Tutorial letter 102/3/2020

Programming: Contemporary Concepts

COS2614

Semester 1 and 2

School of Computing

CONTENTS:
Additional Notes

BAR CODE

Contents

Afrikaanse studente: Hierdie studiebrief is slegs in Engels beskikbaar. U is welkom om die dosente van COS2614 te skakel indien enigiets onduidelik is.

Why Qt?

As stated in MO001, the main focus of COS2614 is on object-oriented programming. There are many vehicles we could have used to cover the most important aspects of OOP – so why did we choose the Qt framework? There are a number of reasons:

- The Qt framework is written in C++. If you did your first level programming modules at Unisa, you should have the necessary background knowledge of C++ to be able to write programs using Qt.
- The Qt framework consists of a large collection of classes which allow one to do some really cool things without having to "re-invent the wheel". Many of the concepts and principles of OOP which we will be covering in COS2614 are illustrated nicely in Qt.
- Qt is a cutting-edge programming technology. Although, the Qt framework is no longer owned by Nokia, it is still used for developing applications for mobile devices. Although we don't specifically cover development of applications for mobile devices in this module, after completing COS2614 and COS3711, you should be in a position to do so quite easily.

So we hope that you are motivated and excited about learning about object-oriented programming using the Qt framework. Be warned, however, that OOP comprises some of the most difficult aspects of programming. You will have to work hard, especially in completing the assignments (both part A and B). Unfortunately there is no short-cut or easy way out. You will have to struggle until you get your head around things. Nevertheless, we believe that you will find it a rewarding adventure!

What's in this tutorial letter?

This tutorial letter provides additional notes on the prescribed book for COS2614 (which we call Ezust). Details of this book are given in MO001. In most cases this consists of comments on or a different view of topics covered in the prescribed sections of Ezust. In other cases, it is a summary or synthesis of topics that are explained in scattered sections of Ezust.

It is not necessary to study these notes in detail before you get going on COS2614. You might like to scan through them initially to get an idea of what they cover, so that you can refer to them as needed. We recommend that you start by installing the Qt Creator software and getting your first Qt program to work as explained in the instructions in DISK2019. Then scan through the section **Using Qt Creator** below briefly (it covers some tricks for using Qt Creator successfully that you will only need as you work through the prescribed book) and proceed directly to Chapter 1 of Ezust which you can read in conjunction with the additional notes provided below.

Using Qt Creator

The prescribed book by Ezust assumes that you are using a Linux platform, whereas we assume that the majority of COS2614 students will be using a MS Windows platform. We therefore prescribed a Qt programming environment which is available on both platforms, namely Qt Creator.

When you start up Qt Creator, you will notice buttons down the left hand side of the screen labelled Welcome, Edit, Design, Debug, Projects, Analyse and Help. These are used to select different *modes* of Qt Creator. Qt Creator starts in Welcome mode. Try clicking on a few of the buttons to switch to the other modes and then return to the Welcome mode.

Creating projects

In Qt Creator, you must create a project to get a program to compile and run. In other words, it is impossible to compile a C++ source code file (to get an executable file) without a project (as one can do in Code::Blocks, for example). The Welcome mode of Qt Creator therefore allows you to open a recent project, open an existing project (not listed as a recent project), or create a new project.

Creating console apps

If you want to create a new console application, select New Project button in the Welcome mode, which will bring up a New Project window. In the New Project Window, choose Applications (under Projects) in the first pane, Qt Console Application in the second pane and click on Choose... In the next window titled Qt Console Application, specify the project name and select the folder where you want to save the project before you select Next. Again click on Next in the next window. In the next window, simply click on Finish button to complete the creation of the Qt console project.

After you have created the console application, Qt Creator displays itself in the Edit mode with the project structure on the left pane and the content of the automatically created `main.cpp` on the right pane. You can remove this file (see below) or edit it.

Creating GUI apps

If you want to create a new GUI application, select New Project button in the Welcome mode, which will bring up a New Project window. In the New Project Window, choose Applications (under Projects) in the first pane, Qt Widgets Application in the second pane and click on Choose... In the next window titled Qt Widgets Application, specify the project name and select the folder where you want to save the project before you select Next. Again click on Next in the next window. In the next window, uncheck the Generate Form checkbox (since we are going to program user interface manually) before you select Next. In the next window, simply click on Finish button to complete the creation of the Qt Widgets project.

After you have created the Qt Widgets application, Qt Creator displays itself in the Edit mode with the project structure on the left pane with three files `main.cpp`, `mainwindow.h` and `mainwindow.cpp` created by default. You can remove these files (see below) if your project does not need these files or edit them.

Entering command line arguments

We do not place much emphasis on command line arguments in COS2614. However, there are some examples in the prescribed book that use them. Here is how to specify command line arguments in Qt Creator: In Projects mode, choose the Build & Run tab and select Run (instead of Build) in the box titled Desktop Qt 5.3.0 MinGW. Enter the command line arguments in the Arguments line edit (under Run). You need to enter the command line arguments before you run the project.

Adding and removing files from projects

The instructions on Disk2019 explain how to create a new source code file and add it to an empty project. However, sometimes you have source code in another file that you want to use. Do the following:

☞ In Edit mode, right-click on the project name (i.e. directory) in the Projects panel, and choose Add Existing Files ... Browse to the source code file (`.cpp`) that you want to add, select it and click on Open. It will be added to the list of source files under the current project in the Projects panel. (In Edit mode, double-click on any file in the Projects panel to get it displayed in the editor on the right.)

Sometimes you need to remove a source code file from your project. Do the following:

☞ In Edit mode, right-click on the name of the file you want to remove (e.g. `unwanted.cpp`) in the Projects panel and choose Remove File ... (You can click the Delete file permanently checkbox if you want to, but this is not recommended as you will effectively be throwing code away.) Click on OK to remove the file from the project. This means that when you compile your project again, this file will not be compiled and its object code will not be linked with the other code in your project to form the executable.

Although it isn't strictly necessary to add all the header files (`.h`) of classes used in a project, to the project, it is necessary for some of them (e.g. header files that contain the MOC keyword `Q_OBJECT` in the class definition). It is therefore safer to add them all.

Note that you must add all the source code (`.cpp`) files used by a project, particularly the implementation of classes included with `#include` statements like the one above.

Getting context-sensitive help

You can get information on any Qt class or member function in your source code by moving the mouse pointer over the class or function name in the editor and pressing the F1 key.

Accessing application programming interface (API) documentation

You can get information on any Qt class using the Index option (type the class name in the Look for: textbox) in the Help menu in Qt Creator.

Compiling Ezust code

Note that the examples in Ezust are written in Qt 4, whereas the prescribed software uses Qt 5. We have updated all the exercises in Ezust to run in Qt 5 and placed them in an easy to access folder structure under Additional Resources on *my*Unisa. For your convenience we have included projects in the exercises. So to compile and execute an exercise, open Qt Creator, click on Open Project button in the Welcome mode. In the Open File window navigate to the .pro file you want to work with. Select the .pro file and click on Open. Then select the Configure Project button. Once the project is open, you can compile and execute the project (using the options in the Build menu).

Chapter 1 of Ezust

Read through Chapter 1 of Ezust. You should compile and execute the programs given in this chapter (use the code provided by the authors – see **Compiling Ezust code** above). Most of the concepts covered in this chapter are covered in the prerequisite modules for COS2614, namely COS1511 and COS1512. This chapter should be regarded as a gentle introduction to a few Qt classes and a revision of concepts and C++ syntax covered in the pre-requisite modules.

You will notice a few differences between the C++ in Ezust and that which you may have been acquainted with so far. For example, the statement `using namespace std;` is often placed inside the `main()` function rather that just after the `#include` preprocessor directives. We prefer placing it earlier, simply because it then applies to all the code in the file rather than just the `main()` function. This aspect is not important (at all) since we will be using the Qt framework rather than standard libraries for almost all the programs in COS2614.

A more important issue is the fact that Ezust assumes that you are programming in a Unix/Unix-like environment and that you compile programs at the command line, whereas Unisa students are expected to work in an MS Windows environment and use an IDE (i.e. Qt Creator) to compile programs (See the instructions and explanations for compiling the first C++ example given just after Example 1.1 in Ezust.). Hence you can ignore the compilation and execution instructions given in Ezust. Instead use Qt Creator to compile and execute programs (see **Using Qt Creator** above).

++ *Section 1.6* explains the processes involved behind the scenes of an IDE (or command line) to successfully create an executable of a project. Read through this section because you need to have a theoretical understanding of the processes involved. You cannot try out all the instructions since they are not written for MS Windows.

++ *Section 1.8* introduces a Qt class `QString`. From now on, you should use `QString` instead of standard library strings.

++ *Section 1.9* introduces the Qt classes `QTextStream` and `QFile`. Work through the section so that you know what these classes are used for since you are expected to use Qt classes instead of standard library classes whenever possible. Note that we prefer to use `qDebug()` for debugging purposes only.

++ *Section 1.11* introduces a few GUI (dialog) classes in Qt. If you would like to know more about these classes access the API documentation via Qt Creator (see **Accessing API documentation** above).

++ *Section 1.13.1* something else you may not have encountered before are command line arguments. To see how they work in practice (Examples 1.19 and 1.20), you should know how to enter `spam eggs "space wars" 123` as command line arguments in Qt Creator (See **Entering command line arguments** above).

++ *Sections 1.15 and 1.16* about pointers and reference variables are important to understand, whereas Section 1.17 is not so important.

Learning outcomes of Chapter 1

You should be able to do the following:
- Write simple Qt programs that uses `QTextStream`, `QString` and `QFile`.
- Write a program that uses Qt dialog classes for input and output.
- Write a program that uses command line arguments.
- Understand the difference between and the uses of the unary operators `&` and `*` to work with pointers and addresses.

Chapter 2 of Ezust

++ *Section 2.1 and 2.2*. You should be familiar with the explanation of classes and their definitions given in Sections 2.1 and 2.2. Note the trick used in the `toString()` member function in Example 2.4 to convert integers to strings. Here the markers `%1` and `%2` are respectively replaced by the strings returned by `arg(m_Numerator)` and `arg(m_Denominator)`. Then a new `QString` is created using the strings returned by the `arg()` functions by placing a `/` in between the strings returned by `arg()`.

++ *Section 2.5*, which introduces UML class diagrams, is important. You might like to look briefly at the section **Notes on UML diagrams** at the end of this tutorial letter. One way of deciding whether there is a composition relationship between two classes is to ask, "Who is responsible for destroying the subobject in the relationship?" In this case when a `Rectangle` is created, we automatically get two `Point` objects. When a `Rectangle` is destroyed, it should automatically destroy both `Point` objects that it is composed of. Therefore the relation is *composition*. If a `Point` object could continue its existence after destruction of the `Rectangle` object to which it was related, then it would be an *aggregation* relationship and we would use a hollow diamond in the diagram. We'll see more examples of aggregation later.

++ *Section 2.6* on friends is interesting, but we will make seldom use of them.

++ *Section 2.7* (although straightforward) is important, especially the notion of a *member initialization list*. An initialization list is better to use than using a series of assignment statements because it is more efficient in the cases where an object is passed as a parameter to the constructor (fundamental types typically have negligible impact on efficiency). If you fail to initialise the data members in an initialization list, they will first be constructed (using their zero-argument constructors). The assignments will then overwrite their initial values. Besides being more efficient, initialization lists are much neater, and in most cases preferred for consistency.

++ *Section 2.8* is important. When an object of a class is destroyed, its default destructor frees the memory of each of its data members. This is illustrated well in Section 2.14. In many cases, the default destructor for a class is sufficient. However, when a class has pointer data members, and particularly when there is a composition relationship between the containing class and the contained class, then the default destructor is not sufficient. In fact, the default destructor will cause a memory leak because it will merely free the memory of the pointers, but not of the objects pointed at by the pointers. In this case, you must define and implement a destructor for the containing class, which explicitly frees the memory of the contained objects. Here is an example:

```
class Whole {
private:
    Part * p;
    // other data members
public:
    // other member functions
    ~Whole();
};
```

```
Whole::~Whole() {
   delete p;
}
```

++ *Section 2.9* discusses static data members, static member functions and static local variables. The most important (and trickiest aspect) of static data members is that they must be initialised in the implementation (`.cpp`) file of the class. See Examples 2.9 and 2.10.

++ *Section 2.10* discusses circular dependencies in classes and how to solve such dependencies. You will encounter circular dependencies in your programs sometime in this module so it is important to know why it occurs and how it can be solved.

++ *Section 2.11*. To summarise, the copy constructor (whether default or programmer-defined) is called (i) whenever the assignment operator is used in a declaration statement for initialising an object, (ii) for call-by-value parameter passing, and (iii) for return-by-value non-void functions.

Like the destructor, the default copy constructor and default assignment operator work well enough for most classes. But things get complicated when a class has a pointer data member. The default copy constructor merely makes a copy of the pointer, and the default assignment operator merely assigns the pointer value of the one object to the other. In most cases, one wants what is called a "deep copy" and a "deep assignment". In other words, the copy constructor needs to make a copy of the object pointed at by the pointer, and the assignment operator needs to assign the value of the one object pointed at by the pointer data member, to the other. Here is an example:

```
class Whole {
private:
    Part * p;
    // other data members
public:
    Whole(const Whole & w);
    Whole & operator=(const Whole & w);
    // other member functions
};

Whole::Whole(const Whole & w)
 : p(new Part(*(w.p)) //, initialisation of other data members { }

Whole & Whole::operator=(const Whole & w) {
    *p = *(w.p);
    // assignment of other data members
     return *this;
}
```

++ *Section 2.13*. The historical explanations in this section about the C language are not important. You should however always use `const` for all member functions that do not change the state of the object. This is good practice, not only to ensure that a member function doesn't inadvertently change the state of an object when it shouldn't, but also as part of a "contract" with client code: It states openly and clearly that the member function won't make any changes to the object it operates on.

++ *Section 2.14* gives the code that implements the UML class diagram given in Section 2.5.1 and illustrates nicely how initialization lists are used in constructors. Note that the destructor of the `Point` class in Example 2.22 is redundant – it could be left out since the default destructor (i.e. the one provided automatically if you don't define one yourself) would be quite sufficient. The class is instructive, however, in that it shows how often the destructor (whether default or programmer-defined) is called. It also shows how the `Point` objects are destroyed in reverse order to their construction as these objects go out of scope.

It is good programming practise to have the definition and implementation of classes in separate header and `.cpp` files with file names matching the class name. Try to put only one class definition in a header file and one class implementation in a `.cpp` file. Include the test function (`main()`) in a separate `main.cpp` file. Ezust does not practise this convention, which may not be a problem in simple examples but can lead to confusion in larger applications.

Something not covered in Section 2.14, but which is important for subsequent chapters, is the question of implementing composition by means of a pointer to an object as opposed to making the contained object a direct data member (as done in the `Square` class of Example 2.22). The decision (i.e. about whether to implement composition by means of a pointer or by a direct data member) depends on two things: whether you need dynamic memory, and whether you need polymorphic assignment.

Dynamic memory is used when you don't know how much memory you will need for the contained object(s). This might be as simple as using NULL if you don't need to store the contained object and a pointer to an object otherwise. In a more complex situation, you might need a whole linked structure (like a linked list or a binary tree) of unpredictable size. Then pointers are essential.

When you have a hierarchy of classes and when the contained object can be an instance of any class in the hierarchy, then you need to have a pointer to the base class, which can then point to any object of the class hierarchy. The ability of a pointer of type base class to point to any objects in its class hierarchy is called polymorphic assignment. You will learn and use this feature extensively later in this module.

The general rule of thumb is to use a direct data member rather than a pointer unless really necessary, since the use of pointers are bug-prone. As explained in the notes on Section 2.9 above, the containing object can easily create a memory leak when it is destroyed if it doesn't deallocate the contained object that its pointer points at.

If you decide that it is important to have a pointer data member for composition, then you must remember to implement the "Big Three" for the containing class, namely a destructor, a copy constructor and an assignment operator. This is explained in the notes on Sections 2.9 and 2.11 above.

This is because, in a strict composition relationship, the containing object is responsible for the lifetime of its contained objects. It should allocate memory on the heap (using `new`) for the objects pointed at by its pointer data members and deallocate their memory (using `delete`) when it is destroyed. The default copy constructor and destructor don't do this. You need to implement them yourself to do this explicitly.

Learning outcomes of Chapter 2

You should be able to do the following:

- Write a class, putting the class definition with the declaration of its member functions in a header file, and the implementation of its member functions in a source file.
- Use the `#include` pre-processor directive to include the header files of all necessary classes used in a header or source file.
- Use the pre-processor directives `#ifndef`, `#define` and `#endif` to ensure that a header file containing a class definition is not included more than once in the source code files of a project.
- Distinguish which members of a class (data members and member functions) should have `public`, `private` or `protected` access specifiers, and which member functions should be `const`.
- Draw UML class diagrams to specify the data members and member functions of classes and possible composition relationships with other classes.
- Write multiple classes that have a composition relationship with one another.
- Use static data members and static member functions correctly in a class.
- Explain, detect and resolve circular dependencies among classes.
- Implement a destructor for a class, particularly when the class has a pointer data member.
- Implement a copy constructor and assignment operator for a class, particularly when the class has a pointer data member.
- Know in what situations copy constructors are called.

Chapter 3 of Ezust

++ *Section 3.1.* We stick to the Qt style guidelines explained in Section 3.1 and strongly recommend that you do so as well in your programs for COS2614. In addition to these guidelines, we use a lowercase letter for the first character of variable or instance names.

++ *Section 3.2.1.* The most interesting thing about Example 3.3 is the `QDate` class, in particular its very powerful `toString()` member function which can convert a date into strings in all sorts of date formats.

Learning outcomes of Chapter 3

You should be able to do the following:
- Write Qt programs that comply with the style guidelines explained in Section 3.1.
- Write console applications that declare an instance of `QTextStream` (called `cout`) for outputting.
- Use the `QDate` class to convert dates to different formats.

Chapter 4 of Ezust

Chapter 4 introduces two containers provided by Qt, `QList` and `QStringList`. More Qt container classes are explained in Chapter 6.

++ *Section 4.2* introduces different iterators. Example 4.1 demonstrates different ways of appending elements to a `QStringList` object, different ways of iterating through a `QStringList` container object as well as the functions `split()` and `join()` of the `QString` class. Example 4.3 demonstrates how to use `QDirIterator` to iterate through a directory structure. Note that you need to provide a path to a directory as command line arguments for Examples 4.2 and 4.3.

In the remaining prescribed chapters of Ezust, we mainly use the 1st style of iteration mentioned in Section 4.2, namely the `foreach` loop.

++ *Section 4.3* explains how to represent the situation where one class has a container of objects as a data member in a UML class diagram. Note the hollow diamond used for the relationship between the `Employer` and `Person` classes in Figure 4.1. This indicates that an `Employer` object is not responsible for the destruction of any `Person` objects that it contains. They can have an existence independent of the `Employer` to which they are related.

Figure 4.2 gives a more complicated example of the same principle. The fact that `Employer` has a composition relationship with `Position` (indicated by the solid diamond) means that the `Employer` class is responsible for freeing any memory used to store the `Position`s it contains.

Sometimes, we can avoid composition by using inheritance. In particular, an alternative to making a container a data member of another class (as in Figures 4.1 and 4.2) is to inherit from the container class. For example, say we wanted a list of `Fraction`s (as illustrated in Figure 2.5). Instead of defining a class which has `QList<Fraction>` as a data member, we could simply inherit from `QList<Fraction>` as follows:

```
class FractionList : public QList<Fraction> {
  public:
    void appendFraction(const Fraction & f) {append(f);}
    Fraction sum() const;
};

Fraction FractionList::sum() const {
    Fraction result(0, 1);
    foreach (Fraction f, *this) {
       result.add(f);
    }
    return result;
}
```

It is good to be able to inherit from Qt classes but such an inheritance relationship is not always a good design choice.

++ *Section 4.4* lists Example 4.4, which is not available in the source code of the textbook but the given code listing is sufficient to understand how the `ContactFactory` and the `ContactList` instances should be used for creating `Contact` instances with random values.

Learning outcomes of Chapter 4

You should be able to do the following:
- Write programs that use the `QList` and `QStringList` classes.
- Write classes that have data members which are instances of the `QList` or `QStringList` classes.
- Make use of iterators in your programs
- Write classes that inherit from `QList`.
- Draw UML class diagrams that indicate aggregation, composition and association relationship between classes.
- Understand the difference between aggregation and composition relationships between classes.

Chapter 5 of Ezust

++ *Section 5.1* explains the notion of function overloading. We believe that this lowers the readability of code, as it is not always clear which definition of the function will be selected during execution. An exception, of course, is with constructors, where we regularly provide a number of versions with different parameter lists to allow an object to be constructed in different ways.

++ *Section 5.2*. Optional arguments are one way to avoid function overloading. We recommend you use them whenever you can, rather than writing two versions of the same function. An obvious exception is of course where there are no sensible default values that may be used for the arguments.

++ *Section 5.3* is interesting and you might have encountered most of it in first year programming modules. It is not important, however.

++ *Sections 5.4, 5.5, 5.6, 5.7 and 5.8*. Even though the material covered in these sections should be familiar to you, they are very important and you should make sure you understand them. You might be surprised that many classes of the Qt framework are passed by value (e.g. `QString` and `QList`). Generally in C++, complex objects are passed by const reference when we don't require changes that a function may make to them to reflect in the actual parameters. The problem with value parameters is that they make a copy of the whole object, which is inefficient in terms of time and memory space. The reason why it is acceptable to use value parameters for these Qt classes is because they utilise so-called "implicit sharing". This is discussed in detail in Section 11.5.

Example 5.12 is instructive. It shows that any change to a pointer passed as a value parameter does not affect the pointer, but that this doesn't prevent the function from changing the value of the thing that the pointer points to.

++ *Sections 5.9, 5.10 and 5.11* are interesting but not important for this module.

Learning outcomes of Chapter 5

You should be able to do the following:
- Overload functions as appropriate.
- Use default parameter values to allow optional arguments for functions.
- Use pass-by-value, pass-by-reference, and pass-by-const-reference parameters as appropriate for functions and member functions.
- Use return-by-value and return-by-const-reference as appropriate in functions and member functions.

Chapter 6 of Ezust

++ *Section 6.1*. It is essential to have a thorough grasp of all the aspects covered in this section. You should read through this section and make sure you understand everything.

The UML diagram in Figure 6.1 is not entirely correct. Can you find the errors and correct it?

Note how `Student` objects are passed to the `finish()` function by pointer in Example 6.5. This is done to illustrate how polymorphism works for the calls to `getClassName()` and `toString()` when they are made virtual in Example 6.6.

++ *Section 6.2* is just as important. Note that the code in Example 6.7 is identical to that in Example 6.5 – the point is that by making the `getClassName()` member function virtual (by simply adding the keyword `virtual` in the superclass) we get run-time binding of the appropriate member function to the object on which it is called.

It is interesting to note that `finish()` will also work correctly (with polymorphism) if the parameter is passed by reference. (Try it!) However, we recommend that you always pass a pointer to the base class as parameter when polymorphism is required, as this is the standard way.

See if you can change Example 6.6 so that `toString()` provides a complete string representation of the respective objects as discussed in page 179.

The exercises in Section 6.2.1 are a good test of your understanding of polymorphism.

++ *Section 6.3* is also very important. Once again, we are not sure why `Shape` objects are passed by pointer to function `showNameAndArea()` in Example 6.16 rather than by reference.

++ *Section 6.4* explains why it is a good idea to have a design plan before coding, especially when a number of classes are involved. The questions at the end of the section are important.

++ *Section 6.5*. Function hiding is explained in this section. It is important to note that a member function in a subclass hides *all* the member functions in its superclass(es) with the same name, irrespective of their signatures. As it turns out, it is very seldom necessary to access a hidden member function, but if you need to do so, Example 6.19 shows you how. Do note that you can't compile and run a project containing Examples 6.18 and 6.19 because the implementations of the classes `Account` and `InsecureAccount` are not provided.

Overriding isn't illustrated nicely in this section. Examples 6.13, 6.14, and 6.15 (in Section 6.1) illustrate it better.

The scope resolution operator is also often used in so-called *partial overriding*. This is where a member function of a derived class overrides an inherited member function of (one of) its superclass(es) but calls that member function itself. So consider the following implementation of the `deposit` member function of `InsecureAccount` as defined in Example 6.18:

```
void InsecureAccount::deposit(double amt, QDate postDate) {
   m_LatestTransaction = postDate;
   Account::deposit(amt);
}
```

This is called partial overriding, because when `deposit` is called for an instance of `InsecureAccount`, the inherited `deposit` is called – some additional functionality is just performed as well. This is very similar to a constructor calling a base class constructor to initialize inherited data members (as in Example 6.22).

++ *Section 6.6*. As explained in the notes on Section 2.14 above, we refer to the three types of member functions discussed in this section as the "Big Three". As a general rule, you need to implement the Big Three for a class when the class has data members which are pointers. In this sense, it isn't necessary to implement the Big Three for the `Account` and `JointAccount` classes given in Examples 6.20 and 6.21 since neither of them have pointer data members. This has just been done to illustrate how they are inherited from a base class.

Note how the `Bank` class calls the `qDeleteAll()` function in its destructor (in Example 6.24). The `qDeleteAll()` function takes a container of pointers as parameter, and uses the `delete` operator on each of the pointers in the container. Calling `qDeleteAll()` in a destructor is necessary when there is a composition relationship between the containing class (e.g. `Bank`) and the contained objects (e.g. `Account`), and the contained objects are accessed via pointers stored in a Qt container (e.g. `QList<Account*>`). The `Bank` class should also have a copy constructor and an assignment operator.

++ *Section 6.7*. Example 6.25 shows the structure of a C program that could be used for accepting command line arguments. It compiles fine but no output is produced.

Test the program in Example 6.28 with the example argument lists (example: `item1 "item2 item3" item4 item5`) used in the sample outputs given just below it. To enter command line arguments in Qt Creator, see section

Entering command line arguments above.

++ *Section 6.8* lists the various Qt containers available. There is also a `Queue` container class that works similarly to `QStack`.

The last two paragraphs of this section are important for associative containers (i.e. `QMap`, `QHash`, `QMultiMap`, `QCache` and `QSet`). You should always make sure that the objects you want to store in an associative container are of an assignable class. This might require implementing a default constructor, copy constructor or assignment operator.

++ *Section 6.9*. The discussion about managed containers as opposed to aggregate containers in this section is a nice illustration of the difference between using a solid versus a hollow diamond in an UML diagram to indicate composition or aggregation. At the risk of sounding like a stuck record, the question to ask is "Who is responsible for destroying the contained objects?" If it is the container class, then it should be a managed container, but if it is the rest of the program, then the container class should be an aggregate container.

++ *Section 6.10*. The discussion about containers of pointers and how to handle them correctly to avoid memory corruption is very important. Refer to the `Library` class implementation (a managed container), which has a composition relationship with `RefItem` with respect to the destruction of `RefItem` objects.

++ *Section 6.11*. Source code for examples 6.44 to 6.48 are not available in electronic format. However, the code segments given in the textbook are sufficient for answering the Review Questions.

Learning outcomes of Chapter 6

You should be able to do the following:

- Draw UML class diagrams that indicate which members of classes are private, public and protected, and that indicate inheritance relationships with other classes.
- Write multiple classes that have an inheritance relationship.
- Write client code that utilises multiple classes in an inheritance hierarchy, using polymorphism when appropriate.
- Overload member functions within a class as appropriate.
- Override and hide member functions derived from a superclass and use partial overriding as appropriate in classes.
- Decide when abstract base classes are appropriate, and design and implement class hierarchies that use them.
- Decide when it is appropriate to implement the Big Three for a class that contains a collection of other objects, and do so.
- Understand how the `ArgumentList` class can be used to process command-line arguments and switches.
- Understand what associative containers are and how they differ from the other Qt containers; understand what assignable data types are and use them appropriately with associative containers.
- Understand the differences between managed and value (i.e. unmanaged) containers, and use them appropriately in programs.

Chapter 8 of Ezust

This chapter introduces the powerful `QObject` class. One of the nice features of Qt is its child management facility for memory management. When using Qt's GUI classes, widgets are automatically added as child objects to the main window or widget via the layout object. However, if you are dealing with objects of non-GUI Qt classes you need to set the parent of an object using the `setParent()` function.

++ *Section 8.1* is important. Read, understand and remember!

++ *Section 8.2*. The UML class diagram in Figure 8.2 is confusing. The `Customer` class and the `CustomerList` class should be removed from this diagram, as these are not part of the Composite design pattern. If we were to apply the Composite design pattern in a program which needed to treat customers and customer lists similarly, then the Leaf class would be represented by `Customer`, and the Composite class would be represented by `CustomerList`. See the section **Design Patterns** at the end of this tutorial letter for more information on the Composite pattern.

++ *Section 8.2.2*. The best way to see the difference between an inheritance hierarchy and a child-parent tree

structure is to get the code in Examples 8.2, 8.3 and 8.4 to work.

This is such an interesting program because it shows how the child management of `QObject` prevents memory leaks which the code in Example 8.4 appears to be riddled with. The general rule to prevent memory leaks is that every object created on the heap with `new` should be deallocated with `delete`. This program creates a single `Person` on the variable stack, namely `bunch`, and then creates a whole lot of other `Person`s on the heap, making sure that they are children (or children of children) of `bunch`.

The parent object `bunch` is destroyed implicitly when it goes out of scope (i.e. when the `main` function terminates) and due to the child management inherited from `QObject`, it deallocates all its children (actually all its descendants) as can be seen in the output provided. This is achieved by the Composite design pattern.

A good test to see whether you understand the parent-child model is to do the exercise in Section 8.2.2.1.

The diagram in Figure 8.4 is completely incorrect if taken to be a UML class diagram, which it is not – don't be confused by the UML like diamonds.

++ *Section 8.3*. Don't worry about the description of the Observer Pattern when you first read this section. This will make more sense when you have seen programs that apply this pattern. See the section **Design Patterns** at the end of this tutorial letter for more information on the Observer pattern.

++ *Section 8.4* explains some important differences between standard C++ programs and programs that use the Qt framework. The reason why a Meta Object Compiler (MOC) is needed, is because there are some programming constructs used by the Qt framework which are not part of standard C++ syntax. For example, the code of a class that defines signals and/or slots (as in Example 8.5) contains the keywords `signals` and `slots` which are non-standard C++ syntax. To allow this, you must use the MOC macro `Q_OBJECT` in the class definition. This tells MOC to fix the code so that it can be compiled by a standard C++ compiler (like MinGW).

++ *Section 8.5.* If you want to have your application respond to events, you have to use the `connect()` function to link a signal emitted by one object to a slot of another object. In many cases, the standard signals emitted by Qt widgets can be used as is (for example, a `QPushButton` object emits a `clicked()` signal whenever the user clicks on it). However, if you want to generate an event when some condition becomes true, then you have to define a class that emits a signal explicitly.

Often, the standard slots provided by Qt widgets can be used as is, as well. The Qt help pages are a good place to look to see what signals and slots the various widgets emit and handle.

++ *Section 8.6* lists some useful tips in managing the lifecycle of a `QObject`.

++ *Section 8.7*. Testing programs is an important aspect of programming. This section briefly explains the `QTestLib` framework to support unit-based testing and the example in this section demonstrates the use of pre-defined macros for testing. You will not be expected to write your own unit-based tests for your programs in this module.

Learning outcomes of Chapter 8

You should be able to do the following:

- Write a class that inherits from the `QObject` class, and write a program that creates a tree of objects of that class to use the child management capability of `QObject` to avoid memory leaks.
- Understand the Composite design pattern, and how the `QObject` class is a simplified application of this pattern.
- Understand the event handling model that the Qt framework provides, implemented by means of signals and slots.
- Understand Qt's support for unit-based testing.

Chapter 9 of Ezust

This chapter contains many useful techniques required for developing Graphical User Interface applications.

If you take a look at the extremely simple Qt program you were told to compile and run in the instructions for installing and testing Qt Creator provided on Disk2019, you will see that the `QLabel` object played the role of the root or parent widget. This is explained in the introduction to Chapter 9.

++ *Section 9.1* lists relevant classes for GUI development. You will be using many of them in your assignments.

++ *Section 9.2* explains how to use the Designer to automatically generate the code for developing the user interfaces. You may choose to design your user interfaces using the Designer but you should be able to manually program the interface as well.

++ *Section 9.3* explains different modes of using dialogs.

++ *Section 9.4.* The `InputForm` class demonstrates how user interfaces can be manually programmed. You need a `main()` function to see the user interface as shown in Figure 9.10.

++ *Sections 9.5 and 9.6*. Examples 9.4 and 9.5 refer to files in Ezust's `cards2` library. Use the files provided on *my*Unisa under Additional Resources.

Note that nothing happens when the user clicks on the Deal and Shuffle buttons – they are just there to illustrate how layouts can be used to arrange widgets on a window.

As explained in Section 9.5, the power of resource files is that images can be compiled and linked into the executable, which means that the final `.exe` file can be distributed on its own without having to copy all the image files with it.

++ *Section 9.6.1* explains how spacing and stretching can be used to get the widgets and layouts on a window to behave better when the window is resized.

++ *Section 9.6.2* explains how the size of widgets can be adjusted.

++ *Section 9.7* The source code given in this section of Ezust is not sufficient to execute the examples. Modified source code files are available on *my*Unisa under Additional Resources.

What is important to note is the way slots are defined for a class (that inherits from `QDialog`) to allow the code to react to signals that are generated when the user performs some action on the dialog (like clicking on a button).

++ *Section 9.9* Once again, Examples 9.13 to 9.16 require some modification to make them work without using the compiled libraries. Use the source code files available on *my*Unisa under Additional Resources.

The `KeySequenceLabel` class demonstrates how one can listen and respond to keyboard and mouse events. It also shows one possible way of making use of a timer (provided by the `QObject` class).

As explained in this section, we rather use signals and slots than custom events whenever possible.

Examples 9.17 to 9.18 demonstrate the use of the `QTimer` class for timed signals. Target strings in this example are generated only when the Exposure time is specified.

++ *Section 9.9* demonstrates how to define a custom `paintEvent()` function for a `QWidget` subclass.

Learning outcomes of Chapter 9

You should be able to do the following:

- Write programs that use dialog boxes with varying numbers of buttons, to provide messages to the user, and to react appropriately to the button the user clicks on.
- Write programs that use input dialogs to allow the user to respond in a richer way (e.g. select an item from a combo box or choose a value using a spin edit) and to react appropriately to the user choice.

- Understand the advantages of using resource files when displaying images in an application.
- Use multiple layouts to arrange widgets on an application window, and use spaces, stretches and struts to ensure that they move appropriately when the window is resized.
- Program graphical user interfaces manually.
- Write programs that listen and respond to low level events
- Make use of timers in your programs
- Understand the role of paint events in widgets

Chapter 10 of Ezust

++ *Section 10.1*. Example 10.2 is a fun little program which shows how to use dialog boxes in a program. The `Dialogs` class basically creates and shows the parent widget. The constructor sets up the menu bar and its dropdown menu, as well as the actions that have to be performed when the user chooses an option.

Note that the program doesn't need to declare or create any objects to display the dialogs. The static member functions `question()`, `information()`, `warning()` and `critical()` of the `QMessageBox` are used instead. The number of buttons that appear on a dialog (between one and three) depends on the number of arguments provided in the call to one of these member functions.

Note the differences and similarities between the `addAction()` member function of the `QMenu` class and the `connect()` member function of the `QObject` class.

++ *Section 10.1.1* Before you try to play the game, note the following while the program is running:

- You can drag the toolbar around the window. (You can put it at the top or the bottom or on either side.)
- The buttons on the toolbar echo the menu options on the menu bar. The corresponding choices are enabled/disabled. For example, if you click on the Use the Force toolbar button (or the Use the Force menu item), three other toolbar buttons and their corresponding menu items are enabled.
- The same tip is displayed in the status bar, whether you hover the mouse pointer over a toolbar button or over its corresponding menu choice.
- Toolbar buttons remain depressed after you have clicked them, and their corresponding menu items are checked.

All of these aspects are done for us by the `QActionGroup` class.

++ *Section 10.2*. Have a look at the Dock Widgets Example in the Qt installation folder. It gives you an idea of the look of `QDockWidget`s.

++ *Section 10.3*. Examples listed in this section are the same as Example 10.1.

When you have got this program running, try resizing (or maximising) the application window, changing the size of the Text Editor (by dragging the bar between it and the Debug Window), and dragging one of the toolbars to the status bar (i.e. to the bottom). Then close the application and run it again. The changes you made during the previous execution of the program should still be in force. This is what is meant by the *persistence* attained by the `QSettings` class.

We are not sure that the `QSettings` class is a particularly typical or good example of the Monostate pattern. In the classic Monostate pattern, a number of instances of a class are created which all share exactly the same state. So if any part of the program refers to any one of them, it gets the same state, and if any part of the program changes any one of them, all instances are affected. This is usually achieved by means of static data members and static member functions. The `QSettings` class allows different instances to have different states within the same program, depending on the arguments provided during construction. In other words, to ensure that multiple instances of `QSettings` share the same state, you must use the same string arguments (for the organisation and application names). `QSettings` preserves the state of the program from one execution to the next, whereas a classic monostate class only maintains one state within one execution of a program. Finally `QSettings` achieves this by saving its state to a file (or to the Registry in the case of Windows – see below) and doesn't use static members.

To see how the state of the application is saved in the Registry, you can do the following:

☞ From the Windows Start menu, choose All Programs, click Accessories, right-click Command Prompt and then point to Run as administrator. In the command prompt that opens, type `regedit.exe`. This will open the Registry Editor. (Please note, this is a rather dangerous thing to do. You should not change anything in the Registry Editor, or your computer might start doing strange things – or even stop working altogether.) Now choose Find ... on the Edit drop-down menu and do a search for `objectlearning`. Expand this folder and you should see Qt4 Sample Main. If you click on this folder, you will see the three registry entries for pos, size and state that were last written by the `writeSettings()` member function. Close the Registry Editor.

++ *Section 10.4*. explains how to access the system clipboard.

++ *Section 10.5*. Take a look at the additional notes on the Command pattern at the end of this tutorial letter, and see if you can identify which Qt classes in Examples 10.11 to 10.14 correspond to the classes in the UML representation.

++ *Section 10.6*. briefly explains Qt support for internationalization.

Learning outcomes of Chapter 10

You should be able to do the following:

- Write programs that use dialog boxes with varying numbers of buttons, to provide messages to the user, and to react appropriately to the button the user clicks on.
- Write programs that use input dialogs to allow the user to respond in a richer way (e.g. select an item from a combo box or choose a value using a spin edit) and to react appropriately to the user choice.
- Use menus, menu bars and toolbars together with actions and action groups to allow users to perform the same action in a program in different ways.
- Write programs that use the `QSettings` class to allow persistence of the program state from one execution to the next.
- Understand the Monostate pattern.
- Understand the `QSettings` class and how it can be used to preserve the settings of an application.
- Understand the Command pattern.
- Understand that Qt supports Internationalization.

Chapter 11 of Ezust

++ *Section 11.1*. Read through and make sure you understand this section (including Subsections 11.1.1 and 11.1.2). Note that Examples 11.1 and 11.2 are not complete and hence you cannot successfully compile them.

++ *Section 11.1.2* The implementation of the `Stack` class provided in Example 11.3 is rather strange. It makes each node in the linked list responsible for deallocating the node attached to it (and hence for all nodes lower in the stack). This is achieved by a recursive destructor.

We suggest a different implementation where the `Stack` class itself is responsible for deleting all its nodes:

1. The destructor of the `Node` class should be removed.
2. The following member functions of the `Stack` class need to be re-implemented:

```
template <class T>
Stack<T>::~Stack() {
   while (m_Head != NULL) {
      Node<T> * head = m_Head;
      m_Head = m_Head->getNext();
      delete head;
   }
}
template <class T>
T Stack<T>::pop() {
   if (m_Head != 0) {
      Node<T> * popped = m_Head;
      T retval = popped->getValue();
```

```
            m_Head = m_Head->getNext();
            --m_Count;
            delete popped;
            return retval;
        }
    }
```

An alternative (and really cool) way is to make use of the child management facility offered by `QObject`. For this, both the `Stack` and `Node` classes need to inherit from `QObject`. Every time a new node is added to (i.e. pushed onto) the stack, the node must set the stack as its parent. For this to work, the destructor for the `Stack` class should be removed to allow the inherited destructor to deallocate all its children.

++ *Section 11.2*. Almost all generic functions (like `power()` in Example 11.1 and `qSort()` in Example 11.6) require some operator to be implemented for the type of objects they are used on.

++ *Section 11.3*. The `QMap` class is a managed container class, so it will free the memory for all values stored in it when it is destroyed. However, if the values are pointers to objects, it will not deallocate the memory for these objects. You (the programmer) need to do that yourself.

The code given in Examples 11.8, 11.9 and 11.10 illustrates this. We recommend that you try to get it to work. Note that the Ezust source code provided for this example is incomplete. You will have to write the constructor and a few of the member functions of the `Textbook` class yourself. (In fact, you only need `getIsbn()` and `toString()` for the purposes of getting `qmap-example.cpp` to work, so you can comment out all the other getters and setters.) We also recommend that you implement a destructor for the `Textbook` class which simply outputs a message that this particular instance is being destroyed. This will show clearly which instances are destroyed and when.

We are not sure why the `Textbook` class inherits from `QObject` (or why it uses the `Q_OBJECT` macro). This is only useful if a different means of cleaning up the memory is used, namely to use `QObject`'s child management facility. So if we make both `Textbook` and `TextbookMap` inherit from `QObject`, we can make the `TextbookMap` instance the parent of each `Textbook` instance that we add to it. Then when the `TextbookMap` is destroyed, it will automatically destroy all its `Textbook`s. See if you can make these changes to these classes.

There are a number of problems with the code in Example 11.10. Firstly, the `main` function has direct access to the books added to the `TextbookMap` object `m`, since it maintains pointers to each of them. This violates the principle of encapsulation, namely that the only way to access the data members of a class should be through its member functions. Secondly, the `TextbookMap` class deallocates all heap memory pointed at by its pointers in its destructor, but the final object (that is removed from `m`) is not deallocated. This is a memory leak. Finally, after `m` is destructed, `t1`, `t2` and `t4` are dangling pointers. In other words, these three pointers point at deallocated memory, and we should not try to dereference them (as is done to `t3` at the end of the `main()` function) as it could give unpredictable results, and even cause memory corruption. It is easy not to make these mistakes here, since the program is short and straightforward, but very often with long and complicated programs, the declaration and allocation of objects is far separated from where they are added to and/or removed from a container. Keeping track of which ones have been deallocated and which pointers are dangling or will cause memory leaks, becomes a nightmare.

To fix these problems, the following would need to be done:

- Remove the declarations of the four pointers at the beginning of `main()`, and then edit the statements that add the textbooks to construct the instances as they are added. For example, the first add instruction should be

  ```
  m.add(new Textbook("The C++ Programming Language","Stroustrup", "0201700735",
                                                                          1997));
  ```

  This will preserve the encapsulation of the `TextbookMap` class.

- Re-implement (i.e. override) the `remove()` member function inherited from `QMap` (or define a function with a different name) so that it deallocates the memory pointed at by the pointer that is removed.

The `foreach` loop provided in the box *More than One Way to Traverse a Map* is inefficient because it steps though `keys()` (which is a `QList` of keys) and does a lookup for the associated value for each one. Each lookup takes $\log n$ time.

If you don't need to display the key with each value, you can rather use `values()`, as in

```
foreach (Textbook * t, values())
    cout << t->toString() << endl;
```

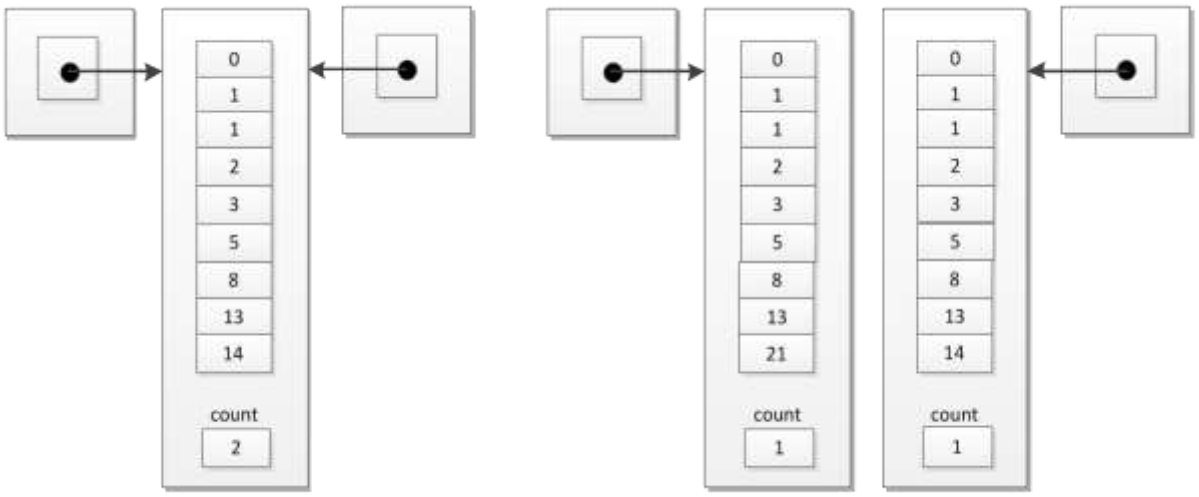
In this case, each lookup will only require constant time.

++ *Section 11.4*. Read through the section. Example 11.13 is incomplete, so it will not compile successfully.

++ *Section 11.5* is an interesting and useful feature of many Qt classes.

Qt uses implicit sharing to save the huge overhead of copying an entire container (and in particular all the objects in it) every time a copy of it is needed. Only when one or other copy is changed, are the contained objects actually copied. As stated in Section 11.5, this is achieved by reference counting. In other words, the data structure storing the objects in the container keeps count of how many pointers are pointing to it. (If there are two copies of the implicitly shared container, there will be at least two pointers to the data structure.) When an attempt is made to change the state of either of the containers in any way (by adding or removing items, or changing the value of an item) then a copy of the data structure is made, and only the one on which the change was being attempted, is actually changed.

The diagram below illustrates this. In the left hand drawing, a copy of the implicitly shared container has been made. Instead of copying the data structure (and all the objects in it), a shallow copy is made. As soon as an attempt is made to change one of the values in either of the containers (in this case the last value in the first container is changed from 14 to 21), a copy of the entire data structure is made, and the change is made to the appropriate value.



The advantage of this arrangement is that sometimes it is not clear whether some code is going to change a copy of a container. Only when an attempt is actually made to change the state of the container in some way, is a copy of the internal data structure actually made.

Note that many of the Qt container classes, e.g. `QList`, `QImage`, `QMap` and `QHash` inherit from `QVariant`, and therefore all implement implicit sharing (see Section 8.1).

Note also that implicit sharing does not solve the problem of implementing the Big Three for containers which store pointers to objects. In other words, if an implicitly shared container is used to store pointers to objects, you still need to implement the Big Three for the container class to avoid memory leaks and to avoid changes to objects in one container from affecting the objects in a copy of the container.

Learning outcomes of Chapter 11

You should be able to do the following:

- Write programs that use function templates, and make sure that the types that you want to use with a function

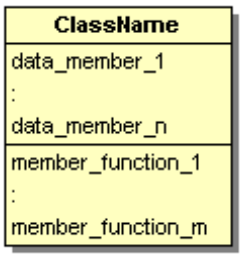
template overload the required operators.
- Understand functors.
- Understand what implicit sharing is and how this is employed in Qt container classes to save copying managed containers when it isn't necessary.
- Understand the Flyweight pattern.

Notes on UML class diagrams

Ezust uses UML class diagrams quite frequently in various places in the text, and different aspects and conventions of these diagrams are explained at different points. We therefore provide a summary of UML class diagrams here, at least those aspects that are needed for this course.

Notation for classes

In UML class diagrams, classes are represented by boxes with three compartments:



The top compartment specifies the name of the class, the second compartment lists all the data members, and the third compartment lists all the member functions.

The name and type of each data member is normally indicated, as is the full signature of each member function (i.e. its name, its parameters and their types, and the function's return type if it isn't void). See Figure 2.1 for an example.

Every member (data member or member function) is preceded by a +, – or # symbol indicating its visibility, i.e. whether it is public, private or protected, respectively. See Figure 2.1 and Figure 6.1 for examples.

Static members are underlined. See Figure 2.3 for an example.

Data members that are constants are written in uppercase.

Abstract classes have their names printed in *italics*, as do the names of abstract (pure virtual) member functions. See Figure 6.3 for an example of an abstract class.

Sometimes, for the sake of brevity, the data members and member functions are omitted from the bottom two compartments of a class box. See Figure 2.4 and Figure 6.2 for examples.

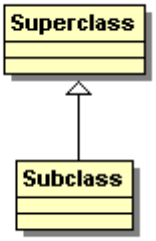
Notation for relationships between classes

In most cases, a number of related classes are indicated in a UML class diagram, and then one needs to indicate the relationships between them. The relationship between two classes is indicated by a line between them, annotated with an arrowhead or some other marking.

Inheritance relationship

When one class has an *is-a* relationship with another class, then the first class (called the subclass) is said to inherit from the second (called the superclass).

Inheritance is indicated with a large, hollow, triangle forming an arrowhead, touching the superclass.



This indicates that **Subclass** inherits from **Superclass**.

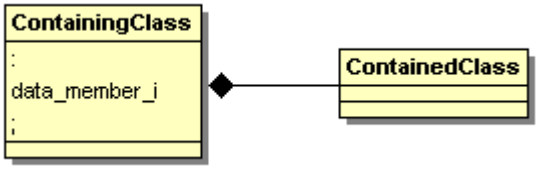
See Figure 6.1 for an example.

A superclass may have a number of subclasses (as in Figure 6.1) and a subclass may inherit from more than one superclass (as in Figure 11.4).

Composition relationship

When objects of one class contain objects of another class, and instance of the containing class is responsible for at least the destruction of instances of the contained class, this is a case of composition.

Composition is indicated by a solid (filled) diamond, touching the containing class.



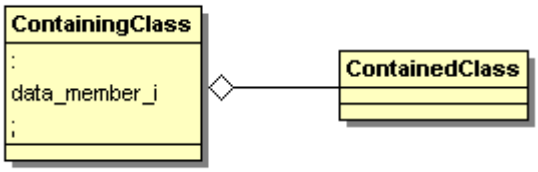
At least one data member of the containing class (for example data_member_i) should be of type **ContainedClass**, or be a pointer to **ContainedClass**.

See Figure 2.2 for an example.

Aggregation relationship

A weaker form of composition is called aggregation. Although the containing class typically has a pointer to an instance of the contained class as one of its data members, the destruction of the contained object is independent of the containing object. In other words, the contained object may be created before the object that contains it, and it might continue to exist after the containing object is destroyed.

Aggregation is indicated by a hollow diamond, touching the containing class.

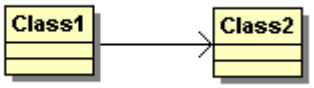


When the aggregation (or composition relationship) is anything other than one-to-one, for example one-to-many, then the line indicating the relationship is often annotated with numbers and/or the * symbol to indicate *multiplicity*. For example, if the relationship is one-to-many, the number 1 is written next to the containing class, and * is written next to the contained class. See Figure 4.1 for an example.

Association relationship

If one class has a direct or indirect reference (often in the form of a pointer) to another class, but this does not indicate a *has-a* relationship (as in composition and aggregation), then this is a case of association. Quite often this indicates a *uses-a* relationship.

A plain arrowhead is used to indicate which class uses which other class.



In this diagram, **Class1** uses **Class2**. The using class (**Class1**) may have one of its member functions take an instance or a pointer to the used class as a parameter, or one of its member functions may declare an instance of the used class.

See Figure 8.2 as an example.

If both classes have references to one another, then no arrowhead is drawn. As with composition and aggregation, the ends of the line may be annotated with numbers or * to indicate multiplicity if the association is anything other than one-to-one.

Notes on Design Patterns

Historically, object-orientation was developed as a technique of organising code to deal with complexity. It was so successful that the complexity goalposts were just moved. As developers were faced with new levels and types of complexity, they found that there were certain tricks which worked, i.e. similar code that kept cropping up in good solutions. Design patterns are an attempt to express that similar code in an abstract way.

According to the bible on this subject (namely *Design Patterns: Elements of Reusable Object-oriented Software*, by Gamma, Helm, Johnson and Vlissides):

> Design patterns ... describe ... elegant solutions to specific problems in object-oriented software. [They] capture solutions that have developed and evolved over time.

Very often, a design pattern specifies a team of classes, which when made to work together in a certain way, can successfully deal with a particular problem of complexity.

An important purpose of design patterns is for improving the reusability and extensibility of software. When you have to write a large application of high complexity, you don't want to have to redesign the wheel when you need to make some changes or adapt the program for some new use. You also don't want to have to scour through ten-of- thousands of lines of code to decide where these changes need to be made. In other words, you want to be able to reuse all the relevant parts in the simplest way possible.

Design patterns often consist of introducing classes that specify the places where changes need to be made. In other words, if you need to reuse a large program but make some changes for your own application, then all you need to do is to implement a class that derives from some base class, or make some changes to some class (for example implement a member function) in the hierarchy of classes provided.

Read Section 7.4 for some further explanations of design patterns.

Large frameworks or APIs (like the Qt framework) are an ideal application of design patterns since they are meant to provide reusable code that can be combined and extended in an infinite number of ways.

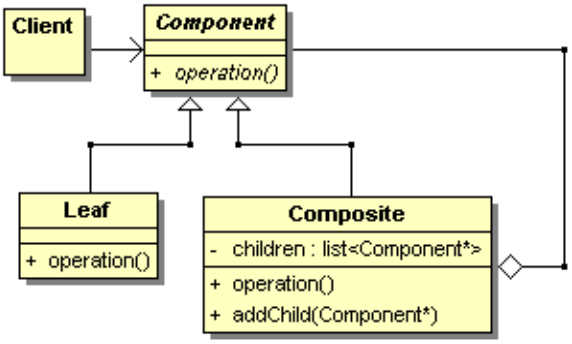
Extensibility is a measure of how easy it is to add enhancements to an application. One of the standard ways of making code extensible, is to provide classes which can easily be subclassed. The Qt framework is a good example of this, in that all its classes are designed so that developers can easily write classes that inherit from them, adding any additional functionality (or data) that is needed.

The chapters of Ezust prescribed for COS2614 cover a selection of design patterns at various points in the book. For COS2614, you need to know about the following design patterns and understand how they have been used (or applied) in various classes in the Qt framework.

Composite pattern

See Section 8.2.

When you require a tree of objects that you want to treat as one object, and call a member function on the root of the tree to get it performed on all the nodes or leaves of the tree, then the Composite pattern is probably a good option to apply.



To build a tree of objects, you need to start with an instance of the **Composite** class at the root of the tree. (You can put a **Leaf** at the root, but then you can't add any children!) Then you can add children to the root node using the addChild() member function – either **Leaf** objects (which can't have any further children) or **Composite** objects (which can have further children).

In the classic Composite pattern, the **Component** class is generally an abstract class which specifies an abstract member function, operation(). Both the **Leaf** and **Composite** classes inherit from **Component** and implement operation(). For the **Leaf** class, operation() performs whatever this type of object should do, e.g. draw itself. For the **Composite** class, operation() calls itself for each of its children. If a child is a Leaf, its version of operation() is called, but if a child is a **Composite**, then its version of operation() is called. In this way, by calling operation() on the root of the tree of objects, it is (recursively) called on all the leaf objects in the tree.

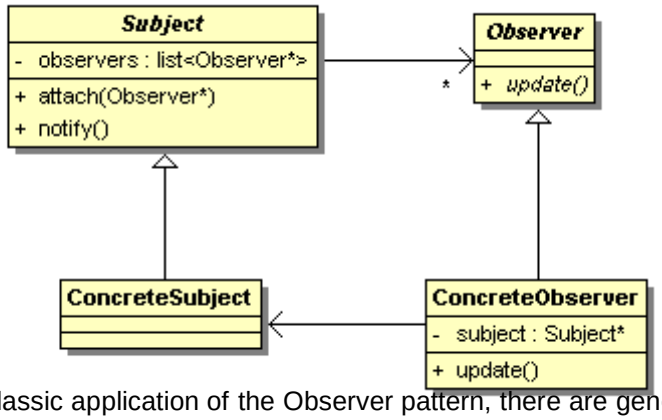
The `QObject` class represents the application of a simplified form of the Composite pattern. `QObject` represents the **Component** and **Composite** classes rolled into one. Every `QObject` maintains a list of its children. You can make one `QObject` the child of another by calling `setParent()`, which automatically adds it to the list of children of its parent. (So there are no member functions to add children, as in the classic Composite pattern.) See Figure 8.3.

Furthermore, `QObject` doesn't have a member function that corresponds to operation(), but it does implement a destructor that recursively destructs all its children. The lifetime of children is therefore dependent on the lifetime of their parent, and so we have a composition relationship between parent and children, not aggregation as in the classic Composite pattern. See Figure 8.3.

Observer pattern

See Section 8.3.

Say one object (called the subject) changes its state, and we want a number of other objects (called observers) to react to this change without the subject explicitly having to inform them all, then the Observer pattern is probably a good bet.



In the classic application of the Observer pattern, there are generally two abstract classes that play the role of Subject and **Observer**, and concrete implementations that inherit from them (**ConcreteSubject** and **ConcreteObserver**, respectively). Subject maintains a list of **Observer**s (which can be added to by an attach() member function). As

indicated in the above diagram, update() is an abstract member function of **Observer**, which has to be implemented by any **ConcreteObserver**. When any change is made to the state of a **ConcreteSubject**, notify() (inherited from **Subject**) is called, which in turn calls update() for each of its observers. **ConcreteObserver** often maintains a reference to its subject, so that it can find out what the new state of its subject is (i.e. when update() is executed).

The `QObject` class represents the application of a simplified form of the Observer pattern, in particular in its implementation of signals and slots. `QObject` represents the **Subject**, **Observer**, **ConcreteSubject** and **ConcreteObserver** classes all rolled into one!

`QObject` has a static member function `connect()` (similar to attach() in the classic Observer pattern) which is used to connect a subject to an observer. It also allows you to specify which change of state (the signal) should cause which update (the slot). Numerous observers can be connected to a single subject, and one observer can be connected to numerous subjects.

Monostate pattern

See Section 10.3.

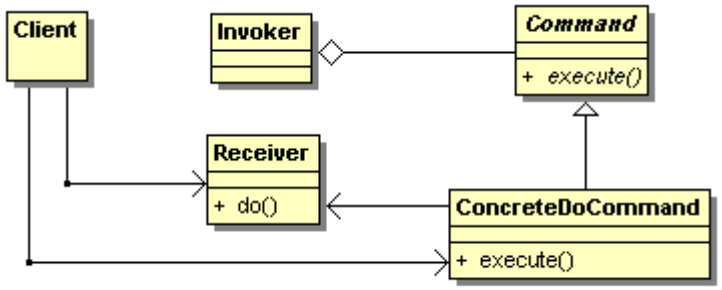
If various parts of a program (or various objects in a program) need to refer to the same object, but they don't know whether such an object has been constructed yet, then if the Monostate pattern has been applied to the class of that object, they can simply construct an instance and use it. The implementation will ensure that all instances of this class share the same state, so that they all appear to be the same object.

The simplest way of implementing the Monostate pattern is to make all the data members of the class, static. In this way, all instances of the class have the same state since they all share the same data.

Command pattern

See Section 10.5.

When you have a situation where a client (either a program or an object) needs to call the member functions of some other classes, but it is not sure whether such other classes are available, or when it would be appropriate to execute such member functions, then it is sometimes useful to apply the Command pattern. With this pattern, the calls to such member functions are represented by objects themselves, so that they can be executed at the appropriate time. Examples of such objects are events or actions.



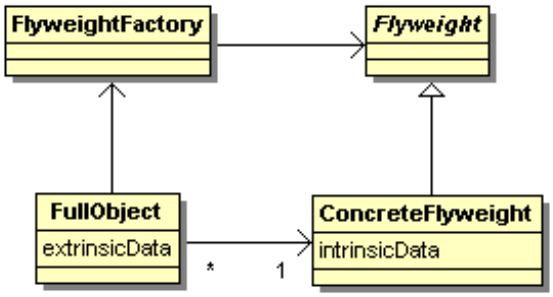
Say the **Client** wants to call a member function of the **Receiver** class. Instead of calling do(), it creates an instance of **ConcreteDoCommand**. (The **ConcreteDoCommand** class is an implementation of the abstract **Command** class, which has an abstract member function execute().) The execute() member function of **ConcreteDoCommand** simply calls do() of some **Receiver** itself. Note that do() hasn't been executed yet. The **Invoker** (which normally runs independently of or concurrently with **Client**) maintains a list (or stack or queue) of all **Command**s (i.e. **ConcreteCommands**) that are constructed, and decides itself when it is appropriate for these commands to be executed. For example, it may maintain a priority queue, and decide which command to execute according to some priority rating. It calls execute() for the chosen **ConcreteCommand**, which calls do() for the appropriate **Receiver**.

`QAction` is a somewhat different application of the Command pattern. All events (e.g. user actions like clicking on a button or a menu choice) are represented by objects of type `QAction`. (`QAction` therefore represents **ConcreteDoCommand** in the UML diagram above. The rest of the classes in the classic Command pattern do not have direct equivalents in Qt.)

Flyweight pattern

See Section 11.5.

The classic Flyweight pattern is useful when you need to save memory, especially when storing a large numbers of objects that are similar. A large number of similar objects have a lot of data in common, so just one copy of the common data (called intrinsic data) needs to be stored, and the data that differs (called extrinsic data) can be stored separately.



In this diagram, the **FullObject** class represents each of the similar objects. The **ConcreteFlyweight** class represents the objects that store the shared, intrinsic data. When an instance of **FullObject** is constructed, **FlyweightFactory** is invoked to return an instance of **ConcreteFlyweight** for storing the intrinsic data. **FlyweightFactory** checks whether an appropriate instance of **ConcreteFlyweight** already exists, and only constructs an instance if it doesn't.

The implementation of implicit sharing in Qt containers applies a simplified form of the classic Flyweight pattern described above. Although it represents a way of saving memory by sharing data that would otherwise be duplicated, there isn't any explicit data.

Furthermore, implicit sharing is really about delaying the duplication of shared data until their values need to be changed. (This is something not addressed by the classic Flyweight pattern.) In other words, when an implicitly shared container is copied, all its elements are shared. They are only copied when one of their values is changed. A trick is used to determine when this needs to be done, namely reference counting. See the comments on Section 11.5 above.