Design Patterns in Modern C++   
by Dmitri Nesteruk

Introduction

* Design patterns are common architectural approached to solving problems.
* They were popularised by four authors who were known as the Gang of Four.
* The patterns have translated to most other OOP languages and even non OOP languages such as Javascipt.
* Some patterns are so popular they became internalised into languages such as the observer pattern and C#’s *event* keyword.
* C++ hasn’t internalised any pattern, but libraries such as Boost do make use of them.
* Creational design patterns: builder, factories, prototype, singleton.
* Structural design patterns: adapter, bridge, composite, decorator, façade, flyweight, proxy.
* Behavioural design patterns: chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method, visitor.
* The instructor will simplify various aspects of the course to save time using non-recommended coding practices: uses public attributes, lack of virtual destructors, passing/returning by value, lack of move operations, etc.
* The course makes use of Boost libraries. To run them with CLion add the following code to the ‘CMakeLists.txt’ file:

set(BOOST\_ROOT "C:/Program Files/boost/boost\_1\_68\_0/")

find\_package(Boost 1.68.0)

if(NOT Boost\_FOUND)

message(FATAL\_ERROR "Boost library not found.")

endif()

SOLID Design Principles

* These design principles were introduced by Robert Martin. There are many other principles, but the S.O.L.I.D. design principles apply in many more contexts.
* S stands for **single responsibility principle**. It means that a class should exist for one specific reason and should not try to expand beyond responsibilities that pertain to that reason.
* An example of this is a *Journal* class. The class should contain methods which are strictly specific to journals, e.g. adding / editing / removing entries. However, implementing methods for saving the data to a file would be bad code because data persistence is another concept which should be represented by another class. This is because many classes in your program may need to make use of persistence, so you will end up having to refactor multiple classes to change the implementation. It’s better to have persistence code localised to one class so that modifications propagate to the other classes as required.
* O stands for **open/closed principle**. It means that classes should be open for extension/inheritance, but closed for modification. In other words, you should never have to go back and make changes to a class, instead, if the code is will designed, you will extend the pre-existing class to introduce new functionality.
* Implementing new methods in classes breaks binary compatibility which makes it difficult to use your library for dependant programs. The entire class will also need to be retested to ensure none of the previously tested code is now broken – this is costly. The OCP can be put into practice by using abstract base classes or interfaces.
* One example of this is an *Employee* class in which the type of employee is passed as an argument. This violates the OCP because every time you want to add a new type of employee you will have to modify various aspects of the class. It’s better to define a general base class that can be inherited from and specialised for each type of employee instead.
* Another example is a *Filter* class which filters a *vector* of *Product*s for products that match specific criteria. One solution is to write a monolithic *Filter* class that has every possible filter implemented. However, requirements can change, or a new requirement may come in that require you to add a new filter. You will be forced to modify the *Filter* class which violated the OCP. In this case you can make use of the specification enterprise design pattern to create new classes every time a new requirement comes in. You can then write combinators that combine multiple filters.
* L stands for **Liskov substitution principle**. It means that derived types should be substitutable for their base types.
* An example of code that would break this is the relationship between a rectangle and a square. Mathematically a square is a special case of a rectangle so you would think to inherit *Rectangle* from *Square*. However, this will lead to you overriding the *setWidth()*, and *setHeight()* methods for *Square* such that the *width* and *height* attributes are equal whenever one attribute is changed. This can lead to unexpected output when passing the *Square* to a function which expects a *Rectangle*. As such doing so violates the LSP.

Builder

* >

Factories

* >

Prototype

* >

Singleton

* >

Adapter

* >

Bridge

* >

Composite

* >

Decorator

* >

Facade

* >

Flyweight

* >

Proxy

* >

Chain of Responsibility

* >

Command

* >

Interpreter

* >

Mediator

* >

Memento

* >

Observer

* >

State

* >

Strategy

* >

Template Method

* >

Visitor

* >

Course Summary

* >