COS3711 – Advance Programming – 2013
Extra notes for Ezust[2] 2nd Edition

The notes provided here pick up on some of the issues that you may come across while working through the prescribed book for this module. Where an example is not mentioned here it means that the program works as described in Ezust[2].

CHAPTER 7 (Libraries and Design Patterns)

Section 7.2: Installing Libraries

Read through this section without downloading or installing any libraries suggested in this section. The installation of the Ezust libraries is detailed in Tutorial Letter 102.

Section 7.4.1: Serializer Pattern

For the Serializer pattern, rather use your notes on the pattern from COS2614. It would be constructive to compare those notes to the approach taken in Ezust[2], and see what the differences and similarities are.

CHAPTER 12 (Meta Objects, Properties, and Reflective Programming)

Section 12.2: Type Identification (playlists.pro)

This application may not want to compile as the `ui_mainwindow.h` (created by Qt) cannot find the `playlistsveiw.h` or `playlistview.h` files, even though they are in the correct folder. This has to do with folder structures, and once you have built the application you may have to change the two lines

```
#include "playlistsview.h"
#include "playlistview.h"
```

to

```
#include "..\..\playlists\playlistsview.h"
#include "..\..\playlists\playlistview.h"
```

(or something similar, depending on where the `ui_mainwindow.h` file is located in relation to the two files it is looking for).

Section 12.4: `QVariant` Class (properties.pro)

This application uses Qt's testing capabilities to check customer properties. The output is either displayed in the Application Output window in the bottom pane of Qt Creator, or (if *Run in terminal* is checked) it will display in a console window.

Section 12.6: MetaTypes

The src\metatype folder contains the `fraction.h` and `metatype.cpp` files needed for this example. To run this example, you will need to set up your own console project and add these two files to it. You need to make some changes before the program will compile and run:

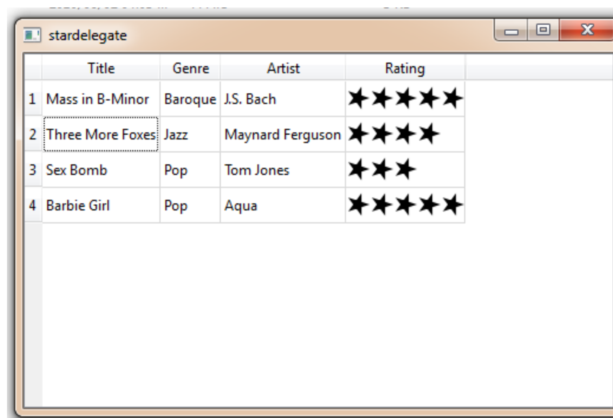
- In `metatype.pro`: make sure that `QT += gui` is included
- In `fraction.h`: Comment out the three overloaded << and >> operator functions below the class (or implement them yourself).

- In `metatype.cpp`:There is a ; missing at the end of the `qDebug()` statement in the middle of `main()`.
- Check the *Run in terminal* checkbox.

CHAPTER 13 (Models and Views)

Section 13.2.3: Delegate Classes (stardelegate.pro)

The example (figure 13.8) refers to `$QTDIR/examples/modelview/stardelegate`. This should be `C:\Qt\2010.04\qt\examples\itemviews\stardelegate`. It produces the following output. Figure 13.8 is a modified version of the example.



Section 13.3: Table models (p414)

Use the files in the folder `shortcutmodel-standarditem` to test the Examples13.10 to 13.12.

Section 13.4: Tree Models (objectbrowser.pro)

The code supplied does not work correctly. Comment out the two lines in `ObjectBrowserModel.cpp` as indicated below, and the program should then work as indicated.

```
QModelIndex ObjectBrowserModel::
index(int row, int col, const QModelIndex& parent) const {
    //if ((row < 0) || (col < 0) || row >= rowCount() ||      // comment out
        //col >= columnCount()) return QModelIndex();         // comment out
    return createIndex( row, col, qObject(parent) );
}
```

Section 13.4.1: Trolltech Model Testing Tool

You can simply read through this section; the programme does not work correctly in a Windows environment (and there are also problems with getting a degugger to work in a Windows environment).

CHAPTER 14 (Validation and Regular Expressions)

Section 14.3 Regular Expressions

At the bottom of page 443 there is a reference to a regular expression tester from Nokia in the src/regex-tester folder. Note that although the folder is there, there is no code for you to use. However, you can find the code in your Qt installation folder in examples\tools\regexp.

Section 14.3: Regular Expressions (regexp.pro)

There are two little things you need to do to get this example to work, both in the `regexp.pro` file.

- Comment out the `include (../common.pri)` line
- Add `CONFIG += console`

CHAPTER 15 (Parsing XML)

Note that XML files can be set up in 2 different ways: one format uses tags (like `book`) with attributes (like `title` and `pages`)

```
<library>
   <book title=”Computer Algorithms” pages=”688” />
   <book title=”C++ unleashed” pages=”918” />
</library>
```

or, using tags (`book`, `title`, and `pages`) with text (like `Computer Algorithms` and `688`)

```
<library>
   <book>
      <title>Computer Algorithms</title>
      <pages>688</pages>
   </book>
   <book>
      <title> C++ unleashed</title>
      <pages>918</pages>
   </book>
</library>
```

An element is anything from the start tag to the end tag of an element. So, `library` is an element, as is `pages`. Strictly speaking, attributes are supposed to supply extra information that is not part of the data (or text). However, there are no rules about when to use attributes and when to use elements. Some argue that using attributes is more limiting, and that using elements is more extensible. Note also that the indenting of the text above is simply to make it more readable.

Further, taking the tree (or hierarchical) structure of XML seriously, an XML document must contain a root element, although it does not matter what this element's name is.

```
<root>
   ....
</root>
```

This root then has branches that are called children.

```
<root>
   <child>
      <sub_child>
          ...
      </sub_child>
      <sub_child>
          ...
      </sub_child>
   </child>
   <child>
      ...
   </child>
</root>
```

Section 15.2: SAX Parsing (sax1.pro)

There are several things you need to note about running this example (which should run as given).

- Add the line `CONFIG += console` to the `sax1.pro` file.
- Note that the xml file that you want to parse should maybe be placed in the *build-desktop* folder instead of in its *debug* folder – it depends on which folders Qt will use as its default folder.
- You can place several xml files here, and then list them in the Arguments edit box in Run Settings – remember to type at least one name (the `samplefile.xml` file, for example) in the Arguments box if you want the program to run.
- If you are getting slightly garbled output, then use the `samplefile.xml` that is supplied with this document in the *Additional Resources* folder (in the *Chapter additional notes* folder) – there is sometimes a problem with the new line and carriage return characters.

The example given here simply parses the xml file and displays it. Suppose you had a `Person` class that had a `name` and an `age`. An xml file for this could look as follows:

```
<people>
   <person name=”Xin” age=”12” />
   <person name=”Ndou” age=”21” />
</people>
```

You thus know the structure of the xml file. If you now wanted to parse this file and then create instances of your Person class, you could do something along the lines of the following:

```
bool StockReader::startDocument()
{
    return true;
}
```

```
bool StockReader::startElement(const QString &namespaceURI, const
QString &localName, const QString &qName, const QXmlAttributes
&atts)
{
    if (qName == "person")
    {
        QString name = atts.value(0);
        int age = atts.value(1).toInt();
        p.setName(name);
        p.setAge(age);
    }
    return true;
}
```

Check the name of the tag

We know that its first attribute is the name, and the second the age

Set values in an object

```
bool StockReader::endElement(const QString &namespaceURI, const
QString &localName, const QString &qName)
{
    return true;
}

bool StockReader::endDocument()
{
    return true;
}
```

You would obviously have to have some way of dealing with `p`, the `Person` object that was created, and that would depend on your individual program.

However, if the xml file is not based on attributes but on text, we could have the following:

```
<people>
   <person>
      <name>Xin</name>
      <age>12</age>
   </person>
   <person>
      <name>Ndou</name>
      <age>21</age>
   </person>
</people>
```

Parsing this is different, and would involve something along the lines of the following. The variables used in the example (`inName` and `inAge`) would need to be members of the `XMLHandler` class.

```
bool XMLHandler::startDocument()
{
    inName = false;
    inAge = false;
    return true;
}
```

Set flags so that we know which tag we are in

```
bool XMLHandler::startElement(const QString &namespaceURI, const
QString &localName, const QString &qName, const QXmlAttributes
&atts)
{
    if (qName == "name") inName = true;
    if (qName == "age") inAge = true;
    return true;
}
```

> Set the appropriate flag

```
bool XMLHandler::characters(const QString& text)
{
    if (inName)
    {
        objectList.append(text);
        inName = false;
    }
    if (inAge)
    {
        objectList.append(text);
        inAge = false;
    }
    return true;
}
```

> Handle the text between the opening and closing tag, and reset the flag

```
bool XMLHandler::endElement(const QString &namespaceURI, const
QString &localName, const QString &qName)
{
    if (qName == "person")
    {
        p.setName(objectList[0]);
        p.setAge(objectList[1].toInt());
        pl.append(p);
        objectList.clear();
    }
    return true;
}
```

> Once we are back at the closing tag for person, we have all the detail we need to instantiate an object

```
bool XMLHandler::endDocument()
{
    return true;
}
```

Section 15.3.1: DOM Tree Walking (domwalker.pro)

Remember to put the `samplefile.xml` file in the *build-desktop* folder so that the example can read this file. Otherwise this code should run as explained in Ezust[2]. This example uses a form of the classic Visitor pattern, but as you are not expected to know this design pattern, this will not be discussed further.

Section 15.3.2: Generation of XML with DOM (dombuilder.pro)

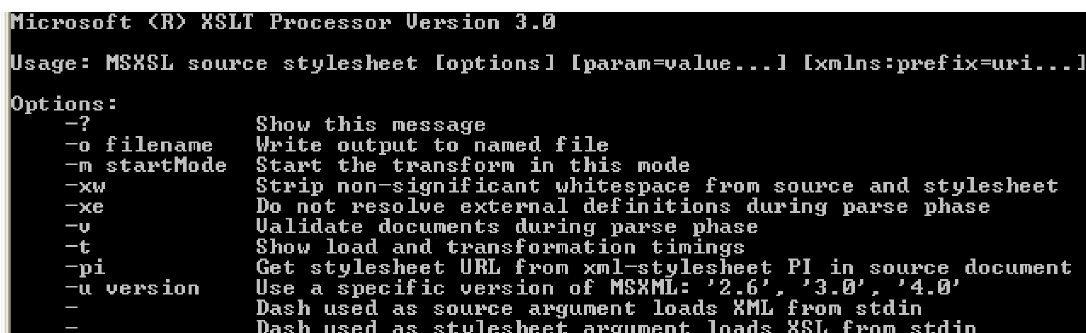
Note the comment at the top of the page: it is better to generate XML documents using an API than hard coding the output to ensure that the output is parsable. Thus, while you can do this

```
toFile << "<people>" << endl;
toFile << "  <person name=\"" << name << "\" age=\"" << age;
toFile << "\" />" << endl;
toFile << "</people>" << endl;
```

it is better to use the more complicated DOM approach.

The example provided should work as described. Example 15.15 refers to folder src/libs/docbook/ – this can be found in the c:\projects\libs\docbook folder, and you can find the files referred to there.

Example 15.14 is a Linux alternative. XSL stands for EXtensible Stylesheet Language, and XSLT stands for XSL Transformations – transforming XML (whose tags are not well understood by browsers) documents to other formats like XHTML (whose tags are well understood by browsers). Windows has its own command line XSLT processor, named msxsl.exe. This file can be downloaded from http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=21714. As you can see from the screenshot below, to use msxsl you need to give the name of the source file and the name of the stylesheet that defines the styles. If you would like to learn XSLT, look at the tutorial at http://www.w3schools.com/xsl/.



More on generation of XML with DOM

If you wanted to manually construct XML from an object, you can use the following outline. Say we wanted to create the following XML document.

```
<people>                 <-- Root element
   <person>              <-- 1st child
      <name>Xin</name>
      <age>12</age>
   </person>
   <person>              <-- 2nd child
      <name>Ndou</name>
      <age>21</age>
   </person>
</people>
```

You will need to create the document and a root.

```
QDomDocument doc;
QDomElement rootElement = doc.createElement("people");
doc.appendChild(rootElement);
```

Create document

Create root element

Add root to document

Then, for each child, you would create it as follows.

```
QDomElement personElement = doc.createElement("person");
rootElement.appendChild(personElement);
```

Create child element and add it to the root

```
QDomElement nameElement = doc.createElement("name");
personElement.appendChild(nameElement);
QDomText nameText = doc.createTextNode(name);
nameElement.appendChild(nameText);
```

Create name element and add it to person

Create the text and add it to the name element

```
QDomElement ageElement = doc.createElement("age");
personElement.appendChild(ageElement);
QDomText ageText = doc.createTextNode(age [as a string]);
ageElement.appendChild(ageText);
```

Now do the same for age

This then creates the XML document in a manner that you can be certain will be parsable.

Tutorial

There is a very nice, concise introduction to SAX and DOM XML parsing in Qt at http://www.digitalfanatics.org/projects/qt_tutorial/chapter09.html. Although it is part of a larger tutorial, it is clear enough to follow. One could, of course, go back to some of the previous tutorials and build the address book application that is used.

CHAPTER 16 (More Design Patterns)

Some specific issues relating to the content in Ezust[2] will be given here. At the end of these notes you will find extra information on many of the design patterns that you are expected to be familiar with.

Section 16.1: Creational Patterns

Ezust[2] discuss both the Factory Method and Abstract Factory design patterns. Watch the videos on these patterns on *my*Unisa, and then make sure that you can describe the differences between these patterns beyond what is given in Ezust[2].

Section 16.1.4: Polymorphism from Constructors

You will need to create your own project to test this little program. Remember to check the Run in terminal box on the Run Settings tab. Otherwise, the program should run as described in Ezust[2].

Section 16.2: Memento Pattern

According to the Gang of Four (E Gamma, R Helm, R Johnson & JM Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software), a memento is an object that stores a snapshot of the internal state of another object, and this state can only be "read" by the original object. This allows an object's state to be restored at some later point, allowing for an "undo" or a rollback operation. What we have in this section is not the memento pattern, but rather an implementation of the serializer pattern.

The Gang of Four do not include the serializer pattern in their set of patterns, although other authors do. Serializer is used to stream objects into other data structures (writing), and is also responsible for restoring objects from such a data structure (reading). It is commonly used to write and restore the state an object to a file or database, allowing for storage and retrieval. Go back and have a look in your study material for COS2614 (see also section 7.4.1 on page 249).

A comment here about `QObjectReader` and `QObjectWriter`. These follow the idea of a serializer well in that neither knows anything about the structure of the objects passed to it, and serializes the object as it finds it.

The `StockControl` project is a running version of an application that uses the `QObjectWriter` and `QObjectReader` classes.

CHAPTER 17 (Concurrency)

Section 17.1: `QProcess` and Process Control (randomNumbers.pro and logRandom.pro)

The `LogTail` project as included in Ezust[2] does not produce the desired output on Windows because the command `tail` is only applicable on a UNIX variant operating system.

A new example is provided in the *Chapter additional notes* folder (in the *QProcess and process control* folder) to demonstrate starting, controlling, and communicating with other processes similar to `LogTail`. To test the new project, follow the instructions below.

- Firstly, open, build, and run the `randomNumbers` project so that `randomNumbers.exe` is generated in the *debug* folder of the project. This project should display 10 random numbers. You can now close this project.

- Secondly, open and build the `logRandom` project. Before you execute the project, copy and paste `randomNumbers.exe` from the *debug* folder of the `randomNumbers` project `RandomNumbers` into the *debug* folder of the `logRandom` project. Run the project to view the output.

Section 17.1.1: Processes and Environment (environment.pro)

A modified version of this example, which works exactly as described in Ezust[2] is provided in the *QProcess and process control\environment* folder. Test the project using the command line argument `–f NULL`. The only difference is that the Windows version uses

putenv rather than setenv, and different environment variables are used (ones that are known in a Windows environment).

Section 17.1.2: Qonsole (qonsole1.pro)

This program will work as discussed in Ezust[2]. However, you will need to create your own project file. In a Windows environment, you can use commands like

- dir (to list directory contents)
- cd (to change to another directory)
- md (to make a directory)
- cls (to clear the screen)
- date (to get the current date)
- help (to get help)

to check that the example is working. Please remember to type in the command exit before you close the window to exit the command line process.

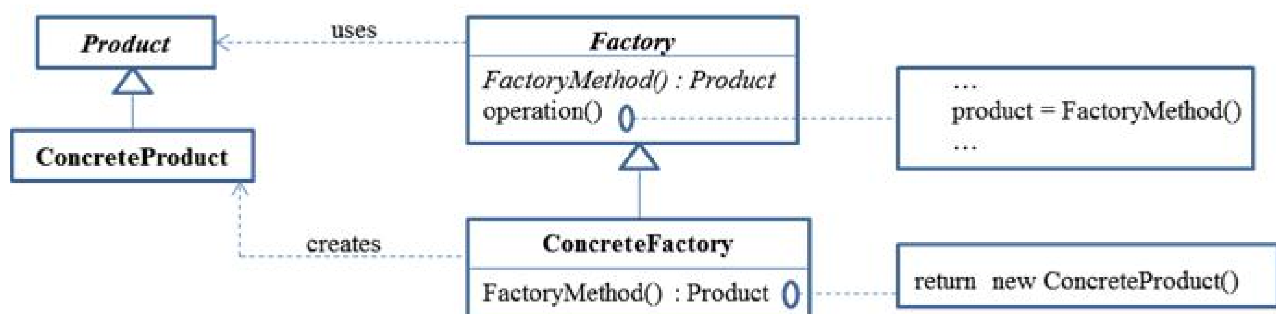
ADDITIONAL NOTES ON DESIGN PATTERNS

Creational Patterns

(1) Factory Method Pattern

This design pattern is applicable when a class needs to create objects but doesn't know which objects to create and the responsibility for creating objects can be transferred to the subclasses. So, this design pattern defines a class with a function for creating an object but the subclasses implement this function to create appropriate objects.

The UML structure of this design pattern is given below:



The main classes included in the UML diagram are:

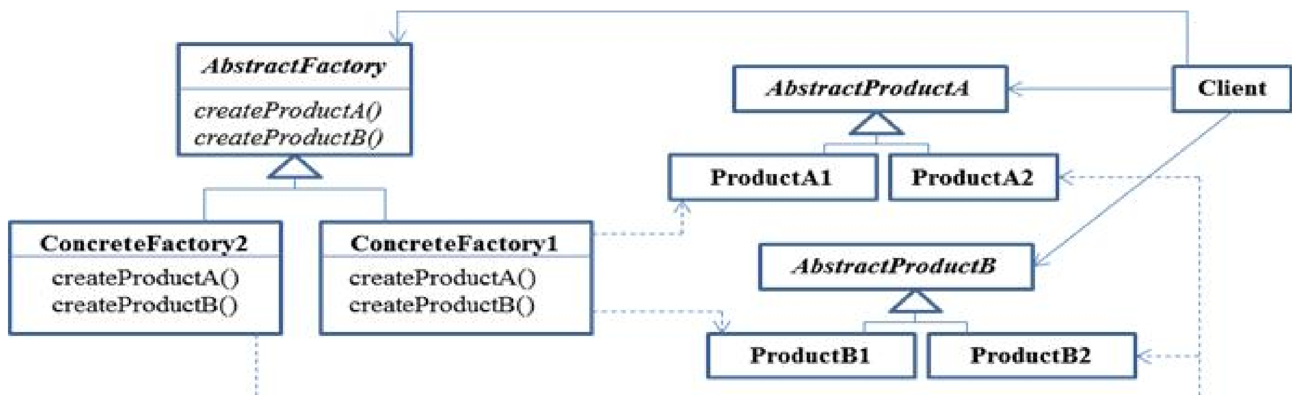
Product: defines the interface of the objects to be created by the Factory
ConcreteProduct: implements the Product interface, which is created by the ConcreteFactory
Factory: declares the factoryMethod(), which returns an object of type Product
ConcreteFactory: overrides the factoryMethod() to return an instance of type ConcreteProduct

There are a number of variations of this design pattern. Firstly, the `FactoryMethod()` in `Factory` may provide a default implementation of the factory method and it may call the `FactoryMethod` in one of its functions (`operation()`). This means the `Factory` is not always necessarily abstract. Secondly, the factory method could take parameters and based on them, several different objects can be created. This means a `ConcreteFactory` may create more than one type of object.

(2) Abstract Factory Pattern

This design pattern is applicable when you need to create families of related or dependent objects. This design pattern mainly has two hierarchies
(1) hierarchies of product classes, whose objects are meant to be used together
(2) then you have a hierarchy of factory classes to facilitate creation of the product classes.

The UML structure of the design pattern is given below:



The main classes included in the UML diagram are:

`AbstractProductA` and **`AbstractProductB`**: abstract classes of two related product classes
`ProductA1` and **`ProductA2`**: concrete product A classes
`ProductB1` and **`ProductB2`**: concrete product B classes
`AbstractFactory`: abstract factory class that declares functions for creating product A and B objects
`ConcreteFactory1` and **`ConcreteFactory2`**: concrete factories, which implement functions declared in the `AbstractFactory` to create concrete product A and B classes.
`Client`: it only uses the functions declared in `AbstractProductA`, `AbstractProductB` and `AbstractFactory`.

Using this design pattern, the client does not know about the implementation details of the concrete product families and it can interchangeably use any of the concrete factories to create the appropriate concrete products.

(3) Singleton

This design pattern is applicable when you want to have a class which has only one instance and this single instance can be accessed via a global access point. This is

generally achieved by making the constructor of the class `private`, creating one instance, and a function that can be used to return the single instance.

Since there is only one class involved, a UML diagram is not include here. Instead, one way of implementing Singleton is demonstrated below:

Class declaration

```
class A{
public:
     static A* getInstance();
private:
     A();
     static A* onlyInstance
};
```

Class definition

```
A* A::onlyInstance = NULL;
A::A(){}
A* A::getInstance(){
     if(onlyInstance == 0)
          onlyInstance = new A();
     return onlyInstance;
}
```

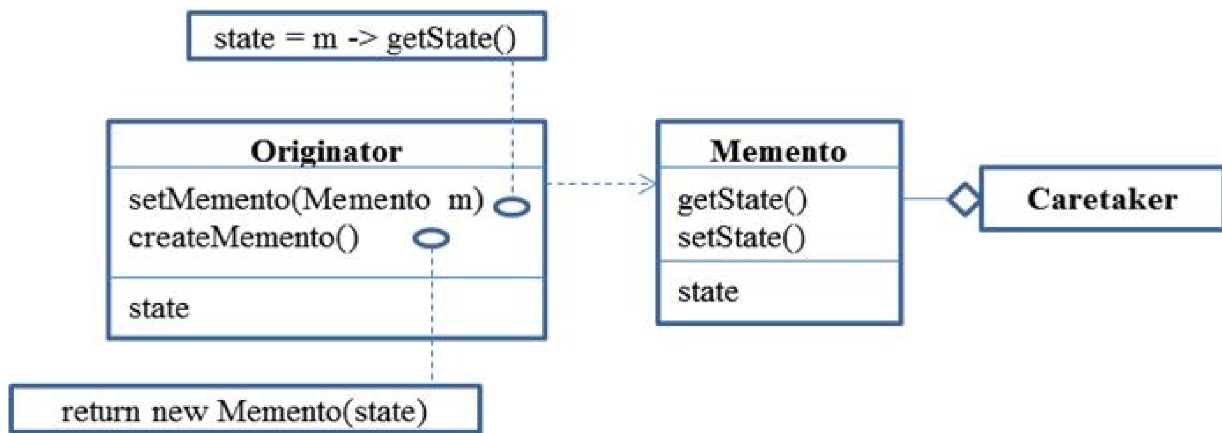
As demonstrated in the code, the single instance of `A` is stored in the `static` variable `onlyInstance`. When a request for an `A` instance is made via `getInstance()`, an instance of `A` is created and stored in `onlyInstance`. All subsequent calls to `getInstance()` do not create new instances but rather return the only instance of `A` stored in `onlyInstance`.

Behavioural Patterns

(1) Memento Pattern

This design pattern is applicable when the state of an object needs to be saved so that its state can be restored later without violating the encapsulation of the class. The data members of the class, which determine the state of its objects, may not all be accessible (via setters and getters) outside the class. Hence the class itself has to be involved in saving and restoring the state of its objects.

The UML structure of the design pattern is given below:



The main classes involved in the UML diagram are:

`Originator`: State of the objects of this class are being saved and restored. To save the state of an object it creates a snapshot of its state by creating a `Memento`, via the `createMemento()` function. To restore the state of the object, it uses a `Memento` via the `setMemento()` function.
`Memento`: stores the state of an `Originator`.
`Caretaker`: it keeps the `Memento`.

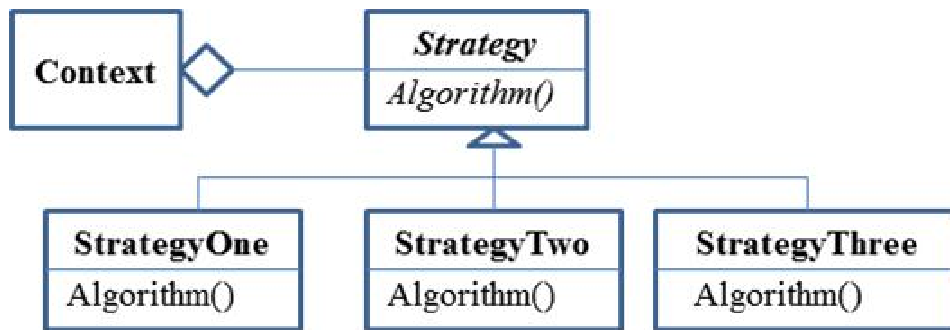
The role of the `Caretaker` is only to store the `Memento` and it is not allowed to access the state of the `Originator` saved in the `Memento`. One way of achieving this is by making the `getState()`, `setState()`,and even the constructor of `Memento`, `private` so that the `Caretaker` cannot access the state of the `Originator`. However `Orginator` needs to be able to access the constructor, `getState()`, and `setState()` of `Memento`. One way of achieving this is by making `Originator` a `friend` of `Memento`.

`Caretaker` requests a snapshot of an `Originator` using the `createMemento()` function. The `Originator` in turn creates a snapshot of its state by passing its state to `Memento` and thus creating a `Memento`, which is then passed to the `Caretaker`. If the state of the `Originator` has to be restored, then `Caretaker` can invoke the `setMemento()` function to restore the state of the `Originator`.

(2) Strategy pattern

The Strategy pattern is applicable when there is a family of algorithms that needs to be made interchangeable based on the context. Using this design pattern each algorithm is encapsulated in a class and the run time selection of a relevant algorithm is made possible.

The UML diagram of this design pattern is given below:



The classes involved in this UML diagram are:

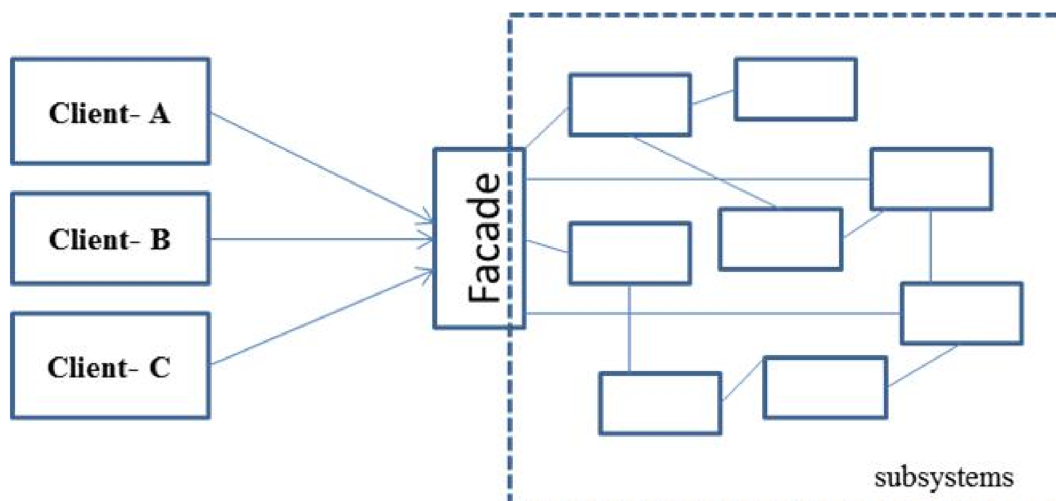
`Strategy`: It is the class which defines a common interface (here the function `Algorithm()`) for the classes that represents the family of algorithms. It is generally an abstract class and sometimes referred to as an abstract strategy
`StrategyOne`, **`StrategyTwo`** and **`StrategyThree`**: These are the concrete strategies, which defines the `Algorithm()` function. These classes are sometimes referred to as concrete strategies
`Context`: Context has a reference to a `Strategy` and it can decide on the concrete strategy to choose based on the requirement.

Structural Patterns

(1) Façade Pattern

The Façade pattern is applicable when you would like to provide a simplified interface to a complicated set of systems but yet allowing the client to access the functionality of the underlying system. This design pattern defines a Façade that interacts and invokes the subsystems to satisfy the request of the client.

A simplified diagram of this design pattern is given below:



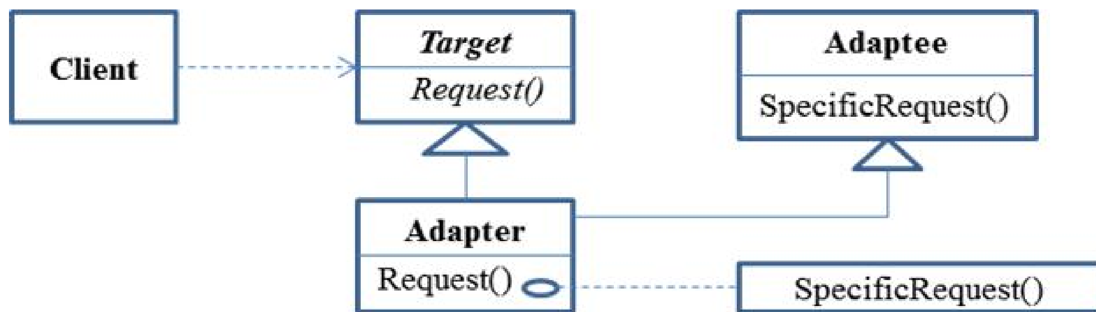
(2) Adapter/Wrapper Pattern

This design pattern is applicable when the functionality of a class can be reused but the interface of the class is not compatible with the existing classes. So this design pattern converts the interface of an existing class into another as expected by the client. This

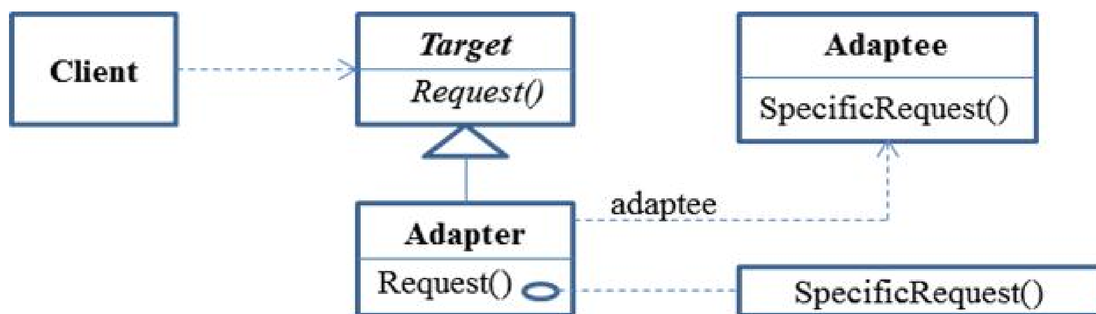
design pattern is sometimes called a Wrapper pattern since it can be seen as wrapping around an existing class to present a different interface to other classes.

There are two forms (Class Adapter and Object Adapter) of the Adapter pattern and the UML diagrams of both forms of the Adapter pattern are given below:

Class Adapter:



Object Adapter:



The main classes included in the UML diagram are:

`Client`: uses objects conforming to the `Target` interface
`Target`: defines the domain-specific interface used by the `Client`
`Adaptee`: defines the interface of an existing class to be reused
`Adapter`: adapts the interface of `Adaptee` to the `Target` interface

In a Class Adapter, `Adapter` adapts the interface of `Adaptee` by inheriting from `Adaptee`. In the `Request()` function of the `Adapter`, it simply invokes the `SpecificRequest()` function in `Adaptee`. Here the client call operates on an `Adapter` instance, which in turn calls `Adaptee` operations (`SpecificRequest()`, for instance) which carry out the operation.

In an Object Adapter, `Adapter` adapts the interface of `Adaptee` by creating an instance of `Adaptee` (`adaptee`). In the `Request()` function of the `Adapter`, it simply invokes the `SpecificRequest()` function on the instance of `Adaptee`. Here the client call operates on an `Adapter` instance, which in turn calls operations on the `Adaptee` instance to carry out the operation.

References:

E Gamma, R Helm, R Johnson & JM Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addision-Wesley

Lasater CG (2007). *Design Patterns*. Wordware Applications Library