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. These patterns encapsulate the object creation process. Thus, object creation logic only has to be modified in one place within the codebase rather than across arbitrary files and classes as this is prone to errors.
* 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.
* It is common to append the name of the pattern at the end of the class name, e.g. *ProductFactory, PersonBuilder, SquareToRectangleAdapter,* etc.
* 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.
* I stands for **interface segregation principle**. It means that interfaces should be concise and contain only methods that are absolutely required. If you don’t follow this principle then implementors will be forced to implement too much code which may be useless for them. The less methods an interface declares, the more general it becomes.
* An example of this is an interface which describes a transaction, it should only contain *deposit()* and *withdraw()* methods as they are directly related. It shouldn’t contain methods to set/get a balance.
* Another example is an interface which describes the capabilities of a printer. It could contain methods for printing, faxing, and scanning, which would be fine for a class which represents a printer that can do all three, but then you may have a *Scanner* class which can only scan, this would make the other methods useless and violate the ISP. It’s better to create a separate interface for all three methods so that a class can pick and choose what it needs.
* D stands for **dependency inversion principle**. It means that high-level modules shouldn’t depend on low-level modules, instead both should depend on abstractions. Also, abstractions should not depend on details, details should depend on abstractions.
* Abstractions refers to interfaces and abstract classes.
* This enables **dependency injection**, as now any custom class with custom overridden methods can be derived from the interface, instantiated, and then be assigned as a data member to another object. DI allows for loose coupling between classes and makes them more extendable. You must use types that enable polymorphism for this to work, i.e. a data members can’t be a value types, they must be a references, or raw/smart pointers.

Builder

* The builder design pattern constructs an object step-by-step.
* Some objects are simple to construct, e.g. *string str(“Hello”)*, other objects may require many arguments. This is makes it difficult to instantiate the class. It’s better to opt for piecewise construction.
* Example: A *HtmlElement* class represents a tag and it’s children tags, rather than using *HtmlElement* directly you can use *HtmlBuilder* to work with *HtmlElement* to build up *HtmlElement* step-by-step.
* A **fluent interface** makes use of method chaining among other techniques to reduce unnecessarily having to use an identifier to access its methods, e.g. *cout << “Hello, “ << world << endl*. Other techniques include *static* methods that return the target type, and type conversion operator overloads.
* Method chaining can be done using references or pointers, but references are generally easier to use.
* Multiple builder classes sharing the same base class can be used to construct a base object via fluent interface.
* Example a *Person* class may contain attributes related to address, work, health records, etc. A builder class can be created for each of these facets/areas to build up a *Person* object using a fluent interface. All of them can inherit from a *PersonBuilderBase* class that contains the reference for the *Person* object to be modified.

Factories

* The factory design pattern selects which derived class to construct based on a method call or supplied arguments.
* Patterns can be combined as needed so a factory pattern can make use of a builder pattern to construct a complex object.
* There are multiple variants, you can have a factory method, a factory class, or an abstract factory set of classes (hierarchy). A factory method is a standalone method that instantiates a class, while a factory class can be used to group a set of *static* factory methods. This makes it easier for the client to find the appropriate method to construct an object with. Abstract factory hierarchies support the OCP whereas factory methods are tightly coupled with various subclasses thus violating OCP.
* Example: A *Computer* class can be difficult to instantiate as there will be many attributes to initialise. You can implement many factory methods that return a specific type of *Computer*. One factory method may return a basic computer with a processor, but no GPU. Another factory method may return a high-end computer with high-spec components.
* Example: A *Point* class may be initialised using cartesian or polar coordinates. In either case the constructor will require two floating types to represent the coordinates: x and y for cartesian, rho and theta for polar. This will be seen as an ambiguous call by the compiler. Factory methods can be used to make it clear how the constructor is going to be used, e.g. *from\_cartesian(double x, double y)*, *from\_polar(double rho, double theta)*.
* Example: You may have many subclass of a *Widget*, but some of them may be used for Linux while others are used for Windows. You can define an abstract factory class called *WidgetFactory* that defines pure virtual methods that return *Button, ComboBox, etc*. You can then inherit from *WidgetFactory* and create classes called *LinuxWidgetFactory* and *WindowsWidgetFactory* that override the pure virtual methods and return the appropriate *Widget*.

Prototype

* The prototype design pattern makes a copy of a partially or fully initialised object.
* This is useful in cases where an object with certain initial attributes is often created, rather than recreating this object every time it’s needed it’s better to create a copy every time to localise creation logic. This pattern is often combined with the factory pattern. The factory pattern just returns the correct derived class, while the prototype pattern creates a copy of an object with the closest initial state to what the client requires. Often the client will use the prototype pattern to create the closest copy, and then change it to match the requirement.
* An example of this is a *Contact* class in which multiple contacts have the same *Address*. *Address* is another complex class that contains various details such as door number, building, street, post code, city, country, etc. One option is to keep recreating *Contacts* and supplying the same information, another option is to create a *ContactFactory* with a *static* method called *new\_trainee()* that returns a copy of a *Contact* that already has the correct *Address* information. Copies are often returned by creating them on the heap via smart pointers to allow polymorphism.
* Another use it to create a copy of an object without knowing its type. If the type is accessed through a base type then just used the copy constructor won’t be enough since it’ll slice the object, a clone method would dynamically return the correct object. To set this up create an abstract base class called *Prototype* with a pure virtual method called *clone()* that returns a *unique\_ptr<Prototype>*. Any subclass can now override this method to dynamically return a copy of itself, e.g. *return unique\_ptr<Derived>(\*this)*. We need to use pointers rather than value types as otherwise the object is sliced. *unique\_ptr* is used because it can be converted to *shared\_ptr* as required, not vice versa. This is also referred to as the clone pattern.

Singleton

* The singleton design pattern ensures that only a single instance of a class exists.
* It’s used when only one instance of a class makes sense, i.e. you may not need a class to load the same data from memory more than once in a program. A constructor call may be expensive so it might be worth to do it only once to load the data in memory, then to get the data from the same instance every time.
* This pattern is controversial and is often considered an anti-pattern. The reason is that it makes it difficult to complete unit tests. However, with dependency injection this problem can be alleviated.
* An alternative to the singleton design pattern is a monostate design pattern in which all of the attributes are marked *static*. Clients can instantiate multiple copies of the class, but the objects will all access the same attributes since they’re only created once/shared. This design pattern isn’t recommended since it’s quite inflexible. It breaks encapsulation as the attributes can be modified by any subclass instance and the changes will be seen by all subclass instances. However, the singleton pattern can be inherited from and each subclass instance maintains its own state.
* To use the pattern: define private constructors only, delete copy constructor and copy assignment operator, define a public static getter method that creates only one instance of the class and returns it.
* Any class that uses the singleton must support dependency injection to so that dummy class instances can be injected to test the code independently from the actual class.

Adapter

* The adapter design pattern adapts an existing interface X to conform to a required interface Y. As such it is also known as a wrapper.
* The focus isn’t to add or remove functionality, instead one interface’s specification is converted to another interface’s specification. As such there must be some relation between the interfaces being adapted. It doesn’t make sense to adapt a random pair of interfaces.
* Works very similar to a power adapter as the output connection is adapted to the input connection.
* Example: A square and rectangle are very similar mathematically, but it’s not recommended to use inheritance since it violates LSP. A square can instead be converted into a rectangle using an adapter.
* To use the pattern: write an adapter class that inherits from the target interface, implement a constructor that stores the source object as an attribute (dependency injection), override interface methods to work with source object’s methods.

Bridge

* The bridge design pattern decouples an interface (hierarchy) from an implementation (hierarchy).
* The pimpl idiom (pointer implementation) allows implementation to be hidden in the source file rather than exposing it to the client through the header file. You can also modify the class used in the pimpl idiom without introducing binary incompatibilities.
* To use the pimpl idiom: create an attribute that’s a pointer to the pimpl class within the main class. Use forward declaration to make the identifier known the compiler. Define the

Composite

* The composite design pattern allows treating individual objects and compositions of objects in a uniform manner.
* Example: You have a *Graphic* class that declares a *draw()* method. It is derived by a *Circle*, *Square, Line*, etc classes. You have a *Render* class that draws *Graphic*s to screen. It accepts only one *Graphic* argument. As such you can only draw one shape at a time. To draw multiple, you can use the composite design pattern. Derive a class called *GraphicsGroup* that has a *vector* of *Graphic*s which can be added to by calling a *add\_graphic()* method. Override the *draw()* method, to call each *Graphic*’s *draw()* method. You can now pass an instance of *GraphicsGroup* into the *Render* class to draw multiple objects at once.
* To use the pattern: inherit from the source class, use a container data member to store multiple source objects, override methods from the source class and call the same method for each object in the container.

|  |  |
| --- | --- |
| class Graphic {  public:  virtual ~Graphic() = default;  virtual void draw() const = 0;  // Other virtual methods.  };  class Circle : public Graphic {  public:  void draw() const override {  // draw a circle.  }  };  class Square : public Graphic {  public:  void draw() const override {  // draw a square.  }  }; | class GraphicsComposite : public Graphic {  vector<shared\_ptr<Graphic>> graphics;  public:  void add\_graphic(shared\_ptr<Graphic> &graphic) {  graphics.push\_back(graphic);  }  void draw() const override {  for(const auto &graphic : graphics) {  graphic->draw();  }  }  // Override other virtual methods.  };  /\* This method will draw a graphic whether it’s a circle, square, or a group/composite of graphics.  \*/  void renderer(shared\_ptr<Graphic> graphic) {  // …  graphic->draw();  // …  } |

* A single object can act as a container and be ‘iterated’ through by implementing *begin()* and *end()*.

|  |  |
| --- | --- |
| MyClass\* begin() {  return this;  } | MyObject\* end() {  return this+1;  } |

* When the code in subclasses is exactly the same it can be refactored into a common base class and removed from the subclasses. When the code in subclasses is mostly the same, the common elements can be refactored into a common base class, then overridden in the subclasses to implement the part of the code that’s unique to the subclass. When the code in subclasses is mostly the same and the only thing that’s different is that it creates instances of the current class, then you can use CRTP to refactor the code into a base class.
* CRTP, or curiously recurring template pattern, allows you to refactor code into a base class even if it depends on type information / attributes / methods from a base class. This can be achieved through a static cast meaning that the compiler binds the call at compile time. As such another name for this is static polymorphism.
* To use it, create a subclass that inherits from a template base class, pass in the derived class as a template argument, anytime information about the derived class is needed in the base class, use a *static­\_cast* on *T*.

|  |  |
| --- | --- |
| template <typename Self>  class Base {    } | class Derived : public Base<Derived> {    } |

* CRTP can be used to implement cloning without having to override a *clone()* method in every subclass.

|  |
| --- |
| class Base {  public:  virtual ~Base() = default;  virtual unique\_ptr<Base> clone() = 0;  // Other virtual methods.  };  template <typename Self>  class BaseCRTP : public Base {  public:  unique\_ptr<Base> clone() override {  return make\_unique<Self>(\*static\_cast<Self\*>(this));  }  };  class Derived : public BaseCRTP<Derived> {  public:  }; |

* Array backed properties can be used to simplify calculations that require multiple/all properties. To use them: store all properties in an array and use an *enum* to keep track of the index of that property.

|  |
| --- |
| class Creature {  enum Stats {\_strength, \_intelligence, \_agility, \_count};  Array<int, \_count> stats;  public:  // Getters and setters.  int get\_strenth() const { return stats[\_strength]; }  void set\_strength(int strength) { stats[\_strength] = strength; }  // Define all setters and getters, and then aggregate methods.  int sum() const { return std::accumulate(stats.begin(), stats.end(), 0); }  // etc…  }; |

Decorator

* The decorate design pattern facilities the addition of behaviours to individual objects.
* Dynamic decorators allow adding behaviour to pre-existing classes without having to inherit from each subclass. The limitation is that you lose behaviour specific to those classes as only the methods defined in a common interface can be accessed.
* To use this pattern: inherit from the base class, use dependency injection to store a reference to a base class object, override relevant methods and implement new methods as necessary.

|  |  |
| --- | --- |
| // Assume these classes already exist.  class Shape {  public:  virtual ~Shape() = default;  virtual void draw() const = 0;  }  class Circle : public Shape {  double radius{};  public:  Circle(double rad) : radius{rad} {}  void draw() override {  cout << “Drawing circle with radius “ << radius;  }  void resize(double factor) {  radius \*= factor;  }  }  class Square : public Shape {  double side;  public:  Square(double side) : side{side} {}  void draw() override {  cout << “Drawing square with side “ << side;  }  }  // … other Shape subclasses. | // Assume you want to add colour functionality without violating OCP, or SRP. You can either subclass each shape subclass, or use dynamic polymorphism.  class ColoredShape : public Shape {  Shape &shape;  string color;  public:  ColoredShape(Shape &s, string c) shape{s}, color{c} {}  void draw() override {  shape.draw();  cout << “, color “ << color;  }  }  // This will work for any Shape subclass – including other dynamic decorators. However, you lose access to specific methods, e.g. Circle::resize().  int main() {  Circle cir{5.0};  ColoredShape red\_cir(cir, “red”);  red\_cir.draw(); // Can’t resize.  return 0;  } |

* Mixin inheritance is when a class inherits from a template type argument.
* Static decorators use mixin inheritance and thus can add behaviour to pre-existing classes via inheritance. However, instead of you having to manually subclass the source class, the compiler does it for you as required. This means you can access class specific methods. The limitation is that you lose dynamic behaviour as binding occurs at compile time.

|  |  |
| --- | --- |
| template <typename T>  class ColoredShape : public T {  string color;  public:  template <typename...Args>  ColoredShape(string c, Args...args)  : T{forward<Args>(args)...}, color{c} {}  void draw() override {  T::draw();  cout << “, color “ << color;  }  } | int main() {  ColoredShape<Circle> red\_cir{“red”, 5.0};  red\_cir.resize(2);  red\_cir.draw();  return 0;  }  // This will also work for any Shape subclass – including other static decorators. You have access to class specific methods, but you lose dynamic binding. |

* Static decorators can be combined with other static decorators, e.g. *TransparentShape<ColoredShape <Circle>> {…}*. Dynamic decorators can be combined with other dynamic decorators, e.g. *TransparentShape ts{colored\_shape, 51}*.

Facade

* The façade design pattern provides a simple user interface over a complex body of code.
* Example: A complex banking system that makes use of 100 classes would be daunting to work with as it could be difficult to work out how the classes are designed to interact and how they should be used. A single façade class can be created to simplify the usage as the client can call methods in the façade to do what they want. This way the user won’t have to worry about which classes to use or about any of the inner workings.

Flyweight

* The flyweight design pattern is a space optimisation technique that saves memory by storing data externally for use by similar objects.
* Example: An MMORPG will have many players that have the same first name, and/or last name. Rather than storing duplicates in memory, it’s more efficient to store unique names in memory only, and then to refer to those names. This concept can be generalised to any data type.
* To use the pattern: you can use static data members, singleton classes, or a shared object that uniquely stores data of a certain type. Anytime that type is instantiated, you must check to see if the data already exists and store its index/iterator instead of storing the actual data in the client.

Proxy

* >

Chain of Responsibility

* >

Command

* >

Interpreter

* >

Mediator

* >

Memento

* >

Observer

* >

State

* >

Strategy

* >

Template Method

* >

Visitor

* >

Course Summary

* >

Extracurricular

* Google Test can be used to perform unit tests on C++ programs.
* Add it to your project’s CMake by following its README:

<https://github.com/google/googletest/tree/master/googletest>

* *#include <gtest/gtest.h>* in your *main.cpp*.
* Initialise it by calling the following methods in the *main* method:

testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

* Define a unit test with the following signature:

TEST(NameOfTestGroup, NameOfTest) {

…

EXPECT\_EQ(expected\_value, actual\_value);

}

* Edit Configuration > Add New Configuration > Google Test > OK.
* Run configuration.
* Unit testing has narrow scope and tests independent sections/units of code.
* Integration testing has wider scope and tests how different components of a system work together.
* Both of these types of tests fall under white box testing as they require the tester to understand the structure/design/implementation of the code being tested. As such the responsibility for these tests falls under software developers.
* Black box testing techniques include acceptance testing and system testing as the code doesn’t need to be understood to test it. The responsibility for these tests falls under dedicated software testers.