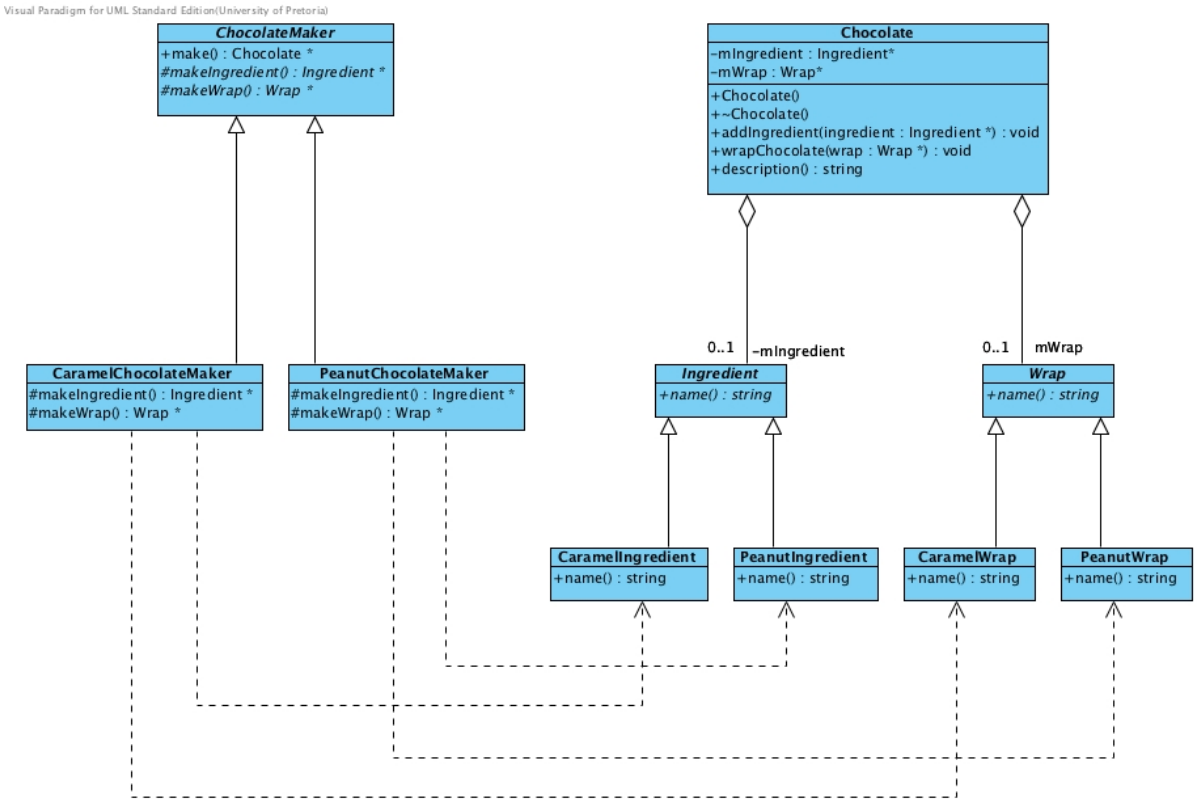


Figure 5: Chocolate Factory

(a) Identify the participants.

(b) Has a template method been used in combination with the factory method?

(c) Write a client program that makes use of this factory method hierarchy.

2. Combine the Memento implementation with the Factory Method implementation discussed here. Hint: You should adapt the memento to internalise the state of any number object.

References

[1] Tony Gaddis. *Starting out with C++: from control structures through objects*. Pearson Education, seventh edition, 2012.

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

[3] Wikipedia. Factory method pattern — wikipedia, the free encyclopedia, 2011. URL `http://en.wikipedia.org/w/index.php?title=Factory_method_pattern`. [Online; accessed 10 August 2011].

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.

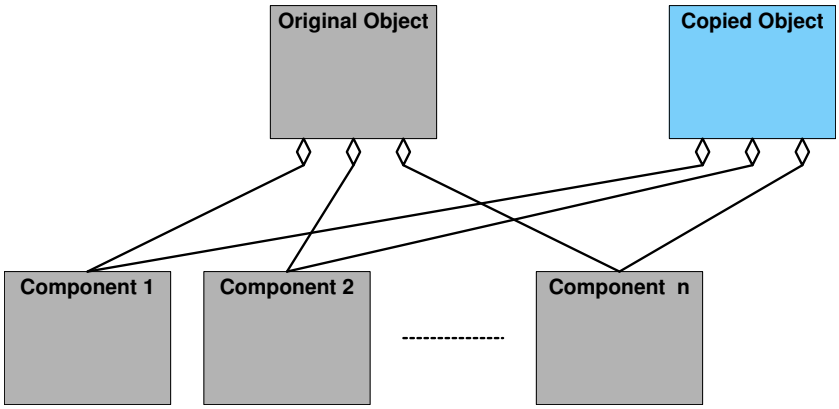


Figure 1: Copy of an object that shares the instance variables of the original

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.

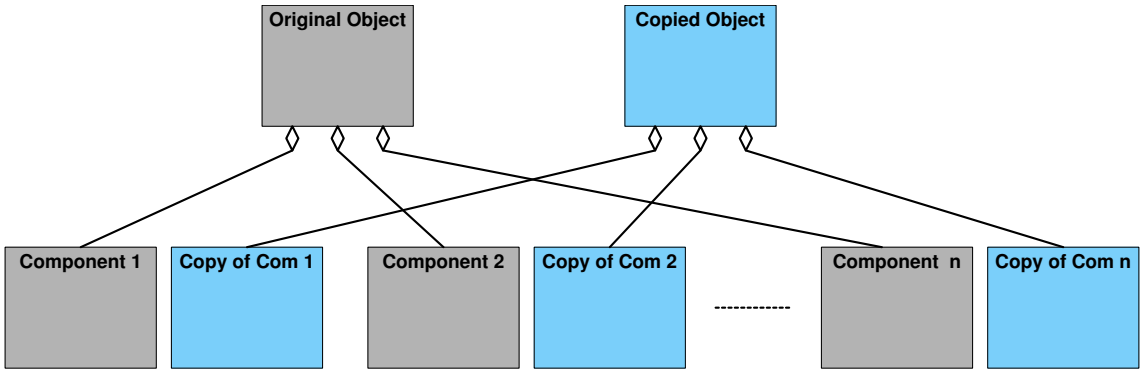


Figure 2: Copy of an object that has its own copies of the instance variables of the original

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

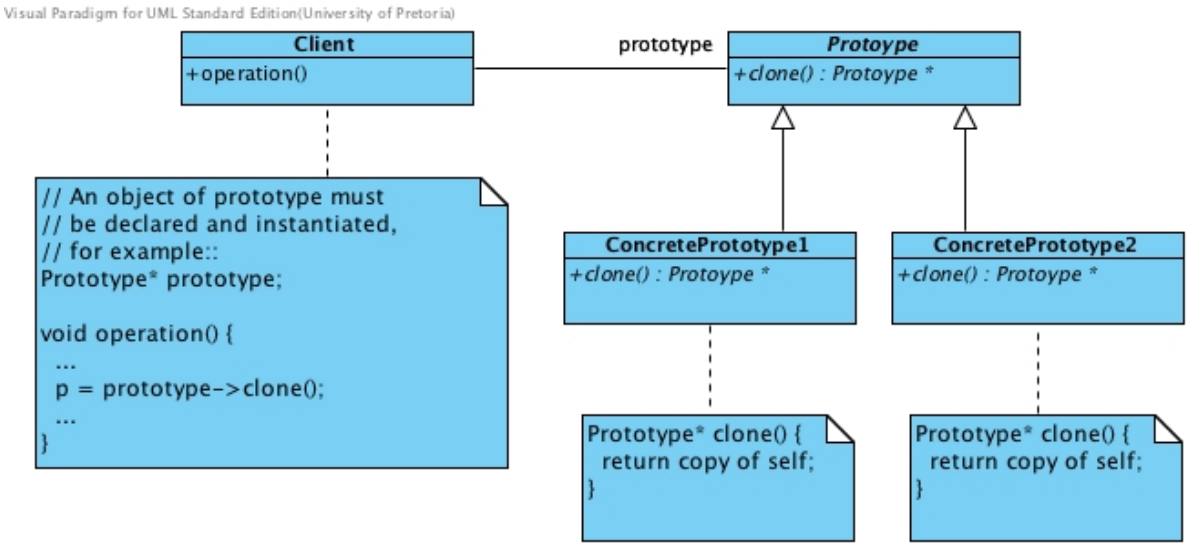


Figure 3: The structure of the Prototype Pattern

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.

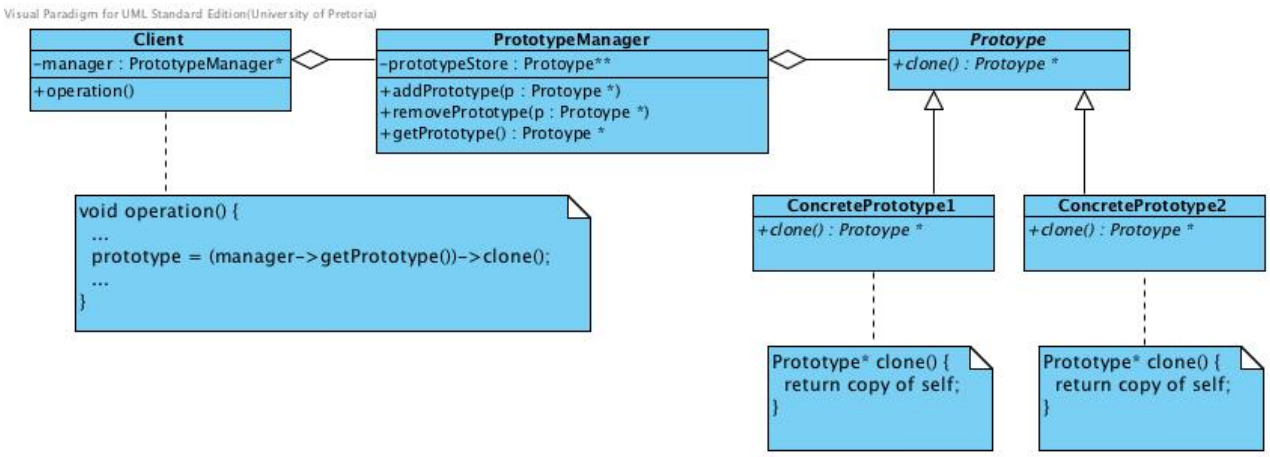


Figure 4: The structure of the Prototype Pattern with a manager

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

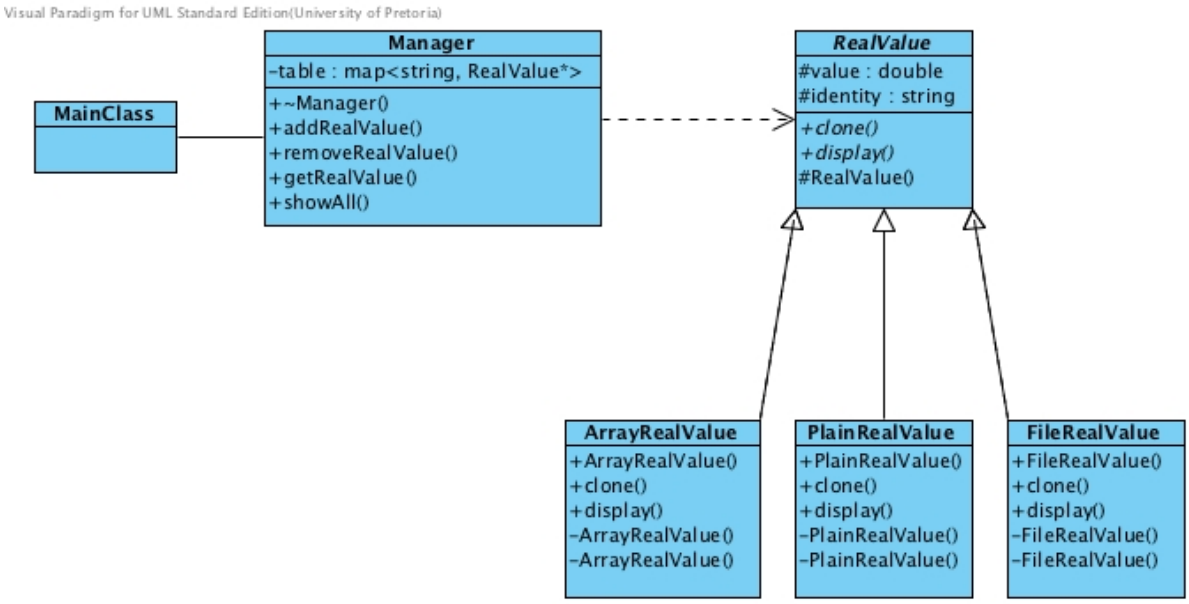


Figure 5: Class Diagram of a system illustrating the implementation of the prototype design pattern

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].

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 6: UML Class and Object Diagrams

Contents

6.1 Introduction

In object-oriented programming, classes do not function in isolation, they interact with each other. UML therefore needs to be able to model how classes interact and for structural UML diagrams, such as class and object diagrams, these interactions are modelled as relationships. Two broad categories of relationships exist, those which are used to model delegation and those used to model inheritance. Examples of delegation relationships are associations and dependencies, which have already been discussed. To model inheritance, the generalisation relationship has been defined in UML. Figure 1 provides an overview of the different relationships which can be modelled in UML [1].

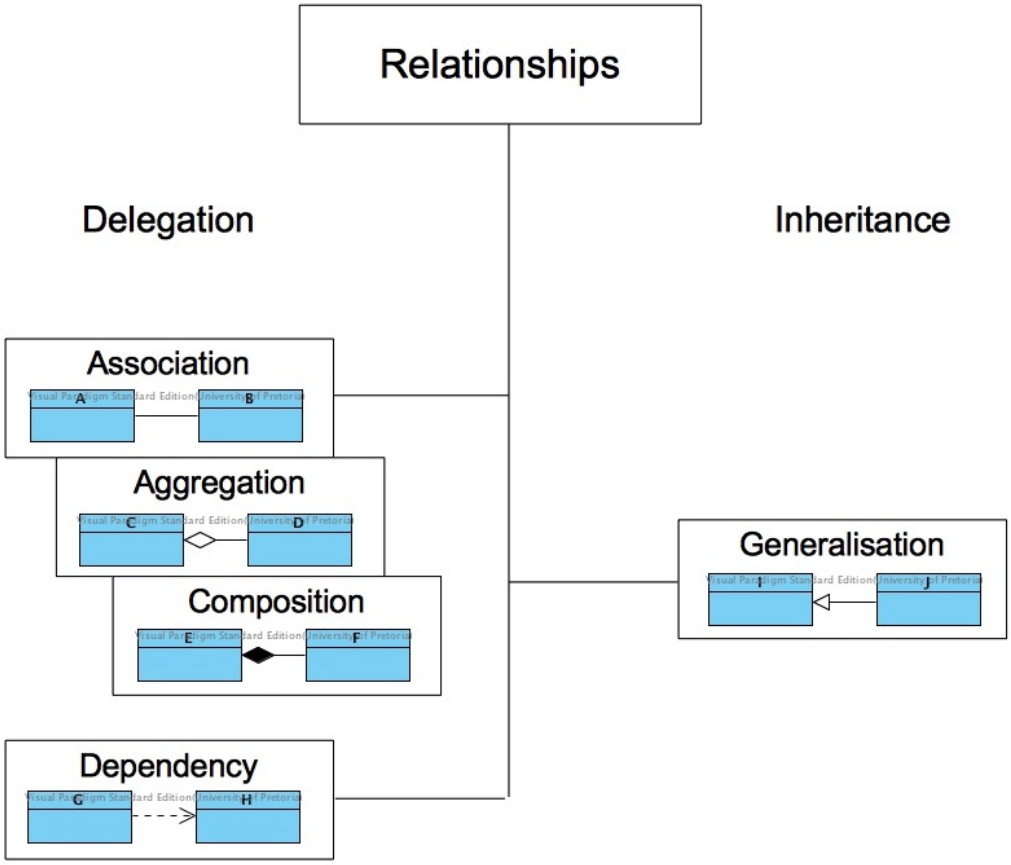


Figure 1: UML relationships

Modelling classes and delegation in UML was discussed in Chapter 2, specifically Sections 2.2.1 and 2.2.2 respectively. In this lecture note, modelling inheritance and objects will be the focus.

6.2 Modelling generalisation

Public inheritance has been used in Chapters 3, 4 and 5 in the code examples and presented in the corresponding UML Class diagrams for the examples. In Section 3.2.1 of Chapter

3, public inheritance was presented as an *is-a* relationship. A differentiation was also made between different types of operations in a class, namely: non-virtual, virtual and pure virtual. In Figure 2, `ConcreteClass` *is-a* `AbstractClass`. `ConcreteClass` inherits from `AbstractClass` and therefore there is a generalisation relationship between the two classes.

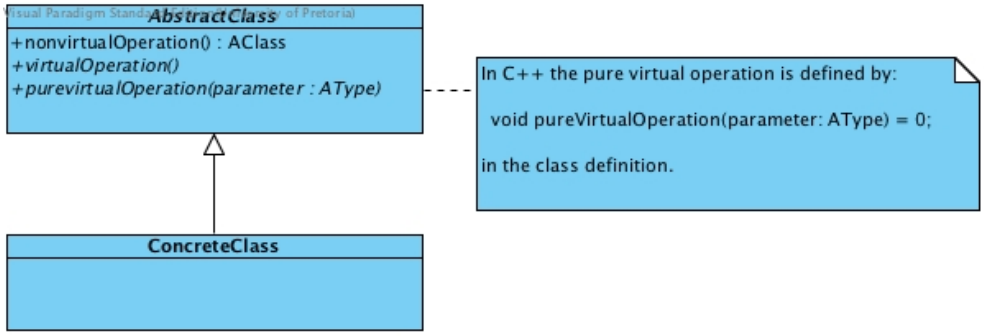


Figure 2: Modelling generalisation in UML

All operations in `AbstractClass` are public, this means that they can be called when an instance of `ConcreteClass`. `nonvirtualOperation`, as defined in `AbstractClass`, can be called by an object of `ConcreteClass`. The operation `virtualOperation` may be redefined in `ConcreteClass`, while `purevirtualOperation` must be implemented in `ClassOperation`. A class with a pure virtual operation cannot be instantiated and is therefore defined as abstract. Under the assumption that the non-virtual operation was not public, but protected or private, then a program or object calling operations of `ClassOperation` would not be access the operation. A operation within `ConcreteClass` would be able to call the operation if it were defined as protected.

6.3 Object diagrams

Object diagrams are derived from class diagrams and are therefore dependent on class diagrams. They represent an instance of a class or classes and are therefore seen as a static view of the class at a specific moment. Object diagrams are concrete in nature, they represent the real-word versus class diagrams which are abstract representations and are seen as a blue-print. An object diagram represents unlimited instances while class diagrams comprise of fixed classes. Object diagrams use the same basic relationships as class diagrams. Object diagrams can be used for forward and reverse engineering of code.

6.3.1 Modelling objects

Refer back to Figure 1 in Chapter 1 showing that both class and object diagrams are classified as UML structural diagrams depicting the relationships between classes and objects respectively.

An example of a diagram showing different styles a single object can be depicted in UML as given in Figure 3. Note the class which these objects are instances of, given in Figure 4, consists of an attribute with object scope (`attribute1`) and an attribute with class scope

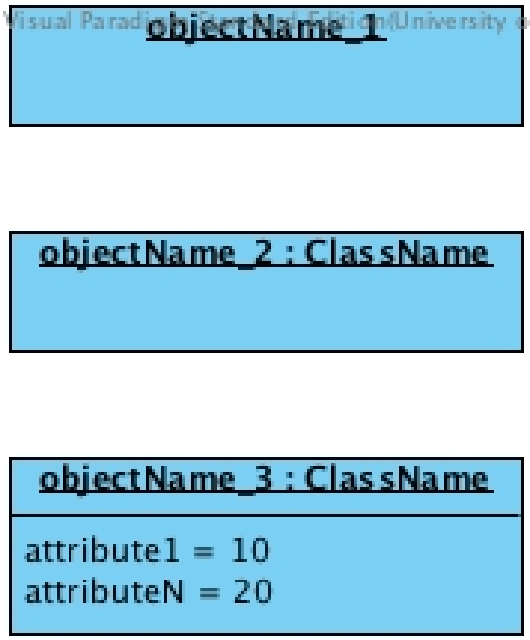


Figure 3: UML object diagrams

(`attributeN`) which will be defined as static. The scope distinction is not evident in the object diagrams.

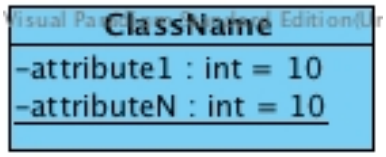


Figure 4: UML class diagram

To determine whether a diagram is a class or an object diagram, it is necessary to consider the name section of the diagram. If the name section is underlined (as with `objectName_1`), an object is being represented. The detail at which the object is being represented can also vary. `objectName_2` specifies which class this representation in as object of, while `objectName_3` details the values of the attributes of the object at specific moment. Operations do not feature in object diagrams.

6.3.2 Modelling relationships

Generalisation relationships are not modelled in object diagrams, but the modelling of associations and dependencies, or just showing that there is a relationship between objects is common *Need to completes*

The are multiple ways of showing multiplicities in object diagrams. Consider the class diagram given in Figure 5. The class `University` has an attribute `students` which has been defined as an array of objects of class `Student`. From the multiplicity it is clear that

an object representing the class `University` may have 0 or more objects of class `Student`. Objects of class `Student` have an `age` and a `degree` attribute.

Figure 5: UML class diagram showing multiplicity

If the detail of each instance in the array is important, then each object of class `Student` associated with the object of class `University` should be shown in the object diagram. This is only practical when the number of objects in the association is not so high that the diagram becomes impossible to understand or when a selection of objects are to be depicted. Assuming we wish to show two students, Alice and Bob, and their affiliation to the the University of Funny Walks (UFW) - apologies to Monty Python [2]. The object diagram representing this situation is given in Figure 6.

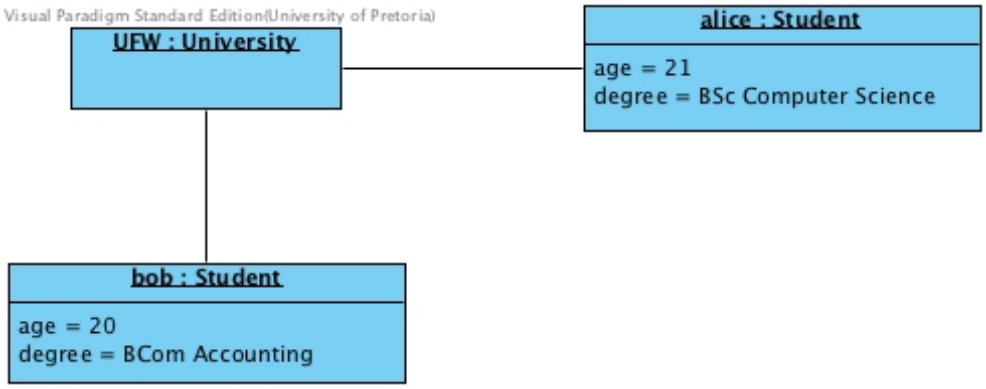


Figure 6: UML object diagram showing multiplicity with object detail

If only knowing which objects have been created is of relavance, then the object diagram can be shown as given in Figure 7 using an attribute textual notation.

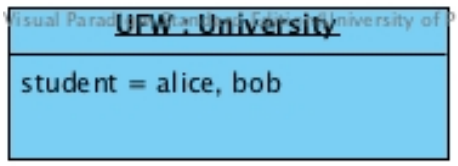


Figure 7: UML object diagram showing object attributes textual notation

Another notation in which the details are not important, yet the multiplicity is shown is given in Figure 8. Structures as these are not native to many modelling tools and therefore are not normally drawn as such or need to be manually drawn.

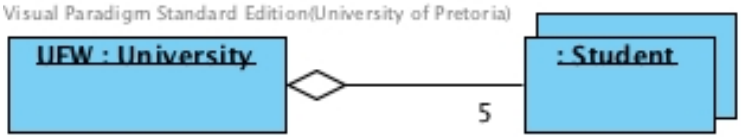


Figure 8: UML object diagram showing object attributes textual notation

6.3.3 Example

Consider the class diagram given in Figure 9 which is an implementation of the Template Method design pattern. Assume that the main program for the classes defined is given by:

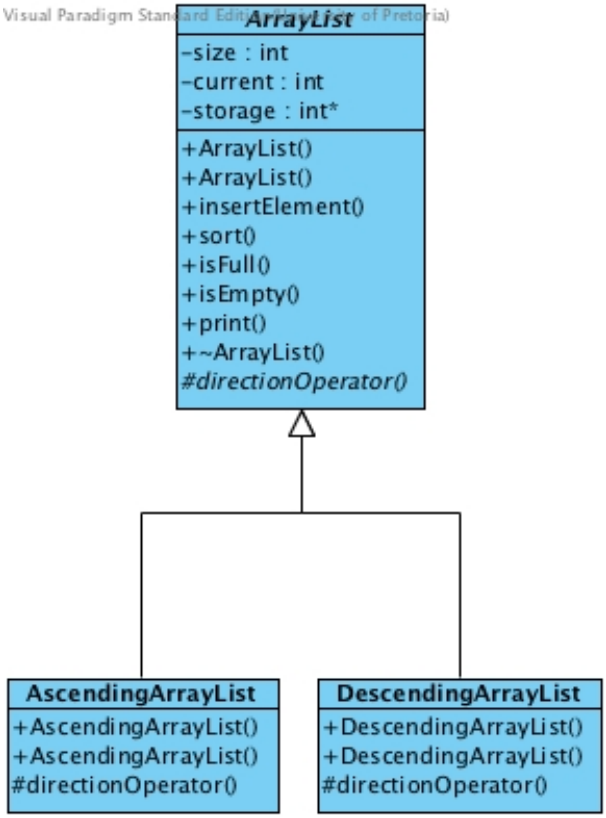


Figure 9: ArrayList, an implementation of the Template Method design pattern

```
1  #include <iostream>
2  #include "ArrayList.h"
3
4  using namespace std;
5
6  int main(){
7
8          ArrayList* arr = new DescendingArrayList(10);
9
10         arr->insertElement(10);
```

```
11          arr->insertElement(20);
12          arr->insertElement(15);
13          arr->insertElement(25);
14          arr->insertElement(5);
15
16          arr->print();
17          arr->sort();
18          arr->print();
19
20          delete arr;
21
22          return 0;
23  }
```

After the object has been created and instantiated in Line 8 of the main program, an object diagram for this state of the program can be drawn. This state is given in Figure 10. After 3 elements have been inserted into storage, that is at Line 12 in the code, the corresponding object diagram for the state of the program at that moment is given in Figure 11. After the elements and `arr` has been sorted, Line 17, the object at that particular moment is given in Figure 12. Note the change in the value of `current` in Figures 10 to 12.

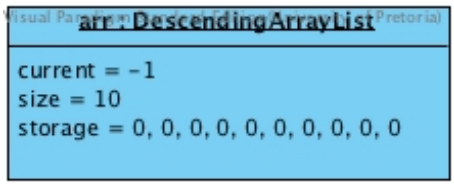


Figure 10: Object diagram on `arr` creation

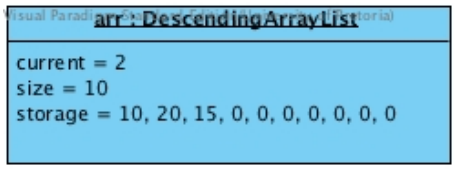


Figure 11: Object diagram after 3 elements have been inserted

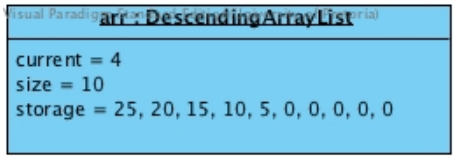


Figure 12: Object diagram after sorting

6.3.4 Object diagrams when modelling behaviour

Objects play a role when behaviour is modelled, using UML Sequence and Communication diagrams. What is depicted in these diagrams is not the state of the object at a given moment, but the communication between objects.

References

[1] Object Management Group. Unified Modeling Language (OMG UML) Version 2.5, 2015. URL `http://www.omg.org/spec/UML/2.5/PDF/`. [Online; accessed 1 August 2015].

[2] Monty Python. Monty Python's - The Ministry of Silly Walks, 2015. URL `http://www.thesillywalk.com`. [Online; accessed 1 August 2015].

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 7: Absract Factory design pattern

Contents

7.1 Introduction

The Abstract Factory design pattern is a creational pattern used to produce product with a common theme [2]. The factories are grouped together under a single interface and linked to differentiated products. Each product hierarchy defines an interface.

7.2 Abstract Factory Pattern

7.2.1 Identification

Name	Classification	Strategy
Abstract Factory	Creational	Delegation (Object)
Intent		
Provide an interface for creating families of related or dependent objects without specifying the concrete classes. (Gamma et al. [1]:87)		

7.2.2 Structure

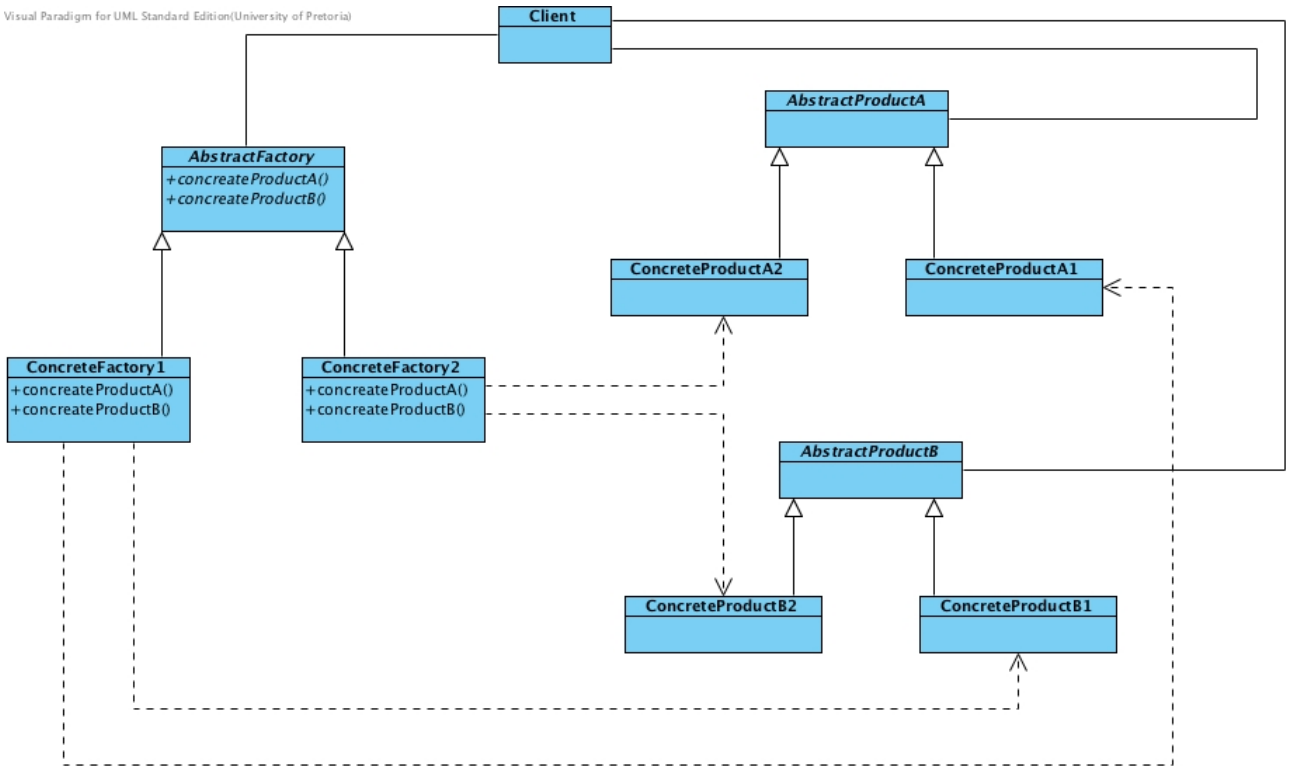


Figure 1: The structure of the Abstract Factory Pattern

7.2.3 Participants

AbstractFactory

- provides an interface to produce abstract product objects

ConcreteFactory

- implements the abstract operations to produce concrete product objects

AbstractProduct

- provides an interface for product objects

ConcreteProduct

- implements the abstract operations that produce product objects that are created by the corresponding ConcreteFactory

Client

- uses the interfaces defined by AbstractFactory and AbstractProduct

7.3 Abstract FactoryPattern Explained

7.3.1 Clarification

The abstract factory comprises of concrete factories. It is the concrete factories that creates product.

7.3.2 Common Misconceptions

Important to note the subtle differences between Factory Method and Abstract Factory. With Factory Method there is a one-to-one relationship between the factory and the product, Abstract Factories exhibit a one-to-many relationship.

7.3.3 Related Patterns

Factory Method or Protoype
The Abstract Factory makes use of the Factory Method or the Prototype for the creation of product. The choice of which route to follow is implementation dependent.

Template Method
May be used within the abstract factory and product hierarchies.

Singleton
Concrete factories may be implemented as Singletons..

7.4 Example

Consider a classification for two-dimensional shapes. 2D shapes are further classified as either polygons or not being a polygon. Figure 2 presents the hierarchy for polygons represented by the system.

Polygons are classified as either quadrilaterals or triangles. All these classes are abstract. The concrete classes are those that inherit from `Quadrilateral` and `Triangle`. The `Polygon` hierarchy forms that product hierarchy for polygons.

The product hierarchy for non-polygons is shown in figure 3. `NonPolygon` is an abstract class, while the `Ellipse` and `Circle` classes are both concrete classes.

As both these hierarchies represent two-dimensional shapes, another abstract class, the `Shape` class is introduced in order to ensure consistency in both hierarchies (refer to figure 4). This class does not form part of the Abstract Factory design pattern, but does not detract from it either. It merely defines the common aspects of all two-dimensional shapes.

The class `Shape` holds two state attributes that are of interest in this example. The first is `LoS`, or lines of symmetry, for each concrete shape the lines of symmetry is stored. The second interesting attribute is `RS` which represents the order of rotational symmetry of the shape. Some shapes have a RS of order 0, while other shapes such as a circle have a RS of infinity.

Till now, the example only comprises of a hierarchy of two-dimensional shapes. Notice that this hierarchy classifies the shapes in terms of their structural characteristics. The attributes of LoS and RS are just that, attributes, and do not form part of the classification. In order to *mesh* the symmetry classification with the structural classification, an abstract factory can be used to produce the shapes according to the symmetry characteristics. The class that represents the *Abstract Factory* participant of the design pattern is in figure 5.

The two concrete classes produce objects that are either line symmetric or rotational symmetric. The classification of polygon and non-polygon is also preserved as each of the classes produce their respective polygon types as well. The code showing how the `ConcreteFactory` classes are implemented is given to show how the classes are linked.

```
class LineSymmetricShapeFactory : public ShapeFactory {
  public:
    LineSymmetricShapeFactory() : ShapeFactory() {};
    LineSymmetricShapeFactory(int lines)  {magnitude = lines;};
    Shape* createPolygonInstance() {
      switch (magnitude) {
        case 0 : return new RightAngledTriangle;
                        // or return new Parallelogram;
        case 1 : return new IsoscelesTriangle;
        case 2 : return new Rectangle;
                        // or return new Oblong;
        case 3 : return new EquilateralTriangle;
        case 4 : return new Square;
        default: return 0;
      }
    };
```

```
    Shape* createNonPolygonInstance() {
      if (magnitude == 2)
        return new Ellipse;
      return new Circle;
    }
};

class RotationalSymmetricShapeFactory : public ShapeFactory {
  public:
    RotationalSymmetricShapeFactory() : ShapeFactory() {};
    RotationalSymmetricShapeFactory(int order) {magnitude = order;};
    Shape* createPolygonInstance()  {
      switch (magnitude) {
        case 0 : return new IsoscelesTriangle;
                         // or return new RightAngledTriangle;
        case 2 : return new Parallelogram;
                         // or return new Rectangle;
                         // or return new Oblong;
        case 3 : return new EquilateralTriangle;
        case 4 : return new Square;
        default: return 0;
      }
    };

    Shape* createNonPolygonInstance() {
      if (magnitude == 2)
        return new Ellipse;
      return new Circle;
    }
};
```

To illustrate how the abstract factory is used to produce product, consider the following main program.

```
int main() {
  ShapeFactory** factory  = new ShapeFactory*[2];
  factory[0] = new LineSymmetricShapeFactory;
  factory[1] = new RotationalSymmetricShapeFactory(2);

  Shape* shapes[4];

  shapes[0] = factory[0]->createPolygonInstance();
  shapes[1] = factory[0]->createNonPolygonInstance();
  shapes[2] = factory[1]->createPolygonInstance();
  shapes[3] = factory[1]->createNonPolygonInstance();

  for (int i=0; i < 4; i++) {
    if (shapes[i] != 0)
      shapes[i]->setState();
```

```
  }

  for (int i=0; i < 4; i++) {
    if (shapes[i] != 0)
      cout << "Area = " << shapes[i]->area() << endl;
  }

  for (int i=0; i < 4; i++) {
    if (shapes[i] != 0)
      delete shapes[i];
  }

  for (int i=0; i < 2; i++) {
    delete factory[i];
  }
  delete [] factory;


  return 0;

}
```

7.5 Exercises

1. Merge the class diagrams given in figures 2, 3, 4 and 5. Make sure that all the delegation, specifically the dependencies, relationships between the concrete factories and the concrete products are included.

2. Consider the class diagram presented in figure 6 and identify the participants.

References

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

[2] Wikipedia. Abstract factory pattern — wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Abstract_factory_pattern`, 2011. [Online; accessed 12-August-2013].

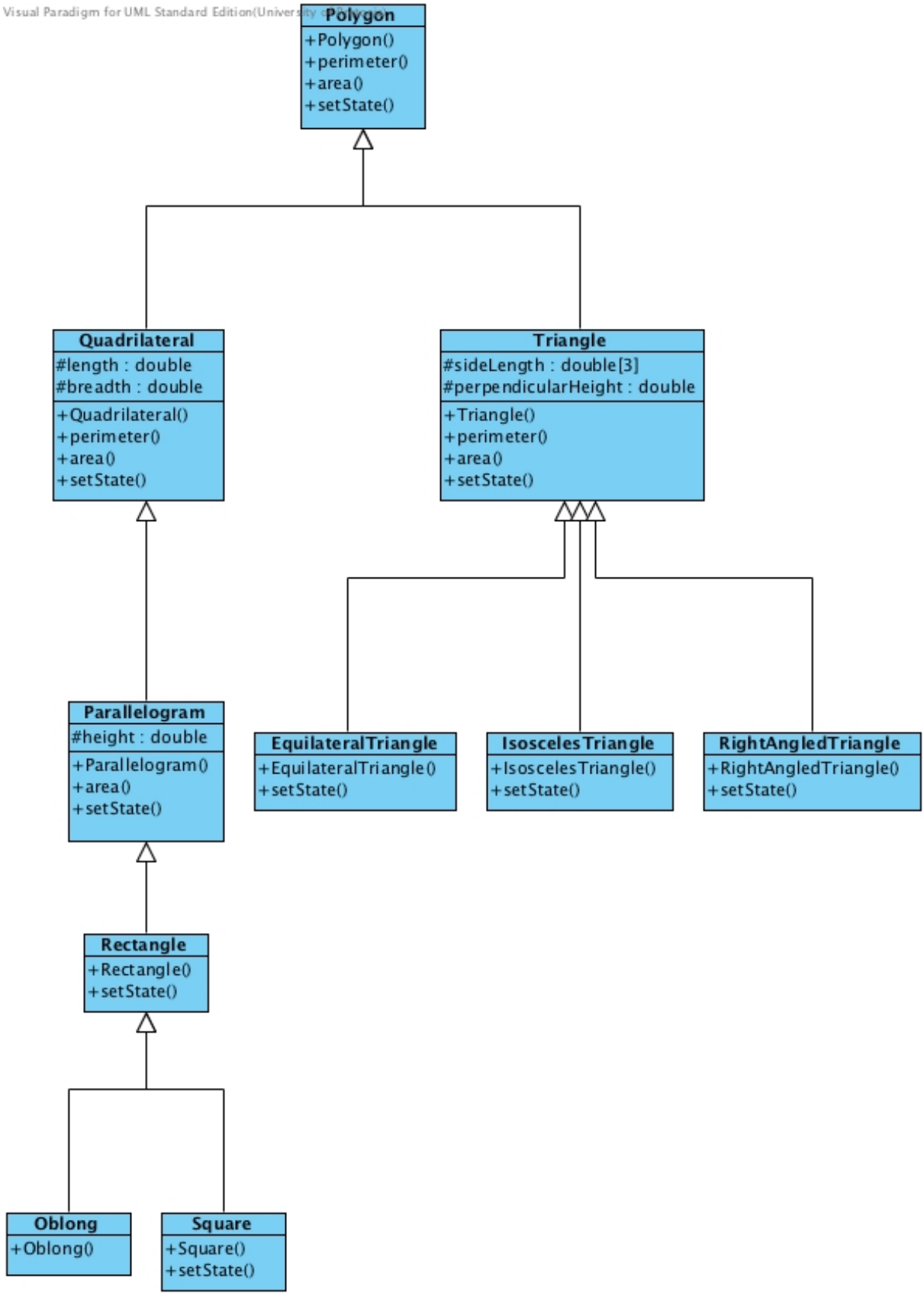


Figure 2: Polygon class hierarchy

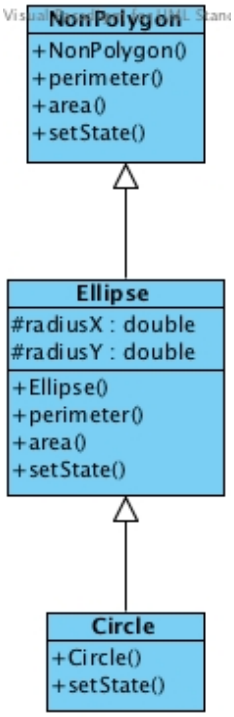


Figure 3: NonPolygon class hierarchy

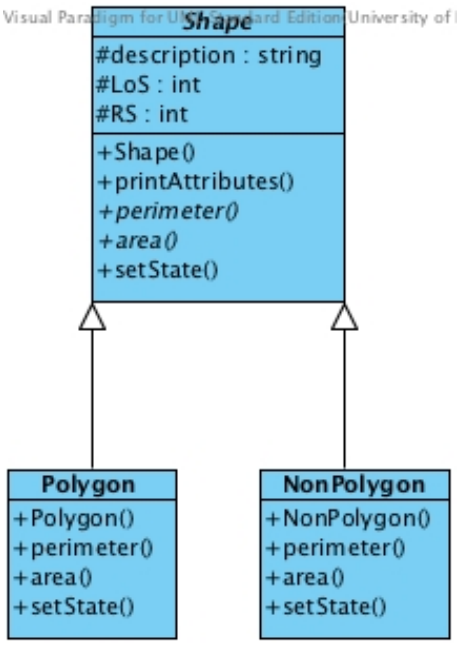


Figure 4: Overarching abstract `Shape` class

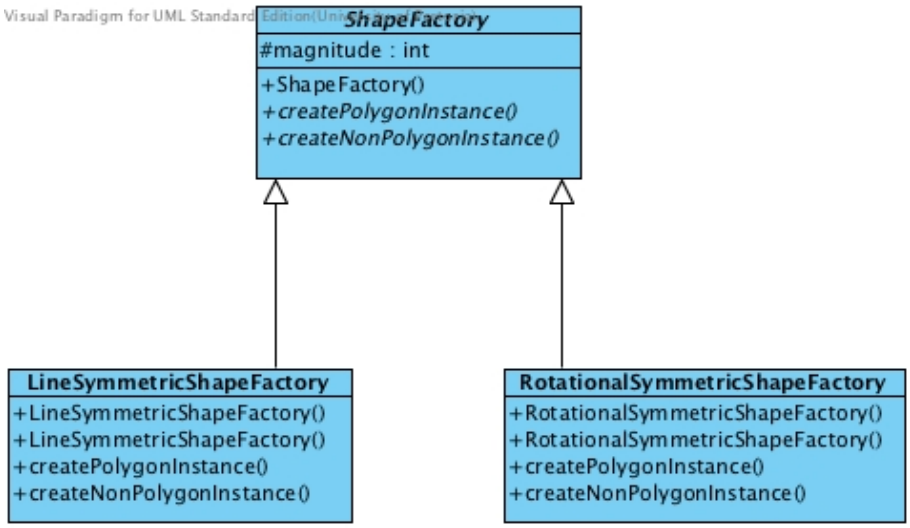


Figure 5: Abstract Factory class hierarchy diagram

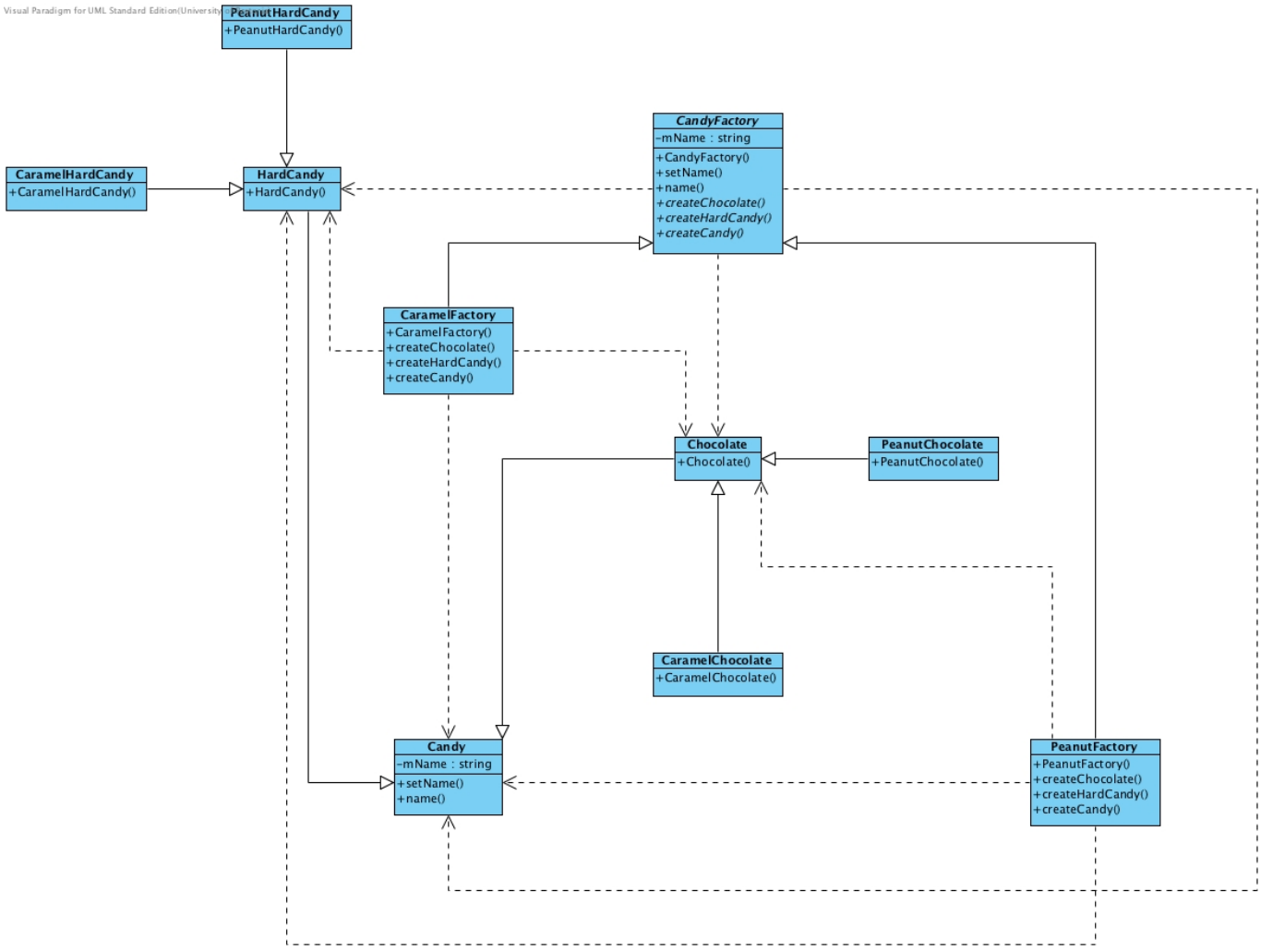


Figure 6: Candy class hierarchy diagram

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 8: Strategy Design Pattern

Contents

8.1 Introduction

In this lecture you will learn all about the Strategy Design Pattern. This pattern has proven to be a very useful pattern. Owing to its elegance it is often used.

We discuss the concept of controlled coupling and why it is considered good practice to control coupling. Most of the design patterns are aimed at controlling coupling in order to enhance maintainability of the code. We illustrate how the Strategy design pattern solves a specific class of problems by applying a standard design solution aimed at controlling coupling.

8.2 Programming preliminaries

8.2.1 Controlling coupling

[3] identify coupling by stating:

> *If changing one module in a program requires changing another module, then coupling exists.*

Coupling impacts negative on the maintainability of a system. The larger the number of modules that needs to be changed to extend or adapt a system, the more difficult it becomes to implement these changes without breaking some code. Coupling occurs when code in one module uses code from another, either by calling a function or accessing some data. It is practically impossible to avoid coupling. It can, however, be controlled. It is important to organise coupling using familiar patterns of dependencies between modules – this is mostly what design patterns are about; i.e. prescribing patterns of dependencies between modules that contributes to the maintainability of the code.

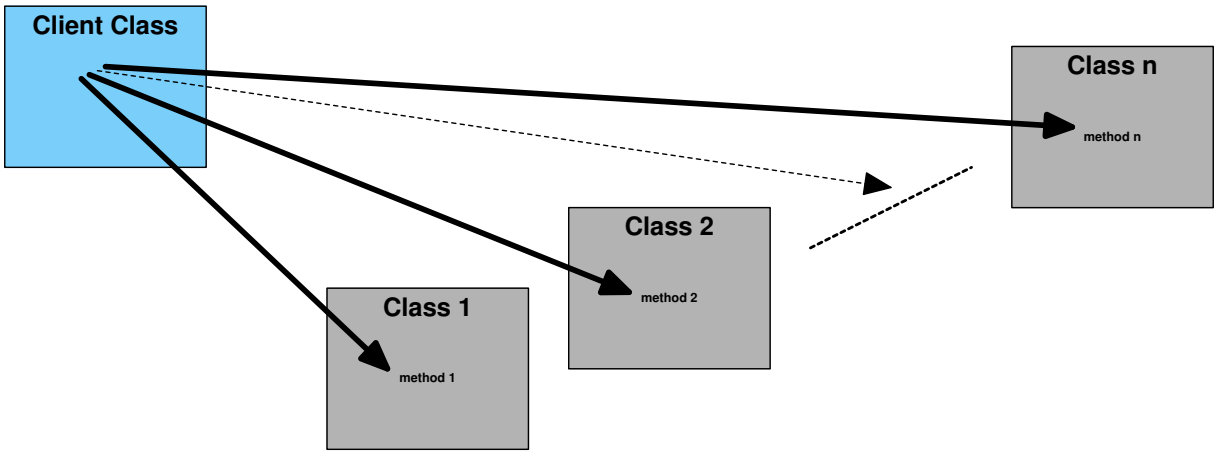


Figure 1: Uncontrolled relations of a client with multiple classes

Defining an interface to serve as a communication hub is a fundamental way to break dependencies and reduce coupling. This strategy is applied in many of the design patterns of which the Strategy design pattern is a very nice example. Figure 1 shows a number

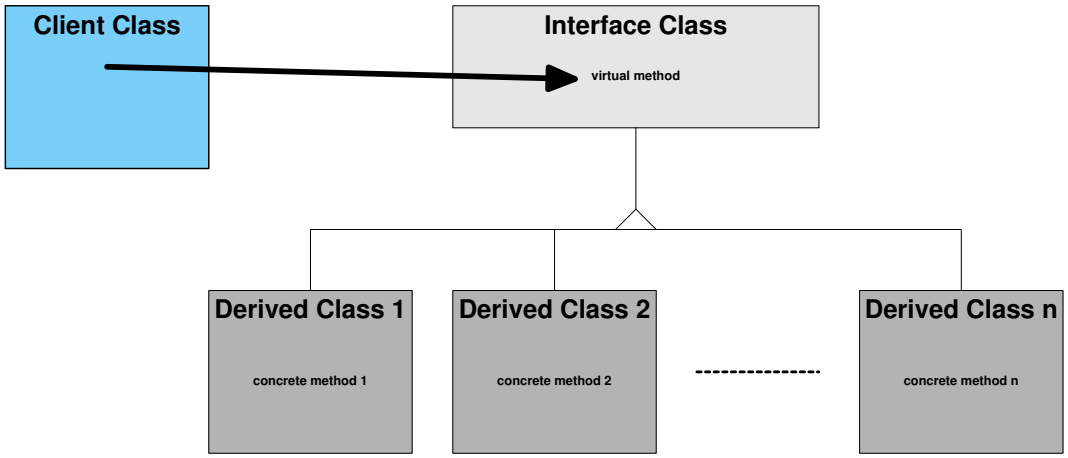


Figure 2: Controlled relation of a client with multiple classes through an interface

of modules who have uncontrolled relations with a client application. Figure 2 shows how this should be organised into a familiar pattern that you will often see in the design patterns.

8.3 Strategy Design Pattern

8.3.1 Identification

Name	Classification	Strategy
Strategy	Behavioural	Delegation
Intent		
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it ([4]:315)		

8.3.2 Structure

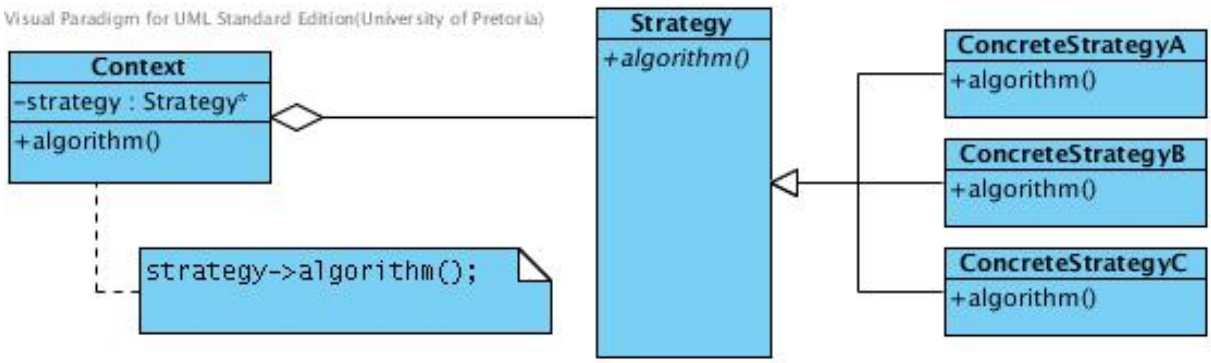


Figure 3: The structure of the Strategy Design Pattern

8.3.3 Problem

An implementation implementing different algorithms to solve the same problem requires uncontrolled coupling with its clients that can be avoided. That is, the implementation consists of a variation of classes for implementing different algorithms to solve a given problem and clients need to couple individually with these classes.

8.3.4 Participants

Strategy

- Declares an interface common to all supported algorithms.
- Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy

- Implements the algorithm defined by the Strategy interface.

Context

- Is configured with a ConcreteStrategy object.
- Maintains a reference to a Strategy object.
- May define an interface that lets Strategy access its data.

8.4 Strategy Pattern Explained

8.4.1 Improvements achieved

- Where various classes provide different implementations of the same routine, interface details is encapsulated in a base class, while the implementation details are provided in derived classes. Clients can now couple themselves to the interface, and will be shielded from changes if algorithms need to be changed: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes.
- Usually the implementation of the Strategy pattern eliminates the need for a complicated conditional statement to select the desired strategy.
- When encapsulating an algorithm in its own class it is possible to define complex algorithm-specific data structures in this class and consequently avoid exposing them to the clients that use the algorithm. This way the implementation of the algorithm is more robust.
- When heuristics are applied to programmatically select an appropriate strategy, this selection my involve the implementation of complex selection structures such as a `switch`-statement or cascading `if`-statements. Encapsulating these in a Context class relieves clients from having to implement them.

8.4.2 Disadvantages

- The coupling between the Strategy interface and the Context class might be wider than always needed. The interface must cater for everything that is needed by each of the strategies. This is the union of the needs of all the strategies. Not all strategies use the whole interface resulting in parameters initialised and never used.

8.4.3 Practical examples

A classic example often used to illustrate the strategy design pattern is to implement a system that offers a method to choose between different algorithms for sorting [1]. This is probably a popular example because most programmers are familiar with the various sorting algorithms and their respective time-space tradeoffs.

There are many other problems for which there are known alternative algorithms to solve the problem, each with their own situational benefits and drawbacks. When a problem requires the system or the user to decide which algorithm should be used to solve the current instance of the problem, the application of the Strategy design pattern is ideal for the implementation of such system. The following are a few practical examples:

- Save files in different formats or compress files using different compression algorithms.
- Provide different routing algorithms in GPS navigational software.
- Provide different validating strategies for input fields.
- Apply different line-breaking strategies to display textual data.
- Apply different routing algorithms to manage network traffic to select to the most effective algorithm based on current traffic patterns.
- Apply different register allocation schemes when compiling code to provide flexibility in targeting code optimisation for different machine architectures.

8.4.4 Implementation Issues

When implementing the pattern one have to provide a way to choose the most appropriate strategy [2]. This can be implemented programmatically or may be implemented to be user-driven. If it is implemented programmatically, the context should implement heuristics to select the most appropriate strategy based on specified criteria. If it is implemented user-driven, the system has to provide an interface for the user to select one of the available strategies. This interface should preferably be implemented by the client. In this case the context only need to provide methods that can be called by the client to change the strategy.

Another thing that needs to be considered is how implemented algorithm is granted access to the data it has to manipulate. The context may pass the data to the algorithm through parameters. If the data structures involved are large, it is desirable to pass a pointer to the data rather than duplicating data. In some cases it may be useful to pass a pointer to the

Context to the algorithm and allow the algorithm to operate directly on the data stored in the Context using a call back mechanism. Alternatively, the strategy can store a reference to its Context, eliminating the need to pass anything a all [4]. This technique couples the Strategy and the Context more closely. The data requirements of the particular problem and its algorithms will determine the best technique.

8.4.5 Common Misconceptions

- When the different strategies are simple methods implemented without encapsulating them in classes that are polymorphic subclasses of an interface, the system implements the desired functionality but is not an application of the Strategy design pattern.

8.4.6 Related Patterns

Factory Method

Both Strategy and Factory Method use delegation through an abstract interface to concrete implementations. However, Strategy performs an operation while Factory Method create an object.

State

The State and Strategy patterns have the same structure and apply the same techniques to achieve their goals. However, they differ in intent. The Strategy pattern is about having different implementations that accomplishes the same result, so that one implementation can replace the other as the strategy requires while the State pattern is about doing different things based on the state, while relieving the caller from the burden to accommodate every possible state.

Flyweight

Strategy objects often makes good flyweights.

8.5 Example

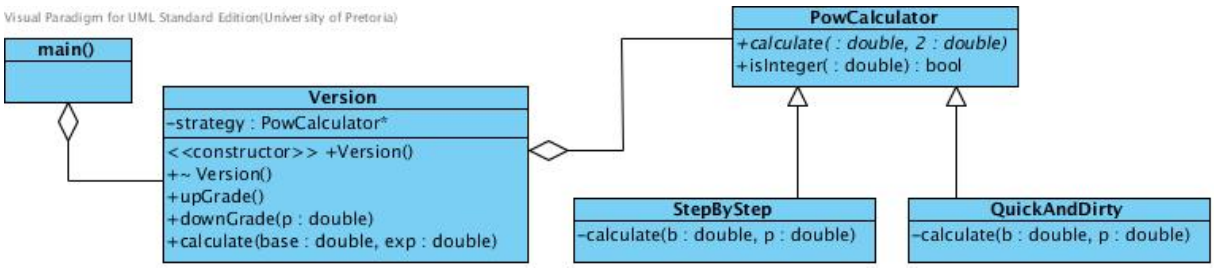


Figure 4: Class Diagram of a system illustrating the implementation of the Strategy design pattern

Figure 4 is a class diagram of an application that implements the strategy design pattern. It implements two different strategies to calculate x^y.

Participant	Entity in application
Context	Version
Strategy	PowCalculator
Concrete Strategies	StepByStep, QuickAndDirty
algorithm()	calculate()
Client	main()

Context

- The `Version` class act as the context.
- The implementation of the `calculate()` method is a redirection to the `calculate()` method of the current concrete strategy. The handle to this method is provided by the `strategy` instance variable of `Version`.
- The `upgrade()` and `downgrade()` methods are provided to enable clients to change the active strategy on the fly.

Stratgey

- `PowCalculator` is an abstract class that implements an interface containing the pure virtual `calculate()` method.
- For the purpose of this application a `isInteger()` method is also defined. This method is used to determine if a given `double` variable holds an integer value. This is used to manipulate the output and also to decide if the step-by-step strategy may be applied.

ConcreteStrategy

- The classes `StepByStep` and `QuickAndDirty` act as concrete strategies.
- The `StepByStep` strategy implements repeated multiplication while the `QuickAndDirty` strategy applies the `math.pow()` function.
- To be able to distinguish between the execution of these strategies, they display their results differently.

Client

- In this application the client has a `Version` object called `application` through which the appropriate implementations of `calculate()` is executed.

8.6 Exercises

1. Write a system that implements the strategy design pattern to allow the user to select different representations of data values that are stored in a text file. Your system should read an unknown number of integer values from the text file and store them in an array in the `Context`. One concrete strategy should write out the values as pairs in set notation. The first coordinate of each pair is an index value while the second coordinate is the value that was read from the file. For example if the file contained the values 5, 7, 2, 12 and 9, the output should be:

{(1,5); (2,7); (3,2); (4,12), (5,9)}

Another strategy should produce a bar chart. For example the above mentioned data should produce the following output:

```
01: XXXXX
02: XXXXXXX
03: XX
04: XXXXXXXXXXXX
05: XXXXXXXXX
```

2. Write a system that can be used for sorting an array of objects using different sorting algorithms. Apply the strategy pattern to allow dynamic switching between algorithms. Implement each algorithm as a template method to enable sorting of different kinds of objects.

References

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

[2] James W. Cooper. *C# design patterns: a tutorial*. Pearson Education, Inc, Boston, 2003.

[3] Martin Fowler. Reducing coupling. *IEEE Software*, 18(4):102 –104, jul/aug 2001. ISSN 0740-7459.

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

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 9: State Design Pattern

Contents

9.1 Introduction

In this lecture you will learn all about the State design pattern. It is a good example of the application of the Protected Variation Principle that is commonly applied to alleviate problems associated with having to change systems during or after development. Therefore, this principle is briefly discussed.

9.2 Protected Variations GRASP principle

General Responsibility Assignment Software Principles (GRASP) is a series of 9 principles proposed by Larman [3] that should be used to govern software design. It is a learning aid to help you understand essential object design and apply design reasoning in a methodical, rational, explainable way. When we discuss the use of the different design patterns, we will refer to the GRASP principles applied by these patterns and introduce these principles as needed.

One of the problems that has been identified as a reason why projects fail is the impact of change during development. Change is inevitable. When mismatches in the expectations and perceptions of different stakeholders are discovered, adjustments are needed to resolve these mismatches in order to deliver a satisfactory system.

The principle we introduce here is called **Protected Variations** (PV). It addresses the problem of the introduction of bugs when code is changed by designing to minimise the impact of code change on other parts of the system. PV suggests that points of predicted variation or instability be identified, and that responsibilities be assigned in such a way that a stable interface is created around them ([3]:427). This is generally done by adding a level of indirection, an interface, and using polymorphism to deal with the identified points of predicted variation.

9.3 State Design Pattern

9.3.1 Identification

Name	**Classification**	**Strategy**
State	Behavioural	Delegation
Intent		
Allow an object to alter its behaviour when its internal state changes.The object will appear to change its class. ([1]:305)		

9.3.2 Structure

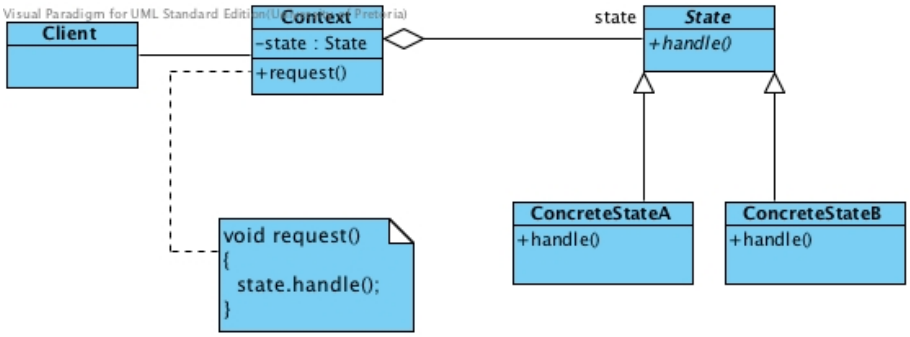


Figure 1: The structure of the State Design Pattern

9.3.3 Problem

The two major problem areas [2] that exist in which the state pattern can make a contribution are:

- when an object becomes large; and
- when there is an extensive number of state changes an object can go through.

The behaviour an object exhibits is dependent on the state of the object. Changing the state of an object will therefore influence the behaviour of the object at run-time. When objects become large, changing their state can become difficult. In order to control the complexity of changing state, the state of the object is managed externally to the object itself.

An object may be required to change state many times and into many different states. When these states become numerous and the flow is controlled by choice-statements (such as `if` or `switch` in C++), the ability to manage the statement flow diminishes. In order to control this complexity, the state of an object can be managed externally to the object itself by modelling the states as objects in their own right.

9.3.4 Participants

State

- Defines an interface for encapsulating the behaviour associated with a particular state of the Context.

ConcreteState

- Implements a behaviour associated with a state of the Context.

Context

- Maintains an instance of a ConcreteState subclass that defines the current state.
- Defines the interface of interest to clients.

9.4 State Pattern Explained

The application of the State Design Pattern is an example of the application of the PV principle. If the system is required to change its behaviour depending on its state, the different behaviours are a candidate for a point of predicted variation which should be wrapped in a stable interface. The `State` participant in the State Design Pattern serves as the stable interface required by this principle while the concrete state classes are isolated in the system to accommodate changes that will not impact on the rest of the system.

The State Pattern applies polymorphism to define different behaviour for different states. Thus, behaviour is altered by executing the code that is implemented in different subclasses. However, the strategy used by this pattern is mainly delegation. The Context class delegates work to be done to these polymorphic classes instead of doing it itself. This way no complicated decision structures are needed in the Context class to cater for different state dependent behaviours.

Client programs should not interact directly with the states. All calls to methods in the concrete state classes are redirections from methods in the context class. Thus, clients will only call methods in the context class. Clients are also not allowed to change the state of the context without the context's knowledge.

9.4.1 Improvements achieved

- **Increased maintainability**
  The State pattern puts all behaviour associated with a particular state into one object. Because all state-specific code lives in a State subclass the code associated with behaviour in a given state is localised and hence easier to maintain. It is also easy to define more states and transitions by defining new subclasses.

- **Eliminate large conditional statements**
  Like long procedures, large conditional statements are undesirable.They're monolithic and tend to make the code less explicit, which in turn makes them difficult to modify and extend. The State pattern offers a better way to structure state-specific code. The logic that determines the state transitions doesn't reside in monolithic `if` or `switch` statements but instead is partitioned between the State subclasses. That imposes structure on the code and makes its intent clearer.

- **Makes state transitions explicit**
  When an object defines its current state solely in terms of internal data values, its state transitions are implicit and have no explicit representation; they only show up as assignments to some variables. Introducing separate objects for different states makes the transitions more explicit. Also, State objects can protect the Context from inconsistent internal states, because state transitions are atomic from the Context's perspective – they happen by rebinding one variable (the Context's State object variable), not several.

9.4.2 Disadvantages

- **Higher coupling**
  The State pattern introduce high coupling. The pattern distributes behaviour for different states across several State subclasses. This increases the number of classes and is less compact than a single class and will require more communication between classes that would be the case if all was implemented in a single class.

9.4.3 Implementation Issues

9.4.3.1 Changing State

When implementing the state pattern one has to decide which class will be responsible to implement the change of state. There are three possible methods that can be applied:

a) **Context applying fixed Criteria**
If the criteria are fixed, then they can be implemented entirely in the Context in the normal flow of events. If this can not be done without conditional statements this approach is probably not appropriate. The disadvantage of this method is that it is not flexible and it is likely that the context code has to change when more states are added to the system. It is generally more flexible and appropriate, however, to let the State sub classes themselves specify their successor state and when to make the transition as described in the following two methods.

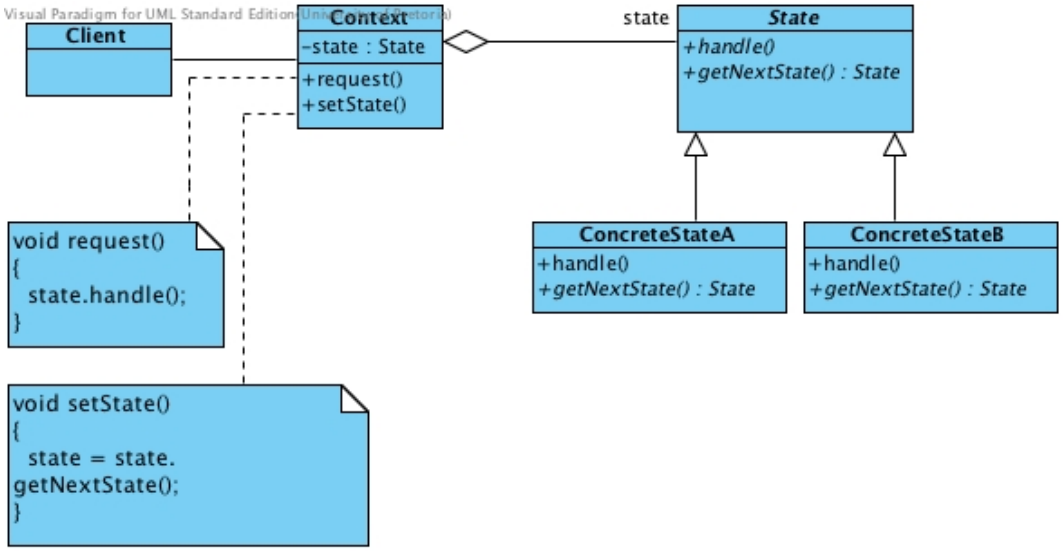


Figure 2: The context is responsible for changing state

b) **Context applying variable criteria**
When the context is responsible for changing the state it will be done in terms of an implementation as shown in Figure 2. Decentralising the transition logic in this way makes it easy to modify or extend the logic by defining new State subclasses. In this case coupling is lower than in the following method since the State is unaware of the Contexts. However, memory management in the context might be difficult. The new

state needs to be requested from the current state and thereafter the current state value has to be updated.

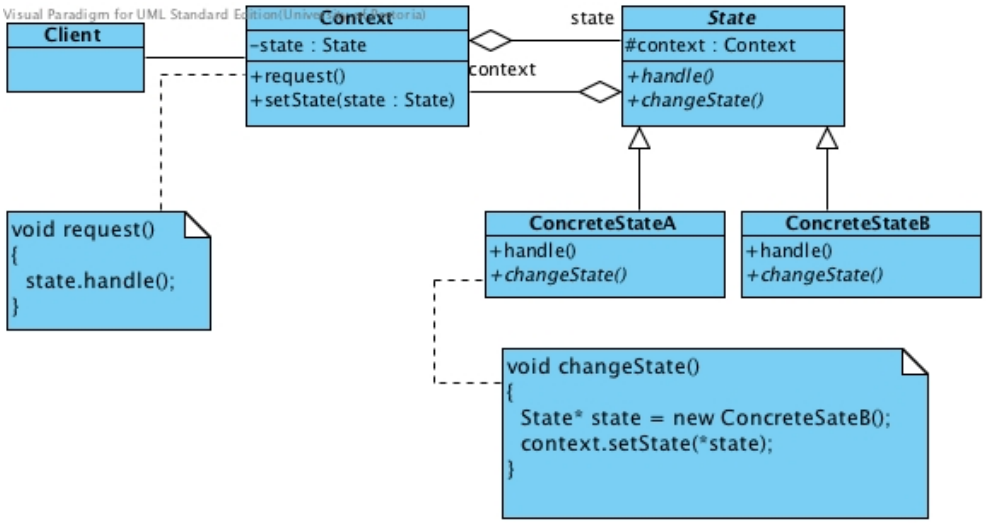


Figure 3: The state is responsible for changing state

c) **Concrete States applying variable criteria**
When the concrete states are responsible for changing the state it will be done in terms of an implementation as shown in Figure 3. This requires adding an interface to the Context that lets State objects set the Context's current state explicitly. In this case the coupling is higher since the State has to be aware of the Context. This can be implemented either by maintaining a reference to its context as shown in this figure, or by passing a pointer to the context as a parameter to the state (as we have done in the example in Section 9.5).

Note that in the last two of the above methods the concrete states are responsible for indicating the new state the context should assume when changing state. They adhere to the Protected Variations Principle while the first method does not. Any changes in the code related to changes in existing states as well as the addition or removal of states when applying the last two methods will not require any changes to the code in the Context class.

9.4.3.2 Sharing State Objects

State objects can be shared. If State objects have no instance variables – that is, the state they represent is encoded entirely in their type – then contexts can share a State object. When states are shared in this way, they are essentially flyweights with non-intrinsic state, only behaviour.

9.4.3.3 Creating and destroying State objects.

A common implementation trade-off worth considering is whether (1) to create State objects only when they are needed and destroy them thereafter versus (2) creating them

ahead of time and never destroying them. The first choice is preferable when the states that will be entered aren't known at run-time, and contexts change state infrequently. This approach avoids creating objects that won't be used, which is important if the State objects store a lot of information. The second approach is better when state changes occur rapidly, in which case you want to avoid destroying states, because they may be needed again shortly. Instantiation costs are paid once up-front, and there are no destruction costs at all. This approach might be inconvenient, though, because the Context must keep references to all states that might be entered.

9.4.4 Related Patterns

Strategy

The Strategy and State patterns have the same structure and both apply the PV Principle to achieve their goals. However, they differ in intent. The Strategy pattern is about having different implementations that accomplishes the same result, so that one implementation can replace the other as the Strategy requires while the State pattern is about doing different things based on the state, while relieving the caller from the burden to accommodate every possible state.

Singleton or Prototype

When implementing the state pattern, the programmer has to decide on how the state objects will be created. Often the application of the Prototype pattern will be ideal. State objects are also often Singletons.

Flyweight

State objects can be shared by applying Flyweight.

9.5 Example

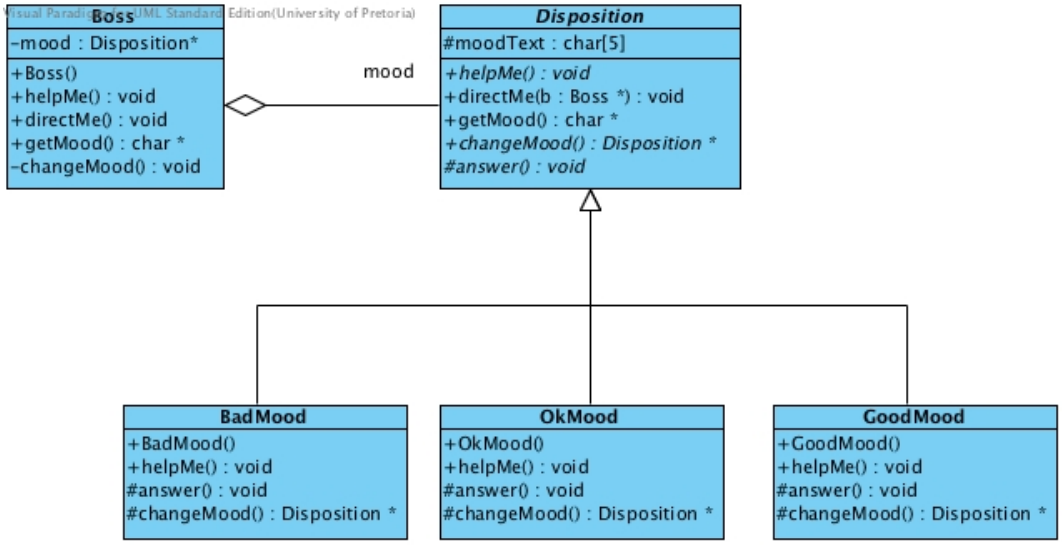


Figure 4: Class Diagram of a moody Boss simulation

Figure 4 is a class diagram of an application that implements the State design pattern. It is a nonsense program that implements three different states a Boss object can assume. The different behaviours of the Boss object is simulated in terms of different messages describing actions in the different states. These `cout` statements are mere placeholders where code that implements different behaviours can be inserted.

Participant	**Entity in application**
Context	Boss
State	Disposition
Concrete States	BadMood, OKMood and GoodMood
request()	helpMe() and directMe()
handle()	helpMe() and answer()

Context

- The `Boss` class act as the context.
- The implementation of both the `helpMe()` method and the `directMe()` method are redirections to the methods with the same names in the `Disposition` interface. The handle to these methods are provided by the `mood` instance variable of `Boss`.
- the `getMood()` method is provided to be able to display the current mood in order to verify the program state during execution.
- The `changeMood()` method is provided to enable a Boss object to change its own disposition. It is defined privately to prevent other objects to be able to change the Boss's state.

State

- `Disposition` is an abstract class that implements an interface containing the methods that implement variations in behaviour that is state dependent.
- `helpMe()` is a pure virtual method. When the `helpMe()` method in the `Boss` class is executed, execution is redirected to the method with the same name in the appropriate concrete state.
- `directMe()` is a template method. When the `directMe()` method in the `Boss` class is executed, execution is redirected to this method. In turn it calls the `answer()` method in the appropriate concrete state and also executes the `changeMood()` method of the appropriate concrete state and use the value returned by this method to manipulate the `mood` variable of the `Boss` class. This is an illustration of how the state is maintained when the concrete states are responsible for changing the state.

Concrete States

- The classes `BadMood, OKMood` and `GoodMood` act as concrete states.
- Each class provides its own implementation of the state dependent actions as defined by the virtual methods that are defined in the `Disposition` interface.
- To be able to distinguish between the execution of these methods, they display different messages.

References

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

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

[3] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice-Hall, New York, 3^{rd} edition, 2004.

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 10: UML State Diagrams

Contents

10.1 Introduction

UML State diagrams are used to describe the behaviour of nontrivial objects. State diagrams are good for describing the behaviour of one object over time and are used to identify object attributes and to refine the behaviour description of an object.

A state is a condition in which an object can be at some point during its lifetime, for some finite period of time [4]. State diagrams describe all the possible states a particular object can get into and how the objects state changes as a result of external events that reach the object [1]. The notation for state diagrams was first introduced by Harel [2], and then adopted by UML.

10.2 Notational Elements

A UML State diagram is a graph in the mathematical sense of the word. It is a diagram consisting of nodes and edges. The nodes can assume a variety of forms each with specific meaning, while the edges are labeled arrows connecting these nodes. In Section 10.2.1 we discuss the basic nodes. The basic nodes are called state nodes. In Section 10.2.2 we discuss the syntax for the edges. They are called transitions. To be able to model complicated transitions special nodes called control nodes are introduced in Section 10.2.3.

10.2.1 State Nodes

There are three kinds of state nodes; initial nodes, state nodes and end nodes.

Figure 1: Initial Node

Figure 1 shows the symbol used to indicate the initial node. It is a filled circle. The initial node is the starting point of a state diagram. It indicates the default state of an object whose behaviour may change over time. A state diagram may at most have one initial node. Although a diagram may be drawn without indicating the starting point it is considered good practice to always have a starting point.

Figure 2: End Node

Figure 2 shows the symbol used to indicate an end node. It is a filled circle with another open circle around it. It indicates where the system terminates. A system may be running infinitely while sometimes changing state. In such case its UML state diagram may have no end nodes. It is also permissible for a state diagram to have many final nodes. Since a state diagram is used to model behaviour that is dependent on events that may or may not happen, it is conceivable that the system may terminate in a variety of ways.

Figure 3: State Node

Figure 3 shows the symbol used to indicate a state. It is a rounded square. In a state diagram each state other than the initial state (initial node) and the final state (end node), should be named. The name of a state is a short descriptive name that can be used to refer to the state. The name of the state in this figure is `Active`. We discuss more detail about states in Section 10.2.4.

10.2.2 Transition edges

Figure 4: Transition Edge

Transition edges in a state diagram are used to indicate transition between states. If an object in a system changes state as a reaction to some event, it is indicated by connecting the current state with a target state with an arrow labeled with the name of the event that triggers such transition. The following detail may be shown on a transition in a UML state diagram. These are all optional i.e. if not required, they may be omitted.

- The event that triggers the transition (text). If the event shown on the transition is detected, the transition will fire and the state of the object will change to the state at the end of the transition. If a transition is shown without an event it triggers immediately after all actions that are associated with entering the state are completed.
- A guard condition that is a prerequisite for transition (text in [ ]). If the guard condition is true the the transition will fire, otherwise it will not fire. When there are more than one transition leaving the same node, the value of the guards my determine which one of them will fire. If no guard condition is shown for a transition, there are no preconditions required for the transition to fire.
- An action that is executed during transition (method name after /). Activities that are executed while moving from one state to the other is shown in the form of a method call on a transition label. Such method call must be preceded by a /. If the transition from one state to another is not associated with some action, no action is specified.

10.2.3 Control Nodes

There are two kinds of control nodes; decision-and-merge nodes to model alternate flows, and fork-and-join nodes to model parallel flows.

Alternate Flows

To model alternate flows, decision-and-merge nodes are used. They are diamonds. They are used both at the beginning and the end of alternate flows. The diamond at the beginning of an alternate flow is called a decision node, while the diamond at the end of the alternate flow is called a merge node. When a decision node is included in a UML State diagram guard conditions are required to indicate the conditions that determine which of the alternative flows will fire.

Figure 5 models a simple heating device. It sensors the current temperature in its sensing state. It remains in sensing state until one of the conditions of the decision node is true. The choice of which alternative flow to follow is determined by the guard conditions. If `[temp < 20]` is true, the top flow will trigger. The device will be turned on as indicated in the action on this transition. The device will remain in the `heating` state until the event `2 min elapsed` occurs. It will then follow through the merge node back to the `sensing` state where it will stay until one of the guard conditions become true. Similarly the device will be turned off when `[temp > 25]` is true whereafter the device will be in `idle` state for a while before returning to `sensing` state.

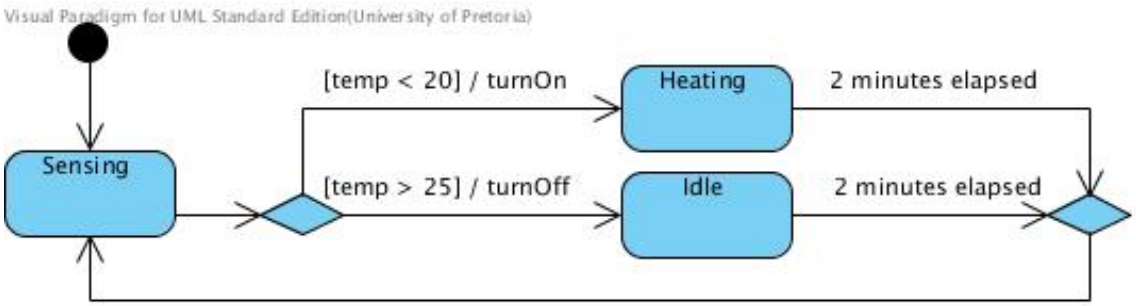


Figure 5: Heating device with alternate flows

Parallel Flows

To model parallel flows, fork-and-join nodes are used. They are heavy vertical or horizontal lines. They are used both at the beginning and the end of parallel flows. The node at the beginning of a number of parallel flows is called a fork node, while the node at the end where the parallel flows meet again is called a join node. When a fork node is included in a UML State diagram, all transitions leaving the fork node fire at the same time creating different threads that execute simultaneously. The join node is used where these parallel threads synchronise. It is also called a synchronisation point. The transition leaving a join node fires only after all threads in the parallel flows meeting at the join have reached the join.

Figure 6 models the stages of a system from implementation through testing to operation. This diagram indicates that the hardware and software testing are executed in parallel. The system will only enter its operation stage after both hardware and software testing are completed.

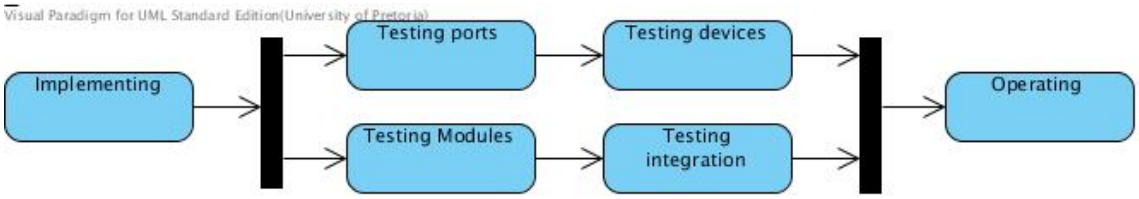


Figure 6: Stages of a system with parallel flows

10.2.4 Actions

Actions can be executed during transition as was explained in Section 10.2.2. It is also possible to indicate actions that are executed while the system is in a state. More often such actions are triggered on entering a state or when leaving a state. Figure 7 shows the syntax for indicating such actions. In this figure the `pickUp` action will execute when the `Receiving` state is reached and the `disconnect` action will execute before a transition to a next state is performed.

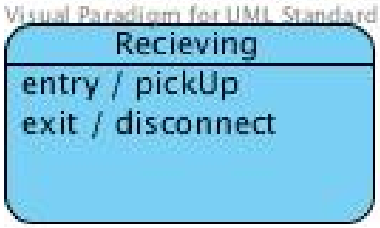


Figure 7: Actions that are executed in a state

10.2.5 Signals

Figure 8: Signal node

In event driven programming events are generated while a system is running. Many of these events are generated by the users of the system, for example when selecting a menu

option or clicking a button. Some of the events can also be generated by actions executed by the system. For example when a timer times-out or when some critical threshold is reached. These events can be modelled using a signal node. Figure 8 shows the syntax for a signal node. When a signal node is reached in a UML State Diagram an event named by the label on the signal node is generated. This event can trigger a transition in any other state in the diagram to fire as a result of this event. When a signal is reached, the event is generated and all transitions in the diagram that is associated with the event will trigger.

10.2.6 Composite States

A UML State Diagram can be nested in a state. The two diagrams in Figure 9 was taken from [5]. They model the same states. In the left diagram the UML State Diagram showing the sub-states of the `Check-PIN` state is shown in detail, while they are hidden in the diagram at the right. Notice how the ∞ symbol us used to indicate that a state is a composite state.

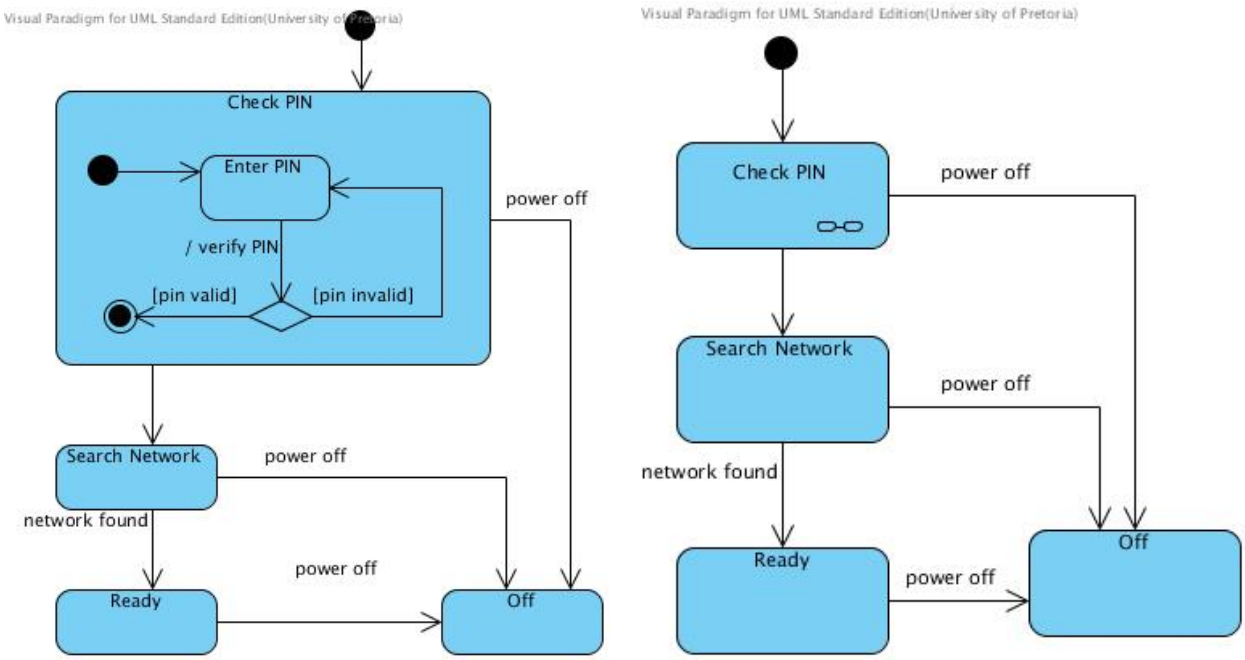


Figure 9: Composite

10.3 Examples

Figure 10 shows a UML State Diagram for a player's turn in Monopoly taken from [6]. It shows all the possible actions and conditions required before taking these actions while making transitions between the states one can be in during one's turn in the Monopoly game. This diagram serves as a good example by itself if you are familiar with the game.

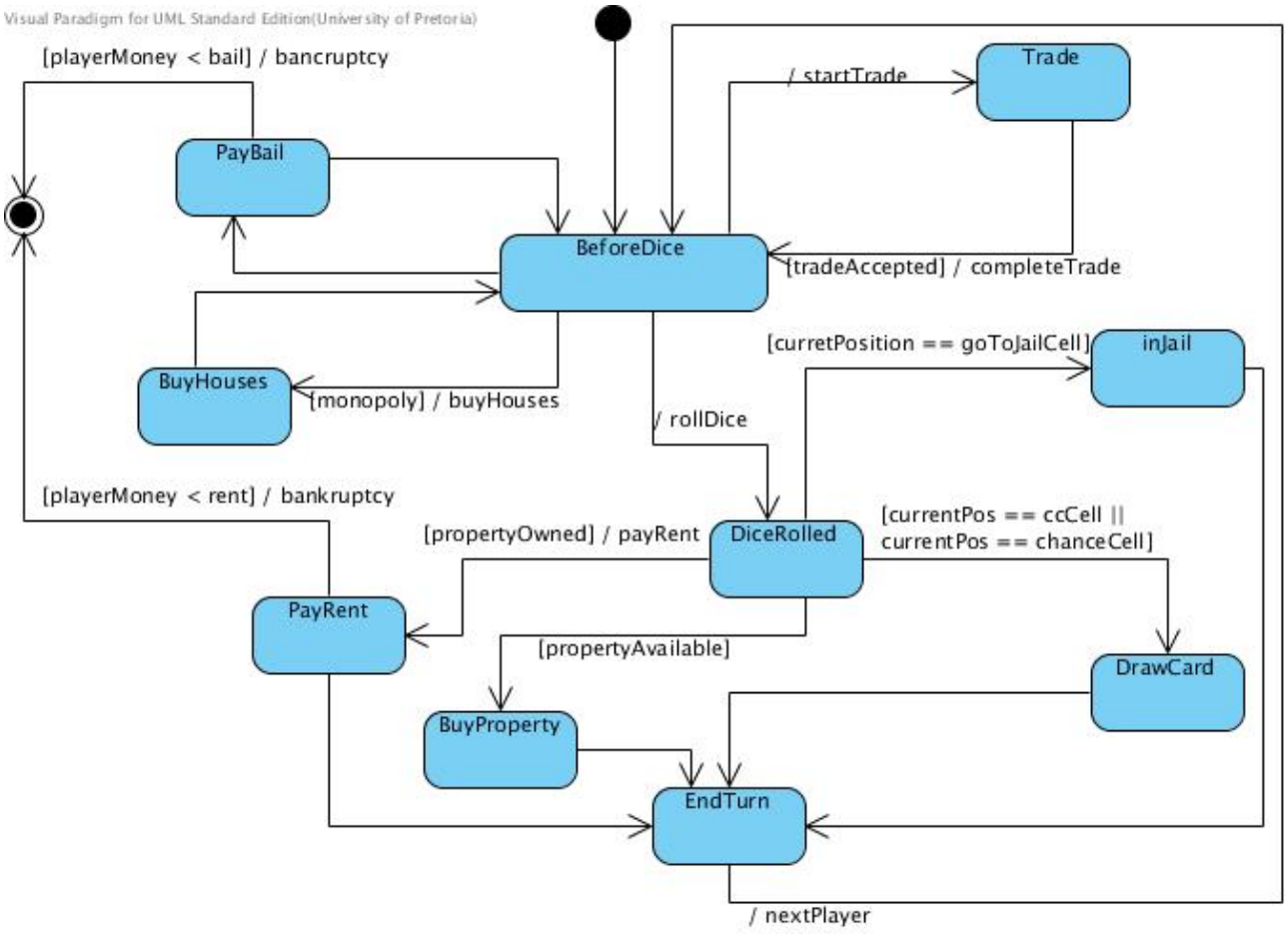


Figure 10: UML State Diagram for a turn in Monopoly

Figure 11 shows a UML State Diagram of a very fancy toaster for toasting bread taken from [3]. It has a timer like most of the common toasters we use in our houses today. In parallel with the timer it uses an on-off cycle similar to the device modelled in Figure 5. Along with all this it features a safety feature that monitors both the absolute colour of the toast as well as the change rate of the colour and will trigger a bomb-out (`Done`-signal) if one of these measures are not OK.

10.4 Exercises

1. Describe all actions and transitions that will execute in the diagram in Figure 10 if a player rolls the dice and lands on a property for which the rent is higher than the value of `playerMoney`.

2. Extend the diagram in Figure 11 model an `Unplugged` state. It should be the initial state.When a `plug-out` event occurs in any of the existing states it should immediately transition to this state. When a `plug-in` event occurs it should transition from `Unplugged` to `Idle` unless it's sensor detects that there is a slice of bread in it. In such a case it should start toasting the bread as if the `start` event had happened.

3. Draw a `UML State Machine` diagram to visualise the states of a torch that is operated with an on-off switch. When the torch is switched on, it shines yellow. It can only be switched off if the switch is turned off while the torch is shining red. The torch will start shining red when it is switched on while it was shining yellow. If the torch is switched off when shining yellow, it start shining white. If it is turned off while shining white, it starts shining yellow.

References

[1] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Reading, Mass, 2003. ISBN 0-321-19368-7.

[2] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[3] Robert C. Martin. Uml tutorial: Complex transitions. `http://www.objectmentor.com/resources/articles/cplxtrns.pdf`, 1998. [Online: Accessed 29 June 2011].

[4] Kendall Scott. *UML Explained*. Addison-Wesley, Boston, Massachusetts, 2001.

[5] n.a. Sparx Systems Pty Ltd. Uml 2 state machine diagram, 2001.

[6] Laurie Williams. An introduction to the unified modeling language: A picture is worth a thousand words. `agile.csc.ncsu.edu/SEMaterials/UMLOverview.pdf`, 2004. [Online; accessed 30-June-2011].

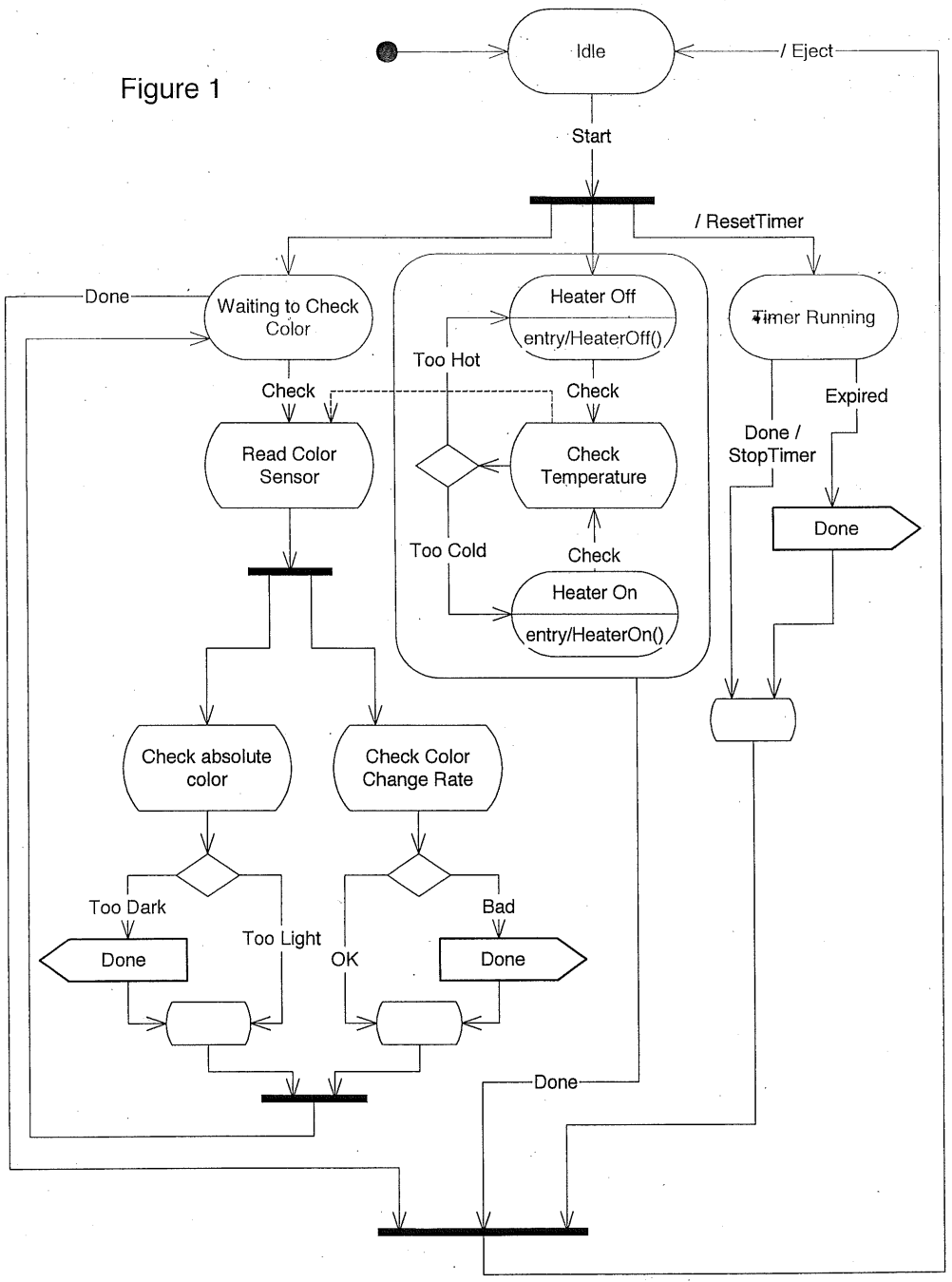


Figure 11: UML State Diagram modeling a toaster

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 11: Composite Design Pattern

Contents

11.1 Introduction

This lecture note will explain the structure of the composite design pattern. The composite design pattern effectively builds a structure of related objects in the form of a tree. These objects can be traversed by making using of the run-time polymorphic properties of object oriented programming. To successfully implement the composite design pattern design issues are considered and the clearing of the entire structure from memory after it goes out of scope. In order to address these issues the C++ Standard Template Library is introduced.

11.2 Programming Preliminaries

11.2.1 C++ Standard Template Library

The Standard Template Library (STL) for C++ is a generic library comprising of containers, algorithms, iterators and functors. The library relies heavily on compile-time polymorphism and therefore template programming constructs for the implementation of data structures and algorithms for multiple object types [4, 1].

Containers are data structures that contain other classes. The classes contained by the containers must at least be copy constructible and assignable. In order to apply algorithms on the containers and to a lesser extent iterators, the classes contained must also be less-than comparable. Stated in other words, the class needs to define a copy constructor, assignment operator and a less than operator. Muldner [3] refer to objects that are canonically constructible, meaning that they have a constructor, copy constructor, assignment operator and destructor defined in them. A good principle to follow when using the STL is to ensure that all objects are canonically constructible and that they are further less-than comparable (have a less-than operator defined).

Containers are classified as either sequence or associative in nature. Every element in a sequence container has either a predecessor and/or a successor with the exception of the first and last element. Examples of sequence containers are: `vector`s, `deque`s and `list`s. Associative containers, as their name refers to, exhibit a relationship between the elements of the container. This relationship may be in terms of another element or as a result of a "key" defined to access the object. Examples of associative containers are: `set`s and `multiset`s, and `map`s and `multimap`s. Other containers exist that make use of the above containers as base but provide a different interface.

In order to traverse containers, the STL provides iterators for the containers. These iterators are built on the principles that will be seen during the discussion of the iterator design pattern. The STL further provides algorithms, such as searching and sorting that along with the iterators can be applied to the containers. Functors or function pointers are pointers to functions and provide the functionality that a function may be sent as a parameter to another function.

In the section that follows two sequence containers in the STL will briefly be discussed. Choosing between the two is dependent on the application for which they are being used. A vector is similar to an array, but will dynamically change its size when elements are inserted or deleted. Elements are inserted at the back of the vector and removed from

Members Category	signature	Vector	List
Construction	constructor	y	y
	destructor	y	y
	operator=	y	y
Accessor	size	y	y
	empty	y	y
	front	y	y
	back	y	y
	operator[]	y	n
	at	y	n
Mutator	push_front	n	y
	pop_front	n	y
	push_back	y	y
	pop_back	y	y
	clear	y	y
Iterator	begin		
	end		

Table 1: Summary of vector and list members

the back as well. A list is optmised for insertion, deletion and moving elements around in the containers. Elements in a list are inserted or deleted either at the front or the back.

11.2.1.1 Summary of Vector and List container members

A summary of the members of the STL vector and STL list implementations are given in table 1. The members are categorised in the following categories: Construction/Destruction, Accessor, Mutator and Iterator. For each of these categories an indication of whether a specific member is implemented for the container or not is given.

Other members also exist for each of the STL containers. Further details regarding the containers are beyond the scope of this lecture note and can be found by searching the internet. There are also good examples regarding the use of the containers available on the internet.

11.2.1.2 Application to previous design patterns

Lists and vectors can easily be used by the caretaker of the Memento pattern in order to store the state of more than one object. They can also be effective be applied to the Abstract Factory example given in Chapter 7 so that all memory is cleared and the example presented in the lecture notes does not exhibit a memory leak.

11.2.2 Anonymous objects

In order to understand the concepts better, some C++ background regarding anonymous objects needs to be explained. An anonymous object is an object for which the code creating the object has not got a handle to the object. Consider the following example:

Listing 1: Example of anonymous object creation

```
1  class A {
2    // all the class stuff
3  };
4
5  class B {
6    public:
7      B(A* in) { a = in; };
8    private:
9      A* a;
10 };
11
12 // Some client code
13  B b(new A());
```

The object created and sent as a parameter to the constructor of class B is anonymous. The client has no way of accessing this object and therefor when it goes out of scope the object is not cleared from memory resulting in a memory leak.

The problem comes down to who takes ownership of the object. If the client keeps ownership, then the client must be able to acquire a handle to the object in order to be able to delete it. This can be achieved by defining a variable for the object `a` as follows and replacing line 13 in listing 1 with the following code:

```
A* a = new A();
B b(a);
```

Another possibility for the client to gain ownership is if class B defines a getter operation that returns a pointer to the object. For both these scenarios, it is now the obligation of the main program to delete the object from the heap before the program goes out of scope. As object `b` is on the stack, it will go out of scope when the main program terminates. The following code will ensure that object `a` is delete from the heap.

```
delete a;
```

Assuming that the design decision made for listing 1 is that ownership of the object is given to class `B`. This means that when object `b` goes out of scope the object of class `A` it points to must all be destroyed. Class `B` must therefore implement a destructor that provides this functionality as the default destructor does not remove objects from the heap memory by including the following code after line 7 of listing 1.

```
virtual ~B() { delete a; };
```

The destructor has been defined as `virtual` to ensure that if another class inherits from `B`, the destructor is still called when that class goes out of scope.

11.3 Composite Design Pattern

11.3.1 Identification

Name	Classification	Strategy
Composite	Structural	Delegation (Object)
Intent		
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. ([2]:163)		

11.3.2 Structure

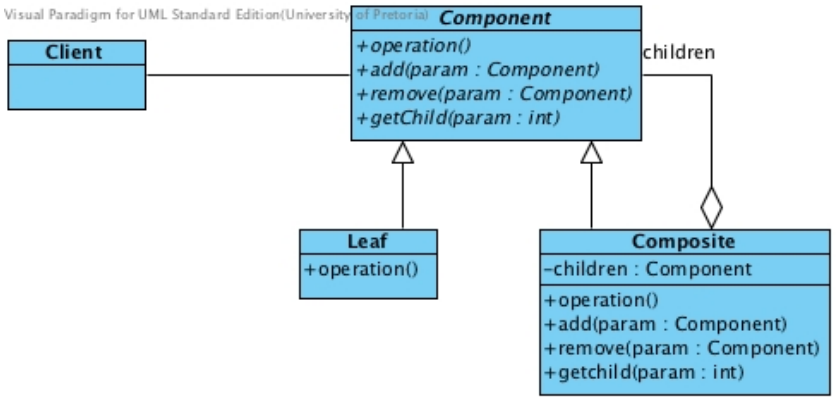


Figure 1: The structure of the Composite Pattern

11.3.3 Problem

Used in hierarchies where some objects are composite of other. Makes use of a store for the children defined by Composite

11.3.4 Participants

Component

- provides the interface with which the client interacts.

Leaf

- do not have children, define the primitive objects of the composition.

Composite

- contains children that are either composites or leaves.

Client

- manipulates the objects that comprise the composite.

11.4 Composite Pattern Explained

The composite pattern inherently builds a tree (refer to Figure 2) with the intermediate nodes being instances of composite and the leaf nodes being instances of the leaf participants.

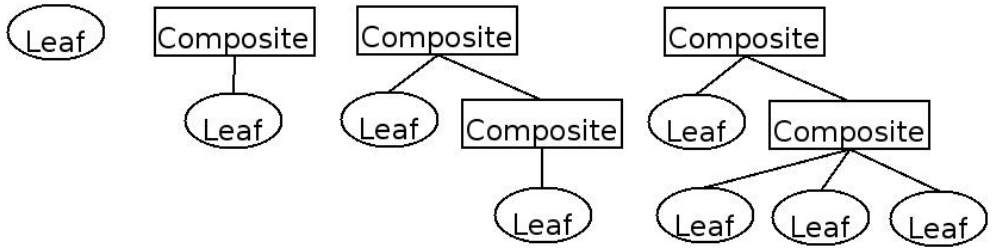


Figure 2: Examples of tree structures

11.4.1 Implementation Issues

During the design of the system a decision must be made regarding the destruction of the composite. This decision will influence how the client may construct objects and pass them to the composite. There are two options with regards to destruction of the composite, the first requires the client to take responsibility. The second destruction design leaves the responsibility of the destruction of objects up to the composite. Both these implementations have their advantages and drawbacks. If the client takes responsibility for the destruction of objects then no anonymous objects may be created. On the other hand, having the composite handle the destruction of objects will allow the client to pass anonymous objects to the composite. In this case the client will need to guarantee that it will not use any objects for which it still has handles after it has destructed the composite.

11.4.2 Related Patterns

Chain of Responsibility
Creates a structure that defines a component-parent link.

Decorator
Used in conjunction with components to add state to the components. When a decorator and a composite are combined, they usually share the same parent class.

Flyweight
Allows sharing of objects, particularly the leaf nodes

Iterator and Visitor
Used to traverse the composite structure.

11.5 Example

11.5.1 Tree

In this example a tree will be built in which each node of the tree is represented by an integer. The UML class diagram is given in Figure 3. The abstract `Tree` class defines operations that can be applied to the tree. In this case a Tree node can be added to the tree or the node can be printed.

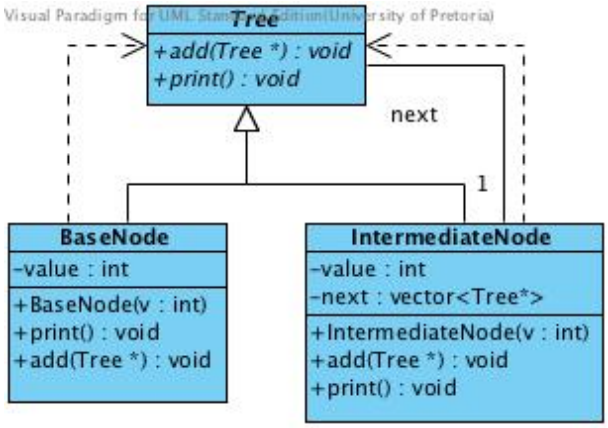


Figure 3: Tree example

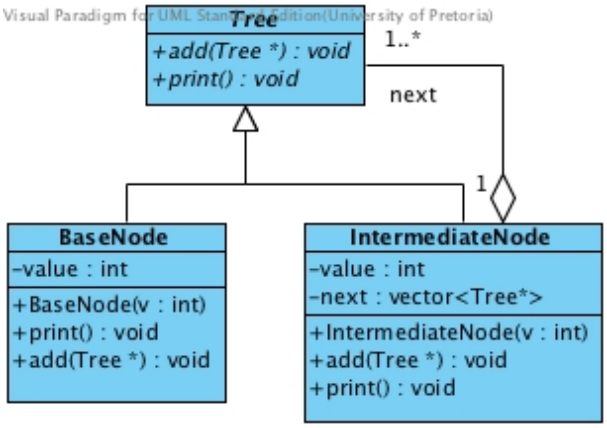


Figure 4: Tree example updated

When the association relationship between the composite participant `IntermediateNode` and the component participant `Tree` is drawn in Visual Paradigm, it is drawn as both a dependency and an association. The dependency is as a result of a `Tree*` being sent as a parameter to `add`, which also accounts for the dependency between `BaseNode` and `Tree`. The association is as a result of the vector `next` of `Tree*`. According to the structure of the pattern given in Figure 1 this association should be aggregation and it is not necessary to include the dependencies. The class diagram given in Figure 4 is more in line with the intended structure of the pattern and needs to be manually updated in Visual Paradigm.

Both the examples in Figures 3 and 4 require the client to take responsibility for the deletion of the objects, unfortunately this can lead to memory leaks. In the example in Figure 5, the deletion of the objects in the hierarchy is the responsibility of the composite.

The implementation of the destructor of the `Tree` class is empty as is that of `BaseNode`. The destructor of `IntermediateNode` must iterate through the `next` vector and delete each element is points to. The implementation of the `IntermediateNode` destructor, along with the implementations of all the other operations of the `IntermediateNode` class shown in Figure 5, is given in the listing below. This code illustrates how elements are inserted into the vector container as well as how to iterate through the container.

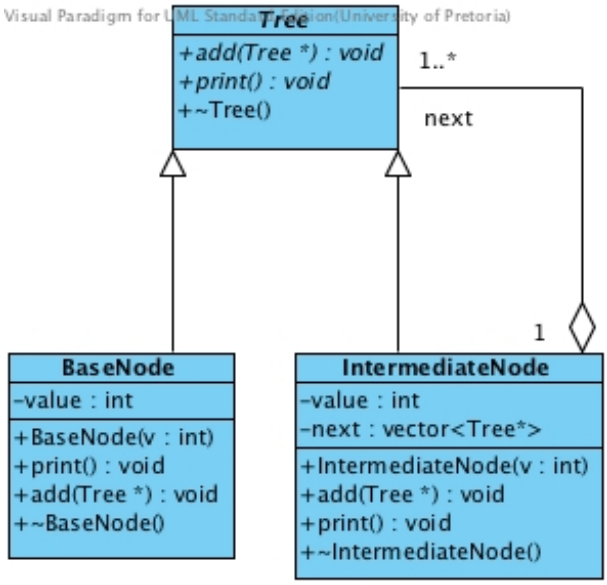


Figure 5: Tree example with the composite taking ownership of destruction

```
class IntermediateNode : public Tree
{
  public:
        IntermediateNode(int v);
        virtual void add(Tree*);
        virtual void print();
        virtual ~IntermediateNode();
  private:
        int value;
        vector<Tree*> next;
};

IntermediateNode::IntermediateNode(int v) : value(v)
{
}

void IntermediateNode::add(Tree* t)
{
        next.push_back(t);
}

void IntermediateNode::print()
{
        cout << "-" << value << "[";
        vector<Tree*>:: iterator it;

        for (it = next.begin(); it != next.end(); ++it)
        {
                (*it)->print();
```

```
        }
        cout << "]";
}


IntermediateNode::~IntermediateNode()
{
        vector<Tree*>:: iterator it;

        for (it = next.begin(); it != next.end(); ++it)
        {
                delete *it;
        }
}
```

11.5.2 Graphics

Consider the Shape Hierarchy that was introduced in the Abstract Factory chapter. Figure 6 serves as a refresher for this hierarchy.

From the hierarchy it is clear that the three (3) concrete triangle classes, the part of the hierarchy from Parallelogram down, and Ellipse and its subclass Circle will form leaf nodes of the Composite design pattern. The Shape class will fulfill the role of the Component participant and a new class that represents the composite participant must be designed. For this design, the area and perimeter of the composite will be determined by calculating the sum of the area and perimeter of its constituents respectively. Figure 7 provides the class structure for the composite shape hierarchy.

11.6 Exercises

1. Refactor the Tree example given in Figure 5 so that `int value` is defined in one place only.

2. How would you modify the composite participant to build a binary tree? A binary tree is a tree where the composite participant only has two children. These children may either be an instance of composite or a leaf participant.

References

[1] cplusplus.com. STL Containers, 2011. URL `http://www.cplusplus.com/stl/`. [Online; accessed 31 August 2011].

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

[3] Tomasz Müldner. *C++ Programming with Design Patterns Revealed*. Addison Wesley, 2002.

[4] Wikipedia. Standard template library, 2011. URL `http://en.wikipedia.org/wiki/Standard_Template_Library`. [Online; accessed 31 August 2011].

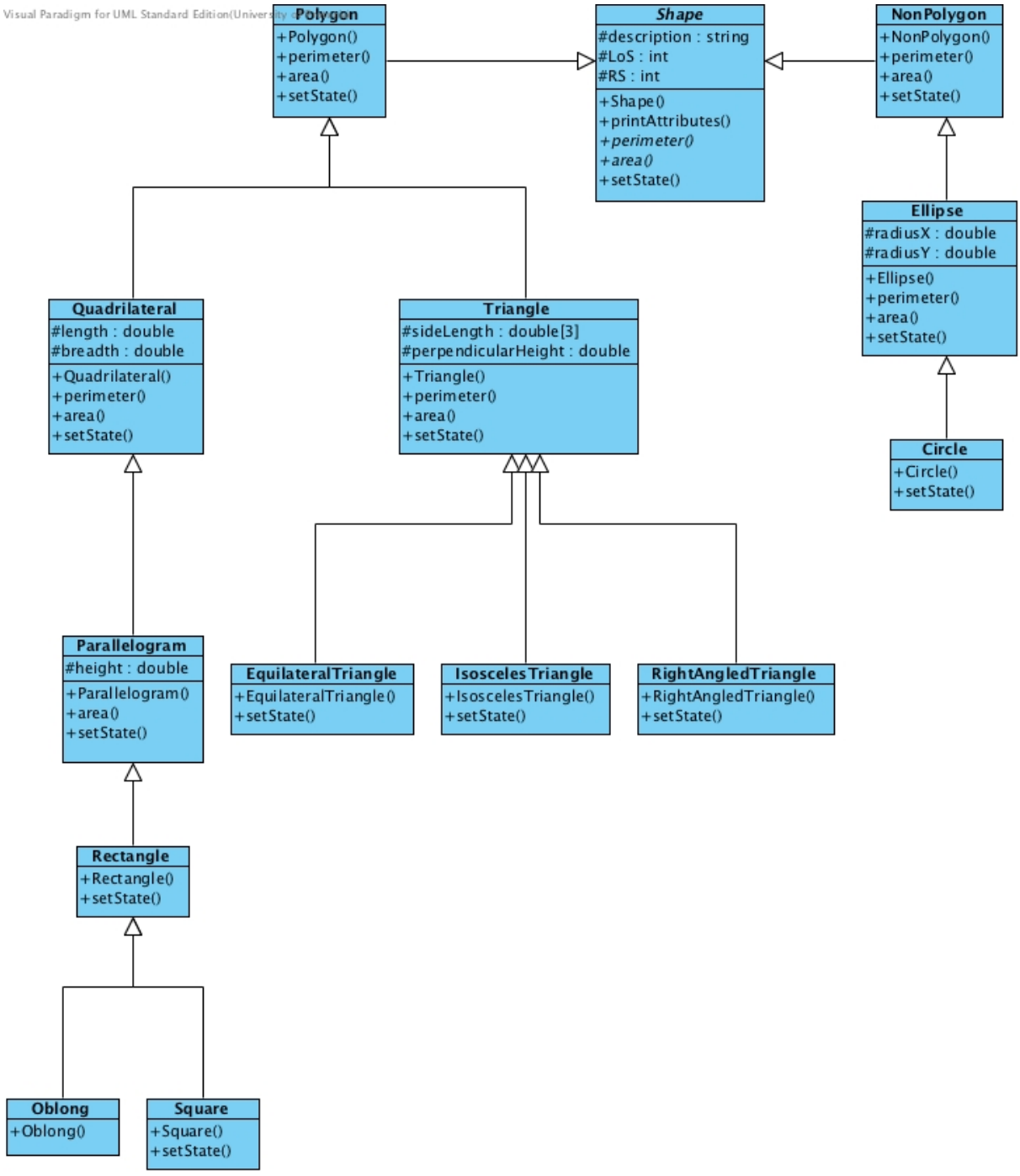


Figure 6: Shape hierarchy

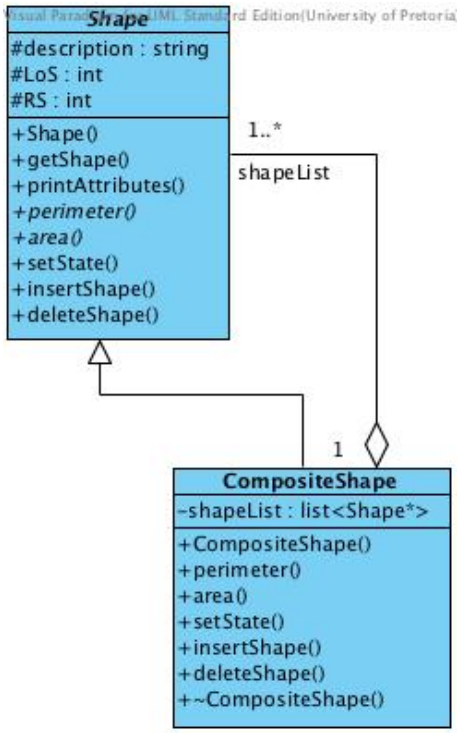


Figure 7: Composite Shape hierarchy

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 12: Decorator Design Pattern

Contents

12.1 Introduction

The decorator pattern is used to extend the functionality of an object without changing the physical structure of the object [2]. The extension is either in terms of elaborating on the state of the object or in terms of behaviour. A combination of both state and behaviour is also possible which implies that these extensions can be stacked on the object and rather than using subclassing, which is a compile-time solution, the extensions can be applied at runtime.

In the sections that follow, an overview of the structure of the decorator pattern will be given along with an explanation of how the decorator can be applied. The two examples presented will illustrate how the decorator can be implemented. The first example will decorate the composite tree developed in Lecture note 14, while the second example will show how decorations of a till slip can change how the till slip is structured.

12.2 Decorator Pattern

12.2.1 Identification

Name	Classification	Strategy
Decorator	Structural	Delegation (Object)
Intent		
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. ([1]:175)		

12.2.2 Structure

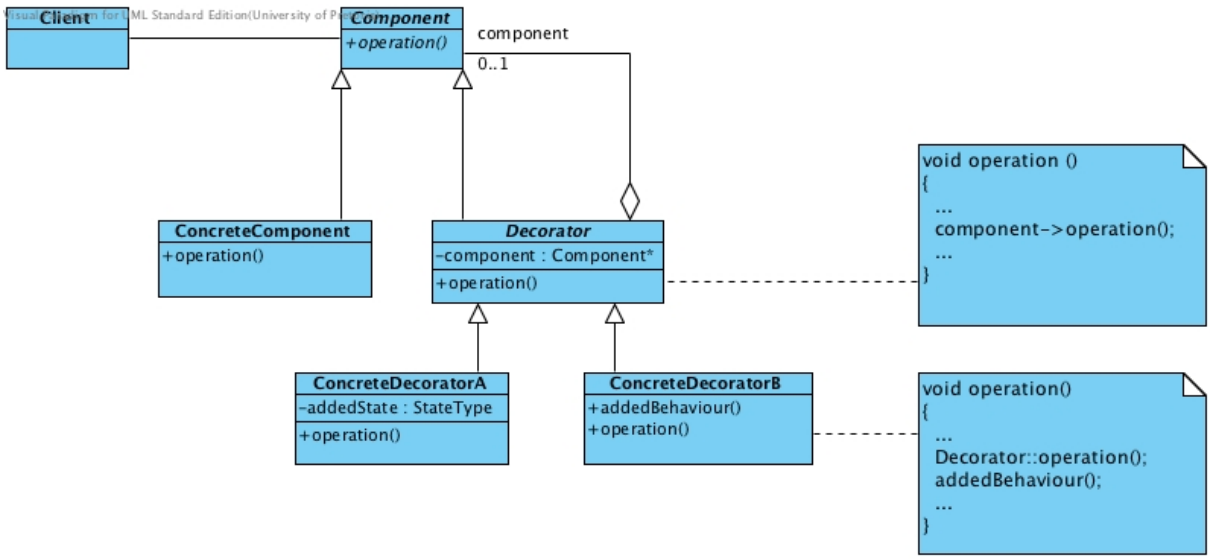


Figure 1: The structure of the Decorator Pattern

12.2.3 Participants

Component

- interface for objects that can have responsibilities dynamically added to them

ConcreteComponent

- the object to which the additional responsibilities can be attached

Decorator

- defines a reference to a Component-type object

ConcreteDecoratorA

- adds state-based responsibilities to the component

ConcreteDecoratorB

- adds behavioural-based responsibilities to the component

12.3 Decorator Explained

The structure of the Decorator is similar to the Composite. The main differences are the number of components related to; and the specialisations the composite and decorator may have. The composite comprises of multiple components, while decorators may or may not comprise of a component. The composite class is defined as a concrete class, while the decorator class is abstract and concrete decorator participants specialise the decorator. The second difference ensures that the composite builds a tree data structure, while the decorator only a list data structure.

As with the composite, it is the concrete components that are to be decorated and there may be multiple of these. A single concrete component object may also have more than decorator instance applied to it. Figure 2 provides a few combinations of decorators that may be applied to the concrete component class.

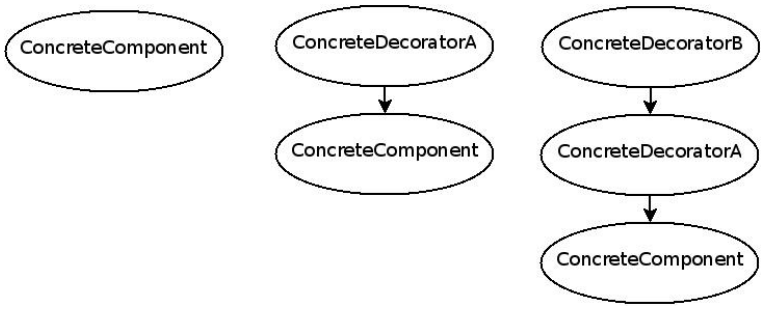


Figure 2: Examples of list structures of objects created by the Decorator

It is conceivable that a list may be decorated with the same concrete decorator more than once, it is however not always practical. The order of the application of the concrete

decorator should also be independent of one another and the net effect should be the same. The reason being that decorated objects should behave as if defined as a single large object with all the additional responsibility embedded in it.

12.3.1 Code improvements achieved

The advantage of applying the decorator design pattern is that objects of the concrete component provide the basic functionality expected of such a component. Any additional responsibility, be it state-based or behavioural-based, can be seen as adding value to the object, but is not embedded in the object. This design separates the concerns of required functionality and "nice to haves".

12.3.2 Implementation Issues

Two types of concrete decorators are defined, those that add state-based responsibilities and those that add behavioural-based responsibilities. It is easier to implement the state-based concrete decorators than it is to implement the behaviour-based responsibility version. It is also conceivable that both these types of responsibilities are included in a single concrete decorator class.

The same issues, as with Composite, arise when dealing with anonymous references.

12.3.3 Related Patterns

Adapter
: Changes the interface to an object while the Decorator only changes responsibilities.

Composite
: A Decorator can be seen as a Composite with only one component that has added responsibility.

Strategy
: The Strategy pattern changes the inner workings of an object while the Decorator changes the looks.

12.4 Example

12.4.1 Tree

To decorate the `BaseNode` of the tree example presented in Chapter 11, the decorator pattern is applied to the `Tree` and `BaseNode` classes as shown in Figure 3. Notice that destructors have been added to the hierarchy in order to ensure that the decorator clears the memory when the first object in the list goes out of scope. The destructors for the classes `Tree`, `BaseNode`, `BehaviourDecorator` and `StateDecorator` are all defined as virtual and an implementation with no statements is provided. The destructor of the `Decorator` class deletes the instance referred to by the attribute `component`.

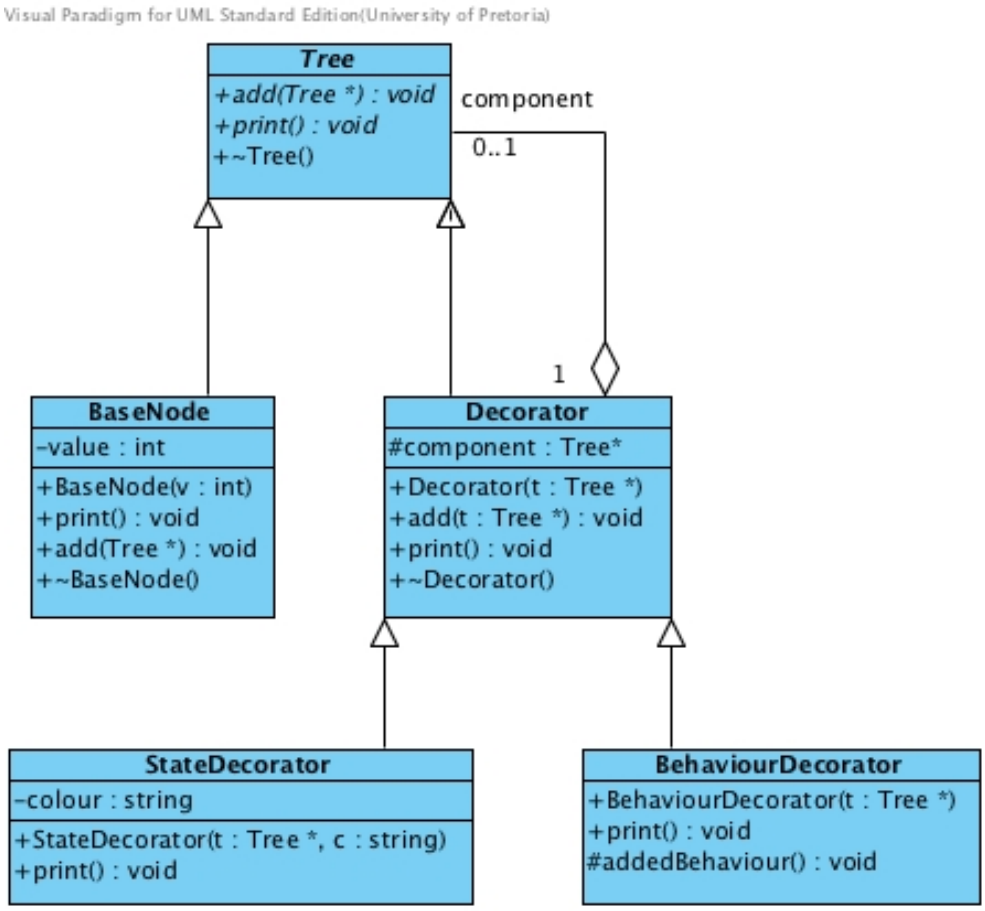


Figure 3: Decorating the Tree: showing only the decorator pattern

Both the `print` functions defined in the concrete decorator participants make calls to the parent `print` function to ensure that all chained prints are executed. Sample implementations of the `print` functions are given.

```
void Decorator::print()
{
    component->print();
}

void StateDecorator::print()
{
    cout << "!" << colour << "-";
    Decorator::print();
    cout << "!";
}

void BehaviourDecorator::print()
{
    addedBehaviour();
    Decorator::print();
}
```

Figure 4 shows how the Composite and Decorator design patterns can be used together. It is now possible to decorate the composite participant, `IntermediateNode`, as well.

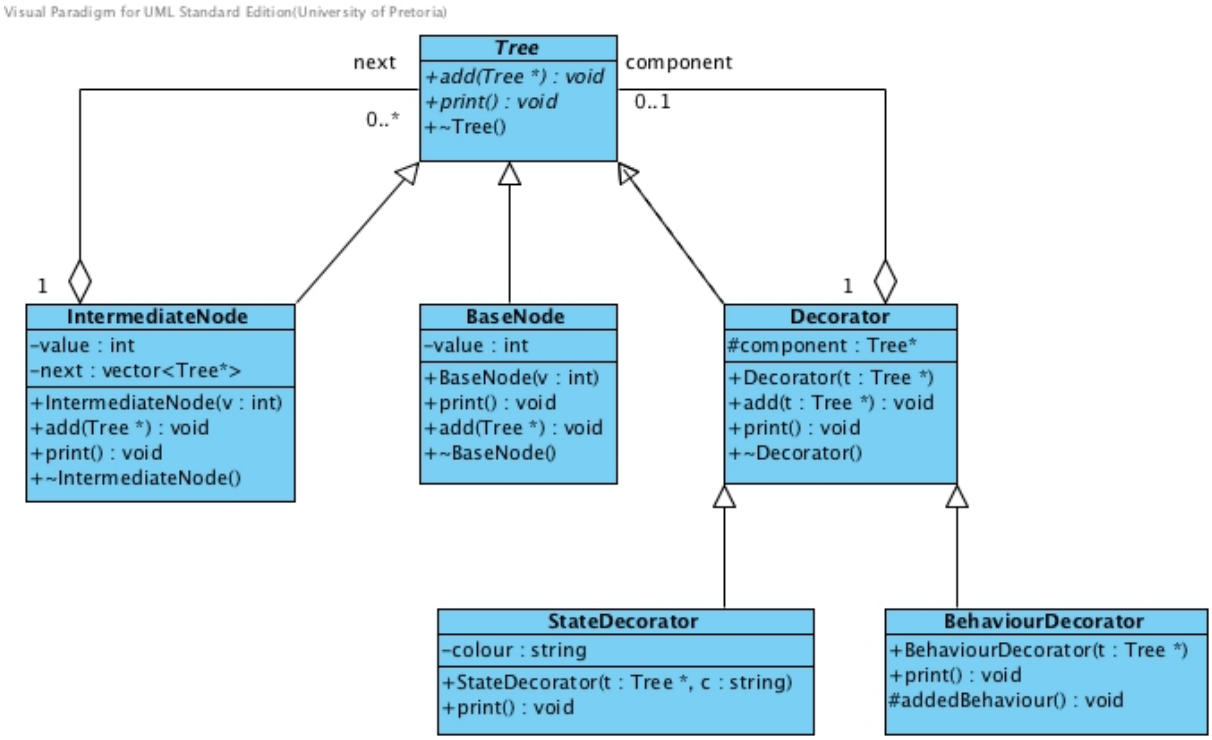


Figure 4: Decorating the Composite Tree

12.4.2 SalesTicket

This example illustrates how the decorator can be applied to change the "look" of a till slip (also referred to as a sales ticket) and customise it for a particular situation. A typical till slip has a header section where the name of the shop is printed, a body where a list of purchases are given and a footer with some friendly message or information. The basic functionality of the till slip is to provide the customer with the items listed in the body of the slip. The shop name displayed in the header and the greeting printed in the footer are "nice to have" and provide an individual identity for the till slip. These added responsibilities can easily be included by decorating the till slip with a header and a footer that is customisable for the particular shop.

Figure 5 presents the UML class diagram for the description of the till slip given above. The class `SalesTicket` represents the *ConcreteComponent* participant of the design pattern. `SomeClass` represents the *Decorator participant* and the classes `Header1`, `Header2`, `Footer1` and `Footer2` the *ConcreteDecorator participant*. *SalesOrder* represents the client for the design pattern.

Understanding how the pattern works can be tricky and therefore some coding aspects of the pattern are highlighted, specifically how `printTicket` is implemented for the participating classes. The `printTicket` of the `SalesTicket` class prints the body of the till slip. `SomeClass` first checks whether the `Component` has been decorated before it called the relevant `printTicket` function for linked component. The functions for both the *Header*

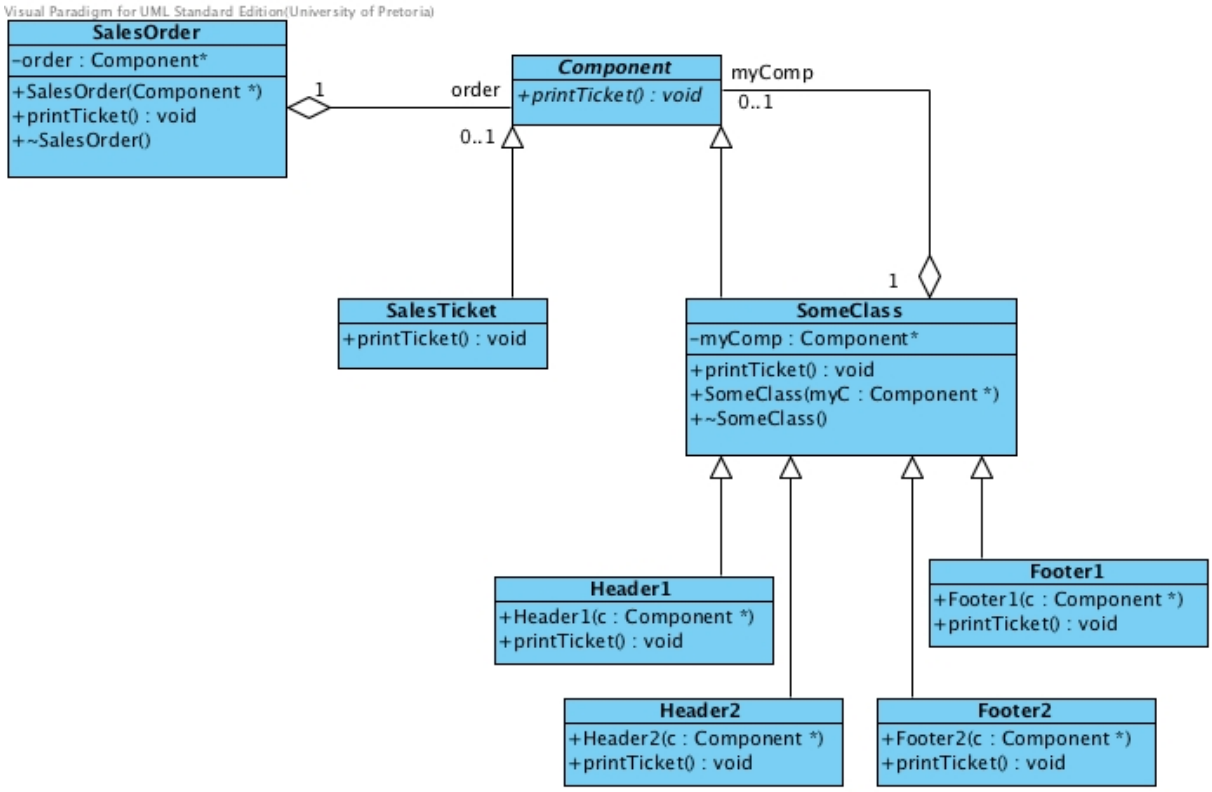


Figure 5: Printing sales tickets with the decorator

classes must first print their message before passing the printing on to the next component. The *Footer* classes do this in reverse to ensure that the relevant text is displayed at the bottom of the till slip.

```
void SalesTicket::printTicket()
{
    cout<<"Cash Sale Ticket"<<endl;
    cout<<"List of items purchased"<<endl;
    cout<<"Item"<<'\t'<<"Quantity"<<'\t'<<"Price"<<endl;
    // print the items out
    cout<<"TOTAL:"<<endl;
}

void SomeClass::printTicket()
{
    if (myComp)
        myComp->printTicket();
}

void Header1::printTicket()
{
    cout<<"Welcome to the Crazy Zone"<<endl;
    SomeClass::printTicket();
}
```

```
void Footer1::printTicket()
{
    SomeClass::printTicket();
    cout << "It was a pleasure doing" << " business with you"<<endl;
}
```

12.5 Exercises

1. Is it possible for the Composite design pattern to be restricted to build a list data structure? Explain.

2. For the combination of the Decorator and Composite given in Figure 4, identify the participants of both patterns.

3. In the sales ticket example given in Figure 5 it is possible to construct the concrete classes using the default constructor. Doing so will cause memory problems within the classes. Explain how you would go about to fix the problem.

References

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

[2] Wikipedia. Decorator pattern, 2012. URL `http://en.wikipedia.org/wiki/Decorator_pattern`. [Online; accessed 27 August 2012].

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 13: UML Sequence diagrams

Contents

13.1 Introduction

UML 2.0 includes a number of interaction diagrams. These are sequence diagrams, interaction overview diagrams, timing diagrams and communication diagrams. In this lecture we will look specifically at sequence diagrams. They are used to model how objects interact with one another in terms of the messages they pass to one another. While all interaction diagrams model these interactions, sequence diagrams emphasise the order of the messages over time.

The way in which object oriented programs systems produce useful results is mainly through passing messages between objects. These messages appear in the form of method calls. A sequence diagram can be used to model the order in which methods are executed as a reaction to some event.

13.2 Notational Elements

We show the notational elements of sequence diagrams in terms of the interaction between two simple classes in the class diagram shown in Figure 1. The Integer class is a wrapper for an integer value. Its getter simply returns its current value while its setter passes a double value which is rounded before it is converted to an integer. The constructor of these classes initialise their respective instance variables to 0. In the case of `Client`, its instance variables are initialised to NULL pointers.

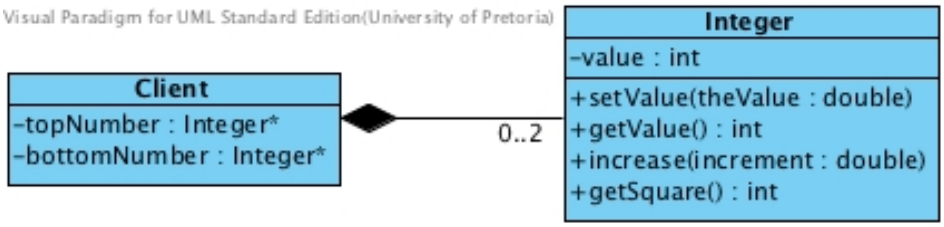


Figure 1: A class containing pointers to another class

13.2.1 Frames

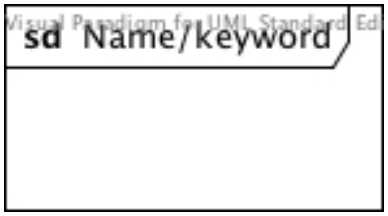


Figure 2: Sequence diagram frame

Sequence diagrams are drawn in frames. Figure 2 shows a frame of a sequence diagram. A frame is a rectangle with a heading in a compartment in the top left corner. The compartment for the heading is drawn as a rectangle with the lower right corner cut off.

The heading is used to name a diagram or to indicate the scope of loop structures (Section 13.4), conditional statements (Section 13.3) and parallel flows (Section 13.5) within a diagram. If it is used to name a diagram it is recommended that the name describe the essence of the interaction modeled in the diagram. When the frame is used within a diagram to show the scope of some subsection containing non-sequential flows, the heading should contain the appropriate keyword.

13.2.2 Lifelines

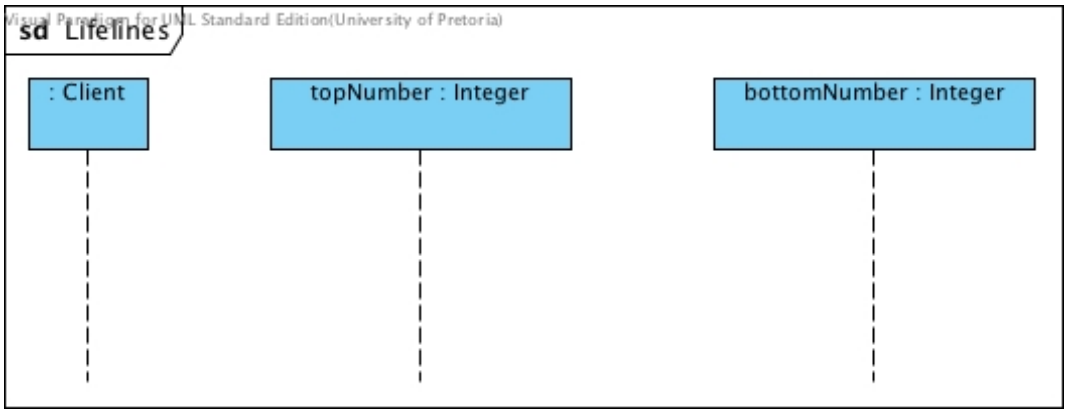


Figure 3: Lifeline notation

In sequence diagrams lifelines represent object instances. They are vertical lines inside the frame. Figure 3 shows three lifelines of objects that might be instantiated in our example. The rectangle at the top of each vertical dashed line identifies the participating object. The syntax is the same as for objects in object diagrams; an object name, a colon and a class name. In this figure the instance of the `Client` class is an anonymous object. Therefore it has only the class name in its identifier. Note that a colon precedes the class name in all cases.

The order of these lifelines is not significant, but by convention general flow starts at the leftmost lifeline and there is a general flow of messages across the diagram from left to right.

13.2.3 Creation and Destruction

It is important to realise that objects in a sequence diagram are *instantiated* instances of classes in a system. Upon creation of the `Client` class the pointers to `topNumber` and `bottomNumber` are initiated to NULL pointers. Therefore, the lifelines of these objects should appear only after they are created.

Assume the following is the main method executed by the `Client` application:

```
int main()
{
        topNumber = new Integer();
        delete topNumber;
        return 0;
}
```

Figure 4 shows how this program should be modeled. A creation action is shown with a dashed arrow with lined arrowhead to the identifier of a new lifeline. The arrow is labelled with the signature of the constructor that is called. Destruction is shown with an unlabeled arrow with filled head from the object that sends the destruction message to the point of destruction of the destructed object. The pont of destruction is shown with a heavy cross at the end of the lifeline.

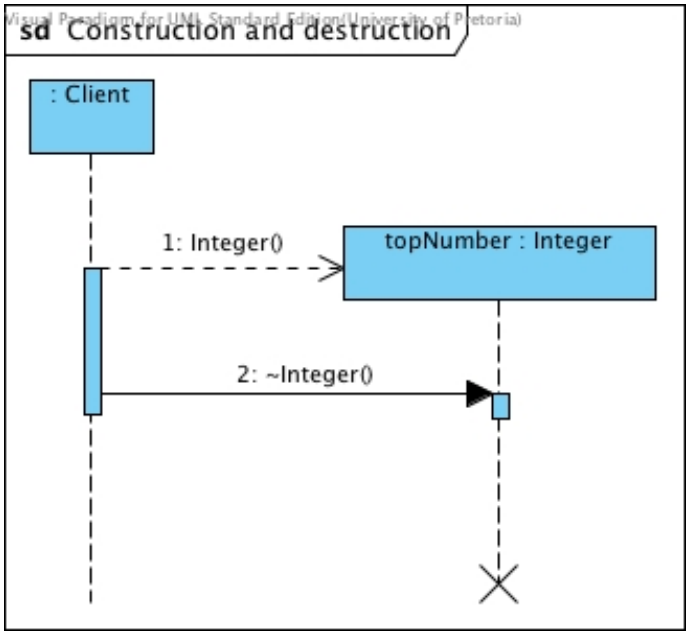


Figure 4: Construction and destruction

13.2.4 Messages

We will show simple messages that can be executed by instances of the two classes in the class diagram in Figure 1.

Assume the following program fragment executed by the `Client` application after `topValue` was created:

```
double aValue = 2.66;
topNumber->setValue(aValue);
```

Figure 5 shows how this interaction is modeled in a sequence diagram. The only interaction modeled here is the call to the Integer instance called `topNumber` using its setter. We assume that the creation and destruction time is not relevant in our model. This is

a simple method call with no return value. This notation should be used for all method calls using methods with void return values.

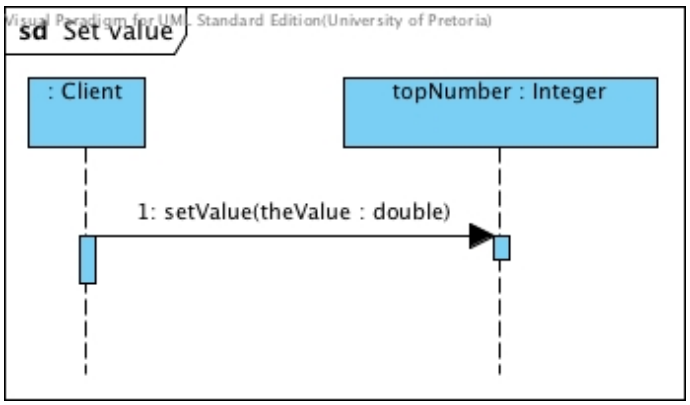


Figure 5: Message call without return value

Note that the label on the message in the diagram is the method signature and does not contain reference to the variable that was used when this message was sent.

Next assume the followingline of code is executed by the `Client` application:

```
int aValue = bottomNumber.getValue();
```

Figure 6 shows how the interaction of this line of code is modeled in a sequence diagram under the assumption that `bottomNumber` was properly instantiated. The interaction that is modeled here is a call to the Integer instance called `bottomNumber` using its getter. This is a method call with a return value. The return of a value is shown using a dashed arrow. This notation should be used for all method calls using methods with return values.

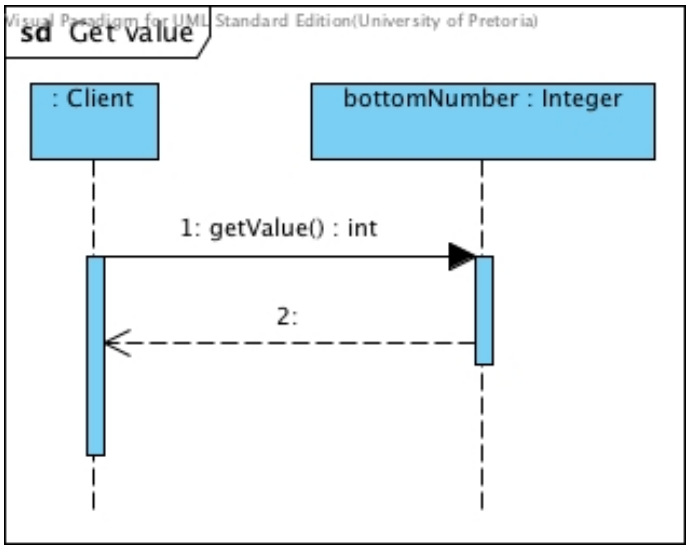


Figure 6: Message call with a return value

13.2.5 Reflexive messages

A reflexive message, referred to as *self message* in Visual Paradigm, is when an object calls a method that is defined in its own class. Typically methods with private scope can only be called by objects that are instances of the class that implements the method. This is the case with the `increase(:double)` and `getSquare():int` methods defined in the `Integer` class in Figure 1. `getSquare():int` has an `int` return value while the return value of `increase(:double)` is `void`. Figure 7 illustrates how these method calls are modelled.

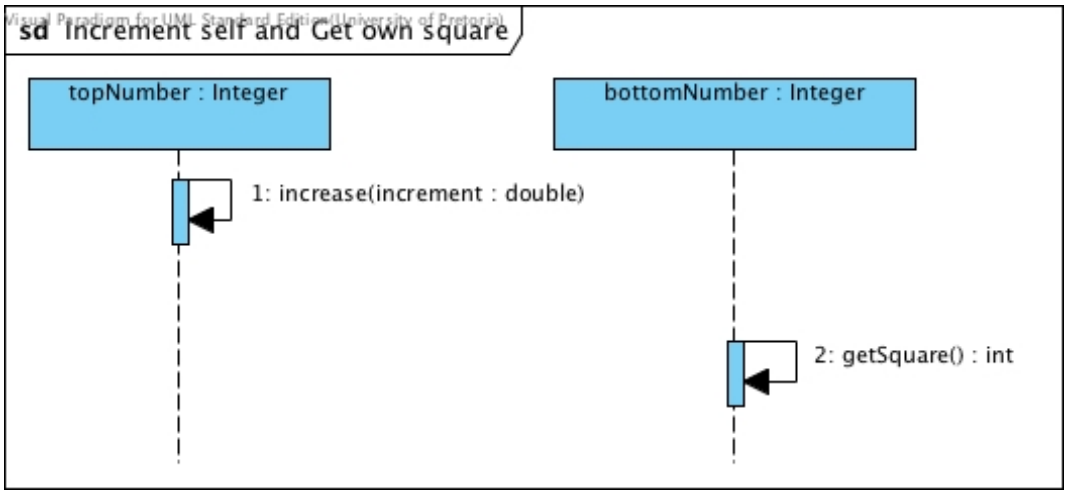


Figure 7: Reflexive message call without a return value and with a return value

13.2.6 Example

We will show an example of interaction between instances of the classes in the class diagram in Figure 8. This application implements the Strategy design pattern. These classes are used in a simple text-based two-player game where the user can select the strategy for his dragon and attack his opponent's dragon.

The modeling of interaction in the system requires knowledge of the implementation of the methods involved in the interaction. The following code shows the implementation of the methods we model in this example:

```
void Dragon :: attack(Dragon* enemy)
{
    strategy->fight(enemy, attackPower);
    float enemyPower = enemy->getAttackPower();
    lifeForce -= strategy->getRecoil(lifeForce, enemyPower);
}

void Dragon :: receiveInjury(float enemyPower)
{
    attackPower
```

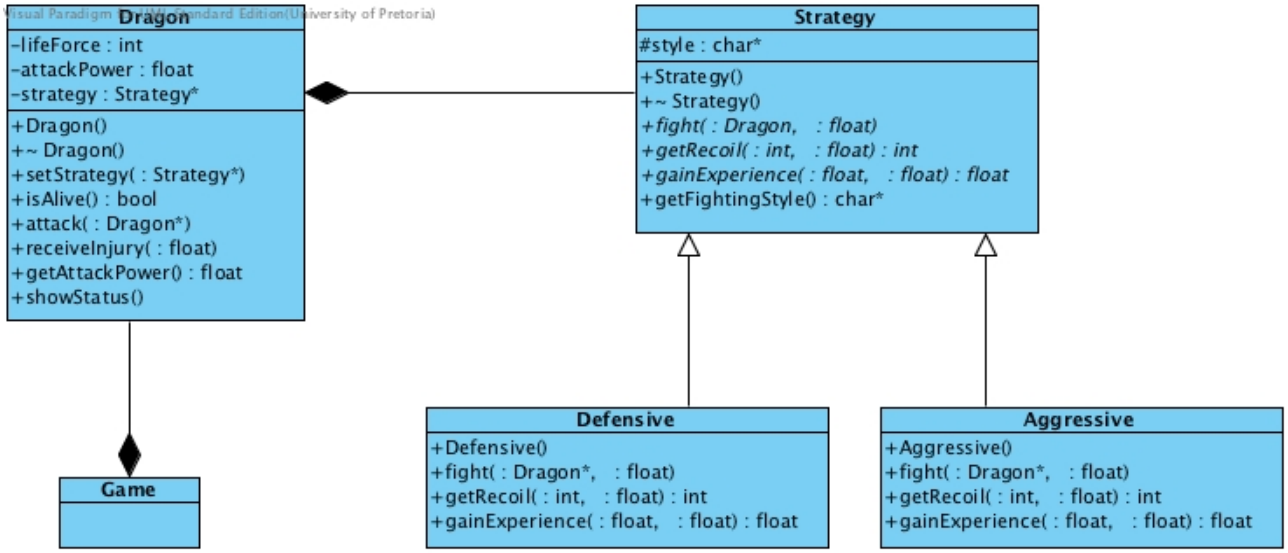


Figure 8: An application of the Strategy Design Pattern

```
    = strategy->gainExperience(attackPower, enemyPower);
}


void Aggressive::fight(Dragon* d, float ownPower)
{
    int num = rand();
    d->receiveInjury(ownPower * (num % 2 + 2));
}

float Defensive::
gainExperience(float ownPower, float enemyPower)
{
    return (ownPower + enemyPower/2);
}
```

Assume the that two dragons (`norbert` and `smaug`) have been instantiated. Further assume that the following statements have already been executed:

```
norbert->setStrategy(new Aggressive());
smaug->setStrategy(new Defensive());
```

Figure 9 shows how the interaction resulting from the following statement is modeled in a sequence diagram.

```
norbert->attack(smaug);
```

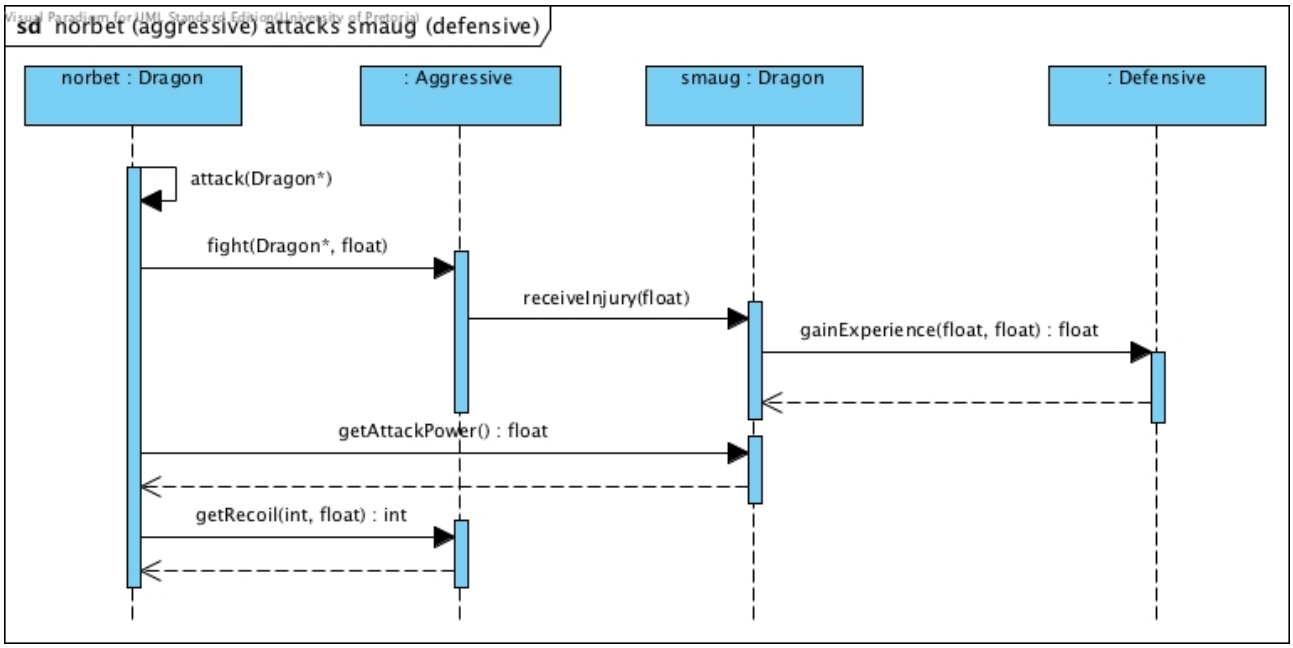


Figure 9: Example of interaction between multiple objects

13.3 Branching

13.3.1 Notation

Branching happens when the program flow contains conditional statements. Figure 10 shows the notation to model an alternate structure. A typical `if` or `switch` statement is modeled by showing the alternative interactions in different sections in a frame with the keyword `alt` in its header. The different sections are separated with dotted lines. The condition that needs to be true for a section to execute is shown as a guard in the top left corner of each section. One may have as many sections as needed. If there is only one section, the frame may alternatively be labeled with the keyword `opt`.

13.3.2 Example

Figure 11 models the interaction for an application for which a `ConnectionController` object has to open up a connection to a `Modem` object. This example was adapted from [2]. If the modem does not respond within a certain amount of time signaled by a `Timer` object, the operation times out and an error dialogue box is created and displayed. If the modem responds, the timer is cancelled and the modem is initialised.

13.4 Iteration

13.4.1 Notation

Iteration happens when the program flow contains looping statements. Figure 12 shows the notation to model recurrence. A typical `for` or `while` statement is modeled by showing

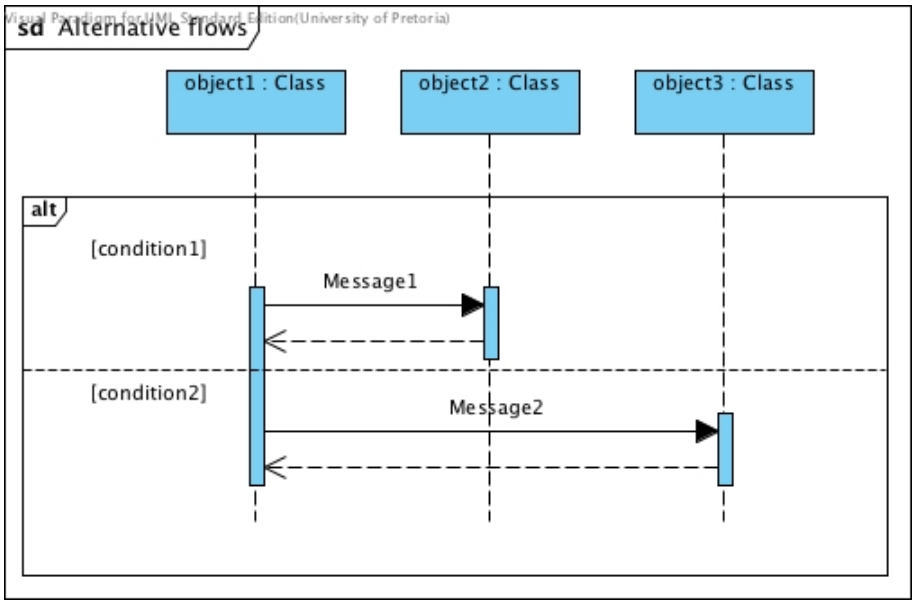


Figure 10: Syntax for alternate flows

the repeated interactions in a frame with the keyword `loop` in its header. The condition that needs to be true for the interactions n the frame to be executed is shown as a guard in the top left corner of the frame. In this example `message 1` returns its result only after `message 2` had completed all its iterations.

13.4.2 Example

The sequence diagram in Figure 13 contains a loop fragment. This example is taken from [1]. The guard in the loop fragment tests to see if the value of `hasAnotherReport` equals true. If the hasAnotherReport value equals true, then the sequence goes into the loop fragment, else the flow proceeds to the first event after the loop. In this case it is the termination of the `getAvailableReports()` message execution. The content of the loop fragment is interpreted the same way one would follow the messages in the loop as you would normally in a sequence diagram.

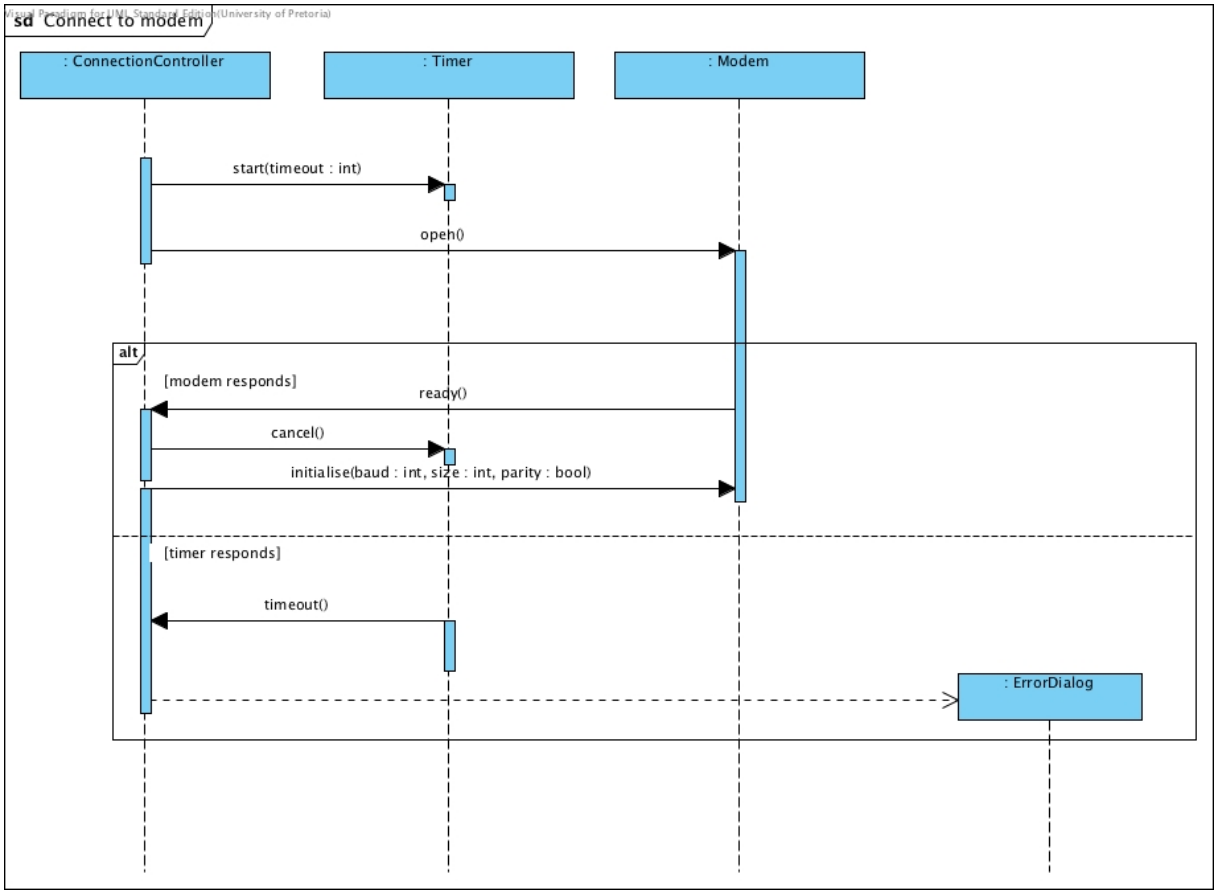


Figure 11: Connection to a modem with alternate flows

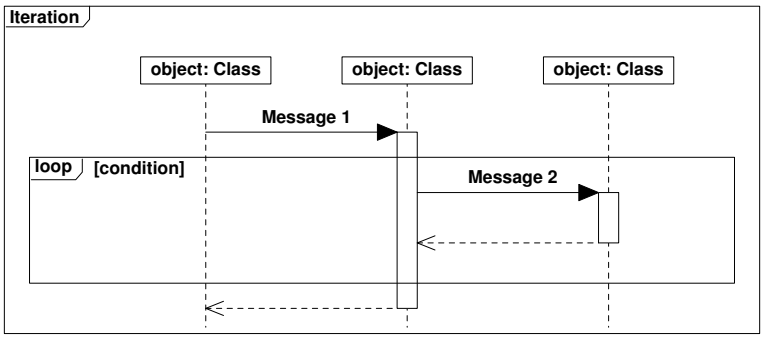


Figure 12: Syntax for a loop structure

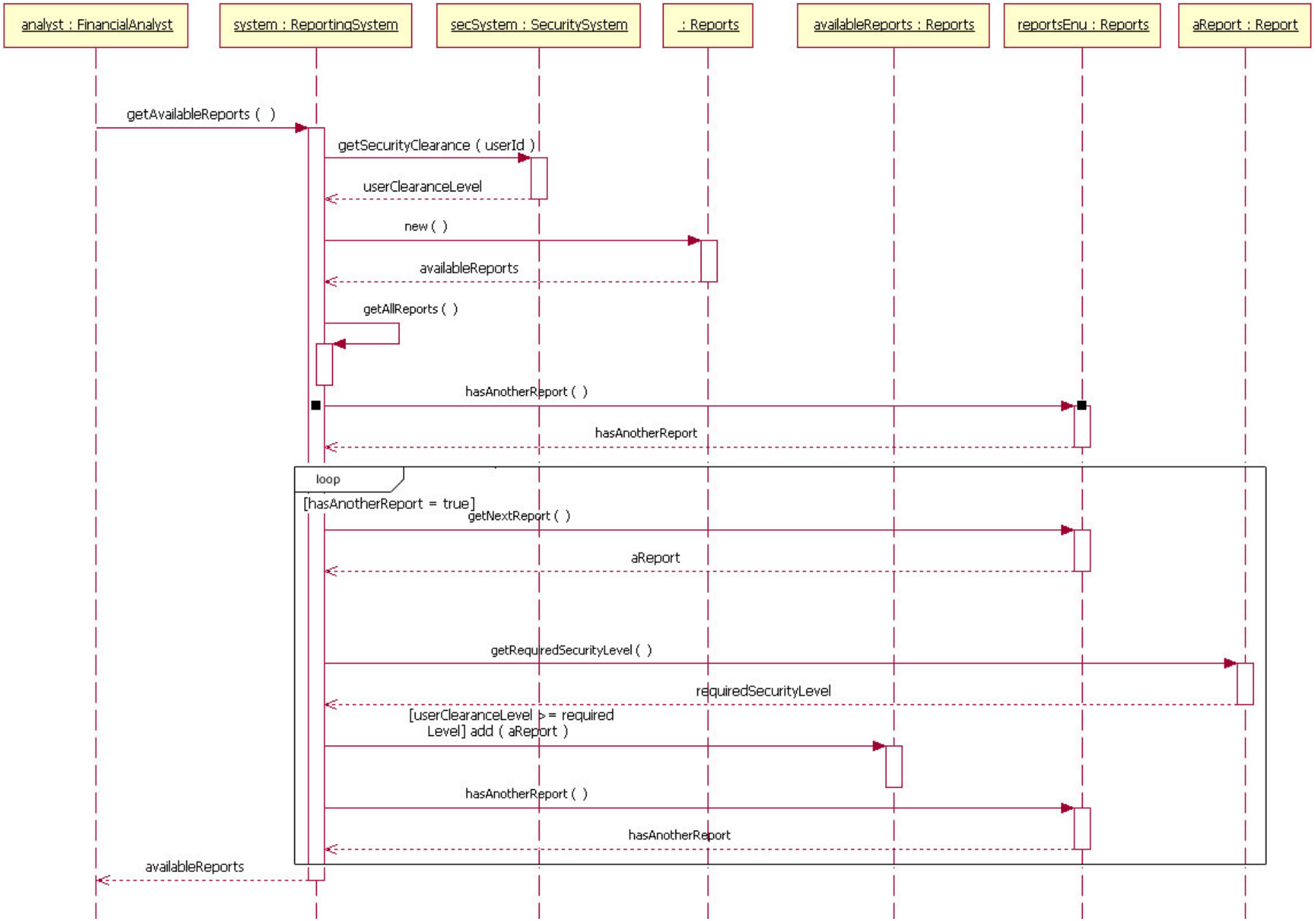


Figure 13: Get all reports

13.5 Parallel actions

13.5.1 Notation

It is also possible to model interactions that are executed at the same time in parallel. Figure 14 shows the notation to model this. It is modeled by showing the parallel interactions in different sections in a frame with the keyword `par` in its header. The different sections are separated with dotted lines. In this case all the sections will be executed at the same time and execution of the interactions after the parallel fragment commences after all the parallel threads has completed their actions. One may have as many sections as needed.

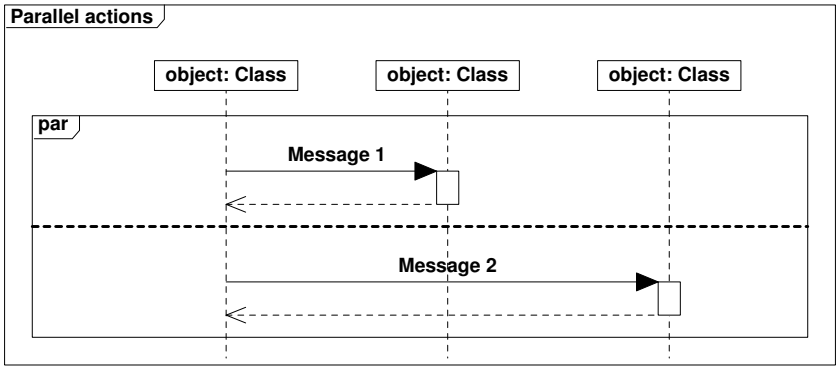


Figure 14: Syntax for parallel actions

13.5.2 Example

The sequence diagram in Figure 15 contains concurrent events as well as a condtional statement. This example was adapted from [3]. When a transaction is created, it creates a transaction coordinator to coordinate the checking of the transaction. This coordinator creates a number (in this case, two) of transaction checker objects, each of which is responsible for a particular check. These checkers are called asynchronously and proceeds in parallel. Thus, the top of the two subframes contains parallel flows.

When a transaction checker completes, it notifies the transaction coordinator about its success whereafter the controller destroys it. The coordinator continuously looks if checkers calls back and will only continue with operation once all the transaction checkers have reported back. This concludes the parallel flows.

The conditional flow is selected after the coordinator have executed a reflexive check to determine if all the checkers were successful. If so, it reports success to the transaction, else it reports failure. After completion of the conditional statement, the transaction destroys the coordinator.

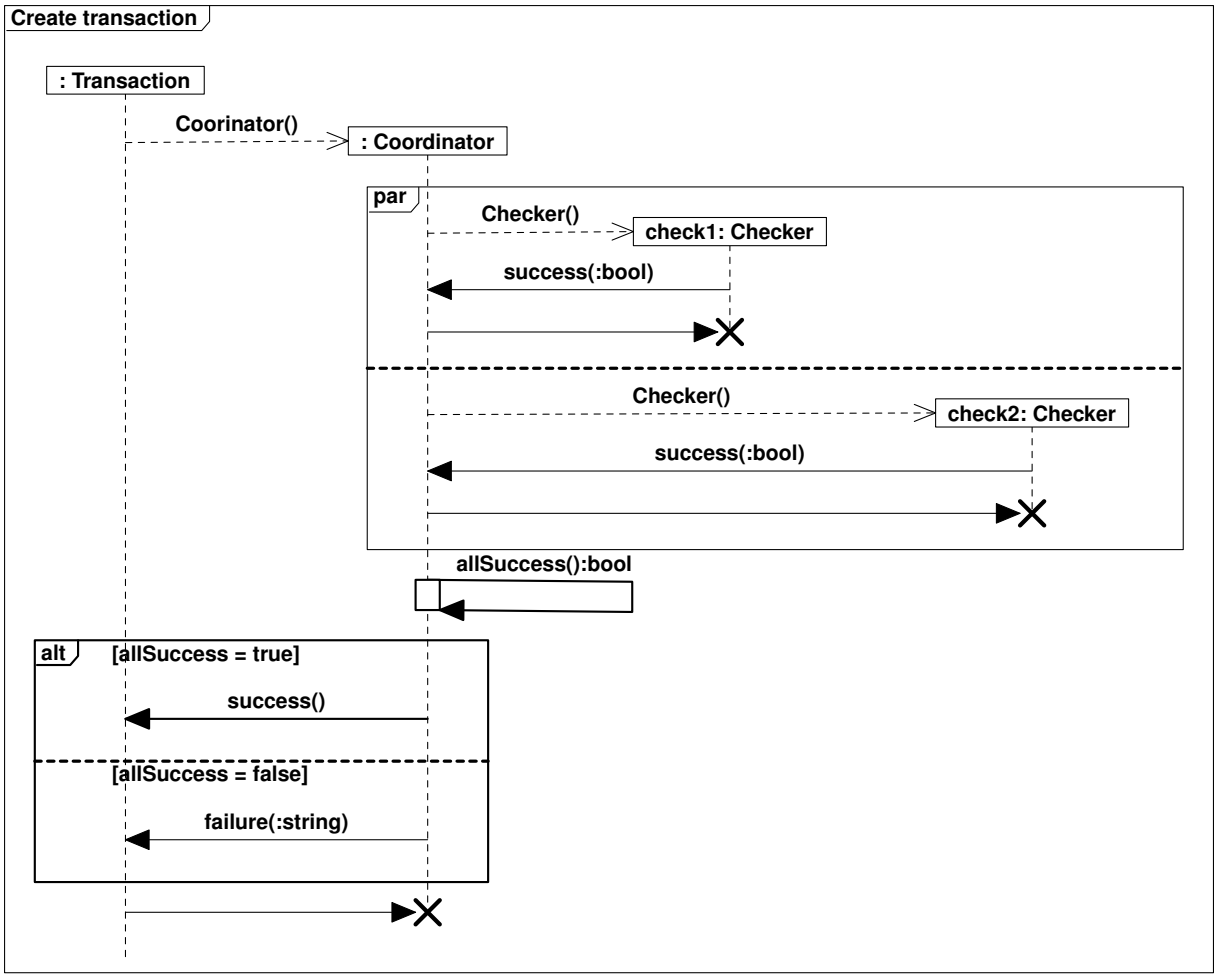


Figure 15: Creating a banking transaction

13.6 Reference to fragments

If a diagram becomes complex, it it advisable to model it in fragments. A fragment is a sub-diagram. A placeholder for a fragment is shown in a diagram using a frame with the keyword `ref` in its header. This placeholder contains only the name of the diagram that contains the detail of the fragment. Figure 16 models the same application as in Section 13.3.2 but by defining parts of the diagram in fragments.

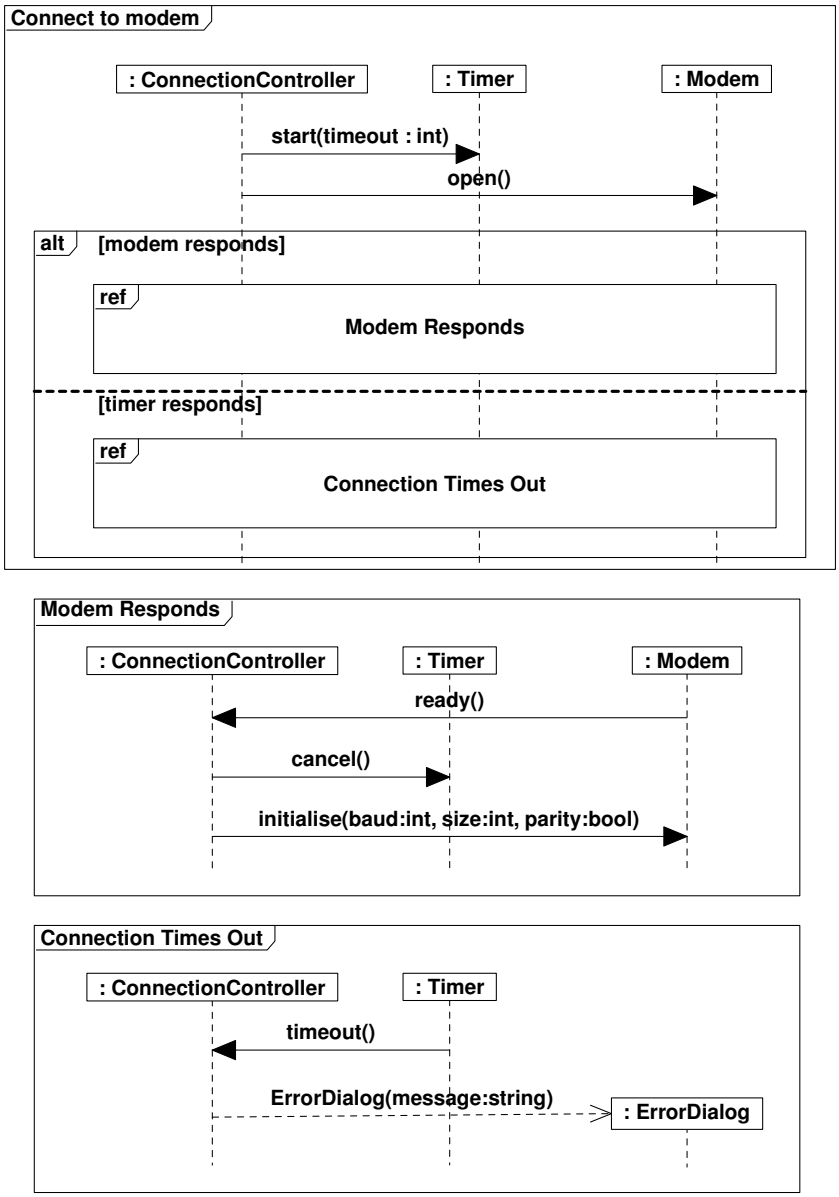


Figure 16: Connection to a modem with alternate flows presented in fragments

13.7 Exercises

1. Use the code given in Section 13.2.6 and the implementations of other relevant methods shown in the class diagram in Figure 8 that can be found in the code that was given for Prac 4. Assume the that two dragons (`norbert` and `smaug`) have been instantiated. Model the interaction resulting from the following code fragment in the `Game` class.

```
norbert->setStrategy(new Aggressive());
smaug->setStrategy(new Defensive());
smaug->attack(norbert);
```

References

[1] Donald Bell. Uml basics: The sequence diagram. `http://www.ibm.com/developerworks/rational/library/3101.html`, 2004. [Online] accessed 2011/08/22.

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

[3] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, Mass, 1997.

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 14: Observer Design Pattern

Contents

14.1 Introduction

This lecture note discuss the observer design pattern. The pattern defines a one-to-many relationship between subjects and their observers. A single subject may have many observers. When the subject changes state all the observers are notified and updated automatically. This pattern would work well in a multithreaded environment which is beyond the scope of this lecture note.

14.2 Observer Pattern

14.2.1 Identification

Name	Classification	Strategy
Observer	Behavioural	Delegation (Object)
Intent		
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. ([1]:293)		

14.2.2 Structure

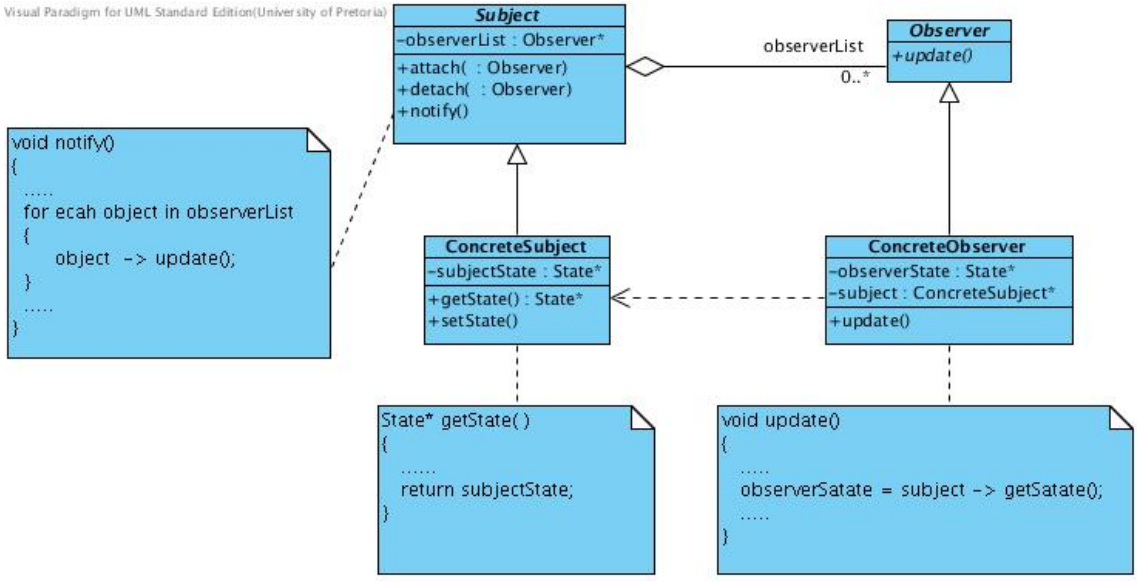


Figure 1: The structure of the Observer Pattern

14.2.3 Problem

The observer design pattern solves the problem of needing to change the code for every observer that is related to the subject during maintainance when observers are changed, added or removed from the system. The pattern facilitates the attaching and detaching of the observers from the subject leaving the subjects code intact.

14.2.4 Participants

Subject

- Provides an interface for observers to attach and detach to the concrete subject.

ConcreteSubject

- Implementation of the subject being observed.
- Implements the functionality to store objects that are observing it and sends update notifications to these objects.

Observer

- Defines the interface of objects that may observe the subject.
- Provides the means by which the observers are notified regarding change to the subject.

ConcreteObserver

- Maintains a reference to the subject it observes.
- Updates and stores relevant state information of the subject in order to keep consistent with the state of the subject.

14.3 Observer Explained

The observer design pattern offers a mechanism by which observers of a subject register with the subject and will be notified when a change occurs in the subject. Figure 2 shows how the communication between the observers and the subject takes place.

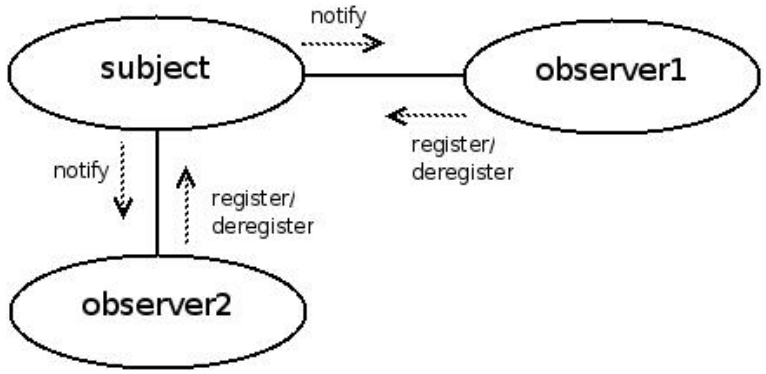


Figure 2: Overview of the interaction between the subject and its observers

An observer will register with the subject. When the subject changes it will notify all the observers registered with it. When an object no longer is an observer of the subject it will deregister from the subject.

14.3.1 Clarification

The pattern comprises of two hierarchies, the objects that are to be observed are represented by the `Subject` hierarchy and the objects that are to do the observation in the `Observer` hierarchy. It is the way in which these two hierarchies interact that brings about the power of the observer pattern. The sequence diagram in Figure 3 shows this interaction.

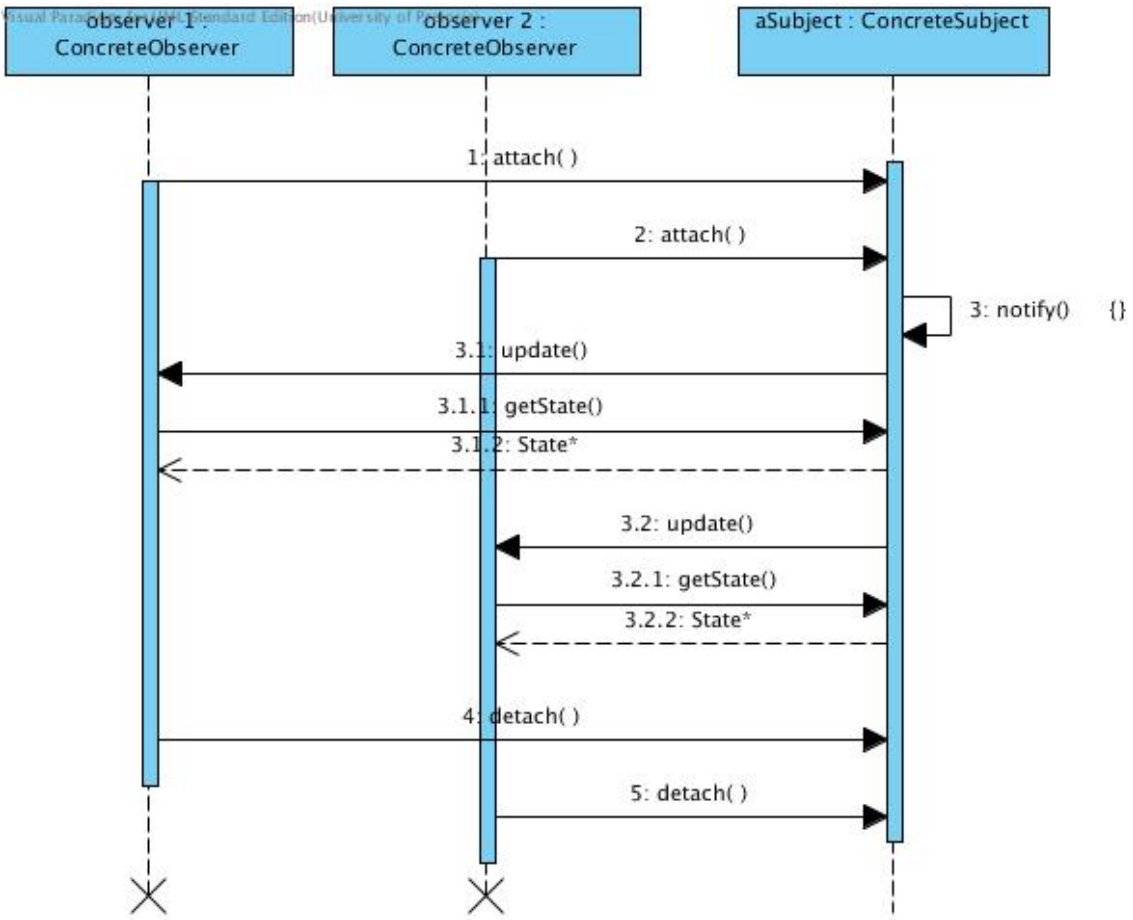


Figure 3: Sequence diagram of the interaction between the participants of the observer

14.3.2 Code improvements achieved

The most significant achievement of the Observer pattern is the separation of concerns. The observers are not embedded, but are independent of the subjects they observe and therefore may register and deregister as the need may arise.

14.3.3 Implementation Issues

The are two design issues that need to be considered when implementing the Observer Design pattern. The first concerns the detaching of the observer from the subject when it goes out of scope. The second issue concerns the how state is transferred to the observer from the subject. These two issues will be briefly addressed in the sections that follow.

Detaching from the subject

The designer of the ConcreteObserver class must ensure that when the related object goes out of scope it detaches from the subject object it is observing. This means that the destructor defined in the ConcreteObserver class needs to call the relevant detach function as defined in the Subject hierarchy. A further important design decision concerns possible extension of the Observer hierarchy, particularly in C++. In the case where the ConcreteObserver is extended to provide additional functionality, the destructor of the ConcreteObserver must be defined as virtual to ensure that the observer object calls the parent destructor and the detachment of the observer actually takes place.

Transfer of state to the observer

The state of the subject can be transferred to the observer by following either a push or a pull model [2]. With the push model, the subject takes responsibility by packaging its public state and providing the observers with this when issuing an update function. The pull model on the other hand requires the subject to provide a public interface by which the observer can request specific state information from the subject. The model depicted in the structure of the design pattern in Figure 1 shows the pull model.

Each of the models have their respective disadvantages. The efficiency of the pull model is lower because it may require a string of function calls to the subject to acquire all the required information. On the other hand, the flexibility of the push model is lower due to the subject needing to keep an additional register of the requirements of the observers and thereby increasing the coupling between the hierarchies.

14.3.4 Common Misconceptions

The observer is used to broadcast events. These events that take place in the subject are only broadcast to the observers that have registered with the subject.

14.3.5 Related Patterns

Mediator

The mediator defines how objects interact and thereby promotes loose coupling between the objects. The objects can therefore interact independently. Mediators are often used to ensure the independent transfer of state between the subject and its observers.

Singleton

By making the subject a singleton, it ensures that the subject has only one access point to it.

14.4 Example

Consider the F1 Grand Prix, or any other motor racing scenario where a pit crew needs to make decisions regarding refueling and changing the tyres of the car that is racing throughout the race. The car in this case is the subject and the pit crew are the observers. Figure 4 provides an incomplete class diagram showing the relationship between the classes. The PitStop hierarchy represents the subject and the PitCrew hierarchy the observers with two concrete observers, the Refueller and the TyreChanger.

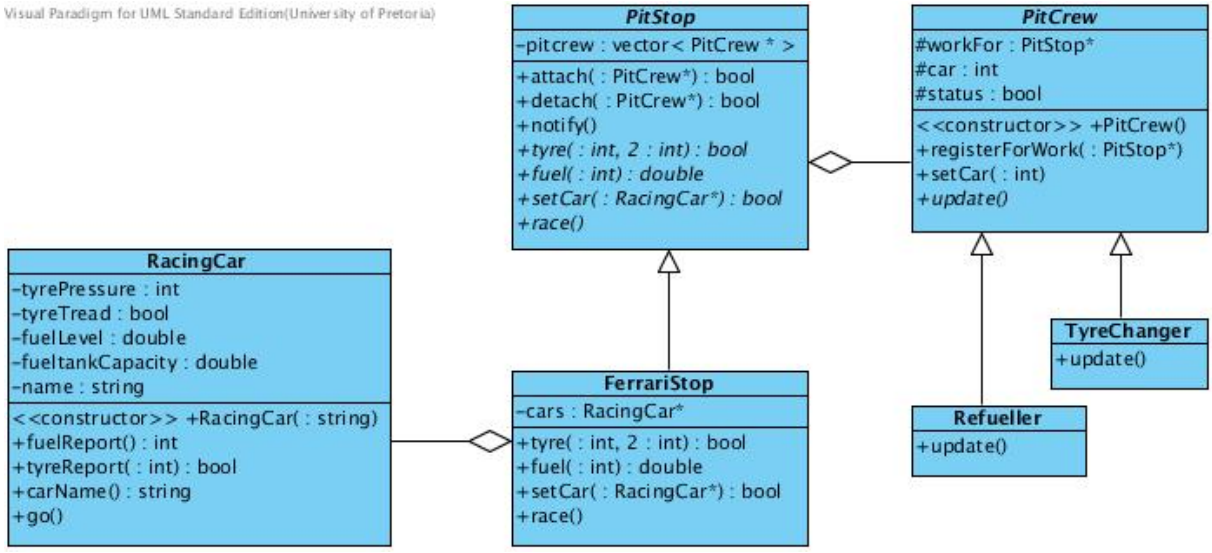


Figure 4: Racing Example

Comparison of the class diagram given in Figure 4 to that of the structure in Figure 1 will show that the dependency between the concrete observer and the concrete subject has been abstracted and included in the Observer participant of the implementation. It is the update function implementations of each of the concrete observer participants that differentiate them from each other not the dependency with the concrete subjects. This being said, a dependency relationship between the `PitCrew` class and the `PitStop` class needs to be included in the figure.

14.5 Exercises

1. Consider the following main program and draw the corresponding sequence diagram for the class diagram as given in Figure 4.

```
int main() {

  RacingCar* car[2];
  car[0] = new RacingCar("Ferrari One");
  car[1] = new RacingCar("Ferrari Two");

  PitStop* ferrariWorkshop = new FerrariStop();

  ferrariWorkshop->setCar(car[0]);
  ferrariWorkshop->setCar(car[1]);

  printWorkshopStatus(ferrariWorkshop);

  PitCrew* refueller = new Refueller();
  ferrariWorkshop->attach(refueller);

  PitCrew* tyreMech = new TyreChanger();
  ferrariWorkshop->attach(tyreMech);

  ferrariWorkshop->race();

  return 0;
}
```

2. Mediator works well with Observer. Show how you would incorporate mediator into the observer structure given in Figure 4.

References

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

[2] Colin Moock. Introduction to design patterns, 2003. URL `http://www.moock.org/lectures/introToPatterns/`. [Online; accessed 1 September 2012].

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 15: Iterator Design Pattern

Contents

15.1 Introduction

To iterate means to repeat. In software it may be implemented using recursion or loop structures such as `for`-loops and `while`-loops. A class that provides the functionality to support iteration is called an *iterator*.

The term *aggregate* is used to refer to a collection of objects. In software a collection may be implemented in an array, a vector, a binary tree, or any other data structure of objects. The iterator pattern is prescriptive about how aggregates and their iterators should be implemented.

Two general principles are applied in the iterator design pattern. The first is a prominent principle of good design namely *separation of concerns*. The other is a fundamental principle of generic programming namely *decoupling of data and operations*. The iterator design pattern suggests that the functionality to traverse an aggregate should be moved to an iterator while functionality to maintain the aggregate remain the responsibility of the aggregate itself. This way the principle of separation of concerns is applied because the functionality concerned with the maintenance of aggregates is separated form functionality concerned with traversal of the aggregates. At the same time the operation to traverse is decoupled from the data structures which are traversed, leading to the creation of a more generic traversal algorithm.

In this lecture we discuss how separation of concerns leads to better cohesion before we proceed to explain the design and implementation of the iterator design pattern.

15.2 Iterator Design Pattern

15.2.1 Identification

Name	**Classification**	**Strategy**
Iterator	Behavioural	Delegation
Intent		
Provide a way to access the elements of an aggregate object sequentially without exposing its underlaying representation ([3]:257)		

15.2.2 Problem

A system has wildly different data structures that are often traversed for similar results. Consequently traversal code is duplicated but with minor differences because each aggregate has its own way to provide the functionality to access and traverse its objects. To eliminate such duplication, we need to abstract the traversal of these data structures so that algorithms can be defined that are capable of interfacing with them transparently [4].

15.2.3 Structure

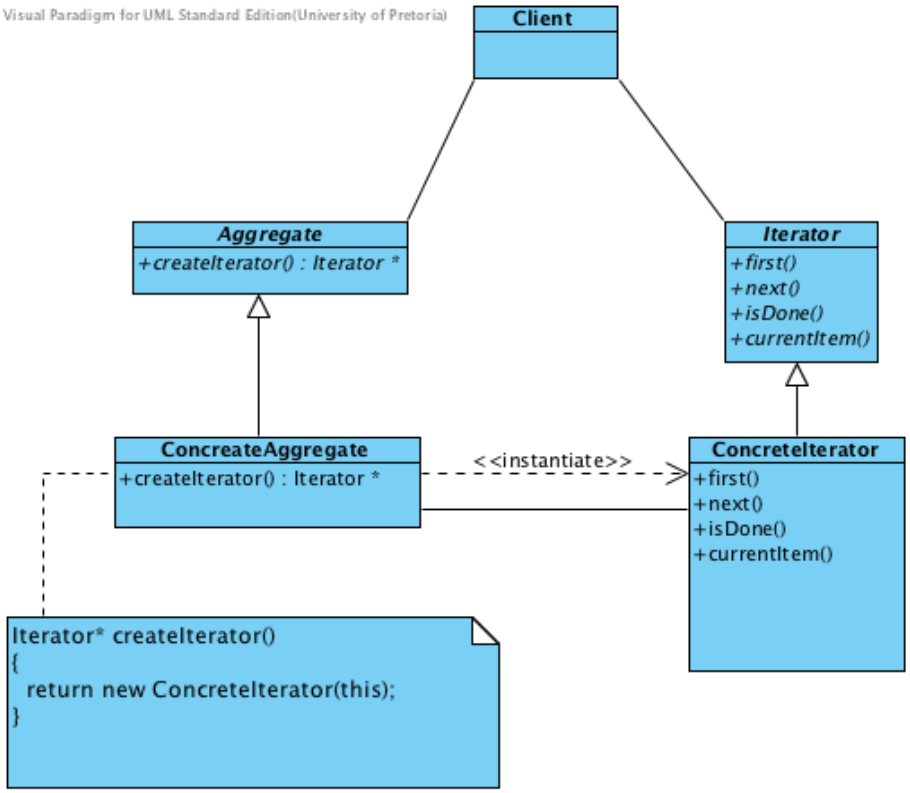


Figure 1: The structure of the Iterator Design Pattern

15.2.4 Participants

Iterator

- Defines an interface for accessing and traversing elements.

Concrete Iterator

- Implements the Iterator interface
- Keeps track of the current position in the traversal of the aggregate

Aggregate

- Defines an interface for creating an Iterator object

Concrete Aggregate

- Implements the Iterator creation Interface to return an instance of the proper concrete iterator.

15.3 Cohesion

When designing a system, it is important to keep its maintenance in mind. Making changes should be easy. One of the design principles that can be applied to avoid the need to change a class, is *separation of concerns*. This means that functionality concerning different aspects should be separated from one another by implementing them in different classes. Thus, the functionality provided by each class in the design should be related to one aspect only. If a single class implements various aspects of functionality, changes in any one of these aspects will result in having to change the class. On the other hand, if a class implements only one aspect of functionality, it will change if and only if that specific aspect changes. Note that the amount of change does not change, only the chance of having to change each class is reduced, meaning that changes are isolated to certain classes.

The term *cohesion* is used to refer to the internal consistency within parts of the design. In object-oriented design the level of cohesion of a system is determined by the level of cohesion of the classes that constitutes the system. A metric to measure the lack of cohesion in methods (LCOM) was proposed by [1]. It is recognised as the most used metric when trying to measure the goodness of a class written in some object-oriented language [5]. When calculating the LCOM of a class, two methods are considered to have a lack of cohesiveness when they operate on disjunct sets of attributes in a class. While it is valid to state that methods are not cohesive when they operate on different attributes, it is not conclusive that they are cohesive if they operate on the same same attributes. Consequently the presence of cohesiveness is much harder to observe than the absence thereof.

We say a class has high cohesion when its methods are related. These methods should be related not only by operating on the same attributes of a class, but more importantly they must be related in terms of the functions they perform. If methods perform functions that are related to different responsibilities, they should be separated by including them in different classes. However, separating responsibility in design is one of the most difficult things to do. Sometimes non-cohesiveness of a class is only realised when the class tends to change more often or in more than one way as the system grows.

Separation of concerns will increase overall cohesion of a system and may reduce the number of classes that has to change when needed. However, it will most likely increase the number of classes in the system. This, in turn will increase complexity as well as coupling between the classes. Finding the best design is illusive. When improving one thing one tends to worsen another! Through experience one learn how to find good solutions that has low coupling without compromising too much in terms of high cohesion.

15.4 Iterator Pattern Explained

15.4.1 Design

The Iterator design pattern applies separation of concerns specifically to aggregates. Usually aggregates have at least two functions. One being its maintenance, and the other its traversal. Maintenance of aggregates includes methods to add and remove elements and the like, while traversal of aggregates concerns only accessing the elements and knowing

an order in which they should be accessed. The iterator design pattern describe a design that separates the mechanism to iterate through the aggregate from the other functions an aggregate may have.

The Iterator design pattern moves the responsibility of traversing objects away from the aggregate to another class called an iterator. The aggregate class, therefore, can have a simpler interface and implementation because it needs only to cater for maintenance of the aggregate and no longer for its traversal [2].

The iterator design pattern takes this good design a step further. Instead of just implementing every aggregate in two classes (one for maintenance, and one for traversal), this pattern is a design that provides a generic way to traverse the objects in aggregates that is independent of the structure of the various aggregates. This is achieved by defining two abstract interfaces – one for iteration and one for the rest of the functionality of aggregates. This way the system is more flexible when either aggregates or their iterations needs maintenance.

15.4.2 Improvements achieved

- Iterators simplify the aggregate interface. All the functionality related to access and traversal is removed from the aggregate interface and placed in the iterator interface resulting in a smaller and more cohesive aggregate.
- Iterators contribute to the flexibility of your code – if you change the underlying container, it's easy to change the associated iterator. Thus, the code using aggregates becomes much easier to maintain. Most changes to the internal structure of the aggregates it uses will have no impact on the code that uses the aggregate.
- Iterators contribute to the reusability of your code – algorithms that were written to operate on a containers that use an iterator can easily be reused on other containers provided that they use compatible iterators. Thus, the same code can be used to traverse a variety of aggregate structures in the same application. This reduces duplication of code in applications that manipulate multiple aggregates.
- It is easy to provide different ways to iterate through the same structure for example traversing breadth first or depth first through a game tree or for example to have an iterator that might provide access only to those elements that satisfy specific constraints.
- It is possible to execute simultaneous yet independent iterations through the same structure.

15.4.3 Disadvantage

A prominent disadvantage of the application of the iterator design pattern is that it becomes complicated the synchronise an aggregate with its iterator. Because the aggregate structure is completely independent of the iteration process, it is thus possible to apply changes to the aggregate while an independent thread iterates through the structure. Such situation is prone to error.

In the example in Section 15.6, `VectorSteppingTool` creates a copy of the state of the aggregate. In this case the iterator can not malfunction even if the aggregate is changed during the iteration process. However, the iteration will complete without reflecting the changes. On the other hand `LinkedListIterator` operates directly on its aggregate. If the `LinkedList` is changed, the iterator will reflect such changes immediately. However, it is prone to error if not synchronised properly. For example, if the current item of the iterator is deleted, the next call to `next()` will cause a segmentation fault. To prevent this the implementation should either disallow the deletion of the current item in all iterations (which might be difficult to implement), or update the current item in all iterations when an item is deleted. To implement this, iterators need to be registered as observers of the delete action – this is also not trivial.

15.4.4 Real world example

The programming use of a controller to select TV channels provide a practical example of a filtering iterator. The TV set has its built in iterator that scan sequentially through all frequencies. The controller can be programmed to have a specific frequency associated with numbered buttons. This corresponds with a `skipTo()` method that is often implemented in iterators. After the channels have been associated with these numbers, the next and previous buttons can be used to request the next channel, without knowing its number. The use of these buttons correspond with the concept of a special iterator that can be used to step through the channels.

15.4.5 Related Patterns

Factory Method
: Both Iterator and Factory Method use a subclass to decide what object should be created. In fact `createIterator()` is an example of a factory method.

Memento
: The memento pattern is often used in conjunction with the iterator pattern. An iterator can use a memento to capture the state of the aggregate. This memento is stored inside the iterator to be used for traversing the aggregate.

Adapter
: Both patterns provides an interface through which operations are performed. They differ in the reason for providing this interface. The adapter do it because it would be otherwise impossible while the iterator do it specifically to generalise iteration of aggregates.

Composite
: Recursive structures such as composites usually need iterators to traverse them sequentially. Although recursive traversal might be very easy to implement without extending the composite pattern, its is strongly advised to create a composite iterator as discussed in Section 3.

15.5 Implementation Issues

15.5.1 Accessing the elements of the aggregate

Concrete iterators must be able to access the elements in the aggregate. The concrete iterator may use this to implement the necessary access in one of the following ways:

- **Make a copy of the aggregate inside the iterator**. This is the most robust solution. This is execution-wise the most efficient, but memory-wise the least efficient. It also has the drawback of not being able to reflect on-the-fly changes to the aggregate.
- **Create an object storing the state of the aggregate inside the iterator**. This more or less boils down to storing a memento (See the memento design pattern) of the aggregate inside its iterator. This is also a robust solution. This might be more efficient than making a copy of the whole aggregate, but not always easy to implement. It suffers the same drawback of not being able to reflect changes to the aggregate that are made after the iterator was created.
- **Keep a pointer to the aggregate inside the iterator and use a call back mechanism to access the elements of the aggregate**. This solution is memory efficient, yet not as robust as the other methods. In this case the methods that needs to be called should be public in the aggregate, or alternatively the iterator can be declared a private/protected friend class of the aggregate and hence be given access to its private/protected methods. This solution will be able to reflect changes that are applied to the aggregate in real time, however it is prone to error if synchronisation between the aggregate and the iterator is not implemented properly. Such close coupling between the aggregate and the iteration also compromises the encapsulation of the aggregate.
- **Use the pimpl**[1] **principle**. This is the most efficient, both in terms of memory and execution time. It is also robust. How this is done is beyond the scope of this module.

15.5.2 Additional functionality

The pattern as given in Figure 1, defines a minimal interface to the Iterator class containing only `first()`, `next()`, `isDone()` and `currentItem()`. When implementing the iterator design pattern it might sometimes be handy to implement some of the following additional methods in the interface:

- `remove()` – This method should remove the current item from the aggregate. It provides the means to synchronise the maintenance of the aggregate with its iterator by using a double dispatch[2].

[1] pointer to implementation

[2] More detail on the double dispatch mechanism is discussed in L34_Visitor

- `previous()` – This method should step backwards instead of forwards to enable iterations that can go in both directions. If this is supported one should also implement to different methods for the prescribed `isDone()`. One for reaching the end while moving forward, and one for reaching the beginning while moving backwards. This is usually implemented using method names like `hasNext()` and `hasPrevious()`.
- `skipTo()` – This method should position the iterator to an object matching specific criteria. This operation may be useful for sorted or indexed collections to enable the implementation of more complicated algorithms to operate on the aggregate. Examples of algorithms that may need this operation are binary search and quick sort.

15.5.3 Internal and External iterators

Iterators are classified as either internal or external depending on who calls the methods that are declared for accessing and traversing the aggregate. If the client calls these methods the iterator is said to be external. An internal iterator, on the other hand, is controlled by the iterator itself. In this case the methods for traversing the aggregate can be declared private to prevent the client from calling them. Internal iterators are less flexible than external iterators because when it is the iterator that is stepping through the aggregate, you have to tell the iterator what to do with the elements while stepping through them. This means that you also need some way to pass an operation to the iterator.

15.5.4 Allocation on the heap or on the stack

Iterator object may be allocated dynamically on the heap. This allows for the creation of polymorphic iterators. In this case the client is responsible for deleting them. This is error-prone, because it's easy to forget to free a heap-allocated iterator object when you're finished with it. Hence they should be used only when there's a need for polymorphism. A more robust solution would be to allocate concrete iterators on the stack.

An alternative that provides the flexibility offered by heap allocation as well is the stability achieved through stack allocation offered by [3] is to apply the proxy design pattern to implement a stack-allocated proxy as a stand-in for the real iterator. The proxy can delete the iterator in its destructor. Thus when the proxy goes out of scope, the real iterator will get deallocated along with it. The proxy ensures proper cleanup, even in the face of exceptions.

15.5.5 Using C++ STL Iterators

The application of the iterator design pattern was taken seriously by the designers of the C++ language. Standard iterators are implemented for the containers in the C++ STL, such as `vector<>`, `list<>`, `stack<>` and `map<>`. When using STL containers it is advisable to use the provided iterators instead of a counter to traverse such container to gain the benefits of using the iterator design pattern mentioned in Section 15.4.2. They may also be able provide a way to access the data in an STL container that don't have obvious means of accessing all of the data (for instance, maps).

When one use one of these containers, a variable of the container type is declared by including the type of the objects as a template parameter. For example use the following syntax to declare a vector of integers called `myVector`:

```
vector<int> myVector;
```

To declare an iterator appropriate for a particular STL template class, you use the following syntax

`std::`*class_name*`<`*template_parameters*`>::iterator` *name*

where *name* is the name of the iterator variable you wish to create and the *class_name* is the name of the STL container you are using, and the *template_parameters* are the parameters to the template used to declare objects that will work with this iterator. Note that because the STL classes are part of the std namespace, you will need to either prefix every container class type with `std::`, or include `using namespace std;` at the top of your program. For example you can create an iterator for the vector `myVector` that was declared in the above mentioned example as follows:

```
std::vector<int>::iterator myIterator;
```

The two loops in the following code fragment are functionally equivalent. The first uses an integer counter to iterate through the vector that was declared in the above mentioned example, while the second uses the iterator that is declared here:

```
for(int myCounter = 0;
        myCounter< myVector.size(); myCounter++)
        cout << myVector[myCounter] << '\t';

for(myIterator = myVector.begin();
        myIterator < myVector.end(); myIterator++)
        cout << *myIterator<< '\t';
```

Note how the elements of the vector are accessed by using the `operator[]` in the first loop, while they are accessed by dereferencing the iterator in the second loop. To move from one element to the next, the increment operator, ++, is used in both cases. Iterators overload all operators. One can use the standard arithmetic shortcuts such as `--`, `+=` and `-=`, and also use `!=`, `==`, `<`, `>`, `<=`, and `>=` to compare iterator positions within the container.

The following are some pitfalls to watch out for when using STL iterators:

- Iterators do not provide bounds checking; it is possible to overstep the bounds of a container, resulting in segmentation faults
- Different containers support different iterators, so it is not always possible to change the underlying container type without making changes to your code
- Iterators can be invalidated if the underlying container (the container being iterated over) is changed significantly

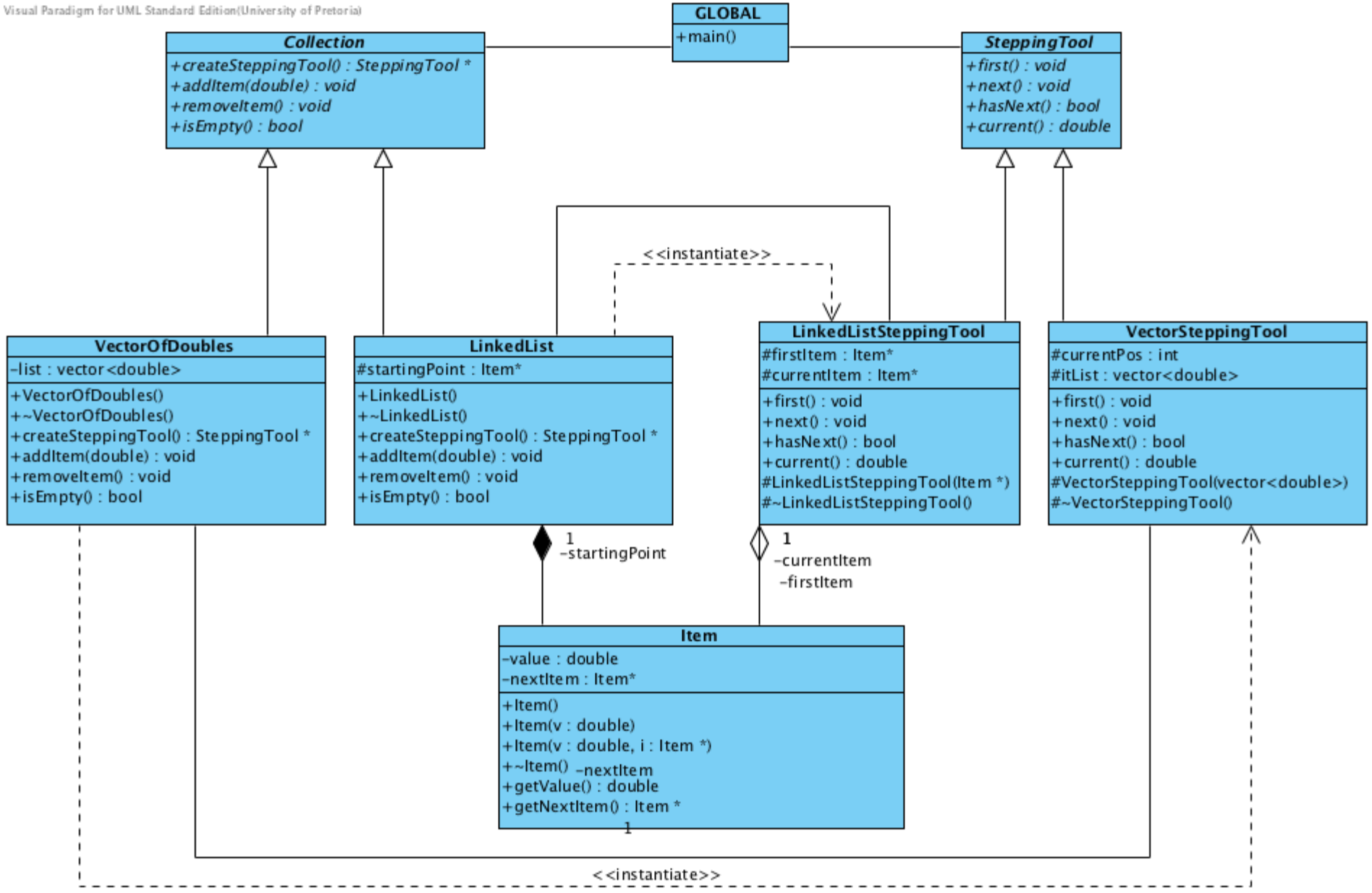


Figure 2: Class Diagram of a system illustrating the implementation of the iterator design pattern

15.6 Example

Figure 15.5.5 is a class diagram of an application that implements the iterator design pattern. It implements two data structures and their respective iterators. The main program uses the same code to manipulate any one of these data structures. It also shows how two independent iterators can be used to traverse the same structure at the same time. The two data structures are a vector and a singly linked list.

Participant	**Entity in application**
Iterator	SteppingTool
Concrete Iterators	VectorSteppingTool, LinkedListSteppingTool
Aggregate	Collection
Concrete Aggregates	VectorOfDoubles, LinkedList
createIterator() :Iterator*	createSteppingTool():SteppingTool*
first(), next(), isDone(), currentItem()	first(), next(), hasNext():bool, current():double
Client	main()

Iterator

- The `SteppingTool` class acts as the iterator interface.
- It defines an interface for accessing and traversing elements exactly as prescribed by the pattern.

Concrete Iterator

- The concrete iterators that are implemented are `VectorSteppingTool` and `LinkedListSteppingTool`.
- Each concrete iterator has an instance variable to enable it to access the elements of its corresponding aggregate. In the case of the `VectorSteppingTool` the whole vector is duplicated[3]. In the case of `LinkedListSteppingTool` it only needs a pointer to its first element, because the rest of the list can be accessed by following the links in its items.
- Each concrete iterator has an instance variable to enable it to refer to the current item during traversal. In the case of the `VectorSteppingTool` it is an integer holding the index value of the current position. In the case of `LinkedListSteppingTool` it is a pointer to the current node in the linked list.
- Each concrete iterator implements the methods defined in the `SteppingTool` interface.

Aggregate

- The `Collection` class acts as the aggregate interface.
- It defines the `createSteppingTool()` method that ensures that an iterator can be created for each concrete aggregate as prescribed by the pattern.

[3]It might have been a better idea to keep only a pointer to the original vector. This way less memory will be used and dynamic changes to the vector can be supported

- It also defines methods to be able to maintain concrete objects of classed derived from this interface. It supports only one insertion and a default deletion of elements, as well as a means to determine if the collection is empty.

Concrete Aggregate

- The concrete aggregates are `VectorOfDoubles` and `LinkedList`.
- Each concrete aggregate implements a default constructor and destructor.
- Each concrete aggregate implements the creation of its specific iterator as prescribed by the pattern. In the case of the `VectorOfDoubles` the vector is passed as parameter to the constructor of `VectorSteppingTool` which in turn makes a copy of it. In the case of `LinkedList` a pointer to the head of the list is passed to the constructor of `LinkedListSteppingTool`. Note that both these cases do not pass a pointer to itself to the constructor of its iterator as suggested in the pattern definition. Instead the pass only the data needed by the iterator to be able to operate.
- `VectorOfDoubles` implements the other methods defined in the `Collection` interface to maintain the collection simply by delegating the actions to the appropriate methods of the `<vector>` class.
- `LinkedList` implements the other methods defined in the `Collection` interface to maintain the collection by creating and deleting `Item` objects and setting their pointers appropriately to maintain a singly linked list. Note in the class diagram that the `LinkedList` has the responsibility to create and delete `Item` objects, while the `LinkedListSteppingTool` only reads them and should not delete them on destruction.

Client

- In this application the client has a `VectorOfDoubles` object called `myCollection` which is initiated with the first 10 integers. Two instances of `VectorSteppingTool` is then used to iterated differently through `myCollection`. The programmer only need to change one line of code to change the instance of `VectorOfDoubles` to an instance of `LinkedList`. Owing to the implementation of the iterator design pattern, the use of the correct iterator for this `LinkedList` will be instantiated without having to change any other code.

15.7 Exercises

1. Change the test harness (main program) of the system given in Section 15.6 to allow the insertion and deletion of nodes during iteration and observe the impact. (See Section 15.4.3).

2. Add a structure that stores a binary tree of double values to the system given in Section 15.6. The values should be inserted in such a way that the left child of every parent is smaller than its parent and the right child is larger than its parent. Duplicates should be ignored. Implement different concrete SteppingTools to allow pre-order, in-order, and post-order traversal of your binary tree.

Write a new test harness. This program should insert random double values simultaneously into a `VectorOfDoubles` and into your binary tree in the order they are generated. Use a `VectorSteppingTool` to display the vector and your different binary iterators to show the different traversals of your binary tree.

3. Implement an iterator for the composite in L13_Composite

   - Implement the `createIteror`-method as an operation of the composite design pattern.
   - Define a composite iterator class as a concrete iterator in the Iterator design pattern.
   - Implement the operations that are defined in the Iterator to be recursive when it is the iterator of a composite.

References

[1] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476 –493, jun 1994.

[2] Eric Freeman, Elisabeth Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Sebastopol, CA95472, 1 edition, 2004.

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

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

[5] Sami Mäkelä and Ville Leppänen. Observation on lack of cohesion metrics. In *Proceedings of the International Conference on Computer Systems and Technologies*. 2006.

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 16: UML Activity Diagrams

Contents

16.1 Introduction

The first structured method for documenting process flow, the "flow process chart", was introduced by [3]. Activity diagrams combine techniques from flow charts, event diagrams [5] and Petri nets [6]. Activity diagrams describe the workflow of a system. The diagrams describe activities by showing the sequence of activities performed. Activity diagrams are useful for analysing a use case by describing what actions need to take place and when they should occur. Usually activity diagrams are used to describe a complicated algorithm or to model the flow in applications with parallel processes. The distinction between state and activity diagrams is only in what they model. Where state diagrams are used to model state-dependent behaviour and conditions for transitions between states, activity diagrams are used to model the flow of actions and the order in which the actions take place.

16.2 Notational Elements

The notational elements used in state diagrams and activity diagrams are the same except for a few subtle differences. The main difference between UML state diagrams and UML activity diagrams is in their intent.

The basic elements of UML activity diagrams can be classified as activity nodes and edges. Activity nodes can either be action, object or control nodes. An activity is shown as a round-cornered rectangle which encloses all the action and control nodes which make up the activity. Action nodes represent a single step within an activity and are also denoted by a round-cornered rectangle. Control nodes model different flows that are controlled by conditions called guards.

16.2.1 Initial and final nodes

The starting point of the flow that is shown in an activity diagram is indicated with a filled circle. An activity diagram must have exactly one initial node. An exit point in an activity diagram is called an final node. There are two kinds of final nodes, referred to as as activity final node and flow final node. The flow final node denotes the end of all flow controls within the activity. The activity final node denotes the end of a single flow control. An activity final node is shown with a filled circle with an outline. Refer to Figure 1. An activity diagram may have multiple final nodes. It is also allowable that it has no final node. If an activity diagram has no final node, it models an infinite activity such as an infinite loop.

16.2.2 Action nodes

An action node in an activity diagram is a rounded square containing a descriptive name for the action. It looks exactly like a state node in a state diagram, as can be seen by comparing `ReceivingState` in Figure 2 with `Action 1` in Figure 3. The difference between a state node and an activity node and more specifically the action nodes, is that a state node in a state diagram may contain actions that are performed whereas activity nodes **are** actions and do not contain state.

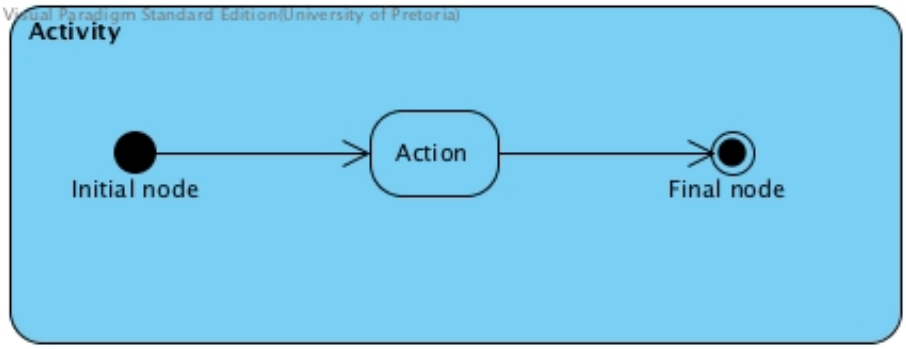


Figure 1: Start and end nodes

A state van have activities An activity does not have states

Figure 2: The notational difference between states and activities

16.2.3 Activity edge

An activity edge is an arrow indicating the flow between two activities, as shown in Figure 3. The arrow points in the direction of the activity that has to execute next. Optionally activity edges may be named using text. Usually there is no need to name activity edges. When decision nodes (Section 16.2.4) are used, each activity edge exiting from the decision node should have a guard condition shown with text in square brackets. A guard condition indicates that the flow to the next activity may only be executed if the specified condition is true. Activity edges are similar to transition edges found in state diagrams. An important difference is that a transition edge in a state diagram my be decorated with an activity while it is not permissible to indicate an activity on an activity edge in an activity diagram.

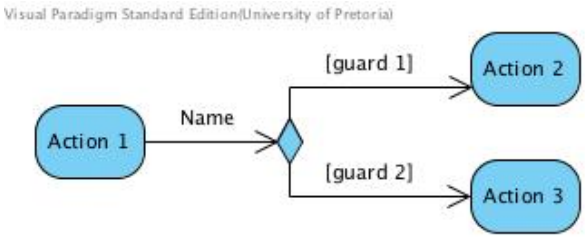


Figure 3: Activity edge

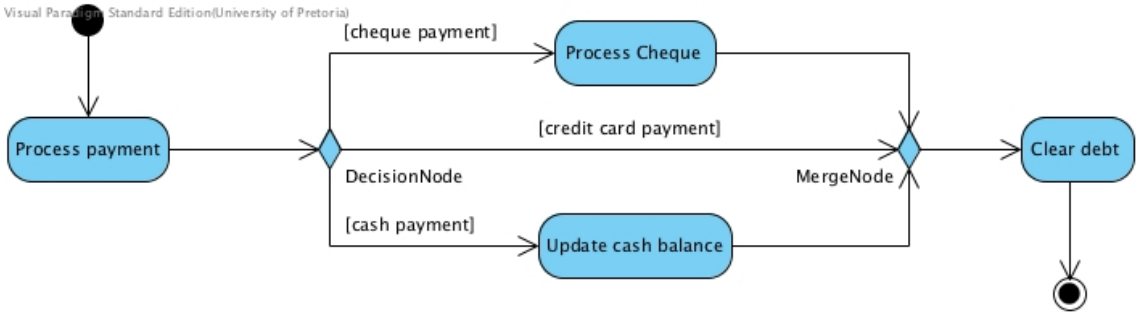


Figure 4: Model of a payment showing the use of decision and merge nodes

16.2.4 Alternate flows

Alternate flows are shown by using decision and merge nodes. Both these are shown as a diamond shape. If different activity edges leave such shape it is a decision node. If different activity edges meet at the shape, it is a merge node. Figure 4 shows a diagram with alternate flows. The guards on the action edges leaving the decision node on the left side of the figure specifies under what conditions each of these flows should execute. In this example there are three alternate flows depending on the kind of payment. Both cash and cheque payments requires some extra activity while the flow immediately proceed to the clear debt activity if a credit card payment is made. In the case of cheque payments the cheque is processed while cash payments require an update of the cash balance before proceeding to the clear debt action. It is important to note that only one of the alternate flows may execute at any time. Therefore, the guards should be mutually exclusive. In programming alternate flows represent conditional actions like in `if` and `while` statements.

16.2.5 Parallel flows

Parallel flows are shown by using fork and join nodes. Both these are shown as a heavy vertical or horizontal line. If different activity edges leave such line it is a fork node. If different activity edges meet at the line, it is a join node. Figure 5 shows a diagram with parallel flows. When a fork is reached in the flow, all the activities to which the activity edges that leave the fork point, are executed at the same time using independent execution threads. This example models a shopping experience by a married couple. They both enter the shop and then proceed each on a separate mission. While the one person (probably the wife) picks up fruit, then meat and then milk, the other person move directly to the news stand to pick up a news paper. After both completed their respective actions, they join and proceed to the till. It is important to note that all the flows leaving a fork are executed in parallel. In programming new threads are spawned for each parallel flow.

16.2.6 Composite activities

A composite activity is an activity that can itself be described in terms of an activity diagram. Sometimes composite activities are shown by drawing the diagram of its activi-

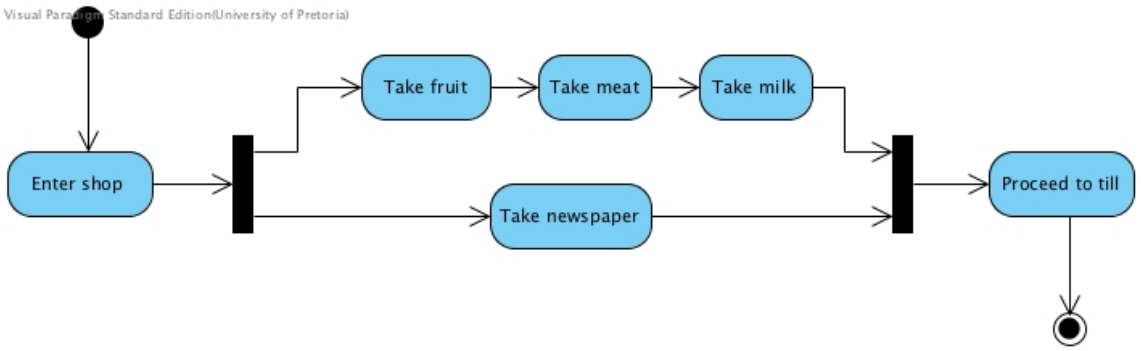


Figure 5: Fork and join nodes

ties inside the action node as shown in the right hand diagram in Figure 6. A composite of which the sub-activities are not shown can be drawn using a rake symbol inside it as shown in the left hand diagram in Figure 6. Such sub-activity is called a *called action*. A called action is like a subroutine call in a program. Called actions are often used to eliminate the need to draw the same sub-activity diagram in more than one place.

16.2.7 Swimlanes

Swimlanes are used to convey which class is responsible for a given activity. A swimlane is a vertical or horizontal zone in which the activities which are the responsibility of a certain class are grouped. Swimlanes are indicated using solid vertical or horizontal lines to indicate a border between two swimlanes and labeling the zones. The diagram in Figure 7 was adapted from [1]. It illustrates the use of swimlanes. In this diagram the consultant is responsible for filling in the expense form and correcting it if needed. The accountant is responsible for validating such form and taking the appropriate actions while the payroll service is responsible for paying the employer of the consultant.

16.3 Examples

16.3.1 Finding and drinking a beverage

The activity diagram in Figure 8 comes from the UML 1.0 documentation and was replicated from [2]. This diagram illustrates the interpretation of an activity diagram on a conceptual level.

There parallel activities `Put Coffee in Filter` ,`Add Water to Reservoir` and `Get Cups` may be interpreted to mean that these activities may be executed in any order or if possible at the same time by interleaving them. The synchronisation point indicated by the join node before `Turn On Machine` should be interpreted that the machine may only be turned on after both the putting the filter with coffee and putting water in the reservoir was completed.

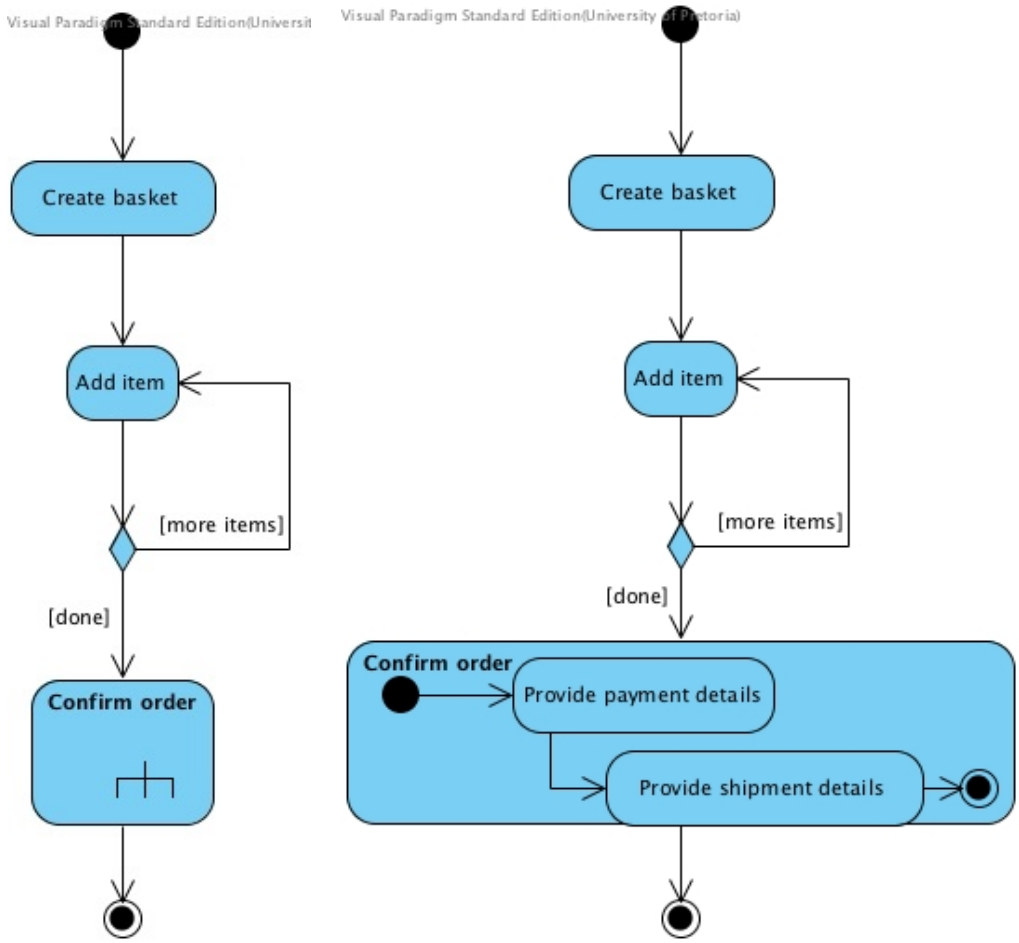


Figure 6: Activities with sub-activities

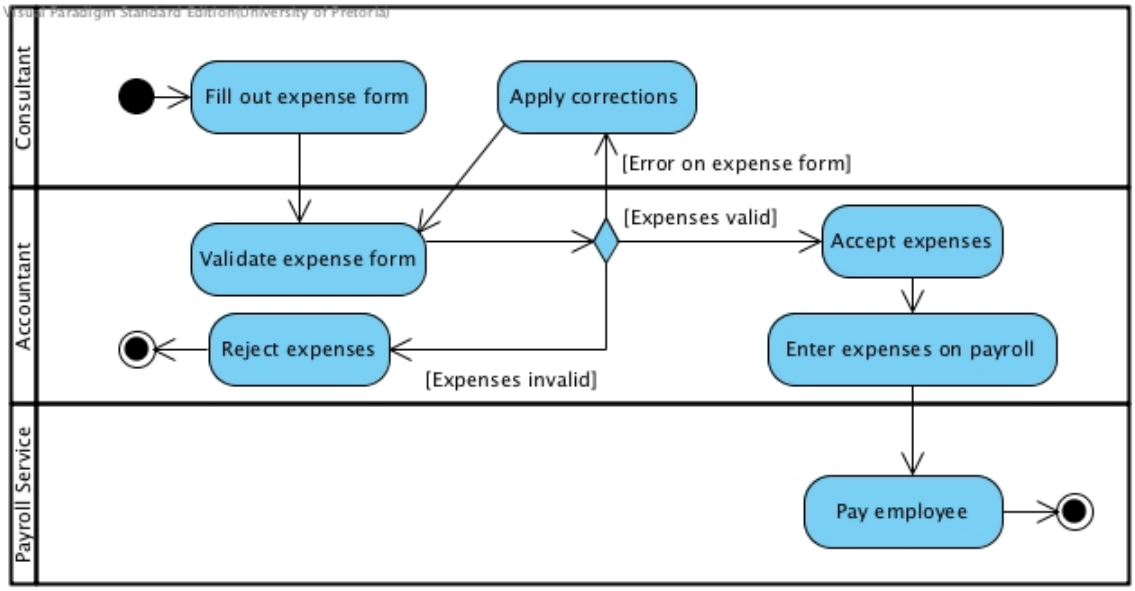


Figure 7: Processing an expense

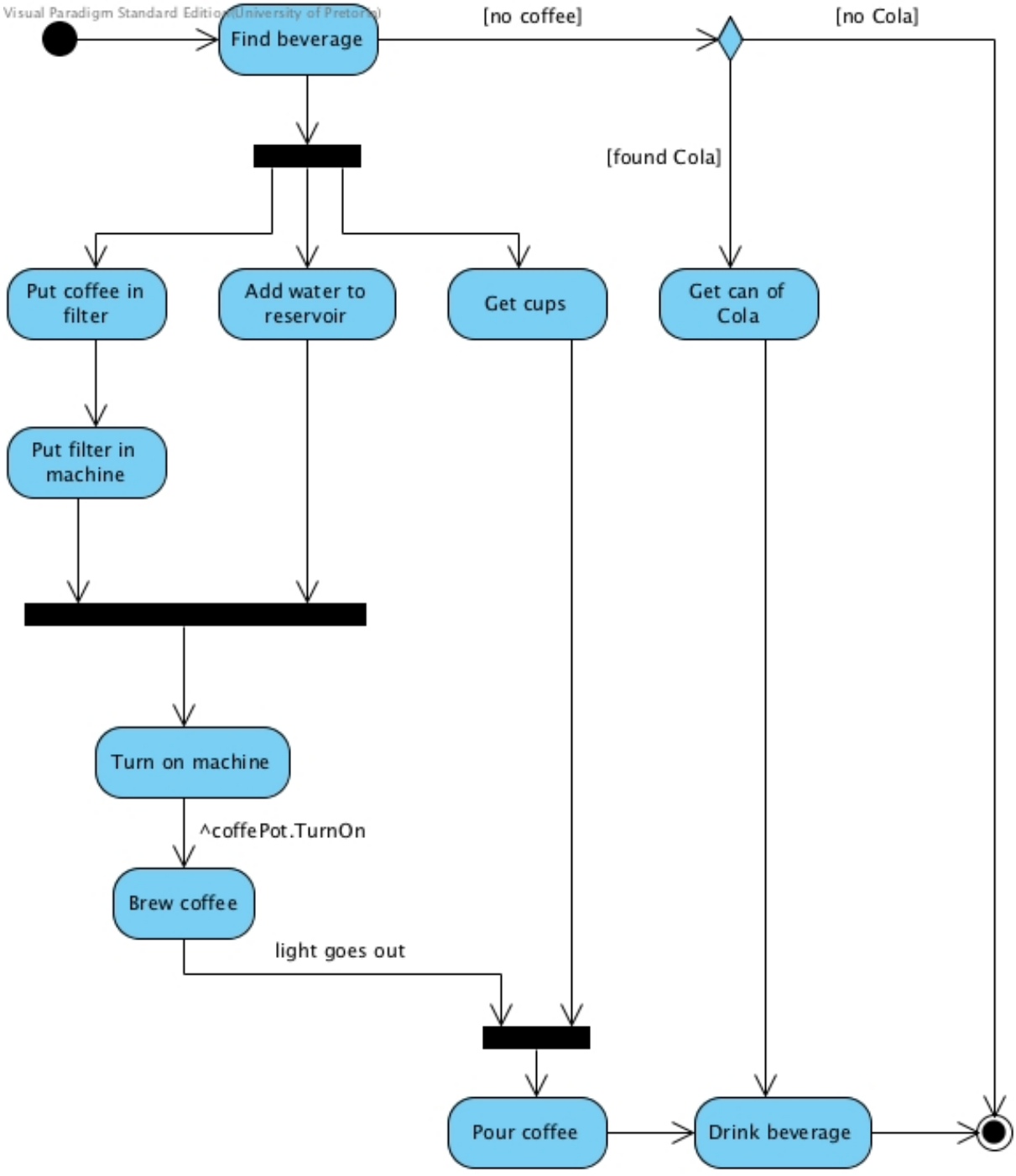


Figure 8: Finding and drinking a beverage

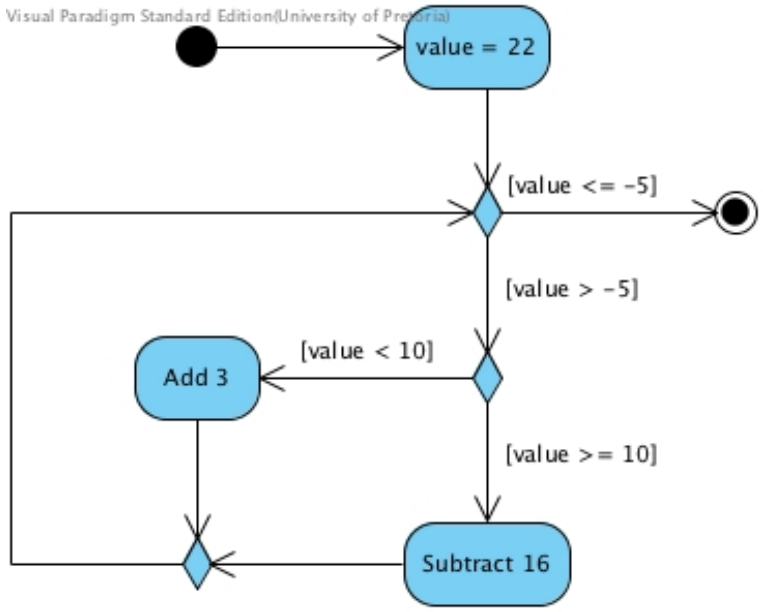


Figure 9: An activity diagram containing a loop

16.3.2 Implementing code for an activity diagram

Activity diagrams can be translated into code. Each activity node is either a program statement or a function call. If it is a function call it can be indicated using the rake symbol. Decision nodes represent conditional statements like `if` or `switch`. If there is some transition from a later activity node back to a decision node, it represents a loop structure which can be implemented using a `while`-loop, a `for`-loop, or recursion.

The activity diagram in Figure 9 contains a loop because the topmost decision node may be reached a number of times. The following code fragment is an implementation of the diagram in Figure 9 using a `while`-loop

```
int value = 22;
while (value > -5)
{
  if (value < 10)
  {
      value += 3;
  }
  else
  {
      value -= 16;
  }
}
```

16.3.3 Activities based on age of the participant

From an implementation-perspective the activities shown in an activity diagram represent methods in classes. To illustrate this we have created code to 'implement' the activities

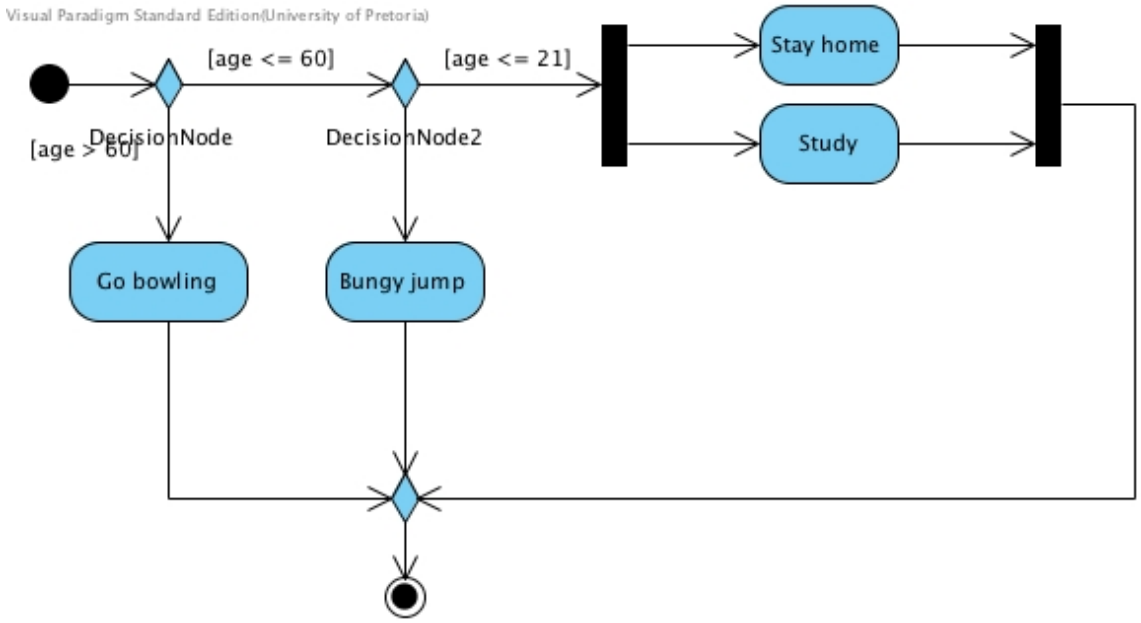


Figure 10: Activities for people of different age groups

in the activity diagram in Figure 10 and came up with the following code:

```
#include <iostream>
#include <boost/thread.hpp>

using namespace std;
using namespace boost ;

void goBowling() ;
void bunjyJump();
void stayHome();
void study() ;

int main() {
        int age ;
        thread stayHomeThread = thread(stayHome);
        thread studyThread = thread(study) ;
        cout << "Please enter your age: ";
        cin >> age;
        if (age > 60)
        {
                goBowling();
        }
        else if (age > 21)
        {
                bunjyJump();
        }
        else
```

```
        {
                stayHomeThread.join ();
                studyThread.join();
                cout << "Actions while threads are running" << endl ;
                stayHomeThread.detach() ;
                studyThread.detach() ;
        }
}

void goBowling()
{
        cout << "Enjoy the day playing bowls" << endl ;
}

void bunjyJump()
{
        cout << "Drive to the precipice , " ;
        cout << "attach the bunjy cord and jump !!!!" << endl;
}

void stayHome()
{
        cout <<  "Stay Home" << endl;
}

void study()
{
        cout << "Study" << endl ;
}
```

An article [4] that was written by the author of the `Boost.Threads` library will get you started on installing and using this library and writing multi-threaded applications using the C++ programming language[1].

References

[1] SW Ambler. Uml 2 activity diagramming guidelines. `http://www.agilemodeling.com/style/activityDiagram.htm`, 2009. [Online; accessed 2011-09-09].

[2] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, Mass, 1997.

[1] In this module you must be able to understand code using threads. However, it will not be expected of you have to write such code yourself

[3] F. B. Gilbreth and L. M. Gilbreth. Process charts: first steps in finding the one best way to do work. Presented at the Annual Meeting of The American Society of Mechanical Engineers, New York, USA, 1921.

[4] William E. Kempf. The boost.threads library. `http://drdobbs.com/cpp/184401518`, 2002. [Online: Accessed 1 Sept 2011].

[5] James Martin and James J. Odell. *Object-Oriented Analysis and Design.* Prentice Hall Inc, Englewood Cliffs, NJ 07632, 1992.

[6] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall Inc, Englewood Cliffs, NJ 07632, 1981.

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 17: Mediator Design Pattern

Contents

17.1 Introduction

The mediator design pattern extends the observer pattern. Where the observer registers observers that get updated whenever the subject changes, the mediator registers colleagues that get updated whenever one of the other colleagues notifies the mediator of an update.

17.2 Mediator Design Pattern

17.2.1 Identification

Name	Classification	Strategy
Mediator	Behavioural	Delegation
Intent		
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. ([1]:273)		

17.2.2 Structure

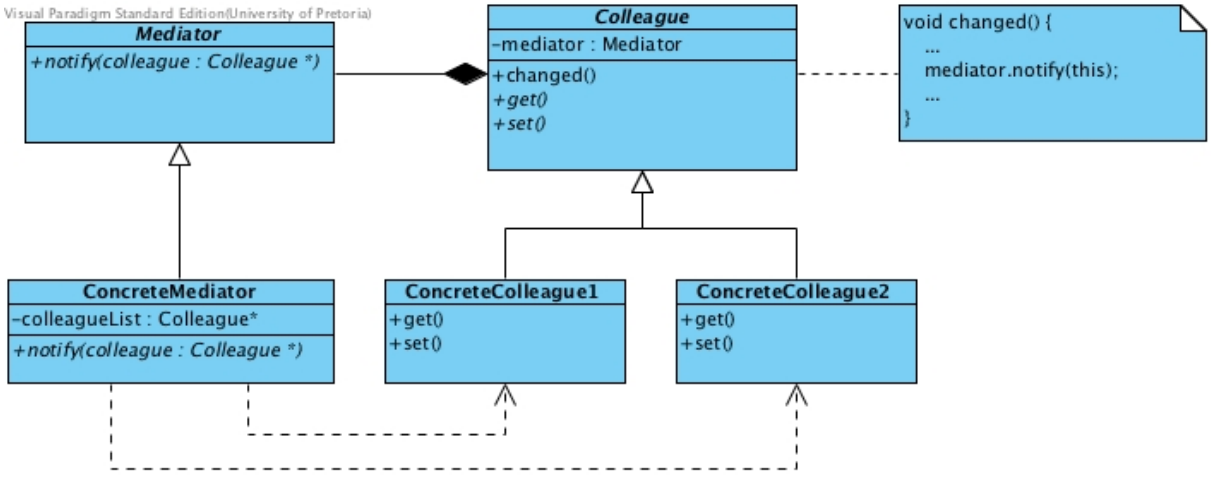


Figure 1: The structure of the Mediator Design Pattern

17.2.3 Participants

Mediator

- defines an interface for communicating with Colleague objects.

ConcreteMediator

- implements cooperative behavior by coordinating Colleague objects.
- knows and maintains its colleagues.

Colleague

- each Colleague class knows its Mediator object.
- each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

17.2.4 Problem

We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the spaghetti code phenomenon. When one wants to reuse only one or a few of the classes in a group of classes, it is virtually impossible to isolate them because they are to interconnected with one another. Trying to scoop a single serving results in an *all or nothing clump* [2].

17.3 Mediator pattern explained

17.3.1 Purpose

Though partitioning a system into many objects generally enhances reusability, proliferating interconnections tend to reduce it again. You can avoid this problem by capsulating the interconnections (i.e. the collective behavior) in a separate **mediator** object.

A mediator is responsible for controlling and coordinating the interactions of a group of objects.

17.3.2 Improvements achieved

Simplification of code updates
If the pattern is not applied and the behaviour of one of the classes in a group is changed it potentially necessitates the update of each class in the group to accommodate the changes made to this one element. The same applies when an element is added to the group or removed from the group. However, if the pattern is applied such changes will only require an update in the mediator class and none of the other classes in the group.

Increased reusability of code
The decoupling of the colleagues from one another increases their individual cohesiveness contributing to their reusability.

Simplification of object protocol
When refactoring into the mediator pattern a many-to-many relationship that exists between the elements in a group of objects is changed to a one-to-many relationship which is easier to understand and maintain.

17.3.3 Implementation issues

changed()

The `changed()` method is implemented in the colleague interface to allow each concrete colleague to call it. This method is used to notify the mediator of changes. It is the responsibility of each concrete colleague to call this method whenever it executes code that may impact on the other colleagues. Its implementation delegates to the mediator with a statement like:

```
mediator->notify(this);
```

notify()

The `notify()` method is called every time one of the concrete colleagues executes the `changed()` method. A pointer to the concrete colleague is sent as a parameter to allow the mediator to have knowledge about the originator of the notification. It is not desirable to send the content or nature of an update of a colleague to the mediator as a parameter of the `notify()` message. This is to insure that this interface is stable and generic enough to allow for different kinds of colleagues. It should get information about the nature and value of the changes that occurred and then propagate the change to all the concrete collegues. The following is pseudo code for the implementation of the `notify()` method:

```
Mediator::notify(originator: Collegue*)
{
    resultOfChange = originator->get();
    for (all colleagues)
    {
        set(resultOfChange);
    }
}
```

17.3.4 Related patterns

Facade

Facade differs from **Mediator** in that it abstracts a subsystem of objects to providea more convenient interface. Its protocol is unidirectional; that is, **Facade** objects make requests of the subsystem classes but not vice versa. In contrast, **Mediator** enables cooperative behaviour that colleague objects don't or can't provide, and the protocol is multidirectional.

Observer

Colleagues can communicate with the mediator using the **Observer** pattern.

17.4 Example

Figure 2 is a class diagram of a system illustrating the implementation of the Mediator design pattern. It is a simulation of the interaction between a number of widgets on a file dialog. It contains simple `cout` statements in the function bodies of most functions

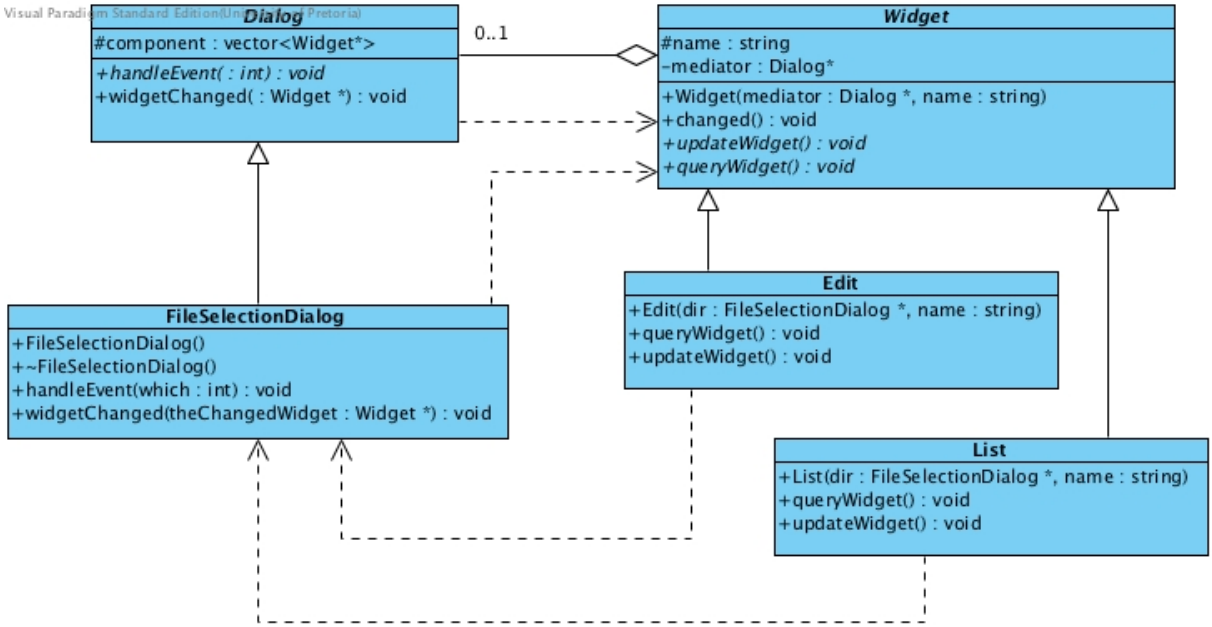


Figure 2: Class Diagram of a partial implementation of a file selection dialog

to be able to observe how the pattern operates. This example was adapted from [2]. The following table summarises how the implementation relates to the participants of this pattern:

Participant	**Entity in application**
Mediator	Dialog
Concrete Mediator	FileSelectionDialog
Colleague	Widget
Concrete Colleague	List, Edit
changed()	changed()
notify()	widgetChanged(: Widget)
get()	queryWidget()
set()	updateWidget()

- In this example, `FileSelectionDialog` is the designated the mediator for all the `Widget` siblings. The `FileSelectionDialog` does not know, or care, who the siblings of `Widget` are.

- Whenever a simulated user interaction occurs in a child of `Widget`, `Widget::changed()` signals it. This method does nothing except to *delegate* that event to the appropriate `Dialog` with the function call `mediator->widgetChanged(this)`. Note that the correct concrete mediator in the `Dialog` hierarchy will be called upon, because `mediator` is a variable which as been assigned the value of the appropriate mediator, when the child `Widget` was constructed. Also note that it passes a pointer to itself to the mediator so that the mediator can know where the change originated.

- `FileSelectionDialog::widgetChanged()` encapsulates all collective behaviour for the dialog box. It serves as the hub of communication. In this example it simply

queries the status form the `Widget` that signalled the change, and propagates the change to all its dependants.

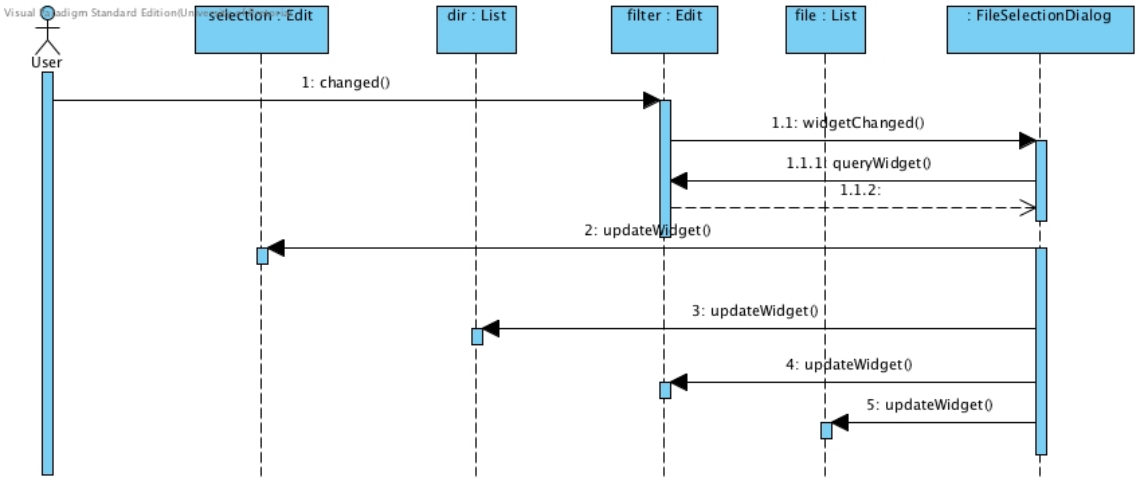


Figure 3: A Sequence Diagram to visualise the time ordering

Figure 3 illustrates how the mediator will govern the interaction between the widgets from the moment the user changes the content of `filter:Edit` until all the `Widget` objects are updated. Note that the sequence diagram shows only instantiated objects. Abstract classes such as `Dialog` and `Widget` are not part of a sequence diagram.

References

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

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

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 18: Command Design Pattern

Contents

18.1 Introduction

This Lecture Note introduces the Command design pattern. The pattern will be illustrated using a TV remote as an example.

18.2 Command Design Pattern

18.2.1 Identification

Name	Classification	Strategy
Command	Behavioural	Delegation
Intent		
Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations. ([1]:263)		

18.2.2 Problem

Used to modify existing interfaces to make it work after it has been designed.

18.2.3 Structure

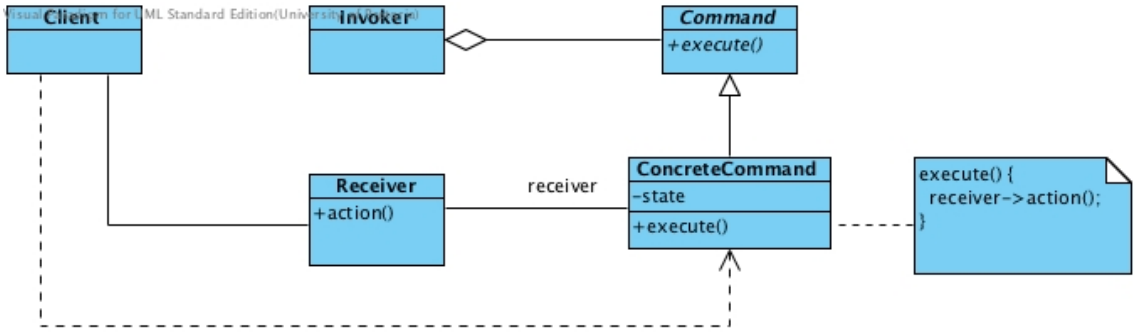


Figure 1: The structure of the Command Design Pattern

18.2.4 Participants

Command

- declares an interface for executing an operation.

ConcreteCommand

- defines a binding between a Receiver object and an action.
- implements execute() by invoking the corresponding operation(s) on Receiver.

Client (Application)

- creates a ConcreteCommand object and sets its receiver.

Invoker

- asks the command to carry out the request.

Receiver

- knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

18.3 Comand Pattern Explained

18.3.1 Related Patterns

Chain of Responsibility:
Makes use of Command to represent requests as objects.

Composite:
MacroCommands can be implemented when combining command with Composite.

Memento:
Makes use of Command to keep state the command requires to undo its effect.

Prototype:
A command that must be copied before being placed on the history list acts as a Prototype.

18.4 Example

18.4.1 TV Remote

This example will model a TV remote. The remote has two buttons, one to flip through channels and one to switch the TV on and off.

Listing 1: Implementation of a TV remote

```
#include <iostream>

using namespace std;

class TV {
  public:
    static void action(char* s) { cout<<s<<endl;};
};

class Command {
```

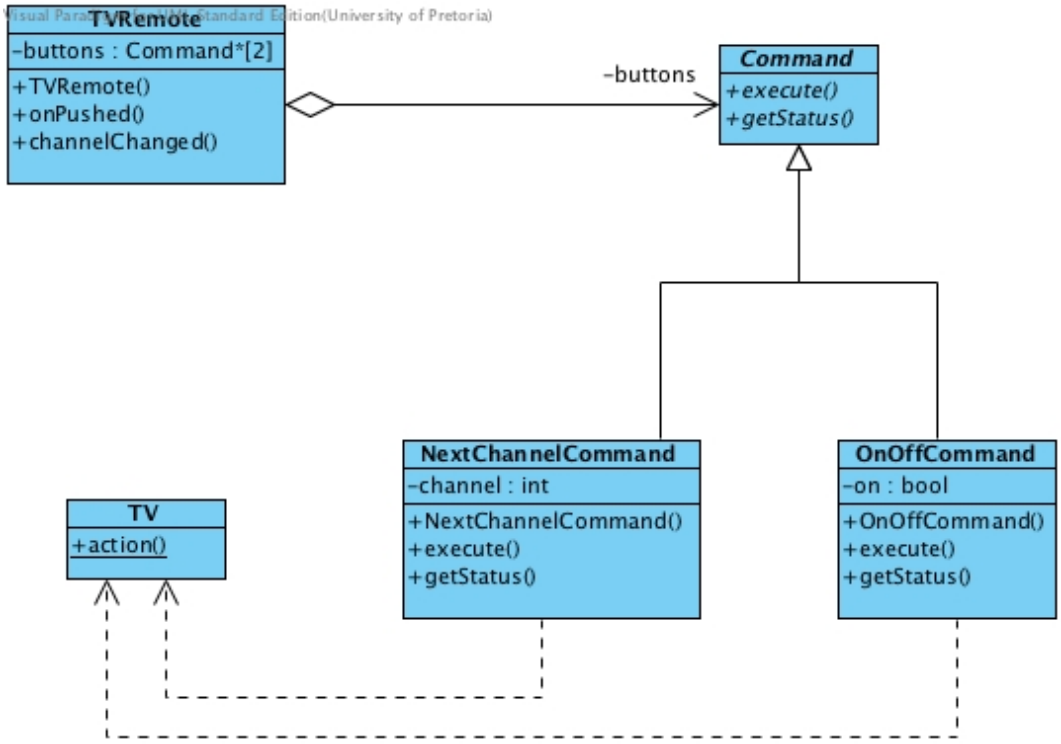


Figure 2: TV remote example

```
  public:
    virtual void execute() = 0;
    virtual char* getStatus() = 0;
};

class OnOffCommand : public Command {
  public:
    OnOffCommand(bool s) : on(s) {};
    void execute() {
      on = on?false:true;
      TV::action(getStatus());  };
    char* getStatus() {
      char* str = new char[20];
      strcpy(str,"The device is ");
      strcat(str,(on==false?"off":"on"));
      return str;
    };
  private:
    bool on;
};

class NextChannelCommand : public Command {
  public:
    NextChannelCommand() {
      channel = 1;
```

```
    };
    void execute() {
      channel = (channel==5)?1:++channel;
      TV::action(getStatus());
    };
    char* getStatus() {
      char* str = new char[20];
      strcpy(str,"The device is on channel ");
      switch (channel) {
        case 1: strcat(str,"1"); break;
        case 2: strcat(str,"2"); break;
        case 3: strcat(str,"3"); break;
        case 4: strcat(str,"4"); break;
        case 5: strcat(str,"5"); break;
      }
      return str;
    };

  private:
    int channel;
};

class TVRemote { // Invoker
  public:
    TVRemote(){
      buttons[0] = new OnOffCommand(false);
      buttons[1] = new NextChannelCommand();
    };

    void onPushed(){
      buttons[0]->execute();
    };

    void channelChanged() {
      buttons[1]->execute();
    };
  private:
    Command* buttons[2];
};



int main(){
  TVRemote* tvr = new TVRemote;
  tvr->onPushed();
  tvr->channelChanged();
  tvr->channelChanged();
  tvr->channelChanged();
```

```
    tvr->onPushed();
    tvr->channelChanged();
    tvr->channelChanged();

    return 0;
}
```

Sample output for the program is given by:

Listing 2: TV Remote Sample Output

18.5 Exercises

1. Consider the following code that illustrated the command pattern. Draw the UML class diagram.

Listing 3: Implementation of the LightSwitch

```
class Fan
{
    public:
        void startRotate() { cout << "Fan is rotating" << endl;}
        void stopRotate() { cout << "Fan is not rotating" << endl;}
};

class Light
{
    public:
        void turnOn( ) { cout << "Light is on " << endl; }
        void turnOff( ) { cout << "Light is off" << endl; }
};

class Command
{
    public:
        virtual void execute ( ) = 0;
};

class LightOnCommand : public Command
{
    public:
        LightOnCommand (Light* L) { myLight = L;}
        void execute( ) { myLight -> turnOn( ); }
    private:
        Light* myLight;
};

class LightOffCommand : public Command
```

```
{
    public:
        LightOffCommand (Light* L) { myLight = L;}
        void execute( ) { myLight -> turnOff( ); }
    private:
        Light* myLight;
};

class FanOnCommand : public Command
{
    public:
        FanOnCommand (Fan* f) { myFan = f;}
        void execute( ) { myFan -> startRotate( );}
    private:
        Fan* myFan;
};

class FanOffCommand : public Command
{
    public:
        FanOffCommand (Fan* f) { myFan = f;}
        void execute( ) {myFan -> stopRotate( );}
    private:
        Fan* myFan;
};

class Switch
{
    public:
        Switch(Command* up, Command* down)
        {
            upCommand = up;
            downCommand = down;
        }

        void flipUp( ) { upCommand -> execute( );};
        void flipDown( ) {downCommand -> execute ( );};

    private:
        Command* upCommand;
        Command* downCommand;
};

int main()
{
    Light* testLight = new Light( );
    Fan* testFan = new Fan();
```

```
    LightOnCommand* testLiOnCmnd = new LightOnCommand(testLight);
    LightOffCommand* testLiOffCmnd = new LightOffCommand(testLight);
    FanOnCommand* testFaOnCmnd = new FanOnCommand(testFan);
    FanOffCommand* testFaOffCmnd = new FanOffCommand(testFan);

    Switch* lightSwitch = new Switch(testLiOnCmnd,testLiOffCmnd);
    Switch* fanSwitch = new Switch(testFaOnCmnd,testFaOffCmnd);

    lightSwitch -> flipUp();
    lightSwitch -> flipDown();
    fanSwitch -> flipUp();
    fanSwitch -> flipDown();

    /** As opposed to
    testLight -> turnOn();
    testLight -> turnOff();
    testFan -> startRotate();
    testFan -> stopRotate();
    */

    return 0;
}
```

References

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

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 19: Adapter Design Pattern

Contents

19.1 Introduction

This Lecture Note introduces the Adapter design pattern. The adapter pattern is also referred to as a wrapper pattern which describes it intent very well. This wrapper is presented in two guises, the first adapts using delegation and the other using inheritance. Both these implementation strategies will be considered. Due to the nature of the inheritance, inheritance access specification other then public will be briefly discussed.

19.2 Adapter Design Pattern

19.2.1 Identification

Name	Classification	Strategy
Adapter	Structural	Inheritance (Class) and Delegation (Object)
Intent		
Convert an interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. ([2]:139)		

19.2.2 Problem

Used to modify existing interfaces to make it work after it has been designed.

19.2.3 Structure

The Adapter design pattern is the only pattern to which one of two structures can be applied. The pattern can either make use of delegation or inheritance to achieve its intent. The delegation structure is referred to as an Object Adapter, Figure 1, and the inheritance structure as shown in Figure 2 for the Class Adapter.

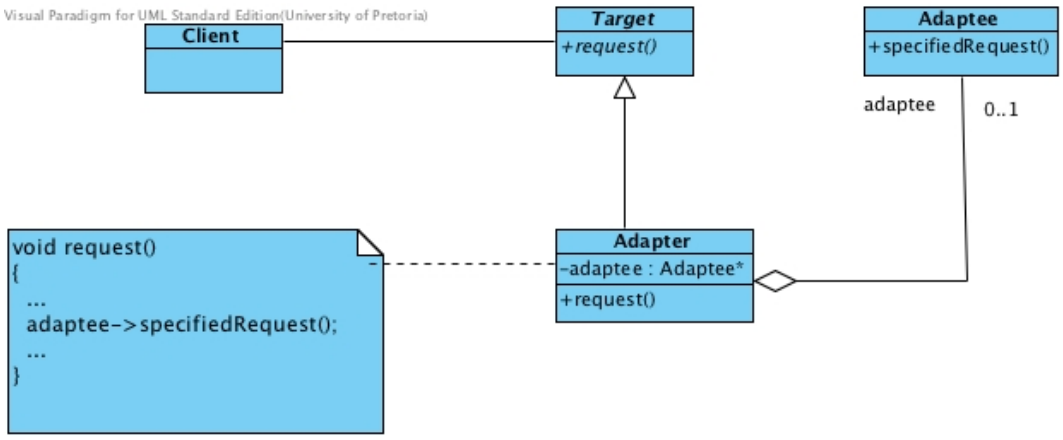


Figure 1: The structure of the Object Adapter Design Pattern

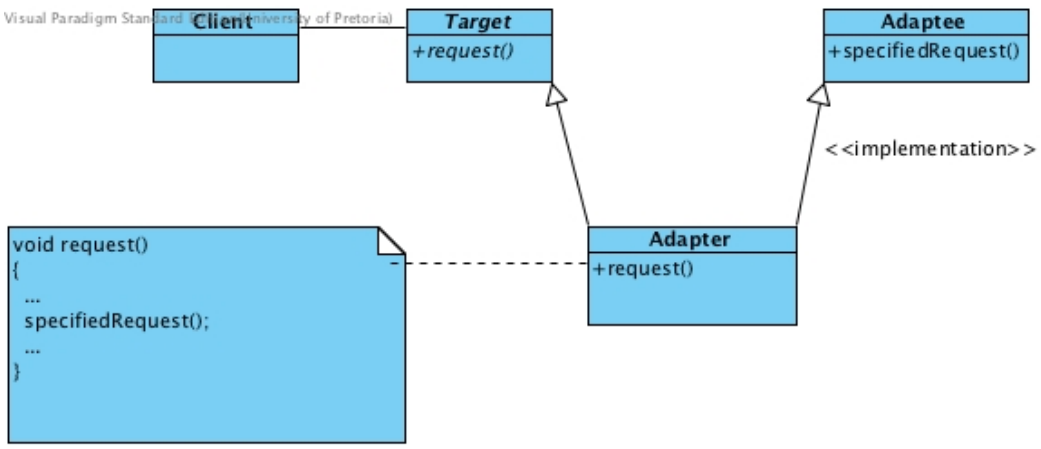


Figure 2: The structure of the Class Adapter Design Pattern

19.2.4 Participants

Adaptee

- The existing interface that needs to be adapted

Target

- Domain specific interface used by the client

Adapter

- Adapts the interface of Adaptee to the Target interface

Client

- Manipulates objects conforming to the interface specified by the abstract class Target

19.3 Protected and private inheritance in C++ explained

So far, the member access specifier (`memberAccessSpecifier` in Figure 3) for inheritance has been `public`. Two other member access specifiers for inheritance may be used, namely `protected` and `private`. Table 1 provides the visibility of the members of the base class in the derived class for each of the member access specifiers.

It can be said that a `memberAccessSpecifier` that is `private` provides the derived class with the functionality defined in the base class. Effectively, the base class is wrapped and no class that may inherit from the derived class will have access to its member functions. Private inheritance in C++ can be seen as a type of *has-a* relationship.

```
class Base {
  ...
};

class Derived : memberAccessSpecifier Base {
  ...
};
```

Figure 3: Example inheritance classes

		Inheritance access specifier of derived class		
		public	protected	private
Base member visibility	public	Derived access specifier is **public**. Derived class can access the member and so can an outside class.	Derived access specifier is **protected**. Derived class can access the member, but there is no access from an outside class.	Derived access specifier is **private**. Derived class can access the member, but there is no access from an outside class.
	protected	Derived access specifier is **protected**. Derived class can access the member, but there is no access from an outside class.	Derived access specifier is **protected**. Derived class can access the member, but there is no access from an outside class.	Derived access specifier is **private**. Derived class can access the member, but there is no access from an outside class.
	private	Derived access specifier is **private**. Derived class cannot access the member and there is no access from an outside class.	Derived access specifier is **private**. Derived class cannot access the member and there is no access from an outside class.	Derived access specifier is **private**. Derived class cannot access the member and there is no access from an outside class.

Table 1: C++ member access specifiers and base class member visibility

19.4 Adapter Pattern Explained

19.4.1 Design

Object Adapter

Object Adapter makes use of object composition to delegate to Adaptee.

Class Adapter

Class Adapter makes use of mixin idiom [4]. A mixin is an object-orientated concept by which a class provides functionality, either to be inherited or just used, but is not explicitly instantiated. Adapter inherits and implements Target (public inheritance). Adapter inherits only the implementation, or functionality, and therefore the use of private inheritance of Adaptee resulting in a *linearisation* of the hierarchy.

19.4.2 Comparison of the approaches

When does one use delegation and when does one use private inheritance. Try to always use delegation. Use composition (inheritance) only when necessary [1].

19.4.3 Real world example

Many instances of the adapter pattern can be found in data structures where one data structure such as a list is wrapped so that it behaves as another data structure, for example a stack.

Further use of the Adapter pattern is when legacy systems are being integrated into a new system. The functionality of the legacy system is therefore encapsulated and used by the new system through an adapter.

19.4.4 Related Patterns

Bridge

Structurally they are similar. However their intent is different, the Adapter changes the interface while the Bridge separates the implementation from the interface.

Decorator

Enhances an object without changing the interface.

Proxy

Defines a surrogate of to an object without changing its interface.

19.5 Example

19.5.1 Billboard

In this example, the electronic billboard is to be adapted to in order to simplify its interface. Electronic billboards can be in one of two states, either on or off with the ability

to toggle between the states. Electronic billboards in the on state display a message, to change this message another setter method is called and then the method to display needs to be called for the message to change.

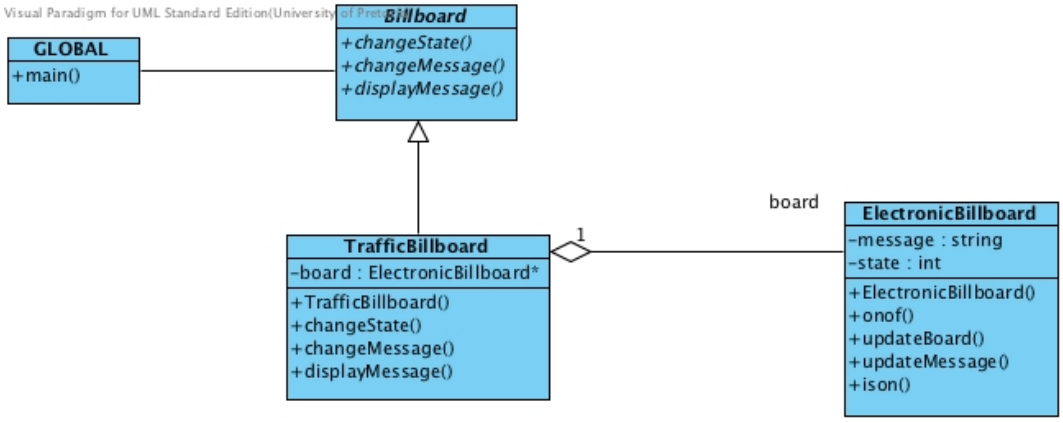


Figure 4: Billboard example - object adapter

The `Billboard` interface (Figure 4) simplifies the the electronic billboard and `TrafficBillboard` implements this simplified interface and uses the `ElectronicBillboard` functionality to do so.

Implementing the Billboard as a class adapter instead of as an object adapter is not difficult. Figure 5 shows the resulting class diagram.

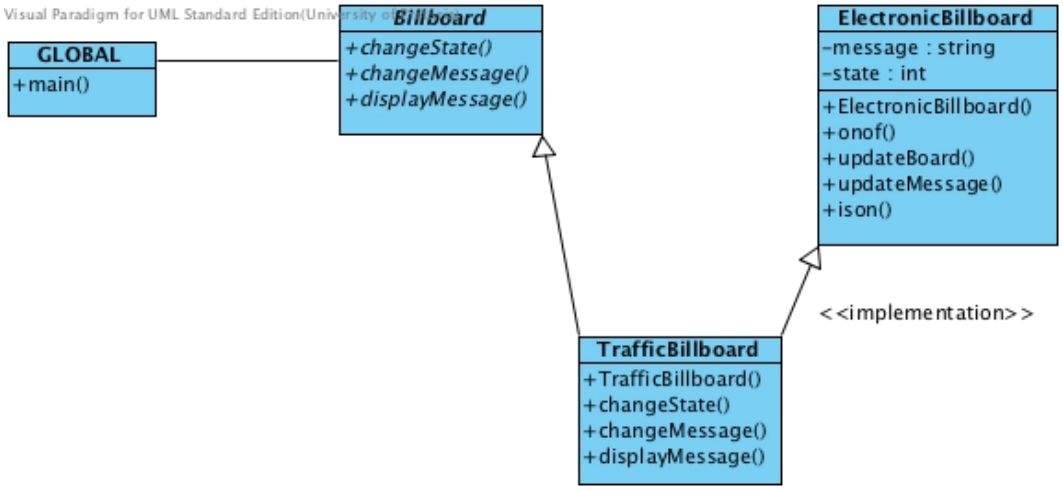


Figure 5: Billboard example - class adapter

By comparing Figures 4 and 5 it can be seen that the most significant difference between the two implementations is that the member attribute, providing the delegation functionality, is not defined in the class adapter. In order to see other subtle differences, it is necessary to look at the implementation level. Listings 1 and 2 represent the Adapter participant of the pattern for the Object Adapter and Class Adapter implementations respectively. The Object Adapter implementation instantiates an object of the Adaptee participant and access the functionality provided by the object through its public interface. The Class Adapter implementation, through private inheritance, uses and effectively

subsumes the functionality of the Adaptee participant. It is important to note, that had there been protected features in the Adaptee participant, these would have been available to the Adapter participant with the Class Adapter implementation and not with the Object Adapter implementation.

Listing 1: Object Adapter Implementation of the Billboard

```
class TrafficBillboard : public Billboard {
    public:
        TrafficBillboard() {
          board = new ElectronicBillboard("all clear");
        };
        virtual void changeState() {
          board->onof();
        };
        virtual void changeMessage(int msgId) {
          switch (msgId) {
             case 1: board->updateMessage("slow traffic ahead"); break;
             case 2: board->updateMessage("accident ahead"); break;
             default: board->updateMessage("all clear");
           }
        };
        virtual void displayMessage() {
           if (board->ison()) {
             cout << "Traffic warning: "; board->updateBoard();
             cout<<endl;
           }
           else cout<<"Board is off"<<endl;
        };
    private:
        ElectronicBillboard* board;
};
```

Listing 2: Class Adapter Implementation of the Billboard

```
class TrafficBillboard : public Billboard, private ElectronicBillboard {
    public:
        TrafficBillboard() {
          updateMessage("all clear");
        };

        virtual void changeState() {
          onof();
        };

        virtual void changeMessage(int msgId) {
          switch (msgId) {
            case 1: updateMessage("slow traffic ahead"); break;
            case 2: updateMessage("accident ahead"); break;
            default: updateMessage("all clear");
          }
```

```
        };

        virtual void displayMessage() {
          if (ison()) {
            cout << "Traffic warning: "; updateBoard();
            cout<<endl;
          }
         else cout<<"Board is off"<<endl;
        };

};
```

19.5.2 Rectangle

This example is available on the internet in different guises [3]. This is yet another adaptation along the same theme and illustrates the implementation of a class adapter. The class diagram for the rectangle is given in Figure 6 and the implementation of the Adapter participant in listing 3. `LegacyRectangle` specifies a rectangle by using 4 values, the first two values represent the x and y coordinates of the top left corner of the rectangle and the last two values the x and y coordinates of the bottom right corner of a rectangle. The adapted rectangle defines a rectangle by its top left corner coordinates and then a width and a height value towards the right and down.

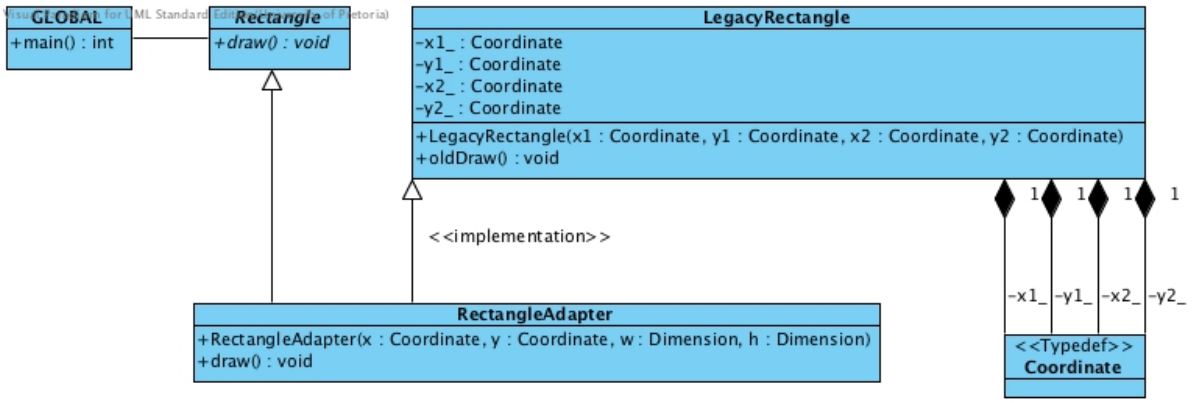


Figure 6: Rectangle example - class adapter

Listing 3: Rectangle Class Adapter Implementation

```
class RectangleAdapter : public Rectangle,
                         private LegacyRectangle {
public:
   RectangleAdapter( Coordinate x, Coordinate y,
                     Dimension w, Dimension h )
                  : LegacyRectangle( x, y, x+w, y+h ) {

    cout << "RectangleAdapter: create.  (" << x
         << "," << y
```

```
            << "), width = " << w
            << ", height = " << h << endl;
    }
    virtual void draw() {
        cout << "RectangleAdapter: draw." << endl;
        oldDraw(); }
};
```

19.6 Exercises

1. Make use of the state design pattern to encapsulate the messages displayed by the traffic billboard.

2. Rewrite the example given in Section 19.5.2 as an object adapter.

References

[1] Marshall Cline. C++ faq: Inheritance - private and protected inheritance, 1991–2011. URL `http://www.parashift.com/c++-faq-lite/private-inheritance.html`. Online; accessed 26 September 2011.

[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] Yannis Smaragdakis and Don S. Batory. Mixin-based programming in c++. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, GCSE '00, pages 163–177, London, UK, 2001. Springer-Verlag. ISBN 3-540-42578-0. URL `http://dl.acm.org/citation.cfm?id=645417.652070`.

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 20: Chain of Responsibility Design Pattern

Contents

20.1 Introduction

The design and implementation of the chain of responsibility design pattern is fairly straight forward. An example that is often used to illustrate the intent of this pattern is the implementation of a cash dispenser as is commonly found in automatic teller machines (ATM) and coin operated machines capable of returning change. In this lecture we discuss this pattern and explain it at the hand of a simulation of the cash dispensing mechanism in an ATM.

20.2 Chain of Responsibility Design Pattern

20.2.1 Identification

Name	**Classification**	**Strategy**
Chain of Responsibility	Behavioural	Delegation
Intent		
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. ([1]:223)		

20.2.2 Problem

There is a potentially variable number of handler objects, and a stream of requests that must be handled. We need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings [2].

20.2.3 Structure

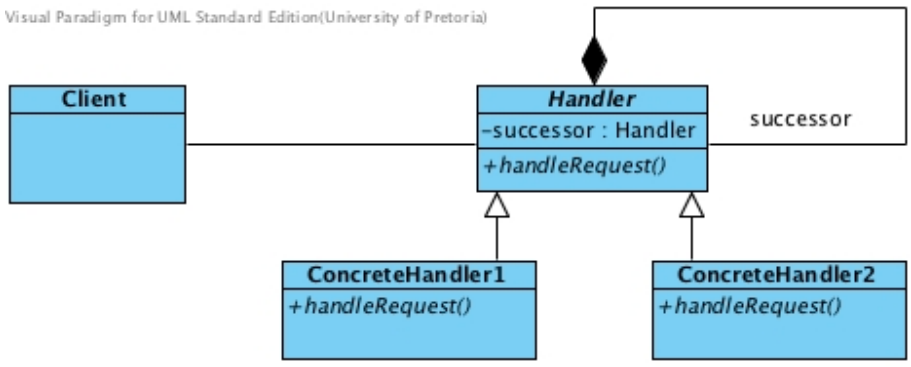


Figure 1: The structure of the Chain of Responsibility Design Pattern

20.2.4 Participants

Handler

- Defines an interface for handling requests.
- Implements the successor link.

Concrete Handler

- Handles requests it is responsible for.
- Can Access its successor.
- If the Concrete Handler can handle the request, it does so; otherwise it delegates the request to its successor via Handler.

Client

- Initiates the request to a ConcreteHandler object in the chain.

20.3 Chain of Responsibility Pattern Explained

20.3.1 Design

A simple way to implement the pattern is to declare the `handleRequest()` in the interface as a virtual method, but not to make it pure virtual. Also provide a default implementation that delegates the request to its successor if it exists. This way if the concrete handler does not implement `handleRequest()`, the request will automatically be delegated to its successor who might be able to handle it. One may also provide a default action that can be taken if there is no successor to prevent the situation that an un-handled request go undetected. The following is the code and related figure (figure 2) for a generic handler interface that applies the chain of responsibility design pattern:

```
class Handler
{
    public:
        Handler(Handler* s) : successor(s) { }
        virtual void handleRequest();
    private:
        Handler* successor;
};

void Handler::handleRequest()
{
    if (successor)
        successor->handleRequest();
    else
        //define action in case of no successor here
}
```

Note the difference in representation between the `Handler` class given in figure 1 with that given in figure 2. The association is implementation dependent. In figure 'refChainStructure the successor is placed in the stack, while in the next implementation is it placed on the heap. When the successor is on the stack, it is tightly coupled to the Handler class

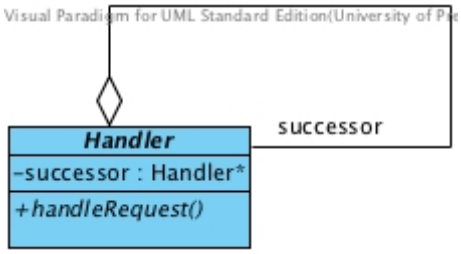


Figure 2: Handler class of the Chain of Responsibility Design Pattern

and will be destroyed when the Handler class goes out of scope. On the heap, it is the responsibility of the Handler class when it destructs.

The implementation of the concrete handlers contains the "intelligence". It can contain code with heuristics to decide to handle the request, ignore the request or pass the request to its successor. The following is the code for a generic concrete handler that applies the chain of responsibility design pattern. If A is true this concrete handler will handle the request. If B is true it will delegate to its successor by calling the implementation of `handleRequest()` in its parent class. If both conditions A and B are false it will ignore the request.

```
class ConcreteHandler : public Handler
{
    public:
        void handleRequest();
}

void ConcreteHander::handleRequest()
{
    if (A)
        //define action to handle the request here
    else if (B)
       Handler :: handleRequest();
}
```

It is important to apply this pattern only if the solution requires multiple handlers for a request that may differ dynamically. Do not use Chain of Responsibility when each request is only handled by one handler, or, when the client object knows at compile time which service object should handle the request [3].

20.3.2 Improvements achieved

- **Reduced coupling**
  The pattern frees an object from knowing which other object handles a request. An object only has to know that a request will be handled. Both the receiver and the sender have no explicit knowledge of each other, and an object in the chain doesn't have to know about the chain's structure. As a result, Chain of Responsibility can simplify object inter-connections. Instead of objects maintaining references to all candidate receivers, each object keeps a single reference to its successor.

- **Added flexibility in assigning responsibilities to objects**
  Chain of Responsibility gives you added flexibility in distributing responsibilities among objects. You can add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time. You can combine this with subclassing to specialise handlers statically.

20.3.3 Disadvantage

- **Receipt isn't guaranteed.**
  Since a request has no explicit receiver, there's no guarantee that it will be handled – the request can fall off the end of the chain without ever being handled. A request can also go unhandled when the chain is not configured properly.

20.3.4 Real world example

source: `http://sourcemaking.com/design_patterns/chain_of_responsibility`

In an ATM there is physical handler for each kind of money note inside the machine. One for R200 notes, one for R100 notes, down to one for R10 notes. While the amount that still needs to be dispensed can be handled by a specific handler, it will dispense a note and reduce the amount. If the amount can not be handled by a specific handler it will simply delegate to the next handler.

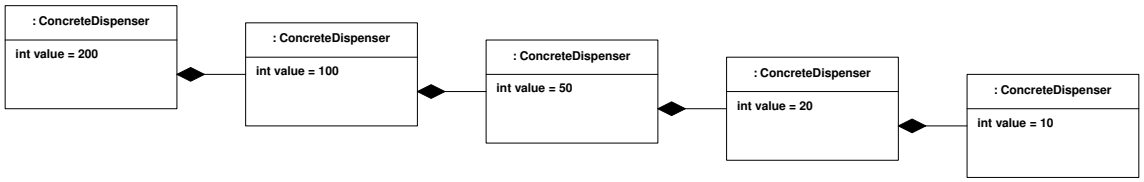


Figure 3: An object diagram showing a chain of ConcreteDispenser objects

An example implementation simulation an ATM can be found in `L25_atm.tar.gz`. The program instantiates five ConcreteDispenser objects and arrange them in a chain as shown

in Figure 3. The following are two sample test runs of this simulation that illustrates the the chain of events for dispensing cash in this simulation:

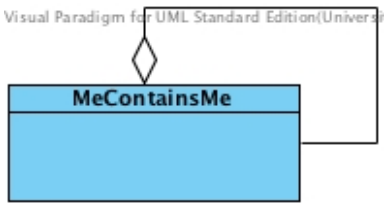
```
Amount to be dispensed: R80
R80 to small for R200 dispenser - pass  on
R80 to small for R100 dispenser - pass  on
R50 dispenser dispenses R50
R30 to small for R50 dispenser - pass  on
R20 dispenser dispenses R20
R10 to small for R20 dispenser - pass  on
R10 dispenser dispenses R10
R0 to small for R10 dispenser - pass  on
Required amount was dispensed

Amount to be dispensed: R605
R200 dispenser dispenses R200
R200 dispenser dispenses R200
R200 dispenser dispenses R200
R5 to small for R200 dispenser - pass  on
R5 to small for R100 dispenser - pass  on
R5 to small for R50 dispenser - pass  on
R5 to small for R20 dispenser - pass  on
R5 to small for R10 dispenser - pass  on
R5 can not be dispensed
```

20.3.5 Related Patterns

Composite

Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor.

Composite and Decorator



The Chain of Responsibility, Composite and Decorator patterns all have recursive composition, i.e. they have a pointer to an object of its own kind as an instance variable.

Command, Mediator, and Observer

Chain of Responsibility, Command, Mediator, and Observer, address how you can **decouple senders and receivers**, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers.

20.4 Implementation Issues

20.4.1 Implementing the successor chain

When implementing the successor chain the link to a concrete handler's successor is defined in the handler interface. This link can be defined private to obligate the concrete handlers to delegate responsibility to their successors through the handler interface.

It is also possible to implement a successor chain through re-use of existing links. When the concrete handlers are already in a structure, for example being elements of an application of the composite pattern or the decorator pattern, the existing links may be re-used to form a successor chain. Using existing links works well when the links support the chain you need. However, if such existing structure doesn't reflect the chain of responsibility your application requires this is not an option.

20.5 Example

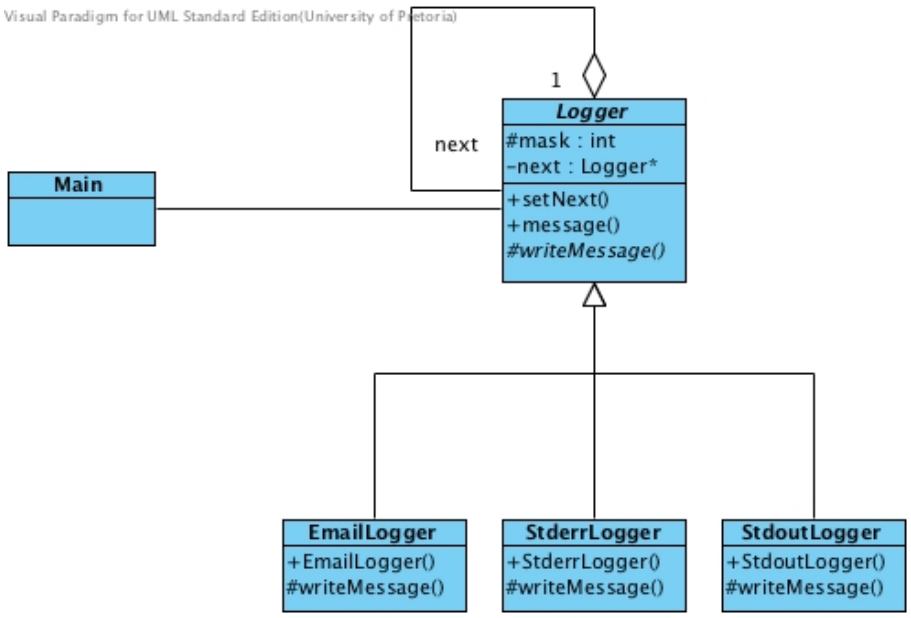


Figure 4: Class Diagram of an Implementation of a Logger

Figure 4 is a class diagram of a system illustrating the implementation of the Chain of Responsibility design pattern. It is a prototype of a system that can be used to log events in a log file. The following table summarises how the implementation relates to the participants of this pattern:

Participant	**Entity in application**
Handler	Logger
Concrete Handlers	StdoutLogger, EmailLogger, StderrLogger
handleRequest()	writeMessage(: string)

Handler

- `Logger` acts as the `Handler` interface.
- It defines an interface for handling and delegating requests.
- It has a private instance variable `next` and a public setter for this variable. They are needed to be able to link Loggers in a chain of responsibility.
- `message()` acts as a template method. It implements the intelligence to execute the request and/or to delegate to the next Logger in the chain.
- It has a protected instance variable `mask` this is set when a concrete Logger is created. It is used to indicate the level of detail that needs to be logged by the specific concrete logger.

Concrete Handler

- The concrete handlers that are implemented are `StdoutLogger, EmailLogger` and `StderrLogger`. Their priorities are set on creation in their respective constructors.
- The implementation of the `writeMessage(:string)` of each of these loggers simply writes a fixed string to stdout. In a real application more detail can be gathered and be written to a log file.

Client

- In this application the client is implemented in the main routine.
- This main routine is a simple test harness to illustrate how the chain of responsibility acts in different situations.
- It sets up a chain with four loggers and then triggers the chain in three different conditions.

References

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

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

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

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 21: UML Communication Diagrams

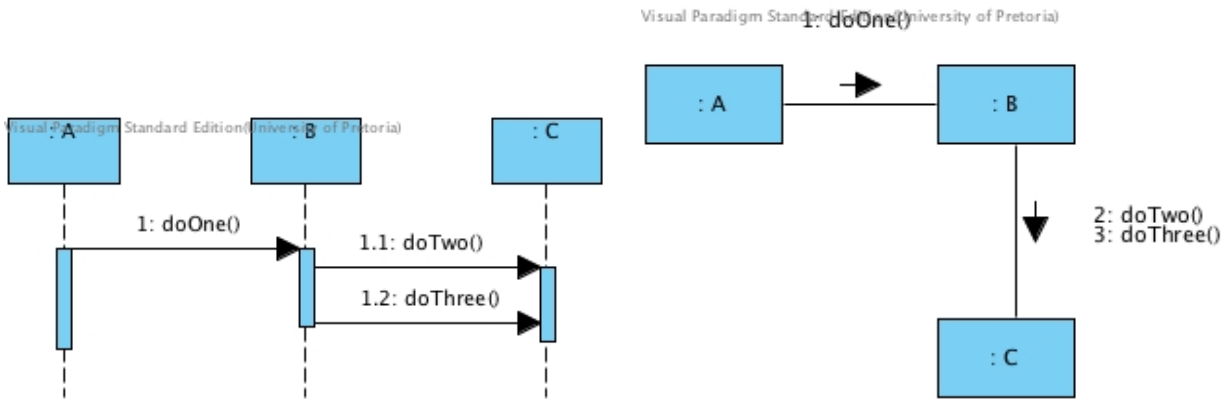
Contents

21.1 Introduction

UML 2.0 includes a number of interaction diagrams. These are sequence diagrams, interaction overview diagrams, timing diagrams and communication diagrams. In this lecture we look specifically at communication diagrams. They are used to model which objects interact with one another in terms of the messages they pass to one another. While all interaction diagrams model these interactions, communication diagrams emphasise relations between the originators and the receivers of messages.

The way in which object oriented programs systems produce useful results is mainly through passing messages between objects. These messages appear in the form of method calls. A communication diagram is used to visualise which objects are involved in the execution of a reaction to some event. In UML 1 these diagrams were called Collaboration diagrams.

Communication diagrams and sequence diagrams are very similar. Figure 1 illustrate the differences and similarities between these types of diagrams. As can be seen in the figure they both model method calls between objects. They use different layouts to show the same information.



Sequence diagram Communication diagram

Figure 1: Comparison of UML Sequence diagrams and UML Communication diagrams

The difference between sequence diagrams and communication diagrams can be summarised as follows:

- sequence diagrams
  - Easier to read call-flow sequence
  - More notation options allows for higher expressiveness
- communication diagrams
  - Easier to observe which objects are involved in the communication
  - Free format spacing allows for easier maintainance

21.2 Basic Notational Elements

The basic elements in a communication diagram are objects. A communication diagram shows that there exist communication between objects by connecting them with a link line. The detail of the communication between objects is visualised in terms of the messages that are passed between these objects. Often the order in which messages are passed are indicated by using sequence numbers.

21.2.1 Objects and Actors

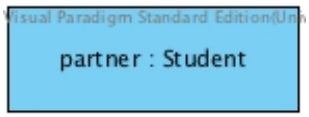


Figure 2: An object

Figure 3: An actor

Figure 2 shows an object. The same notation that is used in UML Object diagrams and UML Sequence diagrams to show objects, are used in UML Communication diagrams. An object is shown as a rectangle containing the object name and the class name of which it is an instance. The object name is separated from the class name using a colon. An object my be anonymous. For example when an object is created but not assigned to a variable. When an object is anonymous it is shown as a rectangle containing only the class name of the class of which it is an instance. In this case the class name is preceded by a colon. The objects shown in Figure 4 are anonymous objects.

Figure 3 shows the same object shown in Figure 2 as an actor. Any object may be shown as an actor. An actor is a stick man labeled with the object name and the class name of which it is an instance. Often actors are anonymous. When anonymous objects are named, technically the class name should be preceded by a colon. However, this colon is often deemed redundant.

The actor notation for objects are usually reserved to represent a user. Such user is labeled with a name (without a colon) to specify the type of user. This usage can be seen in Figure 7.

21.2.2 Link line

Figure 4: A link line between objects

When objects exchange messages at any stage during execution, they are connected to one another using a link line. A link line is a solid line connecting two objects. It merely

indicates that communication between the objects is possible. It is preferable that link lines do not intersect in a diagram.Intersecting link lines, however, are often unavoidable.

21.2.3 Messages

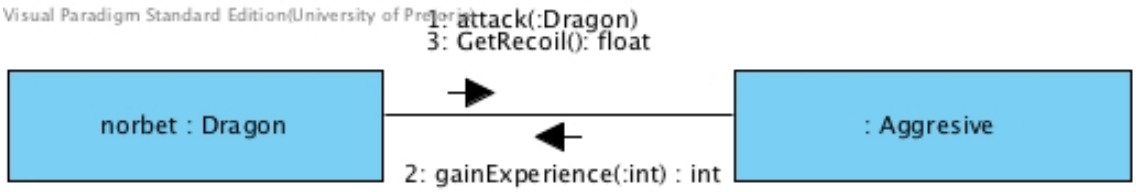


Figure 5: Messages flowing on a link line between objects

The detail of the communication between objects is visualised in terms of the messages that are passed between objects. A message is indicated with a small arrow showing the direction of the method call. The arrow head points toward the object that has the method that is executed. The arrow is usually labeled with the signature of a method that is called. The position of the label is not prescribed. It should, however, be placed in such a way that it is clear which arrow belongs to which label. The signature of the method need only include detail that is relevant to the situation. Usually the data types of its parameters and return type are sufficient. If the return type of a method is `void`, the return type is often omitted. Sometimes descriptive names without data types are used. As can be seen in the label with number 3 in Figure 7, the label may even be a description of an action in natural language.

In UML Sequence Diagrams there is an explicit distinction between methods that are synchronous and methods that are asynchronous. Synchronous methods are those that returns value. As explained in Section **??**, different arrow head shapes are used to indicate the distinction. Contrary to this, UML Communication diagrams do not distinguish between methods returning values to their callers and methods that do not return values. In UML Sequence diagrams a returning dashed line is used to indicate the passing of a return value. In UML Communication diagrams this detail is omitted.

When multiple messages are sent between two objects, they are indicated along the same single link line between objects. As can be seen in Figure 5, messages in *both* directions may flow along the same link line.

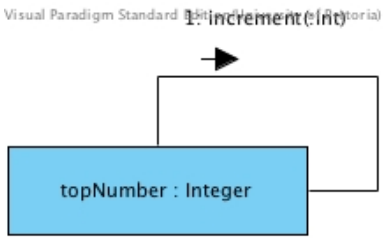


Figure 6: A reflexive message

A reflexive message is when an object calls a method that is defined in its own class. More often than not, reflexive messages are not shown in communication diagrams. It is, however, permissible to show a reflexive message as shown in Figure 6.

21.2.4 Sequence numbers

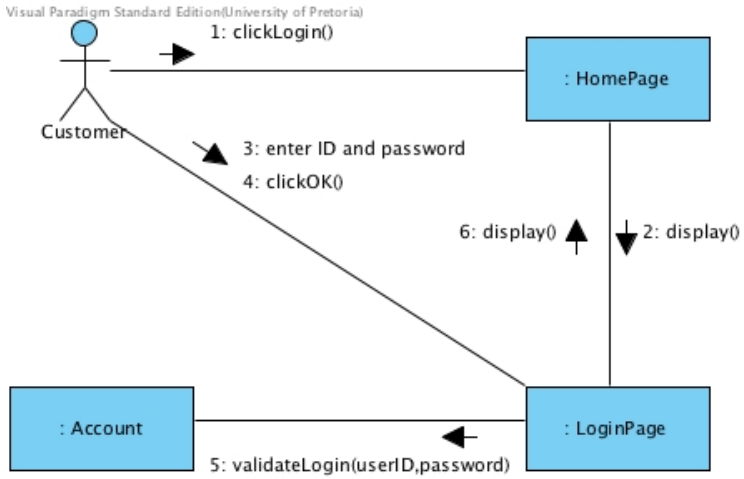


Figure 7: Messages that are numbered to indicate the order of execution

Although the order of messages are usually not relevant in communication diagrams. The order of actions may be shown by numbering the messages in the order that they are executed as shown in Figure 7 taken from [2]. The numbering scheme for numbering messages in a communication diagram is not prescribed and may include sub-numberings. Figure 11 is an example of a communication diagram that uses a decimal numbering scheme with sub-numberings.

21.3 Advanced notational elements

Sometimes it is necessary to indicate advanced detail about communications between objects. In most cases when this is required a UML Sequence Diagram will serve better to model the required detail. Additional detail that can be modelled with some limitations is the modelling of conditional actions and repetition. In this section we deal with these special cases.

21.3.1 Creation and Destruction

It is important to realise that objects in a communication diagram are *instantiated* instances of classes in a system. Creation and destruction of objects are not shown UML communication diagrams.

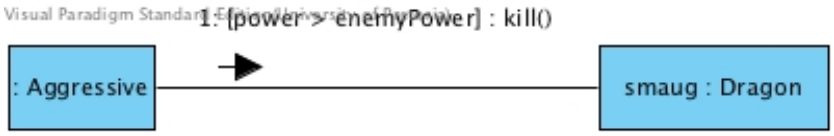


Figure 8: Syntax for a conditional flow

21.3.2 Conditional messages

Branching happens when the program flow contains conditional statements. Figure 8 shows the notation to model a conditional message. The condition that needs to be true for a message to be called is shown as a guard. The syntax to indicate the condition in square brackets is the same as in UML Activity diagrams and UML State diagrams. In this example an anonymous Aggresive object will order a Dragon object named smaug to kill if its power is greater than that of its enemy.

21.3.3 Iteration

Figure 9 contains notations to model iteration. Iteration is modeled by showing a * before the method that is repeatedly executed. The condition that needs to be true for operation to be repeated may be shown as a guard next to the *. The guard condition may be shown as

- a boolean expression (See Figure 10)
- starting and ending values (See Figure 9)
- a counter name (See Figure 9)
- a natural language expression (See Figure 11)

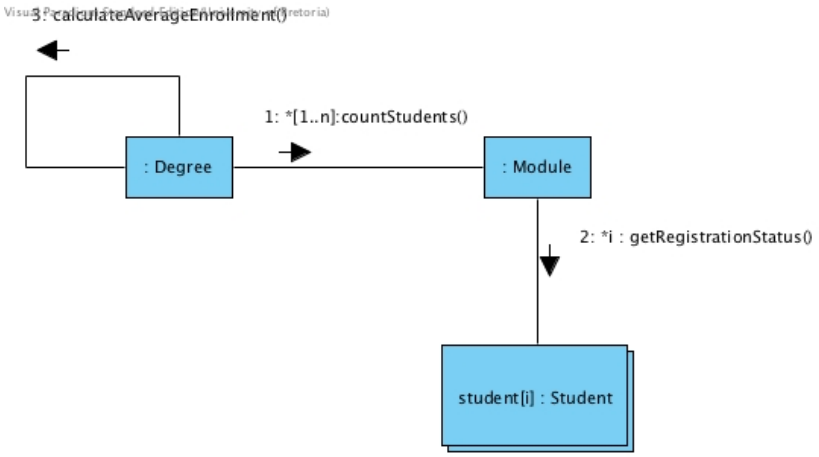


Figure 9: Communications including loops

Figure 9 models the communication of a system that loops through a number of modules and within each module loops through a number of students in order to calculate

the average enrolment per module for a certain degree. In this example the message `countStudents()` is executed with a starting value of 1 and ending value of n. In this case n represents the number of modules. The `getRegistrationStatus()` statement is executed for each value of i. In this case the counter i is used to iterate over the students in the module.

The guard condition for a repeated statement in a UML Communication diagram is optional (it may be omitted). In Figure 13 the guard is omitted because the condition for the loop is obvious from the context and therefore deemed redundant.

21.4 Examples

21.4.1 Get relevant reports

The communication diagram in Figure 10 correlates with the loop fragment of the sequence diagram shown in Figure 14 of L16_Sequence.pdf. Note how the condition for execution of the `add()` message is given as a guard in terms of a boolean expression. The guard of the loop fragment is shown as a guard for the repetition of `getNextReport()` message.

Note that although the `getRequiredSecurityLevel()` message appears in the loop fragment in Figure 14 of L16_Sequence.pdf, it is not shown as an action that is repeated, because this message is only sent once to a specific instance. Every time the loop is executed, it executes with another instance of `Report` that gets the message only once. Similarly the `add()` message as well as the `hasAnotherReport()` messages are also not shown as a repeated statements. Having them starred here would mean that they are loops nested inside the loop indicated by the star preceding the `getNextReport()` message, which is not the case here.

Figure 10: Get all reports and add the relevant ones to `availableReports`

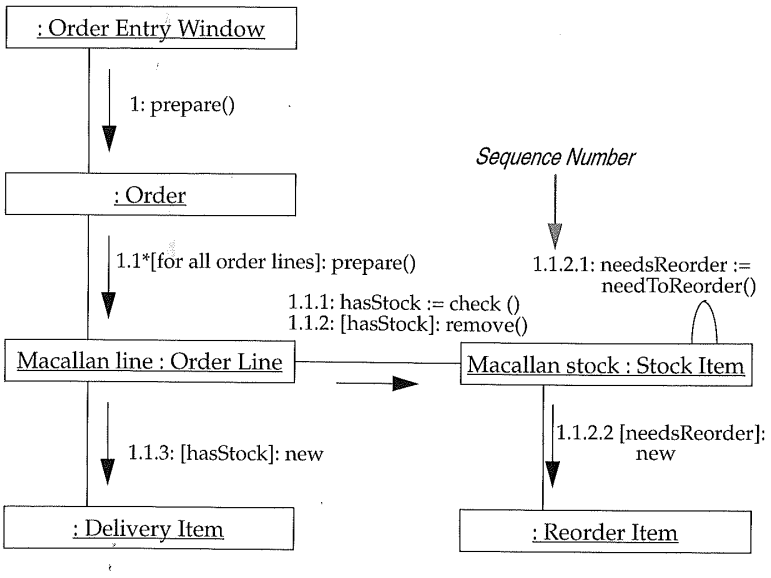


Figure 11: Preparing an order

21.4.2 Preparing an order

The communication diagram in Figure 11 was taken form [1]. It is actually a Collaberation diagram in UML 1.2. It shows the communication between objects when executing the `prepare()` method in the `Order` class.

Note that the object names are underlined. This was required in UML 1.2 Collaboration diagrams. In UML 2.0 object names are no longer underlined.

In this diagram decimal numbering with sub-numbers to model the order in which the operations are executed.

The * in operation number 1.1 indicates that this operation is executed repeatedly. The guard condition for this repetition is given as a natural language expression. It indicates that this operation should be executed for all the order lines that exist in the system.

Operations number 1.1.2, 1.1.2.2 and 1.3 are conditional statements. The guards for all these conditional statements are boolean values. Operations number 1.1.3 and 1.1.2.2 are creational operations.

21.4.3 Sending mail to a mailing list

Figure 12 is a sequence diagram showing the actions of a portion of a system that creates a `Mailer` object and a `Mailing List` object. It then applies the Iterator design pattern to iterate through the items on the mailing list to send an email message to each of them.

Figure 13 is the communication diagram to model the communication between the objects in the same portion of the system that is modelled in Figure 12.

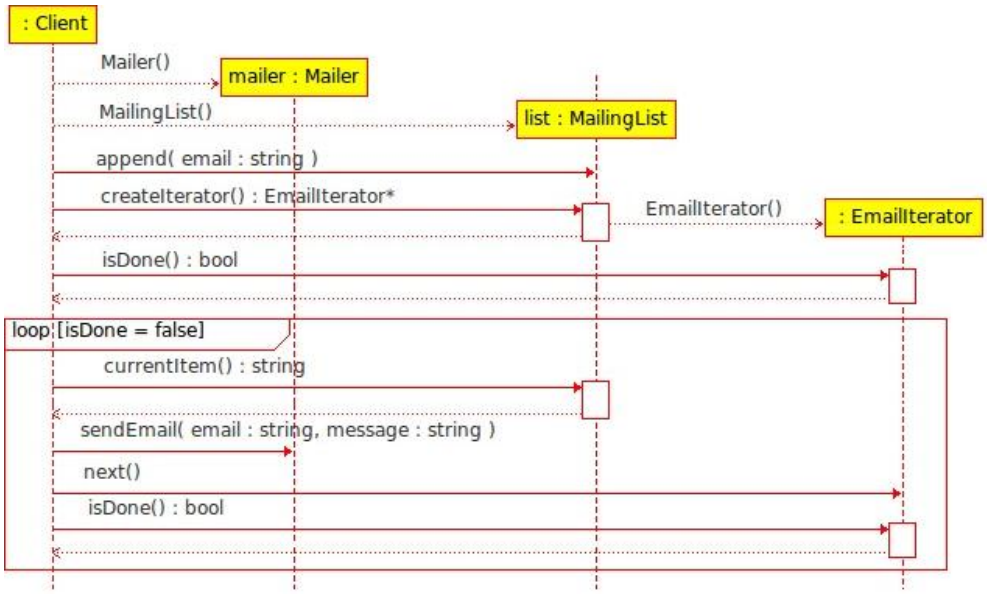


Figure 12: Sequence diagram

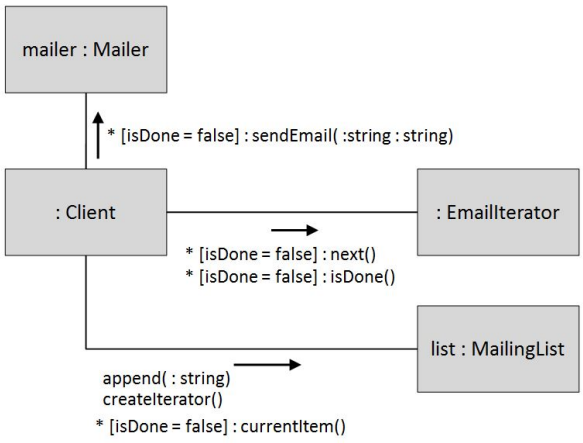


Figure 13: Communication diagram correlating with Figure 12

References

[1] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, Mass, 1997.

[2] Kendall Scott. *Fast Track UML 2.0*. Apress, Berkeley, CA, 2004.

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 22: Builder Design Pattern

Contents

22.1 Introduction

The builder pattern is a creational pattern that adds an additional level of abstraction in order to separate the process of construction of a complex object from the representation of the object. This allows the designer to easily add or change representations without having to change the code defining the process. It will also be possible to change the process without having to change the representations.

22.2 Builder Design Pattern

22.2.1 Identification

Name	Classification	Strategy
Builder	Creational	Delegation
Intent		
Separate the construction of a complex object from its representation so that the same construction process can create different representations. ([1]:97)		

22.2.2 Structure

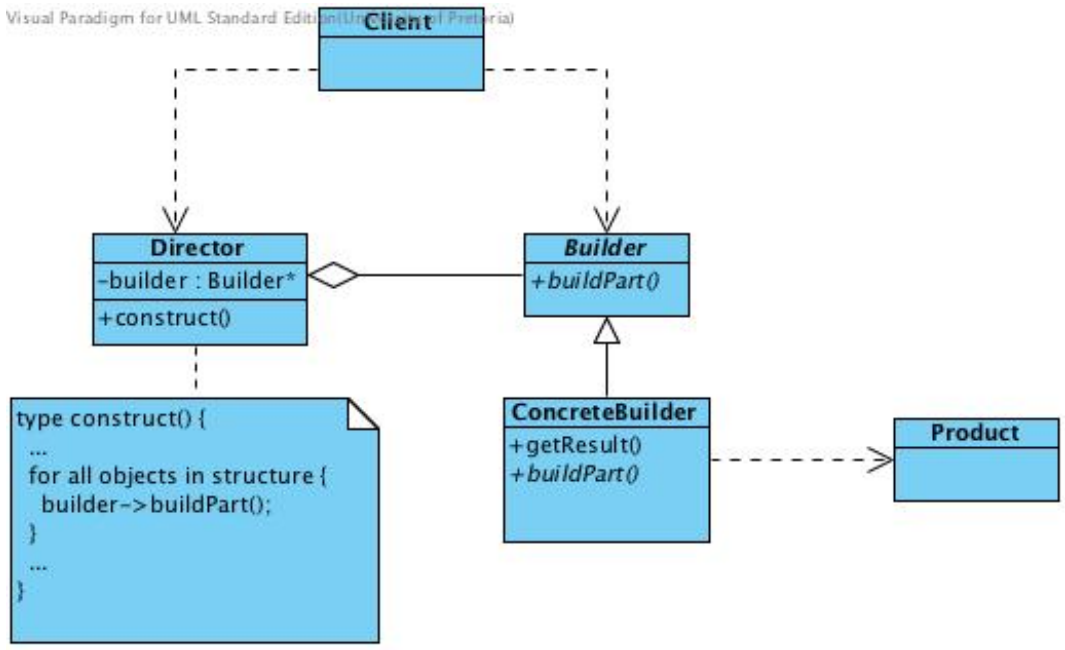


Figure 1: The structure of the Builder Design Pattern

22.2.3 Participants

Builder

- specifies an abstract interface for creating parts of a Product object.

Concrete Builder

- constructs and assembles parts of the product by implementing the Builder interface.
- defines and keeps track of the representation it creates.
- provides an interface for retrieving the product

Director

- constructs an object using the Builder interface.

Product

- represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
- includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

22.2.4 Problem

An application maintains a complex aggregate and provide for the construction of different representations of the aggregate. There is a need to design the application in such a way that the addition of more representations of the aggregate would require minimal modification of the application.

22.3 Practical Example

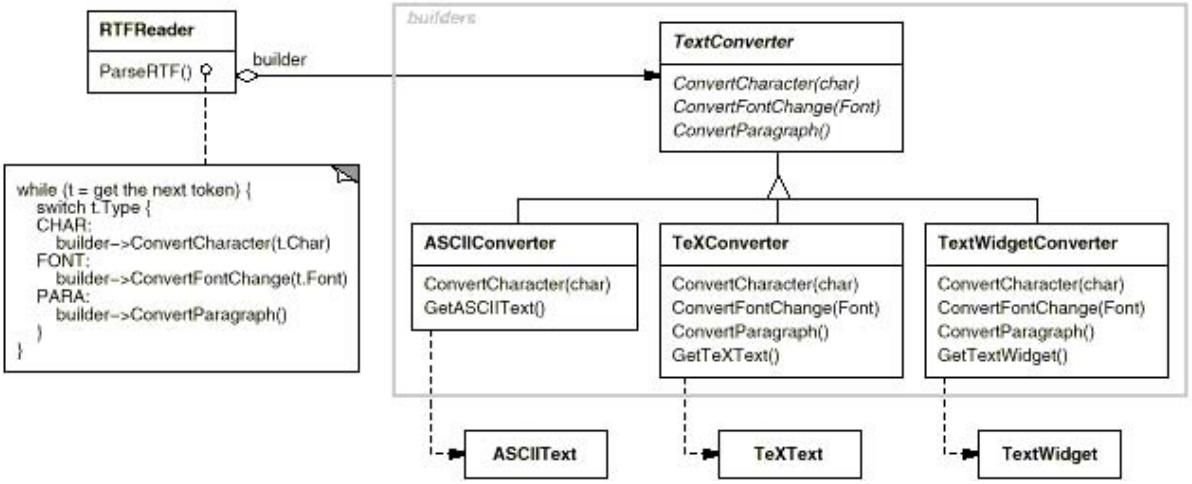


Figure 2: The design of a text converter

An example where the application of the builder pattern is useful offered by [1] is a parser of documents in RTF format that is used to produce documents in ASCII text format,

in TEX format or in a custom format for a text widget. Figure 2 is the class diagram of this application. The participants of the builder pattern in this implementation can be identified as follows:

Participant	**Entity in application**
Director	RTF Reader
Builder	TextConvereter
Concrete Builders	ASCIIConverter, TeXConverter, TextWidgetConverter
Products	ASCIIText, TeXText, textWidget
construct()	parseRTF
buildPart()	convertCharacter(), convertFontChange(), convertPraragraph()
getResult()	getASCIIText(), getTexText(), getTextWidget()

The extension of this system to produce other text formats will entail creating a concrete builder for the required format. For example to add a DocConverter that can create a MS Word document. It will be also easy to re-use these converters by another director. For example if you have built your own LATEXeditor, you can re-use these builders to convert the LATEXsource to any one of the supported formats.

22.4 Builder compared with Factory Method

The following can be observed when the structure of the Builder pattern is compared with that of the Factory Method pattern:

- The Builder pattern contains a Director class, which does not exist in the Factory Method pattern. The `construct()`-method that is defined in the Director participant of the Builder pattern, is the equivalent of the `anOperation()`-method as defined in the Creator participant of the Factory Method pattern. Thus the Builder Pattern requires an additional class in which the algorithm describing the process to construct an object is defined.
- The Builder pattern does not have an abstract product as does the Factory Method pattern. It is explained in [1] that when the Builder Pattern it used it is likely that the concrete products are likely to be so diverse that there is little to gain from giving these products a common parent class. This implies that if the application requires the created products to have a common interface, the Builder design pattern is probably not the most suitable pattern to use for the application.

22.5 Builder Pattern Explained

22.5.1 Interaction and Collaboration

Figure 3 is a sequence diagram that illustrates how the participants of the Builder pattern cooperate to create an object and give the client access to the created object. The client has to create or be given a concrete builder capable of constructing the required product. The client also has to have access to a director which defines the process to construct the

required product and knows the correct concrete builder that will assemble the required product. If all this is in place, the client simply issue a command to the director to construct the required product and retrieve it from the concrete builder when completed.

The build process as defined in the director is executed in terms of a series of calls to the concrete builder which will create the product and assemble it by adding parts to it. The process of assembling the product as well as the internal structure of the product is hidden from the client.

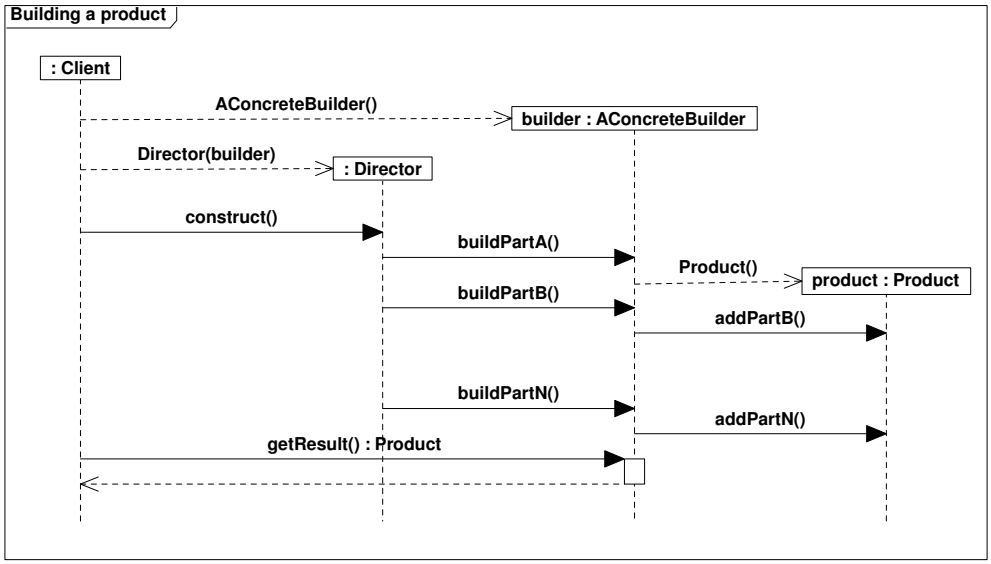


Figure 3: Cooperation of the participants of the Builder Pattern

22.5.2 Improvements achieved

[1] offers the following consequences of the application of the builder design pattern:

- **Variation product's internal representation**
  Builder object provides the director with an abstract interface for constructing the product. The interface lets the builder hide the representation and internal structure of the product. It also hides how the product gets assembled.

- **Separation of code for construction and representation**
  The Builder pattern improves modularity by encapsulating the way a complex object is constructed and represented. Clients need not know anything about the classes that define the product's internal structure.

- **Finer control over the construction process**
  Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the director's control. Only when the product is finished does the client retrieve it from the builder.

22.5.3 Common Misconception

Programmers are often under the impression that the application of a complicated algorithm for the construction of multi-part objects constitutes the application of the Builder pattern. However, if this algorithm is implemented in the abstract class of the 'Concrete Builder' objects, it is in fact an implementation of the Factory Method pattern. Thus, we do not agree with [2] who states that "directors can actually be the builder themselves". To be an implementation of the Builder Pattern, this algorithm has to be implemented in a separate 'Director' class.

22.6 Implementation Issues

22.6.1 Creating a product

Each concrete builder has the responsibility to define its own process to create a product in terms of the methods defined in the abstract builder. Each time a product is created, it has to be created from scratch. This can be done either by creating a default product in the first method that is executed by the director or by always having a default product handy.

The option to create a new default product in the first method that is executed by the director is less versatile since it it prescriptive in what method the director should always execute first. However, it is more robust because it is easier to ensure that the copy of the product under construction was not altered in a previous use of the concrete constructor, especially if the same instance of a concrete constructor is re-used by different directors.

The option to have a default product handy can be achieved by instantiating a default product on construction of the concrete builder. This is a viable option if the concrete builder is destroyed after creating an instance of a product and recreated each time it is needed. In situations where the same instance of a concrete builder is re-used, the option to create a product in a method call issued by the director is a better option.

22.6.2 Model for constructing a product

Builders construct their products in step-by-step fashion. Therefore the Builder class interface must be general enough to allow the construction of products for all kinds of concrete builders. A key design issue concerns the model for the construction and assembly process. A model where the results of construction requests are simply appended to the product is usually sufficient. But sometimes you might need access to parts of the product constructed earlier. In that case, more methods to enable communication between the builder and the director is needed to enable the director to retrieve parts, modify them and pass them back to the builder.

22.6.3 Extending a product

Each concrete builder creates a unique product. A concrete builder is allowed to define and add parts to a product that is not controlled by the director. Concrete builders usually

define and maintain instance variables that can eliminate the need for the director to pass many values by means of parameters to the methods that assemble the product.

22.6.4 Varying construction

Since the code for construction and code for representation is separated from one another the design allows exchanging the construction process. Thus, different directors can use the same concrete builders in different ways to build product variants from the same set of parts.

22.7 Related Patterns

Composite
: Builder usually construct composite objects.

Abstract Factory
: Abstract Factory is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract Factory's emphasis is on families of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately.

22.8 Example

In a real application of the builder pattern a director may depend on data that specify the detail of the aggregate. The process of creating a new product involve interpreting the data and issuing commands related to this interpretation to a concrete builder. Different concrete builders are able to create different variations of a product through different implementations of these commands issued by the director. It is important to note that in many cases concrete builders implement only the operations they need and omit the others.

Since the interpretation of data that specifies the aggregate is not part of the pattern, our example assumes a small hard coded aggregate (a soft toy) that has exactly five parts (name, body, stuffing, heart and voice).

Figure 4 is a class diagram of our example implementation. It is a nonsense program that implements the builder structure to illustrate how different directors can use the same concrete builders to create variations of the products that are produced. The different products deliberately have different internal structures and different interfaces to illustrate how this pattern allows for the creation of divers products by the same director when using a different concrete builder.

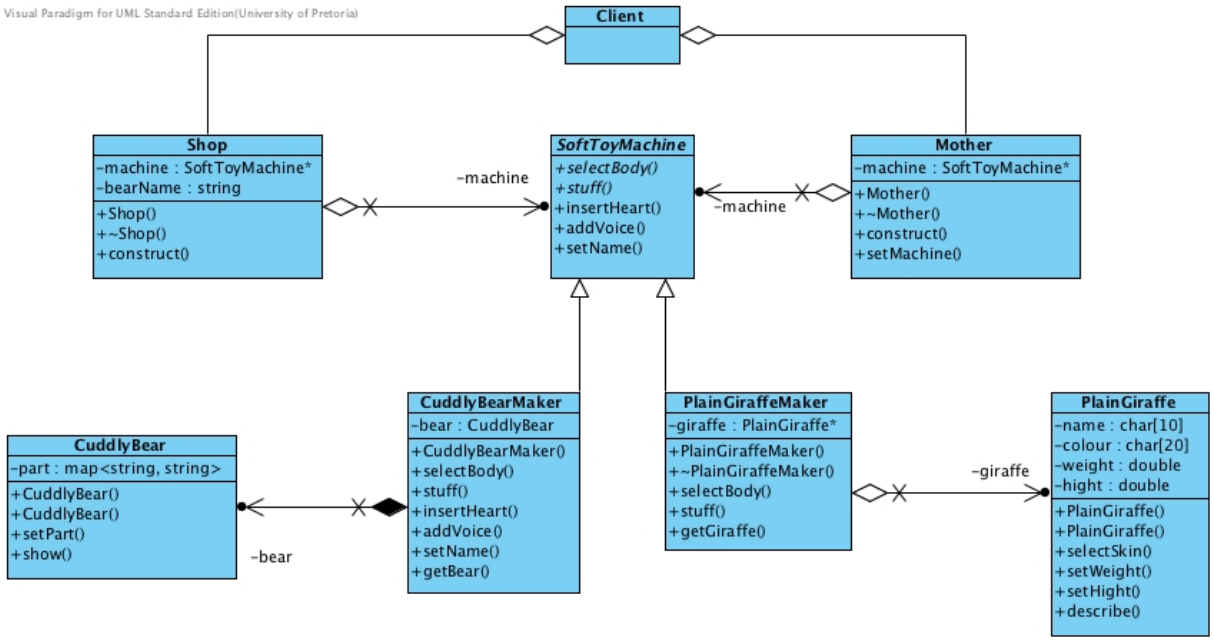


Figure 4: Class Diagram of a soft toy builder nonsense program

Participant	**Entity in application**
Directors	Mother, Shop
Builder	SoftToyMachine
Concrete Builders	PlainGiraffeMaker, CuddlyBearMaker
Products	PlainGiraffe, CuddlyBear
construct()	construct()
buildPart()	selectBody(), stuff(), insertHeart(), addVoice(), setName()
getResult()	getGirrafe(), getBear()

Directors

- The `Mother` class and the `Shop` class are different directors. Both are implemented to use instances of the same concrete builders to create products.
- The `Mother` class has a method that allows the client to equip a `Mother` object with another concrete builder on the fly, while the `Shop` class is instantiated with its concrete builder on construction. The only way to equip a `Shop` object with another concrete builder is by recreating it.
- The `construct()` methods of these two classes are different. The `Shop` class calls all the methods in the interface and make use of parameters to specify high quality products while the `Mother` class omits some of the methods and calls other with default values.

Builder

- The `SoftToyMachine` class act as the builder. It defines the union of all operations needed by the different concrete builders. Those that are not necessarily required are provided with empty implementations.

- This interface specifies the methods that has to be implemented by the concrete builders. All its methods are virtual to allow concrete builders to override them.
- `selectBody()` and `stuff()` are pure virtual. Each concrete builder is required to implement these.
- `insertHeart()`, `addVoice()` and `setName()` has default empty implementations. Usually most of the methods specified in a builder should be specified as such, to allow a concrete builder to omit them if they are not required in the products created by the concrete builder.

Concrete Builders

- The classes `CuddlyBearMaker` and `PlainGiraffeMaker` act as concrete builders.
- Each of these classes provides its own implementation of the building process. They implement the common interface that is defined in `SoftToyMachine` to adapt the methods in the concrete products to the methods defined in `SoftToyMachine`.
- `CuddlyBearMaker` instantiates its product on construction. It is therefore not reusable. For this reason different instances of this class is used by the different directors in this example.
- `PlainGiraffeMaker` instantiates its product in the `selectBody()` method. It is therefore required that each director should call this method first in its `construct()` method. Notice that it deletes any previous instance of the product (if it exists) before creating a new one. This is done to avoid a memory leak. Also notice how its `getGiraffe()` method returns a copy of this product, rather than the product itself. This is done because the product that is created will be destroyed when this concrete builder is reused. The copy is owned by the client and is destroyed by this concrete builder.

Products

- The products are PlainGirrafe and CuddlyBear. You will notice that these products do not share a common abstract interface. This is a distinct feature of the situation where the builder design pattern is deemed appropriate.

Client

- The client constructs instances of directors, concrete builders and products. It then illustrates how the different directors uses the same concrete builders to create different variants of the products. The output is the detail of the products that was created by the different directors.

22.9 Exercise

1. Draw a class diagram showing the participants of the builder pattern to implement dynamic context sensitive creation of menus in a word processing program.

References

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

[2] Christopher G. Lasater. *Design Patterns*. Wordware Publishing Inc., Texas, USA, 2007.

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 23: Interpreter Design Pattern

Contents

23.1 Introduction

The interpreter design pattern is a behavioral pattern relying on its inheritance structure to achieve its purpose. It represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes. When the interpreter is executed on a composite node in the hierarchy it propagates to all the descendants of such node.

In this lecture we assume a basic knowledge of grammars. We illustrate by example how the pattern is applied to translate given grammars into implementations.

23.2 Interpreter Design Pattern

23.2.1 Identification

Name	**Classification**	**Strategy**
Interpreter	Behavioural	Inheritance
Intent		
Given a language, define a represention for its grammar along with aninterpreter that uses the representation to interpret sentences in thelanguage. ([2]:243)		

23.2.2 Structure

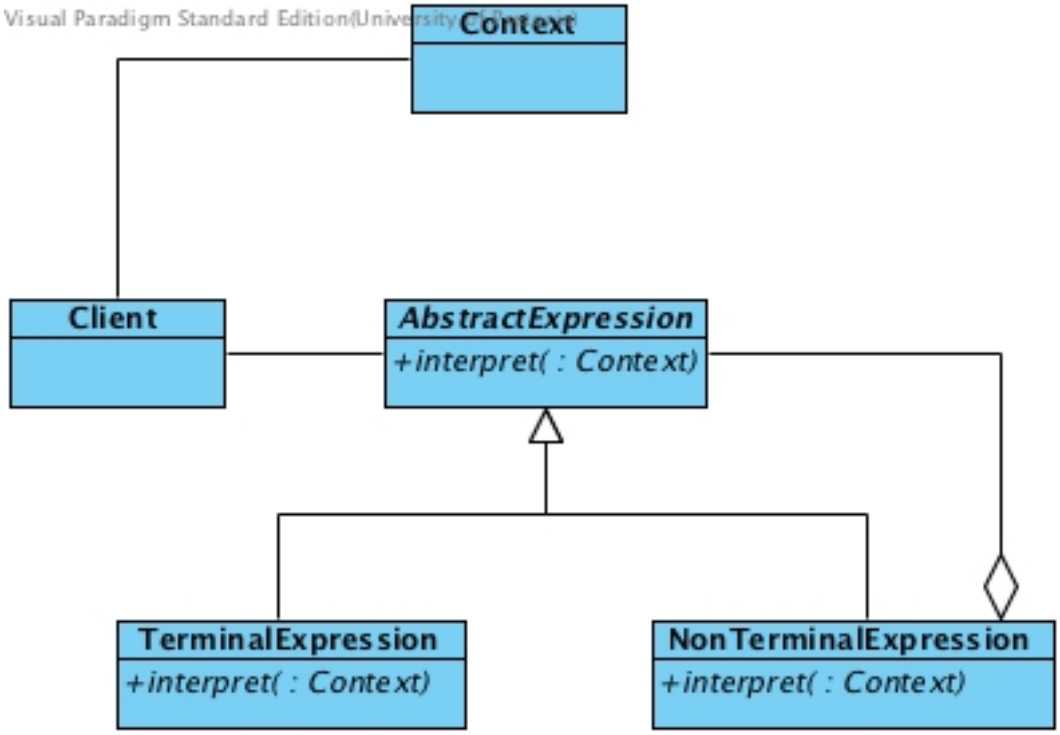


Figure 1: The structure of the Interpreter Design Pattern

23.2.3 Participants

AbstractExpression

- declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.

TerminalExpression

- implements an Interpret operation associated with terminal symbols in the grammar.
- an instance is required for every terminal symbol in a sentence.

NonterminalExpression

- one such class is required for every rule R ::= R1 R2 ... Rn in the grammar.
- maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
- implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.

Context

- contains information that's global to the interpreter.

Client

- builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes.
- invokes the Interpret operation.

23.2.4 Problem

A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a language that can be expressed in terms of a formal grammar, then problems could be easily solved with an interpretation engine representing the grammar. [3].

23.3 Interpreter pattern explained

23.3.1 Improvements achieved

Improved adaptability

It's easy to change and extend the grammar. Because the pattern uses classes to represent grammar rules, you can use inheritance to change or extend the grammar. Existing expressions can be modified incrementally, and new expressions can be defined as variations on old ones.

Implementation can be automated

Once a grammar is defined, the class design and its implementation is completely determined by the rules of the grammar. The generation of code for classes defining nodes in the abstract syntax tree can often be automated with a compiler or parser generator using the grammar as input.

23.3.2 Situations where other solutions may be more suitable

The pattern is not suitable for complex grammars. If the grammar is complex the class hierarchy for the grammar becomes large and unmanageable. Tools such as parser generators are a better alternative in such cases. They can interpret expressions without building abstract syntax trees, which can save space and possibly time.

If you need multiple *interpreter* operations, then it might be better to use the Visitor pattern. Here every *interpret* operation can be implemented in a separate *visitor* object. For example, a grammar for a programming language will have many operations on abstract syntax trees, such as type-checking, optimization, code generation, and so on. It will be more likely to use a visitor to avoid defining these operations on every grammar class.

23.3.3 Common Misconception

The pattern require the existence of an abstract syntax tree of the expression that has to be interpreted. It is often assumed that the compilation of the abstract syntax tree of the sentence that needs to be interpreted is part of the pattern and that the pattern can only be implemented if a parser of text input is involved. However, the Interpreter pattern doesn't explain how to create an abstract syntax tree.

The pattern does not address parsing. The abstract syntax tree can be created by a table-driven parser, by a hand-crafted (usually recursive descent) parser, or directly by the client. Thus the abstract syntax tree is generated from input provided through a GUI or a series of prompts. For example the syntax tree of a mathematical expression may as well be compiled using the following code assuming that the methods that are called in the switch statement will also prompt the user to enter appropriate information.

```
Expression* inputExpression(string prompt)
{
        Expression* exp;
        cout << "Specify the type of the " << prompt << endl;
        cout << "1. A value" << endl;
        cout << "2. A variable" << endl;
        cout << "3. An operation" << endl;
        cout << "Enter your choice: ";
        int choice;
        cin >> choice;
        switch (choice)
        {
                case 1: exp = inputValue(); break;
                case 2: exp = inputVariable(); break;
                case 3: exp = inputOperation(); break;
```

```
        }
        return exp;
}
```

In this case the `inputOperation()` method will typically prompt the user for an operation and then recursively call this method for the left- and right operands of the operation.

23.3.4 Related Patterns

Template Method
: The template method and the interpreter design patterns are the only two Gamma:1995 behavioural patterns that uses inheritance as its basic strategy to achieve its purpose. All other Gamma:1995 behavioural patterns uses delegation.

Composite
: Interpreter is an application of the Composite pattern. It ads specific behaviour to the composite structure namely to interpret the elements of the composite structure. Note that is more specific than just an operation distributed over a class hierarchy that uses the Composite pattern. It is specifically aimed at dealing with problems that are specified in terms of grammars.

Flyweight
: Both Flyweight and Interpreter share symbols. Interpreter share terminal symbols within the abstract syntax tree.

Visitor
: Both Interpreter and Visitor adds behaviour to a composite structure. Where interpreter adds a simplistic behaviour, visitor allows for more generic and adaptable behaviour. The similarity between these patterns is deep. The operations of an interpreter can always be refactored into *interpreter* visitors.

State
: The interpreter design pattern and the state design pattern are different ways to solve interpretation. Where the interpreter pattern provide a way to interpret parse trees directly while it is possible to first transform a parse three into a state machine and then applying the state pattern to solve the same problem.

23.4 How to implement the interpreter pattern

The implementation of the interpreter pattern is useful to achieve one of the following:

- to verify if a given sentence complies with the grammar
- to generate a sentence that complies with a grammar
- to determine a value of a sentence that complies with a grammar

In this section we explain the process of implementing this pattern at the hand of an example that verifies if a given sentence complies with a language. In Section 23.5 we present an example that applies this pattern to generate a sentence that complies with a grammar. Lastly, in Section 23.6, we present a tutorial that guides the reader through the process to implement the pattern to calculate the value of a mathematical expression.

23.4.1 Define a grammar

The interpreter pattern is only applicable if the solution to the problem at hand can be expressed in terms of a formal grammar. If the problem offers a justifiable return on investment one can define a grammar and then build an interpretation engine using this pattern to process the solutions.

23.4.2 Use the grammar to design the system

The Interpreter pattern uses a class to represent each grammar rule. Symbols on the right-hand side of the rule are instance variables of these classes. Thus the grammar rule `alternation ::= expression '|' expression`, should be represented in the design with the classes shown in Figure 2.

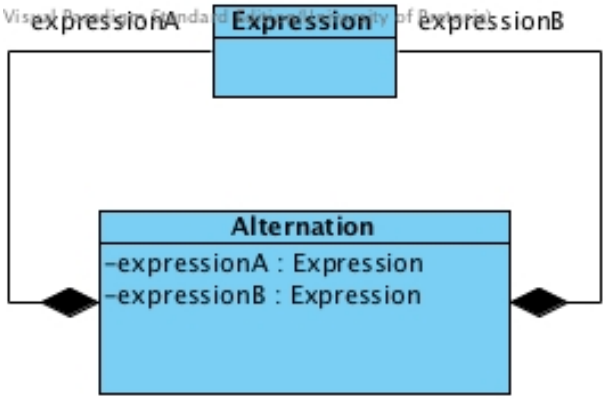


Figure 2: Classes representing `alternation ::= expression '|' expression`

23.4.3 Implement the design using the grammar

The constructor of each class should instantiate its instance variables. This is typically done by implementing only one constructor that requires parameters to specify the values for all its instance variables. The following is an example of the constructor of this `Alternation` class assuming the `Expression` class implements an assignment operator.

```
Alternation::Alternation(Expression a, Expression b)
{
    expressionA = a;
    expressionB = b;
}
```

The `interpret()` method of each class should be implemented. For example the `interpret()` method of an application that verifies if a sentence complies will return a boolean value. It will return true if the input matches, and will return false if it does not. Assume the the `Alternation` class given above must check if the input matches any of its alternatives. Below is the implementation of this method. Note how the interpretation is delegated to its instance variables.

```
bool Alternation :: interpret (string input){
    bool a = expressionA.interpret(input);
    bool b = expressionB.interpret(input);
    return a || b;
}
```

23.5 Example

The interpreter pattern is the translation of a grammar to an implementation. Usually grammars are associated with parsing and interpretation of parsed text. Most examples illustrating the interpreter pattern includes a parser and illustrates the pattern at the hand of grammars that identify expressions that are parsed before they are interpreted. It is, however, clearly stated in [2] that the pattern does not include parsing.

23.5.1 The problem

For our example we chose an example given by Huston [3]. It does not include the parsing of expressions. We deem it a suitable example to illustrate the design of a grammar and the translation of such grammar to an implementation. It is an expression of the well known Towers of Hanoi puzzle in terms of a grammar that is used to implement a solution of the problem.

The Towers of Hanoi puzzle consists of three pegs and a number disks with different sizes. The goal is to reposition the stack of disks from one peg to another peg by moving one disk at a time, and, never placing a larger disk on top of a smaller disk. An interactive example to solve the puzzle is available at [1]. This is a classic problem for teaching recursion in data structures courses.

Here a language is designed that characterizes this problem domain. The language is then mapped to a grammar, and the grammar is implemented with an interpretation engine that apples the interpreter patten. This application can now be applied to solve the puzzle. The implementation writes the solution in terms of the moves required to achieve the required end state.

23.5.2 A language

The language required to describe a solution can be defined in terms of moves. A simple move is described in terms of the source peg and the destination peg. Executing a move described as `A B`, means the top disk on peg A is moved to peg B. It will be valid if peg B is empty or the top disk on peg B is larger than the top disk on peg A.

A complex move is defined as the combination of moves needed to move a stack of n disks from one peg to another. Executing this move can be described as `n A B`. Such move is valid if the bottom disk of top n disks on peg A is smaller than the top disk on peg B.

Every solution to move n disks from one peg to a specified peg can be now described in terms of a complex move, followed by a simple move, followed by a complex move. For example the complex move `4 A C` can be refined as `3 A B` followed by `A C` followed by `3 B C`. The logic is moving the top 3 disks out of the way, move the bottom of the four disks to its required destination, and thereafter move the top three disks to the required destination.

23.5.3 A grammar

In the above description of the solution to the general problem we identified types of moves and a production rule. Assuming the pegs are labelled A, B and C, this can be formulated in the following grammar to define the language to describe the solution:

```
1  Move ::= complexMove | simpleMove
2  simpleMove ::= peg1 peg2
3  complexMove ::= size peg1 peg2
4  size peg1 peg2 ::= size2 peg1 peg3, peg1 peg2, size2 peg3 peg2
5  1 peg1 peg2 ::= peg1 peg2
6  peg1 ::= A | B | C
7  peg2 ::= (A | B | C) and !(peg1)
8  peg3 ::= (A | B | C) and !(peg1) and !(peg2)
9  size ::= (an integer)
10 size2 ::= (value of (size - 1))
```

Rules 1, 2 and 3 are general rules while rules 6 to 10 are specific rules stating the allowable values and relationships between terminal symbols. Rules 4 and 5 are production rules specifying how a complex move can be expressed in terms of other moves. It includes detail about the values and relationships between the terminal symbols comprising the expression. The following expressions are a generalisations of these rules:

```
4  complexMove ::= complexMove, simpleMove, complexMove
5  complexMove ::= simpleMove
```

23.5.4 Mapping the grammar to a design

When mapping the grammar to a design. One look at the general rules and their relation to one another. You will notice that the classes in the class diagram in Figure 3 correspond with the named items (Move, simpleMove and complexMove) in the grammar defined in Section 23.5.3. The other items like the peg names and the values in the variable called `size` in the given grammar feature as instance variables of these classes.

The design represents the abstract rules defined in the given grammar. These are the abstract versions of rules 1 to 5. The application of other rules specifying the detail about peg names and values and their relation to one another will be observed in the implementation.

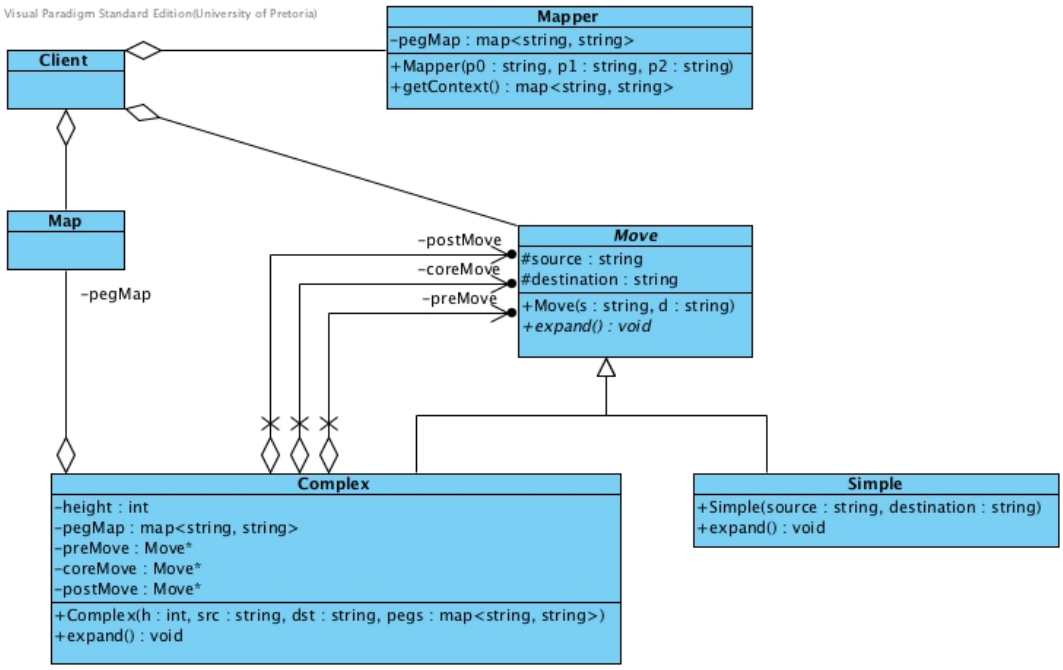


Figure 3: Class design of the grammar in section 23.5.3

The following table identifies of these classes as participants of the interpreter design pattern:

Participant	**Entity in application**
Abstract Expression	Move
Terminal Expression	Simple
Non-Terminal Expression	Complex
Context	map
Client	Client
interpret()	expand()

23.5.5 Implementing the design

Client

- This client prompts the user to specify names for the source peg, the destination peg and an auxiliary peg. It also prompts the user for the number of disks that is to be moved from the specified source peg to the required destination peg.
- The input data is passed to the `Mapper` class to compile the context (see later).
- The client is now able to produce the solution to the puzzle for the given context in a single call to the `expand()` method of the complex move to move the entire stack of disks from the source disk to the destination disk.

Mapper

- The `Mapper` class seems as if it should act as the context. However, in this implementation it is merely a helper class to construct the context.

map

- In this implementation the map serves as context. It is a lookup table that specifies the name of the spare peg when the source and destination pegs are known.
- We deviate from the design of the interpreter pattern in how we use the context. The design of the interpreter pattern suggests that the context be passed as parameter to the `interpret()` method. However, in this implementation this context is also needed by the constructor of the `Complex` class. It is also not used by the `Simple` class. It was therefore decided to pass it to the constructor of the `Complex` class and keep it as an instance variable of a `Complex` object.

Move

- This is the interface to all possible moves.
- It defines the virtual method `expand()` that interprets the specific move.
- Since all moves in the puzzle specifies a source peg and a destination peg. It was decided to locate these as instance variables of a move.

Simple

- This class represents a terminal expression. This is because it provides a concrete move of one disk from one peg to another.
- It implements the virtual method `expand()` that interprets the specific move. This method contains a `cout` statement describing the move. In other more typical implementations if the interpreter pattern the `interpret()` method will typically return single result.

Complex

- This is the heart of the interpreter pattern. It is specified that a class should be implemented for each of the production rules in the grammar. In this example rule 4 of the grammar defined in Section 23.5.3, is the only production rule. Therefore, we implement only this one Non-Terminal expression.

- The constructor applies the associated production rule to create its sub-ordinates as instance variables of the class.
- The abstract version of the language rule is reflected in the definition of the instance variables of this class.
- The detailed version of this rule and its consequences are applied to construct these instance variables when an object of this class is instantiated.
- As prescribed in the interpreter pattern, the implementation of the `expand()` method simply calls the `expand()` methods of its instance variables.

The C++ code of this example can be found in the tarball called `L30_Interpreter.tar.gz`.

23.6 Tutorial

We present an example as a tutorial. We only provide partial solutions for the the development steps and leave the completion of these steps to the reader.

23.6.1 The problem

We need to write an application that can evaluate mathematical expressions written in postfix notation. Only binary operations +, - and * with their usual interpretation are supported. The operands for the operations may only be integers or variable names.

23.6.2 A grammar

The language for mathematical expressions is well established. In this context we will allow variable names to be strings of any length consisting of only alphabetical characters. The following rules are part of our grammar:

```
expression ::= sum | difference | product | variable | number
sum ::= expression expression '+'
number ::= int
```

The last rule simply specifies the data type of this terminal symbol. The reader is invited to write rules for the compound expressions `difference` and `product` and also provide a rule for the terminal of type `variable`.

The eager reader can also add rules for the integer operations `quotient` with operator / and `mod` with operator %.

23.6.3 Mapping the grammar to a design

When mapping the grammar to a design, each grammar rule is represented by a class.

There should be a class for each terminal symbol. In this example they are `variable` and `number`. These classes are directly derived from the abstract class `Expression` because

they appear in the right hand side of the first rule indicating that they are kinds of expressions. Note in Figure 4 that the `Number` class representing `number` contains an instance variable of the type as specified by the grammar rule. It also implements a constructor and the `interpret()` method. The reader is invited to add a class representing `variable` to the class diagram in Figure 4.

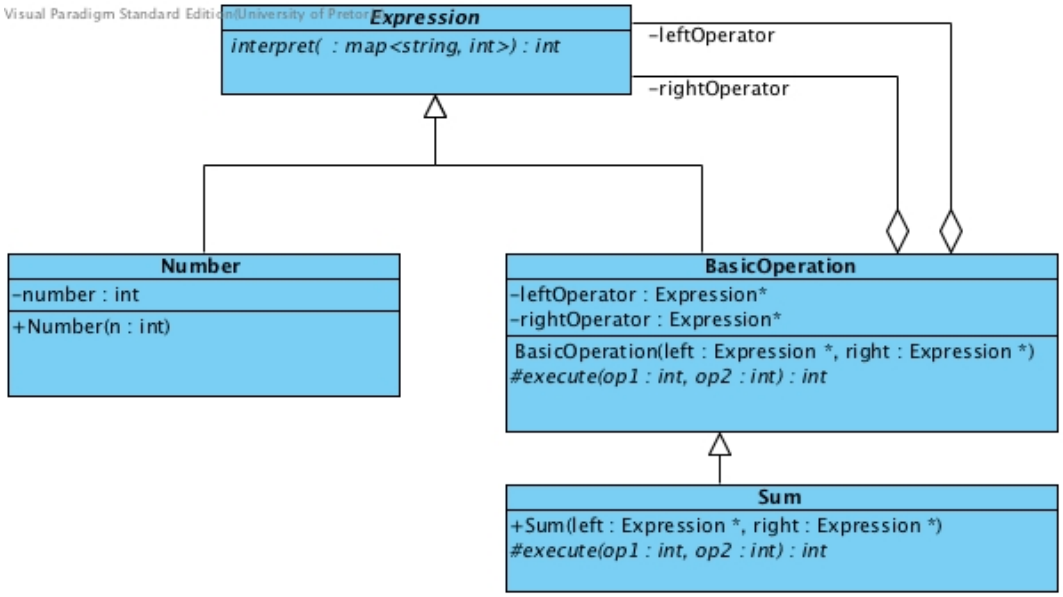


Figure 4: Class design of the grammar in section 23.6.2

There should also be a a class for each production rule. The class `Expression` represents the first rule. Classes that should derive from it are the classes representing `sum`, `difference`, `product`, textttvariable and `number`. In our example we have observed that all the classes representing operations will be similar to the class representing `sum`. Therefore, we have defined an abstract interface for them. They will all have two operands that may be any kind of expression. We have defined these variables in this abstract interface. This abstract interface must, similar to the classes representing terminals, implement a constructor and the `interpret()` method. In this case we have implemented it as a template method that will delegate the core of the operation to its derived class using the virtual `execute()` method.

The reader is invited to add classes for the rules for the compound expressions that he/she added to the grammar.

23.6.4 Implementing the design

When implementing the design one need to have an abstract syntax tree of the expression that needs to be interpreted, compile the context needed for interpretation and inplement the contsructors as well as the interpret method in each of the classes in the design.

Syntax Tree

The compilation of the syntax tree is usually done by implementing a parser. The implementation of a parser or other means of compiling the syntax tree is left as an exercise.

Hint: see Section 23.3.3.

Context

In this example the context entails the values of the variables. It is advised to define the context as a `map<string, int>`. A `pair<string, int>` can be inserted to this map for each variable that is encountered in the expression. This will provide a lookup table that can be used in the implementation the interpret method of the `Variable` class. Since the creation of this context is closely related to the process to build the abstract syntax tree this part of the implementation is left to the reader.

Constructors

Each constructor needs to instantiate its instance variables as specified by the class design. Here we list the constructors of all the classes in Figure 4 that has instance variables. The implementation of the constructors of the rest of the classes are left for the reader to complete.

```
Number::Number(int value): number(value)
{}

BasicOperation::BasicOperation(Expression* left, Expression* right)
{
        leftOperator = left;
        rightOperator = right;
}

Sum::Sum(Expression* left, Expression* right)
     : BasicOperation(left, right)
{}
```

Interpret method

Each class has to implement the `interpret()` method. It is assumed that the context is passed as parameter to the interpret method. Typically the interpret method of terminal symbols simply return its value while composite nodes will recursively call the interpret method of its instance variables. Here we list the implementation of this method of the `Number` and `BasicOpertion` classes in Figure 4 as well as the `execute()` method of the `Sum` class to illustrate how this can be achieved. The implementation of the interpret/execute methods of the rest of the classes are left for the reader to complete.

```
int Number::interpret( map<string, int>)
{
     return number;
}

int BasicOperation::interpret(map<string, int> variables)
{
```

```
    return execute(leftOperator->interpret(variables),
        rightOperator->interpret(variables));
}

int Sum::execute(int left, int right)
{
    return left + right;
}
```

References

[1] Anonymous. Tower of hanoi. `http://www.mathsisfun.com/games/towerofhanoi.html` edited by Rod Pierce. [Online: Accessed 3 October 2011].

[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].

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 24: Bridge Design Pattern

Contents

24.1 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.

24.2 Bridge Design Pattern

24.2.1 Identification

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

24.2.2 Structure

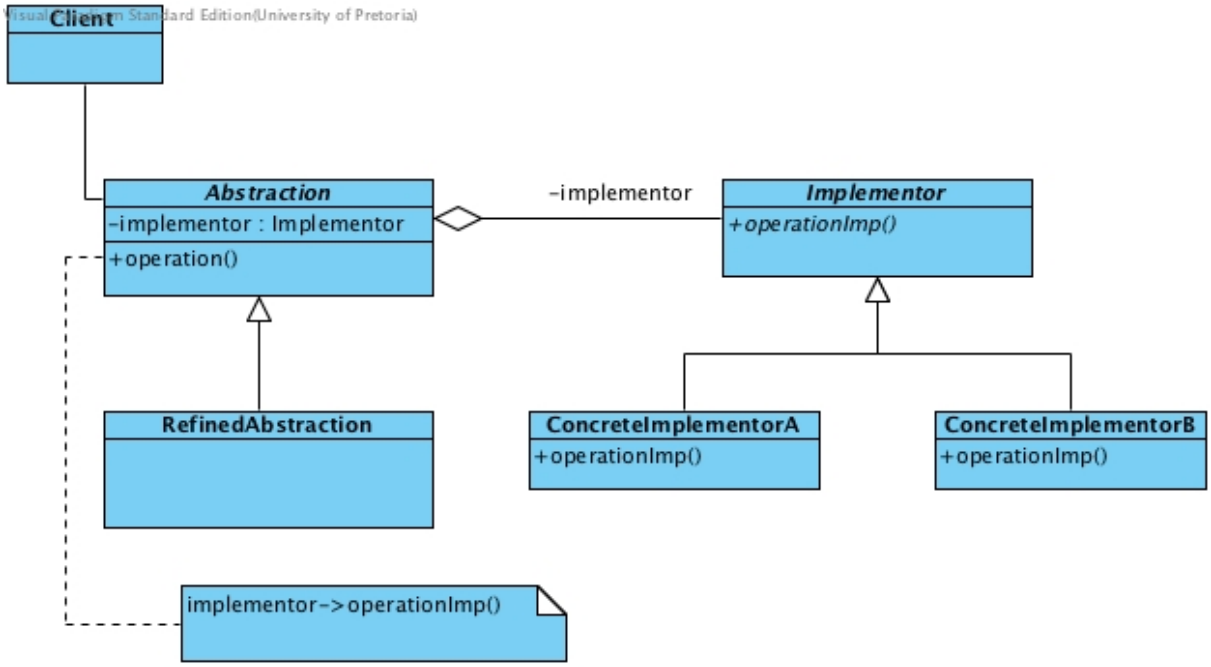


Figure 1: The structure of the Bridge Design Pattern

24.2.3 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.

24.2.4 Problem

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

24.3 Bridge pattern explained

24.3.1 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

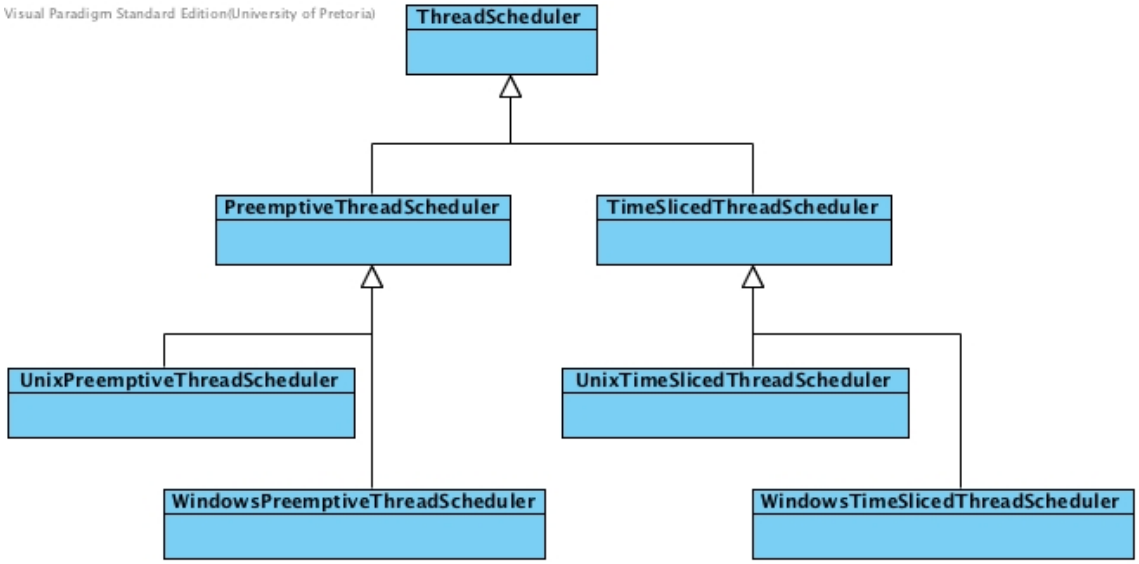


Figure 2: Proliferation of classes to accommodate different schedulers for different platforms

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.

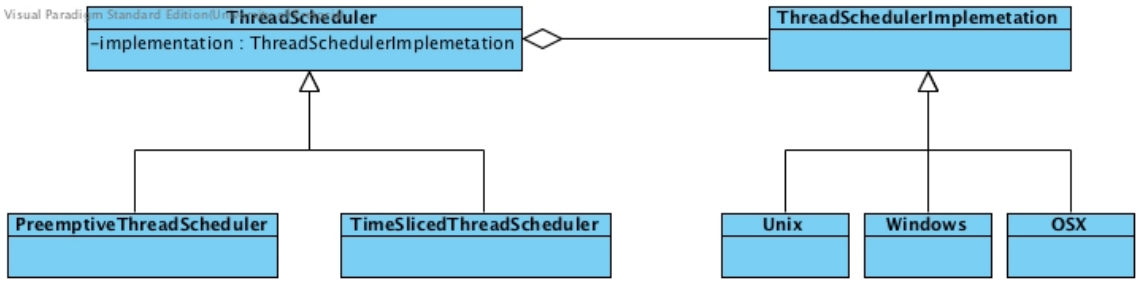


Figure 3: A bridge to accommodate different schedulers for different platforms

24.3.2 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]

24.3.3 Improvements achieved

Greater flexibility

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

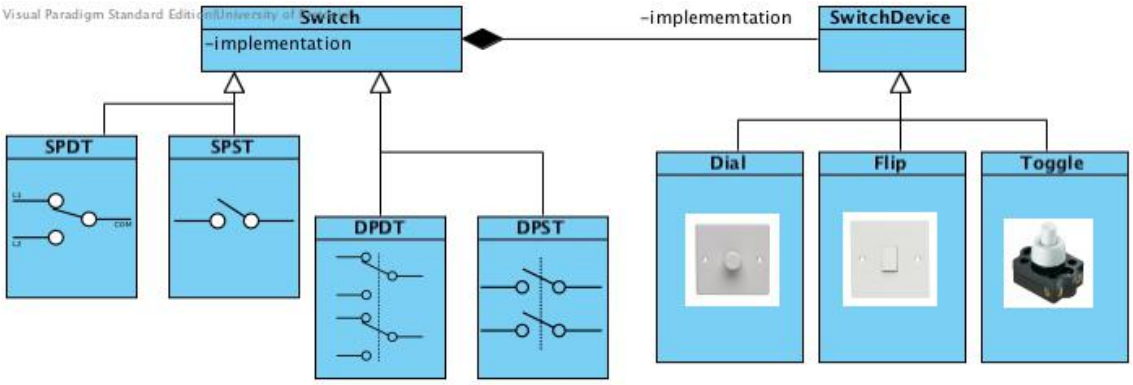


Figure 4: Household switch abstraction and implementation hierarchies

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.

24.3.4 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.

24.3.5 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.

24.4 Example

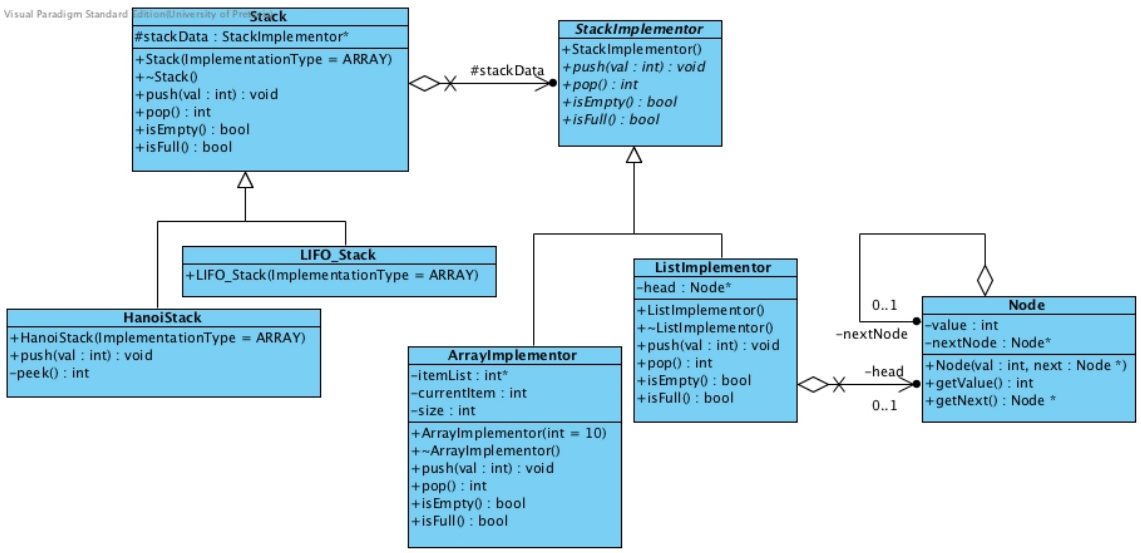


Figure 5: Different kinds of stacks implemented with different data structures

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.

24.5 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, 1995.

[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].

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

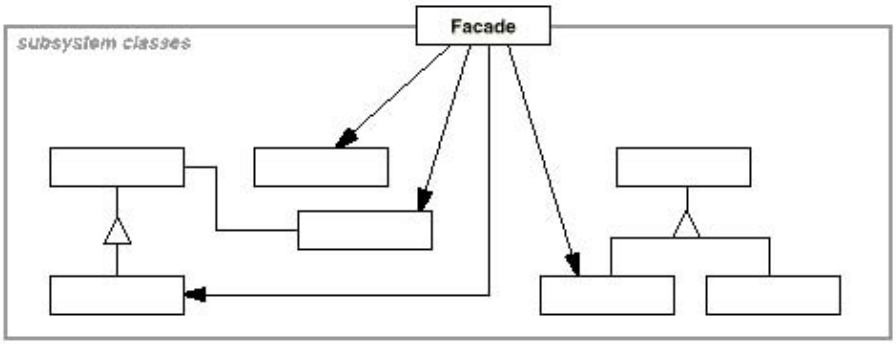


Figure 1: The structure of the Façade Design Pattern

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

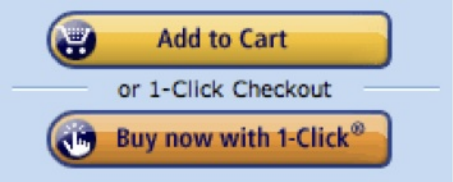
The Façade is applied to automate a process that consist for a sequence of steps where the different steps in the process may be executed by different subsystems in different classes.

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

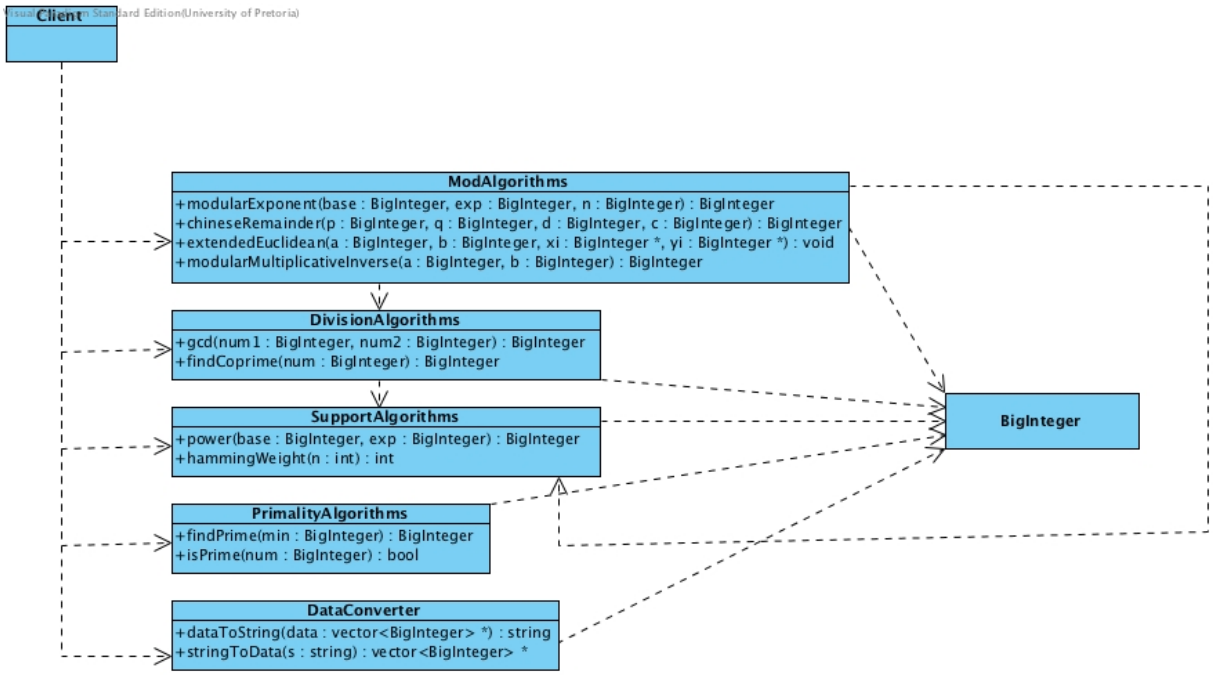


Figure 2: Class Diagram a system before implementing a façade

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

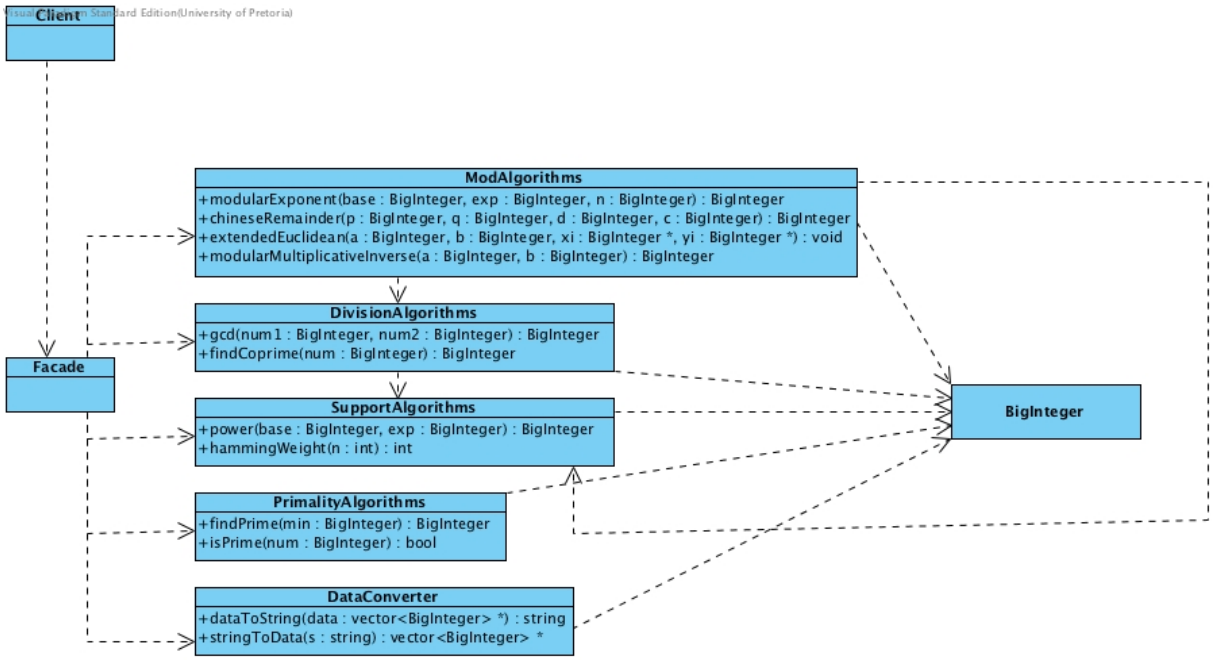


Figure 3: Class Diagram a system after implementing a façade

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].

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 26: Visitor Design Pattern

Contents

26.1 Introduction

In this lecture you will learn about the Visitor Design Pattern. This pattern separates the behaviour of the elements in an aggregate from the state of these elements. This is done with the intention to simplify the maintenance when the behaviour of these elements have to be changed or extended. This is done with extreme elegance. When implemented correctly neither the elements of the aggregate nor the clients using the aggregate need to be recompiled when new functions are added to the system. Unfortunately the application of this pattern complicates the maintenance of the aggregate itself. It is difficult to add classes to the aggregate. Thus, this pattern is more applicable in a system with changing processing needs and a stable internal structure of elements. I.e. a system where you will seldom add new classes but have the need to often add new functions to some derived classed in an aggregate and consequently new virtual functions to existing interfaces to the aggregates.

26.2 Programming Preliminaries

26.2.1 Single dispatch

With the exception of the Visitor design pattern, all the design patterns that uses delegation as their strategy, the concrete function that is called from a function call in the code depends on the dynamic type of a single object and therefore they are known as single dispatch calls, or simply virtual function calls.

Single dispatch is a natural result of function overloading. Function overloading allows the function called to depend on the type of the argument. Function overloading however is done at compile time where the compiler creates code for each overloaded version of the function. Consequently there is no runtime overhead because there is no name collision. Calling an overloaded function goes through at most one virtual table just like any other function.

26.2.2 Double disapatch

In the Visitor design pattern the mechanism that dispatches a function call to different concrete functions depends on the runtime types of two objects that are involved in the call. This mechanism is known as double dispatch.

When double dispatch is applied two overloaded functions are involved in the process to execute the correct concrete function. Assume there are two class hierarchies respectively with abstract classes called `HierarchyA` and `HierarchyB`. Further assume that `functionA(:HierarcyB)` and `functionB(:HierarchyA)` are virtual functions respectively defined in `HierarchyA` and `HierarchyB`. A double dispatch call to a concrete class derived from `HierarchyB` involves calling `functionB(:HierarchyA)` and passing a pointer to a concrete class derivied from `HierarchyA` via the parameter. This will result in the execution of the correct concrete implementation of `functionB(:HierchyA)`. The next step in the double dispatch process will now use the parameter that was passed to `functionB(:HiercahyA)` to call back. i.e. if `objectA : HierrachyA` is the argument

that was passed to `functionB(:HierchyA)`, the body of `functionB(:HierchyA)` will include a a statement like `objectA->functionA(objectB)`. This call will in turn use the type of its argument to determine the correct concrete implementation of `functionA(:HierarchyB)` to be executed.

26.3 Visitor Design Pattern

26.3.1 Identification

Name	Classification	Strategy
Visitor	Behavioural	Delegation
Intent		
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. ([2]:331)		

26.3.2 Structure

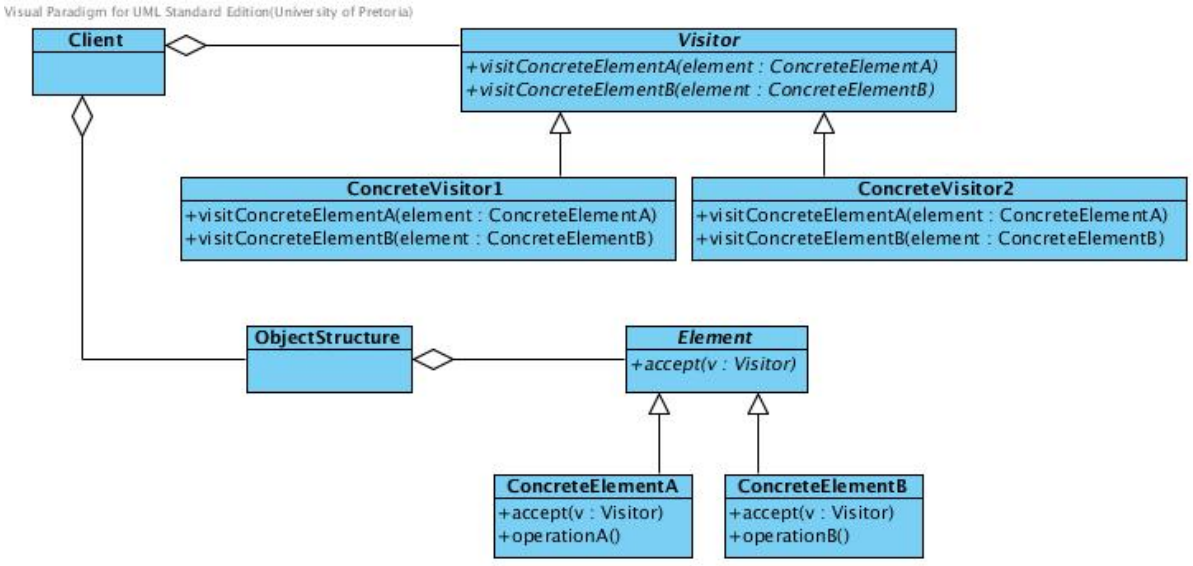


Figure 1: The structure of the Visitor Design Pattern

26.3.3 Problem

Many distinct and unrelated operations need to be performed on node objects in a aggregate structure that may be heterogeneous. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation [3].

26.3.4 Participants

Visitor

- Each class of ConcreteElement has a `visit()` operation declared for it.
- The operation's signature identifies the class that sends the `visit()` request to the visitor.
- The particular class is then accessed through the interface defined for it.

ConcreteVisitor

- Implements the operations defined by visitor.
- May store information about objects that are visited.

Element

- Defines an `accept()` operation that takes an object of Visitor as a parameter.

ConcreteElement

- Implements the `accept()` operation that takes an object of Visitor as a parameter.

ObjectStructure

- Has a highlevel interface that allows the Visitor access and traverse its elements.
- This structure may be a Composite or a collection such as an array, list or a set.

26.4 Visitor Pattern Explained

26.4.1 Improvements achieved

- The operations of a conceptual operation is kept together rather than being scattered in different classes in an aggregate. Thus cohesion is increased because related operations are logically grouped in different visitors.
- The different players are independent. This independence reduces coupling [1]
- Aside from potentially improving separation of concerns, the visitor pattern has an additional advantage over simply calling a polymorphic method: a visitor object can have state. This is extremely useful in many cases where the action performed on the object depends on previous such actions [5].

26.4.2 Disadvantages

- It is difficult to change the aggregate. When classes in the aggregate are changed *all* visitors need to be changed. This is the case even when the changes are part of the aggregate which is of no particular interest to the visitor [4].
- The encapsulation of the concrete elements is diminished to allow the visitors to perform their operations [2].
- Owing to double dispatch (i.e. twofold redirection), the application of the Visitor pattern may introduce a significant performance penalty. One might say that this is the price of complete flexibility (which may or may not be worth paying).

26.4.3 Implementation Issues

The Object structure is dependant on the Visitor to compile as it has visitors as parameters to its `accept()` methods. The Visitor in turn is also dependant on the Concrete Elements in the Object structure to compile for the same reason. This is called a *cyclic dependancy*. Fortunately they depend only on the names of the classes. Therefore, the problem can be avoided by including a forward declaration of the Visitor class in the Object structure and first compiling the Object structure or vice versa.

Note that it is required that the Visitor participant defines a separate `visit()` function for each of the concrete element types in the aggregate. This function also takes a parameter of the type of this concrete element. Owing to C++ overloading, these functions may all have the same name and can be distiguished by their differing parameter types. It is, however, important to note that the `accept()` methods in the elements still need to be implemented in the concrete classes despite the fact that they seem to be identical. If you would implement it in the interface, the static type `*this` would be the base class which would not provide the compiler with the needed type information.

26.4.4 Related Patterns

Composite
: The Composite is supportive to the Visitor. Visitors can apply an operation over an object structure defined by the Composite pattern.

Iterator
: Iterator and Visitor has similar intents. Visitor, however, is more general than Iterator. 'n Iterator is restricted to operations on elements of the same kind while Visitor can operate on elements of different types.

Interpreter
: The Visitor pattern may be applied to do the interpretation.

Abstract Factory
: Abstract Factory and Visitor has similar structure. Abstract factory applies the structure to create families of objects while Vistor applies this structure to perform a group of related operations.

Bridge

Both Bridge and Visitor separates state and behaviour of objects. Bridge applies single dispatch while Vistor applies double dispatch.

26.5 Example

Figure 2: Class Diagram of a system illustrating the implementation of the Visitor design pattern

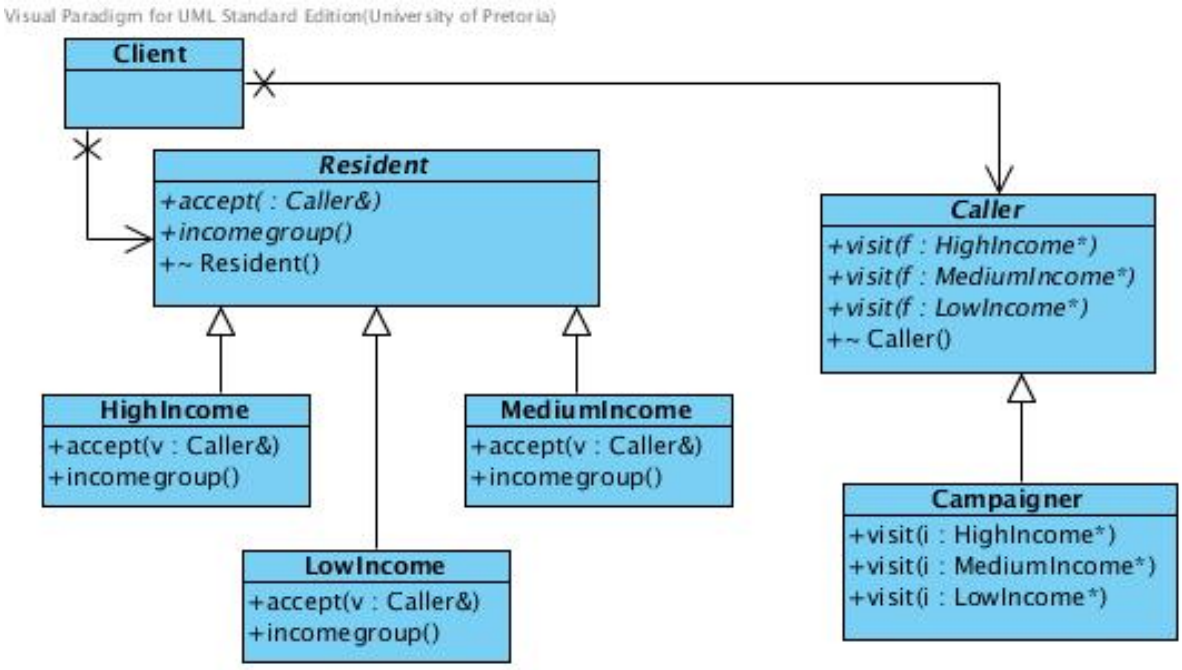


Figure 2 is a class diagram of an application that implements the visitor design pattern. It illustrates an implementation of the Visitor design pattern by showing that a Campaigner might have different actions to take while visiting different kinds of voters. In this system it will **not** be easy to change the structure of the Resident class hierarchy but it will be very easy to change the behaviour of the Campaigner or add a different kind of visitor without having to change any code in the Resident class hierarchy.

Participant	**Entity in application**
Visitor	Caller
Concrete Visitor	Campaigner
Element	Resident
Concrete Elements	HighIncome, LowIncome, MediumIncome
ObjectStructure	*Not implemented*
visitConcreteElement()	visit()
accept()	accept()
Client	main()
operation()	incomegroup()

Visitor

- The `Caller` class act as the visitor.
- The definition of the `visit()` method is overloaded to provide the functionality to visit the designated concrete elements (`HighIncome, MediumIncome` and `LowIncome`)

Concrete Visitor

- The `Campaigner` class acts as a concrete visitor.
- All variations of the `visit()` method can be implemented. It is possible to implement empty methods or rely on default implementations of the`visit()` function in cases where a visitor should not visit certain elements.
- It is easy to alter/add/remove concrete visitors from the system with no need to recompile the client or any of the elements in the object structure.

Element

- The `Resident` class acts as the element.
- The definition of the `accept()` method is polymorphic and allows for different implementations in `HighIncome, MediumIncome` and `LowIncome`.
- The `incomegroup()` method represent methods that are defined in the `Element` participant of the Visitor pattern. This method is typically called in the body of the `visit()` method that is implemented in the concrete visitors.

Concrete Element

- The `LowIncome, MediumIncome` and `HighIncome` classes act as the concrete elements.
- The implementation of their respective `accept()` methods is the second dispatch of the application of double dispatch. It calls the `visit()` method of the `Caller` it received as parameter. The implementation of this method that should appear in each concrete `Resident` is shown below.
- The `incomegroup()` method represent methods that are defined in the `Element` participant of the Visitor pattern. This method is typically called in the body of the `visit()` method that is implemented in the concrete visitors.

The following code is the implementation of the `accept()` method that should appear in each concrete `resident`.

```
void accept(Caller& v) {
    v.visit(this);
}
```

Although the code is identical in each of these classes they are different from a compiler point of view because the parameter that is passed in its parameter is of a different type in each of these cases.

In a typical execution the client calls the `accept()`-method of a concrete `Resident` and passes a pointer to a specific `Caller` as parameter. The `accept()`-method will then call

the `visit()`-method of the concrete `Caller` that was passed as parameter and passes a pointer to itself to this `Caller`. Lastly the `visit()`-method will call back to the concrete `Resident` by executing (in this case) its `incomegroup()`-method.

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].

[4] Jens. Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings. The Twenty-Second Annual International*, pages 9 –15, aug 1998.

[5] Wikipedia. Visitor pattern — wikipedia, the free encyclopedia, 2012. URL `http://en.wikipedia.org/w/index.php?title=Visitor_pattern&oldid=516592783`. [Online; accessed 21-October-2012].

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 27: Proxy Design Pattern

Contents

27.1 Introduction

This lecture note introduces the Proxy design pattern. One of the most prevalent implementations of the proxy pattern in C++ is in the implementation of smart pointers. Therefore an overview of smart pointers is presented in the programming preliminaries section before introducing the pattern. Other than using the proxy to implement smart pointers, three other implementation strategies are presented.

27.2 Programming Preliminaries

One of the uses of the Proxy design pattern is to manage smart references. In C++ there has been the notion of a Smart Pointer. As from C++11 this notion has been refined.

A C++ smart pointer is an abstract data type that simulates a pointer while providing additional features intended to reduce bugs caused by the misuse of pointers while retaining efficiency [2]. Smart pointers are used to keep track of the objects they point to for the purpose of memory management. This includes bounds checking and automatic garbage collection.

Many libraries exist that implement a type of smart pointer. The library under consideration in this section is the STL for C++11. The standard for C++11 was accepted in August 2011 and is being implemented in C++ compilers. In this standard, there are 3 kinds of smart pointers, `unique_ptr`, `share_ptr` and `weak_ptr`. `unique_ptr` can be replaced with `auto_ptr` as defined in the C++ standard prior to C++11. As with `auto_ptr`, it is defined in the header `<memory>`. The latter two are based on the implementation of smart pointers in the Boost library.

auto_ptr - `auto_ptr` is not implemented in C++11. It is however discussed here for the sake of completeness. Up till C++11 it was the only smart pointer implementation available in C++. In the way `auto_ptr` is implemented, the copy constructor and assignment operators do not copy the stored pointer. They copy the pointer, leaving the copied or assigned object empty. This strategy effectively transfers ownership of the pointer. It however does not provide a solution when copy semantics are required.

```
auto_ptr<int> value(new int(10));

cout<<value.get()<<endl; // access to the pointer
cout<<*value<<endl;      // access to the value of the pointer
```

unique_ptr - unique ownership, move constructible and move assignable. Changing the example given for `auto_ptr` can be adapted to make use of C++11 `unique_ptr` by replacing all references to `auto_ptr` with references to `unique_ptr`. Note, it may be necessary when compiling and linking to inform the compiler that C++11 is required by specifying the correct flags for your compiler if it supports the C++11 standard.

```
unique_ptr<int> value(new int(10));
```

```
cout<<value.get()<<endl; // access to the pointer
cout<<*value<<endl;      // access to the value of the pointer

unique_ptr<int> newValue = move(value); // Transfer ownership

newValue.reset() // deletes the memory
value.reset() // nothing to delete
```

shared_ptr - counted pointer, object is deleted when the use count goes to zero

```
shared_ptr<int>  value(new int(10));
shared_ptr<int>  newValue = value;
                  //newValue and value both own the memory

cout<<value.get()<<endl; // access to the pointer
cout<<*value<<endl;      // access to the value of the pointer

value.reset(); // memory still exists because of dual ownership
newValue.reset(); //last to own the memory, deletes the memory
```

weak_ptr - robust unowned pointer which is managed by a shared pointer

```
shared_ptr<int> value(new int(10));
weak_ptr<int> newValue = value; //value owns the memory
            // newValue holds a reference
```

27.3 Proxy Design Pattern

27.3.1 Identification

Name	Classification	Strategy
Proxy	Structural	Delegation
Intent		
Provide a surrogate or placeholder for another object to control access to it. ([1]:207)		

27.3.2 Problem

The proxy holds off using and therefore possibly also creating the real subject until it is necessary. This means that the cost of accessing the real subject is deferred. The availability of the real subject is therefore on demand. This means that the proxy can be seen as a replacement object where there is a requirement for a more sophisticated reference to an object than just a pointer.

27.3.3 Structure

The structure of the Proxy design pattern is given in Figure 1.

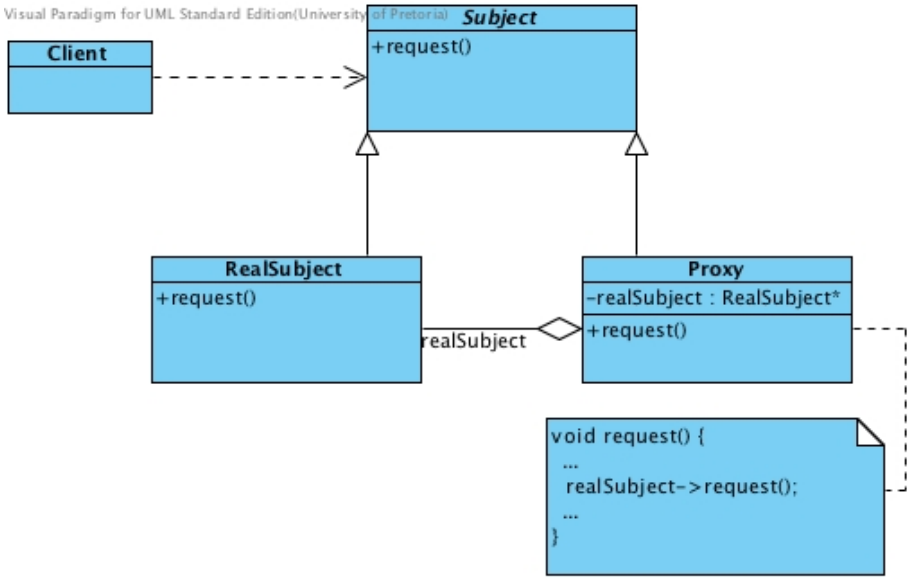


Figure 1: The structure of the Proxy Design Pattern

27.3.4 Participants

Subject

- Defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

RealSubject

- Defines the real object that is represented by the proxy.

Proxy

- serves as substitute for the real subject
- maintains a reference to the real subject
- controls access to the real subject
- may be responsible for creating and deleting the real subject
- more responsibilities specific to its kind

27.4 Proxy Pattern Explained

The proxy pattern may be applied to a number of situations. The most common situations where a proxy may be applied are given below.

Remote proxy: The remote proxy provides a local representation of an object in a different address space. It therefor is used to hide the fact the the object may be on a different computer. The remote proxy is responsible for:

- encoding a request and its arguments; and

- sending the encoded request to the real subject in a different address space.

Virtual proxy: The virtual proxy provides a local placeholder for an object that is expensive to create and maintain. It is used to postpone access to the expensive object until it is really needed. The virtual proxy is responsible for:

- creating expensive objects on demand; and
- caching information about the real subject so that access to it can be avoided if possible.

Protection proxy: The protection proxy controls access to the real object. It is used to control access rights to the real subject. Different users/objects may have different access rights. The protection proxy checks the access rights of a user/object for the particular request being issued. It may perform additional housekeeping tasks when the object is accessed.

Smart reference: When a proxy is implemented as a smart reference, it replaces a bare pointer and performs additional actions when the object is accessed. The typical uses of a smart pointer are:

Memory management - count the number of references to the real object

Load on demand - load a persistent object into memory on first reference

Safe updating - lock the real object before it is accessed

27.4.1 Related Patterns

Adapter

Adapter and Proxy both provide an interface to access another object. However, the reasons for doing this are different. Adapter a different interface to the object it adapts. Proxy provides the same, or diminished interface to its subject.

Decorator

Decorator and Proxy both describe how to provide a level of indirection to an object and forward requests to it. However, they have a different purpose. Decorator provides a dynamic attach or detachment of an object through recursive composition. The component provides only part of the functionality. One or more decorators provide furnish the rest. Proxy provides a stand-in win it is inconvenient of undesirable to access the subject directly. With the proxy, the subject provides the key functionality. The proxy provides (or refuses) access to this functionality.

Prototype

Prototype and Proxy both offer a solution to a problem related to an object that is expensive to create. However, the solutions are different. Prototype keeps a copy of the object handy and clones it on demand. Proxy creates a stub for the object and created it on demand.

Flyweight

Flyweight and Proxy both apply a smart reference to manage access to an object. However, the purpose of the reference is quite different. Flyweight controls multiple pointers to a shared instance. It can be related back to a C++11 share_ptr. A proxy controls single access to a specific object. This relates to a C++11 unique_ptr.

27.5 Example

One example is presented in two versions. Additional suggestion for other versions of the application of the proxy design pattern is also given after the code for version 2 has been given.

27.5.1 Message Server Example - Version 1

The following code implements a message server. The message server is seen as the RealSubject participant. A proxy participant takes the messages from the client and passes them on to the actual message server. In this example, each proxy message server links to its own actual message server.

```
// Subject
class MessageServer {
public:
  virtual void connect() = 0;
  virtual bool message(string) = 0;
  virtual void disconnect() = 0;
  virtual ~MessageServer() {}
};

//RealSubject
class ActualMessageServer : public MessageServer {
public:
    ActualMessageServer() {
        messages = 0;
    }
    void connect() {
        cout << "Connected..." << endl;
    }
    bool message(string msg) {
        cout << "Message is " << msg << endl; messages++;
        return true;
    }
    void disconnect() {
        cout << "Disconnected, " << messages
             << " message(s) have been received." << endl;
    }
private:
    int messages;
};

//Proxy
class ProxyMessageServer : public MessageServer {
    ActualMessageServer* implementation;
    bool connected;
public:
```

```
    ProxyMessageServer() {
        implementation = new ActualMessageServer();
        connected = false;
    }
    ~ProxyMessageServer() {
        delete implementation;
    }
    void connect() {
        if (!connected) {
            implementation->connect();
            connected = true;
        }
    }
    bool message(string msg) {
        if (connected)
            return implementation->message(msg);
    }

    void disconnect() {
        if (connected) {
            implementation->disconnect();
            connected = false;
        }
    }
};
```

27.5.2 Message Server Example - Version 2

This example adapts the implementation given in Version 1 so that there is only one message server, rather than one per proxy. It will require the proxy to 'register' with the message server when it is ready. This 'registration' can be simulated by sending the actual message server object through to the Proxy. It will also mean that the proxy may no longer delete the actual message server. An alternative implementation to using a pointer to the actual message server would be to:

- make use of a shared pointer; or
- make the actual message server a Singleton.

Figure 2 provides a possible solution to the description given above.

An example of how the proxy can be implemented to achieve the requirements for the example in version 2 is given in the following listing.

```
class ProxyMessageServer : public MessageServer {
    ActualMessageServer* implementation;
    bool connected;
    bool registered;
```

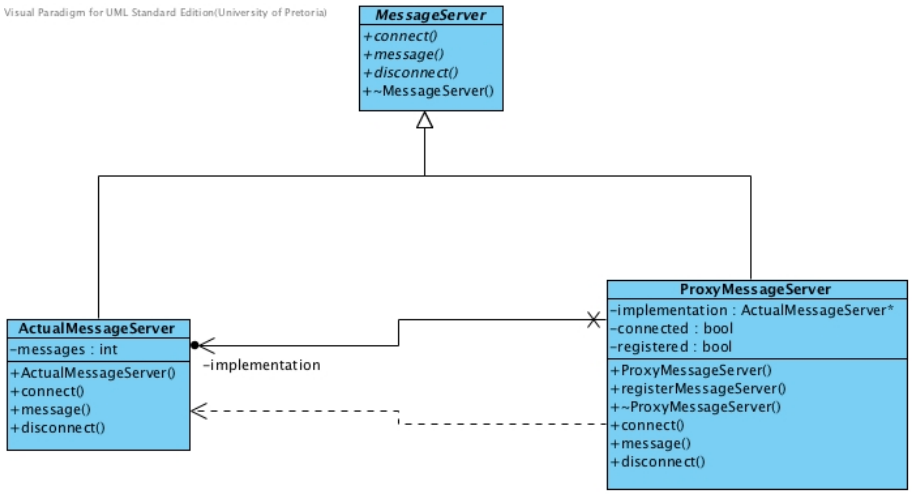


Figure 2: Proxy message server example 2

```
public:

    ProxyMessageServer() {
        registered = false;
    }

    void registerMessageServer(ActualMessageServer* ms) {
        implementation = ms;
        registered = true;
        connected = false;
    }

    ~ProxyMessageServer() {
    }
    void connect() {
        if (registered && !connected) {
            implementation->connect();
            connected = true;
        }
    }
    bool message(string msg) {
        if (registered && connected)
            return implementation->message(msg);
    }

    void disconnect() {
        if (registered && connected) {
            implementation->disconnect();
            connected = false;
        }
    }
```

};

References

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

[2] Wikipedia. Smart pointer — wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Smart_pointer`, 2013. [Online; accessed 14-October-2013].

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 28: Singleton Design Pattern

Contents

28.1 Introduction

In this lecture you will learn about the Singleton Design Pattern. When looking at it in terms of its class diagram and design, it is the simplest pattern. Its class diagram has only one class. However, it turns out to be quite complicated when considering its implementation and the consequences of the various ways in which it can be implemented. The singleton is probably the most debated pattern of all the classical design patterns.

The Singleton pattern is applicable in situations where there is a need to have only one instance of a class instantiated during the lifetime of the execution of a system. Sometimes because it makes logical sense to have a single instance of an object and sometimes because it might be dangerous to have more than one instance alive at the same time. Dangerous situations arise when resources are shared. For example if a system would have two instances of its file system running at the same time it can easily create a state where the two instances destroy the operations of one another resulting in unpredictable system state. The Singleton Design Pattern provide the means to write code that is guaranteed not to instantiate more than one instance of a specific class.

The Singleton pattern should be used with caution. We agree with the following remark by [2]

> *If you use a large number of Singletons in your application, you should take a hard look at your design. Singletons are meant to be used sparingly.*

28.2 Programming Preliminaries

28.2.1 Information hiding

The Singleton Design Pattern applies information hiding to ensure that its constructors are used in a strictly controlled manner. We assume that the reader is familiar with the concept of information hiding and the visibility of members of a class (i.e. public, protected and private) and their respective consequences. These are also discussed in more detail in Section **??**.

28.2.2 Destructors and virtual destructors

In C++ a destructor is generally used to deallocate memory and do some other cleanup for a class object and its class members whenever an object is destroyed. Destructors are distinguished by the tilde, the $\sim$ that appears in front of the destructor name. When no destructor is specified, the compiler usually creates a default destructor. In our experience we have learnt that relying on the default destructor is sometimes disastrous. Consequently we have specified in our coding standards that every class definition of a class with dynamic instance variables should explicitly containing a default constructor, copy constructor, assignment operator and destructor.

In a situation where a derived class is destroyed through a pointer to its base class, the base class destructor will be executed. This destructor will be unaware of any additional memory allocated by the derived class, leading to a memory leak where the object is

destroyed but all extra memory that was allocated beyond the base object memory, will not be freed. To remedy this, the base class destructor should be virtual. If it is virtual the destructor for any class that derives from base will be called before the base class destructor. For this reason [8] specify that destructors of polymorphic base classes should be declared virtual. An implication of this important design principle of class design is that if a class has one or more virtual functions, then that class should have a virtual destructor.

28.2.3 Static variables and class members

In C++ the `static` keyword is used to specify that a local variable in a function, an instance variable of a class or a member function of a class is static.

The use of static keyword when declaring a local variable inside a function means that once the variable has been initialised, it remains in memory until the program terminates. Although the variable is local to the function and cannot be accesses from outside the function it maintains its value in subsequent calls to the function. For instance, you can use a static variable to record the number of times a function has been called simply by including the following code in the body of the function:

```
static int count = 0;
count++;
```

The memory for local variables of a function are allocated each time the function is called and released when the function goes out of scope. When declaring a local variable of a function static, however, the memory for it will be allocated once and will not be released when the function goes out of scope. Thus, the line `static int count = 0;'` will only be executed once. Whenever the function is subsequently called, `count` will contain the value that was assigned to it during the previous call the the function.

Variables declared inside a class are used to specify the state of an instance of the class. Thus each instance of a class, allocates memory for all its variables and each of these variables can have a different value in the different instances of the class. On the contrary, a static instance variable has the same value over all instances of the class. Its memory is allocated only once and is shared by all the instances. It acts like a global variable that is visible only to instances of the class that defines it. In fact its memory is allocated even if the class is never instantiated.

Methods of class a may be declared static. Static methods can be accessed without instantiating the class. The following example code defines a utility class in Figure 1 called `FinanceTools`. It contains a number of static member functions.

```
class FinanceTools
{
    public:
        static double calculateSimpleInterest
            (double principal, float rate, int term);
        static double calculateCompoundInterest
            (double principal, float rate, int term);
        static double calculatePayment
            (double loan, float rate, int term);
};
```

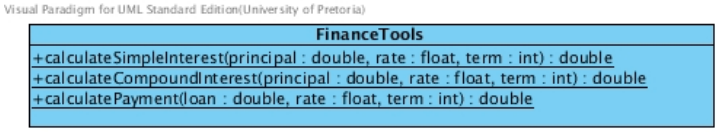


Figure 1: The FinanceTools class

Note that the member functions are underlined in the class diagram. It shows that they are declared static. Because all the member functions of this class are static and the class has no instance variables, it is not required that the class be instantiated. The member functions can be used without an instance of the class. Thus there is no need to implement constructors or a destructor for this class.

To use the static member functions of a class one can simply refer to them through the use of a class name and the scope operator, `::`. For example the following code uses the `calculatePayment` function without instantiating the class:

```
double installment;
installment = FinanceTools :: calculatePayment(220000, 23.7, 40);
```

28.2.4 Lazy initialisation

Sometimes classes that are implemented as part of a system are not always used while an application executes. Often software applications may contain functionality that most users of the program hardly use. An example may be the mail-merge functionality in a word processor that may not be needed when the user is using the word processor to write a scientific report. Another example is the math equation editor that may not be needed if the user is someone who is typing the minutes of a meeting.

To save resources the concept of lazy initialisation may be applied. This means that classes are not instantiated unless they are required. This is achieved by instantiating the classes only when their functionality is requested by the user.

28.3 Singleton Design Pattern

28.3.1 Identification

Name	**Classification**	**Strategy**
Singleton	Creational	Delegation
Intent		
Ensure a class only has one instance, and provide a global point of access to it. ([4]:127)		

28.3.2 Structure

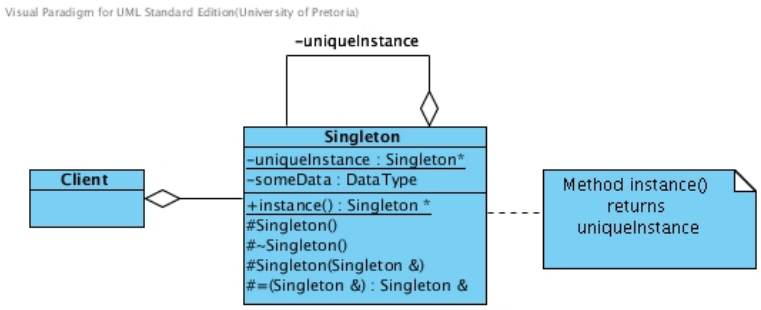


Figure 2: The structure of the Singleton Design Pattern

28.3.3 Problem

The Singleton design pattern address problems such as inconsistent results, unpredictable program behaviour, and overuse of resources that may arise when more than one object operate on some shared resources.

Examples of objects that only one instance is desired are thread pools, caches, dialog boxes, objects that handles preferences (like registry settings), objects used for logging, device drivers, etc. [2]

28.3.4 Participants

The Singleton pattern has only one participant

Singleton

- defines an `instance()` operation that lest clients access its unique instance. This method is a static member function in C++.
- may be responsible for creating and destroying its own unique instance.

28.3.5 Discussion

The Singleton pattern takes control over its own instantiation by hiding its constructor. This is done by declaring the constructor private or protected. If the constructor is private no class other than the singleton class itself can call the constructor. The consequence is that the class can not be instantiated at all because it cannot call its own constructor if it does not yet exist. This problem is solved by providing a public static member function that can be called by any class that needs to use the instance of the singleton class. Recall that a static method can be called by using the class name and the scope operator :: without the need to instantiate the class. This member function is responsible for creating the instance. The UML activity diagram shown in Figure 3 shows the creational logic that has to be implemented by this member function.

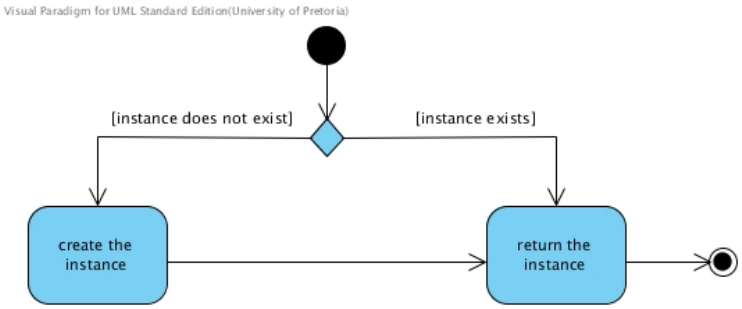


Figure 3: The creational logic of the Singleton Design Pattern

To allow the option of having derived singletons, the constructor can be declared protected. When allowing this it is assumed that the programmer still intend to have only one concrete instance of the singleton to to be instantiated, but would like to decide at runtime what variation is to be instantiated. The system should not allow the co-existence of different classes that are derived from the abstract singleton. The use of derived singletons are beyond the scope of these notes. [2] point to the dangers of the practice to subclass a singleton. The interested reader can find examples of how subclassing of singletons should be done in [5].

28.4 Singleton Implementations in C++

28.4.1 GoF Implementation

Various authors have implemented of the Singleton design pattern with subtle differences to address some of the consequences and flaws of the original implementation suggested by [4] shown in the following code:

```
class GoFSingleton {
  public:
    static GoFSingleton* getInstance();
  protected:
    GoFSingleton();
  private:
    static GoFSingleton* onlyInstance;
};

GoFSingleton* GoFSingleton :: onlyInstance = 0;

GoFSingleton* GoFSingleton :: getInstance() {
  if(onlyInstance == 0) {
      onlyInstance = new GoFSingleton();
  }
  return onlyInstance;
}
```

Observe the following:

- The `onlyInstance` member is declared static. Since this member is accessed for the first time before the class is instantiated it is declared static to ensure that it is assigned memory before the class is instantiated.
- The `onlyInstance` member is initiated outside the constructor. This is required because it is is declared to be static. If this initiation is not done, a call to the singleton will cause a segmentation fault.
- The constructor is protected. This hides it from other classes but allows for subclassing the `GoFSingleton` class.

28.5 Larman implementation

A serious consequence of the combination of multithreaded environments and optimising compilers is unpredictable behaviour where complicated creational logic is implemented. The implementation of the Singleton pattern is particularly at risk. The problems that may arise as a consequence of lazy initialisation can be remedied by applying **eager initialisation**. This is the opposite of lazy initialisation. The object is initialised at the beginning when the program is executed regardless if it is needed or not. The application of eager instantiation avoids complex conditional creation logic at the cost of instantiating an object that might not be used during execution. The following code adapted from [6] shows how this can be implemented:

```
class LarmanSingleton{
    public:
        static LarmanSingleton* getInstance();
    protected:
        LarmanSingleton();
    private:
        static LarmanSingleton* onlyInstance;
};
LarmanSingleton* LarmanSingleton ::
    onlyInstance = new LarmanSingleton();
}

LarmanSingleton* LarmanSingleton :: getInstance(){
    return onlyInstance;
}
```

The above code differs from the code in Section **??** in the following ways:

- Instead of intiating the `onlyInstance` to 0, and instantiating it only on the first call to the `getInstance()`-method it is initiated at startup.
- Because it is guaranteed that `onlyInstance` already exist when the `getInstance()`-method is called, the need to apply the prescribed creational logic is eliminated.

This, however, is usually not preferred because the instantiation of the object which is never actually accessed may be wasteful. Besides, one of the aspects of the intent of the Singleton pattern is to apply lazy initialisation. Technically this implementation is not really as Singleton and does not differ from having a static global variable pointing to the Singleton class.

28.5.1 Müldner Implementation

[11] points out that the unique instance of an object is not guaranteed by hiding only the constructor of the class. It might still possible for other classes to instantiate copies of the Singleton object through calling a copy constructor or assignment operator. Consequently the Singleton pattern is unable to comply with its intent to guarantee that only one instance can exist. This problem is remedied in the following code that is adapted from a Singleton implementation given by [11]:

```
class MuldnerSingleton {
  public:
    static MuldnerSingleton* getInstance();
    void updateSingletonData(int);
    void printSingletonData();
  protected:
    MuldnerSingleton();
    virtual ~MuldnerSingleton(){cout << "Destructing" << endl;};
    MuldnerSingleton(const MuldnerSingleton&){};
    MuldnerSingleton& operator=(const MuldnerSingleton&){};
  private:
    static MuldnerSingleton* onlyInstance;
    int  singletonData;
};

void MuldnerSingleton :: updateSingletonData(int i){
  singletonData = i;
}

void MuldnerSingleton :: printSingletonData(){
  cout << "Singleton data: " << singletonData << endl;
}

MuldnerSingleton :: MuldnerSingleton() : singletonData(0){}

MuldnerSingleton* MuldnerSingleton :: onlyInstance = 0;

MuldnerSingleton* MuldnerSingleton :: getInstance() {
  if(onlyInstance == 0) {
      onlyInstance = new MuldnerSingleton();
  }
  return onlyInstance;
}
```

The above code differs from the code in Section 27.4.1 in the following ways:

- The copy constructor and assignment operator are also hidden by declaring them protected. Since they are now declared, the compiler requires an implementation for each. The implementations provided, however, are empty because they can anyway not be called. They are just stubs to prevent the public ones from being created.
- Since its constructors are protected it is assumed that subclassing may be applied. It is declared virtual to comply with the principle discussed in Section 27.2.2.
- Additional instance variables and member functions were implemented to show how the rest of the Singleton class (its actual functionality) should be implemented, and also to provide detail to be able to see how it behaves during execution.

If the `getInstance()` method of the Singleton class returns a pointer to the only instance of the class, [11] argue that the client should avoid using a variable holding this pointer. One should rather call the operations of the Singleton class through the instantiating operation. For example in the above implementation the `printSingetonData()` operation should be called with the following statement:

```
getInstance()->printSingetonData();
```

This is to avoid the danger that the variable (which is of pointer type) can be used to delete the Singleton object while it is still in use. However, it is generally not a good idea to rely on programmers to treat the objects of your classes properly. Therefore, it would be much better to provide a mechanism to ensure that the Singleton is properly destructed. Understanding the destruction issues and how to address them is beyond the scope of these notes. The interested reader may read [13, 3, 1]

28.5.2 Meyers Singleton

[14] discuss a singleton he calls the Meyers Singleton. It is based on a remark by Meyers in an earlier version of [7]. The remark by Meyers was not intended to be a suggestion about how the Singleton pattern can be implemented. It was intended to offer a solution to address problems related to not being able to control the order of initiation of non-local static objects. Incidentally it provided a neat implementation of the Singleton pattern. This implementation overcomes most problems that may occur when Singletons are constructed. It also eliminates the destruction problems associated with most other implementations of the Singleton pattern. This solution is shown in the following code on the next page. This code makes use of references rather than pointers. It differs from the code in Section 27.5.1 in the following ways:

- The `onlyInstance` variable is no longer a static pointer to `self`. Instead it is now a static local variable (not a pointer) of the `getInstance()` method. The global initialisation of the `onlyInstance` variable is thus no longer required.
- The `getInstance()` method is changed. Instead of returning a pointer, it returns the Singleton itself as a reference. The creation logic is no longer needed. Lazy instantiation of the Singleton is achieved as a consequence of the language feature that static variables of functions are instantiated when the function is first called.

- Since `onlyInstance` is not a pointer, its memory is allocated on the stack. Consequently the destruction of the Singleton is no longer an issue.
- The rest of the implementation (not shown here) is *mutatis mutandis* the same as the implementation shown in Section 27.5.1.

The following is an implementation of the Meyers Singleton adapted from an implementation of this principle given by [11]

```
class MeyersSingleton {
  public:
    static MeyersSingleton& getInstance();
    void updateSingletonData(int);
    void printSingletonData();
  protected:
    virtual ~MeyersSingleton(){};
    MeyersSingleton();
    MeyersSingleton(const MeyersSingleton&){};
    MeyersSingleton& operator=(const MeyersSingleton&){};
  private:
    int singletonData;
};

MeyersSingleton& MeyersSingleton :: getInstance(){
  static MeyersSingleton onlyInstance;
  return onlyInstance;
}
```

Unfortunately the Meyers Singleton is not thread-safe [14]. The reader is referred to [9] and [10] for an interesting discussion about creating thread-safe Singleton.

28.6 Singleton Pattern Explained

28.6.1 Improvements achieved

- If the Singleton class provide access to resources that are shared. The usage of these resources is more robust than would be without the pattern. The Singleton encapsulates its sole instance, hence it can have strict control over how and when clients access it.
- If the Singleton class is resource intensive, the application of the pattern may contribute to more optimal resource usage.
- The Singleton Pattern is an improvement over global variables. Although global variables can provide global access (part of the intent of this pattern) it cannot guarantee that only one instance is instantiated (the more crucial aspect of the intent of this pattern).

28.6.2 Disadvantages

- The application of the Singleton pattern violates the *One class, one responsibilty* principle to an extent. This principle is aimed at increasing cohesion in classes. If a class has unrelated responsibilities the cohesion of such class is low. See Section **??** for a discussion of cohesion and coupling and why it is important. Inadvertently the Singleton class of the Singleton pattern has two unrelated responsibilities, namely to manage its own creation (and possibly destruction) as well as providing the functionality it was designed for in the first place.

- In multithreaded applications the creation step in the singleton design pattern should be made a critical section requiring thread concurrency control. If it is not properly managed it may happen that two different threads both create an instance on the class which may lead to undesired consequences. Furthermore, the overhead when implementing and executing the creation step in a thread-safe manner is needed only once (when the function is called the first time) but may be executed with every call to the function. If not implemented properly it may lead to unacceptable performance degradation. The implementation of the singleton in a thread-safe manner is beyond the scope of these notes. The interested reader can find an example of how this should be done in [12].

28.6.3 Implementation Issues

1. The intent of the Singleton design pattern can in some cases be obtained by means of a static or global variable. This, however, is deemed bad practice. Besides a global variable always applies **eager initialisation**. Thus, the benefits of lazy instantiation is forfeited when a global instance is used as opposed to applying the singleton pattern. Furthermore, sometimes the initialisation of the Singleton object may need additional information which might not be known at static initialisation time, which make it impossible to to apply a static global variable to implement the intent of the Singleton design pattern.

2. When [4] first proposed the Singleton pattern, different techniques for implementations with subclassing was discussed but nothing was said about the destruction of singletons. The problems related to creating Singletons in a multithreaded environment was also not addressed.

28.6.4 Related Patterns

- Singleton and Prototype both sometimes apply lazy instantiation. Singleton does this to save memory while Prototype uses it to save processing power.

- Abstract Factory, and Builder can use Singleton in their implementation.

- Façade objects are often Singletons because only one Façade object is required.

- State objects are often Singletons.

28.7 Example

Figure 4 is a class diagram of a PrintSpooler class. It implements the Muldner implementation of the Singleton Pattern.

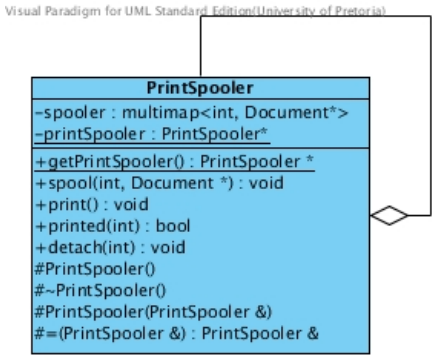


Figure 4: Class Diagram of a PrintSpooler Class

The following table shows the participant and its prescribed members.

Participant	**Entity in application**
Singleton	PrintSpooler
onlyInstance	printSpooler
getInstance()	getPrintSpooler

28.8 Conclusion

The Singleton pattern is a innocent looking pattern that provides a simple solution to ensure that only one instance of a class is instantiated for the duration of program execution. Unfortunately it can potentially give rise to many problems. It is extremely difficult to subclass, its destruction is controversial and it is almost impossible to implement it in a thread-safe manner. If the need to have only one instance of the class is not an absolute necessity, it is probably not worth the risk to implement the pattern.

References

[1] Andrei Alexandrescu. *Modern C++ design : generic programming and design patterns applied*. Addison-Wesley, Boston, MA, 2001.

[2] Eric Freeman, Elisabeth Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Sebastopol, CA95472, 1 edition, 2004.

[3] Evgeniy Gabrilovich. Destruction-Managed Singleton: a compound pattern for reliable deallocation of singletons. `http://www.cs.technion.ac.il/~gabr/papers/singleton_cppr.pdf`, 2000. [Online: Accessed 10 November 2012].

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

[5] Greg Ippolito. C++ Singleton design pattern. `http://www.yolinux.com/TUTORIALS/C++Singleton.html`, 2008. [Online: Accessed 10 November 2012].

[6] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice-Hall, New York, 3^{rd} edition, 2004.

[7] Scott Meyers. *Effective C++: 55 Specific Ways to Improve your Programs and Designs*. Pearson Education Inc, Upper Saddle River, NJ 074548, 3^{rd} edition, 2005.

[8] 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.

[9] Scott Meyers and Andrei Alexandrescu. C++ and the Perils of Double-Checked Locking: Part I. *Dr. Dobb's Journal*, 29(7):46 – 49, July 2004.

[10] Scott Meyers and Andrei Alexandrescu. C++ and the Perils of Double-Checked Locking: Part II. *Dr. Dobb's Journal*, 29(8):57 – 61, August 2004.

[11] Tomasz Müldner. *C++ Programming with Design Patterns Revealed*. Addison Wesley, 2002.

[12] Arena Red. Revisiting the Thread-Safe C++ Singleton. `http://www.bombaydigital.com/arenared/2005/10/25/1`, October 2005. [Online: Accessed 10 November 2012].

[13] John Vlissides. To Kill a Singleton. `http://www.research.ibm.com/designpatterns/pubs/ph-jun96.txt`, June 1996. [Online: Accessed 10 November 2012].

[14] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998.

Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 29: Flyweight Design Pattern

Contents

29.1 Introduction

29.2 Programming Preliminaries

29.3 Flyweight Design Pattern

29.3.1 Identification

Name	Classification	Strategy
Flyweight	Object Structural	Delegation
Intent		
"User sharing to support large numbers of fine-grained objects efficiently." ([1]:195)		

29.3.2 Problem

29.3.3 Structure

The structure of the Flyweight design pattern is given in Figure 1.

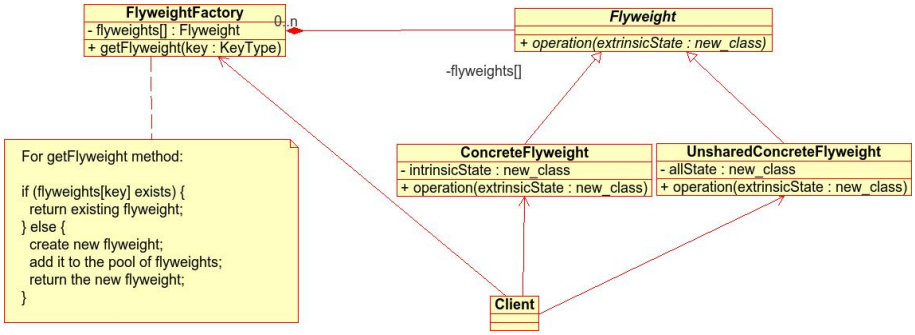


Figure 1: The structure of the Flyweight Design Pattern

Flyweights have both *intrinsic* and *extrinsic* state. Intrinsic state refers to the internal state of the flyweight and can be shared as it is independent of the context in which the flyweight is. For example: a flyweight may represent a letter. Extrinsic state refers to the context within the flyweight is and therefore cannot be shared. For example: flyweights are ordered in terms of the context to form words.

29.3.4 Participants

FlyweightFactory Creates an instance of a flyweight if it does not exist or supplies an existing one.

Flyweight Defines the interface through which flyweights are instantiated

ConcreteFlyweight Implements the interface and adds intrinsic (shareable) state storage.

UnsharedConcreteFlyweight Not all flyweights need to be shared. Therefore not all need to store intrinsic state. UnsharedConcreteFlyweights may have ConcreteFlyweights as children

29.4 Pre-knowledge

29.5 Flyweight Pattern Explained

29.5.1 Related Patterns

Composite In combination with flyweights, can be used to model directed-acyclic graphs.

States can be implemented as flyweights.

Strategies can also be implemented as flyweights.

29.6 Example

29.6.1 Balls

29.6.2 References

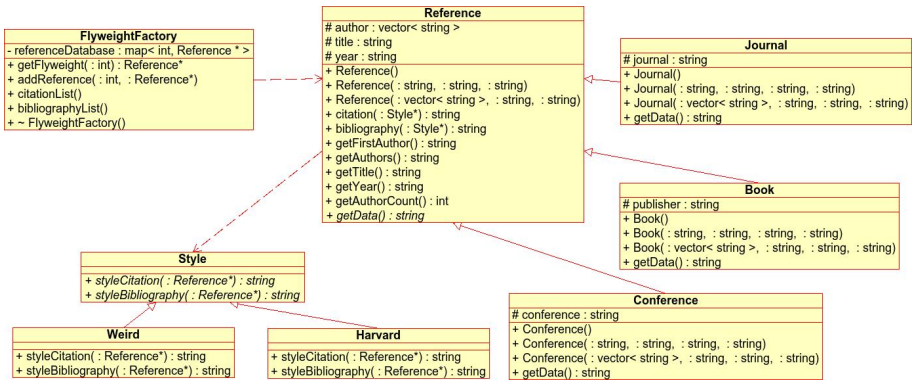


Figure 2: References example

Participants - Flyweight Design Pattern FlyweightFactory - FlyweightFactory Flyweight - Reference ConcreteFlyweight - Journal, Conference, Book ConcreteColleague - Student, Lecturer

Participants - Strategy Design Pattern Provides the Extrinsic state Context - Reference Strategy - Style ConcreteStrategy - Weird, Harvard

References

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