Design Patterns in Modern C++

Dmitri Nesteruk

Design Patterns in Modern C++

Dmitri Nesteruk

This book is for sale at http://leanpub.com/design-patterns-modern-cpp

This version was published on 2018-06-10

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. Introduction

The topic of Design Patterns sounds dry, academically constipated and, in all honesty, done to death in almost every programming language imaginable – including programming languages such as JavaScript which aren't even properly OOP! So why another book on it? I know that if you're reading this in a book store (ha-ha, real-world book stores, seriously, this is an e-book!), you probably have a limited amount of time to decide whether this is worth the investment.

I guess the main reason this book exists is that C++ is great again. After a long period of stagnation, it's now evolving, growing, and despite the fact that it has to contend with backwards C compatibility, good things are happening, albeit not at the pace we'd all like (I'm looking at you, C++17).

Now, on to Design Patterns – we shouldn't forget that the *original* Design Patterns book[1] was published with examples in C++ and Smalltalk. Since then, plenty of programming languages have incorporated design patterns directly into the language: for example, C# directly incorporated the Observer pattern with its built-in support for events (and the corresponding `event` keyword). C++ has *not* done the same, at least not on the syntax level. That said, the introduction of types such as `std::function` sure made things a lot simpler for many programming scenarios.

Design Patterns are also a fun investigation of how a problem can be solved in many different ways, with varying degrees of technical sophistication and different sorts of trade-offs. Some patterns are more or less essential and unavoidable, whereas other patterns are more of a scientific curiosity (but nevertheless will be discussed in this book, since I'm a completionist).

Readers should be aware that comprehensive solutions to certain problems (e.g., the Observer pattern) typically result in overengineering, i.e., the creation of structures that are far more complicated than is necessary for most typical scenarios. While overengineering is a lot of fun (hey, you get to *really* solve the problem and impress your co-workers), it's often not feasible.

[1] Erich Gamma et al. (1994), *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison Wesley

Preliminaries

Who This Book Is For

This book is designed to be a modern-day update to the classic GoF book, targeting specifically the C++ programming language. I mean, how many of you are writing Smalltalk out there? Not many, that would be my guess.

The goal of this book is to investigate how we can apply Modern C++ (the latest versions of C++ currently available) to the implementations of classic design patterns. At the same time, it's also an attempt to flesh out any new patterns and approaches that could be useful to C++ developers.

Finally, in some places, this book is quite simply a technology demo for Modern C++, showcasing how some of its latest features (e.g., coroutines) make difficult problems a lot easier to solve.

On Code Examples

The examples in this book are all suitable for putting into production, but a few simplifications have been made in order to aid readability:

- Quite often, you'll find me using `struct` instead of `class` in order to avoid writing the `public` keyword in too many places.
- I will avoid the `std::` prefix, as it can hurt readability, especially in places where code density is high. If I'm using `string`, you can bet I'm referring to `std::string`.
- I will avoid adding `virtual` destructors whereas, in real life, it might make sense to add them.
- In very few cases I create and pass parameters by value to avoid the proliferation of `shared_ptr`/`make_shared`/etc. Smart pointers add another level of complexity, and their integration into the design patterns presented in this book is left as an exercise for the reader.
- I will sometimes omit code elements that would otherwise be necessary for feature-completing a type (e.g., move constructors) as those take up too much space.

- There will be plenty of cases where I will omit `const` whereas, under normal circumstances, it would actually make sense. Const-correctness quite often causes a split and a doubling of the API surface, something that doesn't work well in book format.

You should be aware that most of the examples leverage Modern C++ (C++11, 14, 17 and beyond) and generally use the latest C++ language features that are available to developers. For example, you won't find many function signatures ending in `-> decltype(...)` when C++14 lets us automatically infer the return type. None of the examples target a particular compiler, but if something doesn't work with your chosen compiler[2], you'll need to find workarounds.

At certain points in time, I will be referencing other programming languages such as C# or Kotlin. It's sometimes interesting to note how designers of other languages have implemented a particular feature. C++ is no stranger to borrowing generally available ideas from other languages: for example, the introduction of `auto` and type inference on variable declarations and return types is present in many other languages.

On Developer Tools

The code samples in this book were written to work with modern C++ compilers, be it Clang, GCC or MSVC. I make the general assumption that you are using the latest compiler version that is available, and as a consequence, will use the latest-and-greatest language features that are available to me. In some cases, the advanced language examples will need to be downgraded for earlier compilers, in others it might not work out.

As far as developer tools are concerned, this book does not touch on them specifically, so provided you have an up-to-date compiler, you should follow the examples just fine: most of them are self-contained `.cpp` files. Regardless, I'd like to take this opportunity to remind you that quality developer tools such as the CLion or ReSharper C++ greatly improve the development experience. For a tiny amount of money that you invest, you get a wealth of additional functionality that directly translates to improvements in coding speed and the quality of the code produced.

[2] Intel, I'm looking at you!

Piracy

Digital piracy is an inescapeable fact of life. A brand new generation is growing up right now that has never purchased a movie or a book. Even this book. There's not much that can be done about this. The only thing I can say is that, if you pirated this book, you might not be reading the latest version.

The joy of online digital publishing is I get to update the book as new versions of C++ come out and I do more research. So if you paid for this book, you'll get free updates in the future as new versions of the C++ language and the Standard Library are released. If not... oh, well.

Important Concepts

Before we begin, I wanted to briefly mention some key concepts of the C++ world that are going to be referenced in this book.

Curiously Recurring Template Pattern

Hey, this is a pattern, apparently! I don't know if it qualifies to be listed as a separate *design* pattern, but it's certainly a pattern of sorts in the C++ world. Essentially, the idea is simple: an inheritor passes *itself* as a template argument to its base class.

```
1  struct Foo : SomeBase<Foo>
2  {
3    ...
4  }
```

Now, you might be wondering *why* one would ever do that? Well, one reason is to be able to access a typed `this` pointer inside a base class implementation.

For example, suppose every single inheritor of `SomeBase` implements a `begin()/end()` pair required for iteration. How can you iterate the object inside a member of `SomeBase`? Intuition suggests that you cannot, because `SomeBase` itself does not provide a `begin()/end()` interface. But if you use CRTP, you can actually cast `this` to a derived class type:

```
template <typename Derived>
struct SomeBase
{
  void foo()
  {
    for (auto& item : *static_cast<Derived*>(this))
    {
      ...
    }
  }
}
```

For a concrete example of this approach, check out the Composite chapter.

Mixin Inheritance

In C++, a class can be defined to inherit from its own template argument, i.e.:

```
template <typename T> struct Mixin : T
{
  ...
}
```

This approach is called *mixin inheritance* and allows hierarchical composition of types. For example, you can allow `Foo<Bar<Baz>> x;` to declare a variable of a type that implements the traits of all three classes, without having to actually construct a brand new `FooBarBaz` type.

For a concrete example of this approach, check out the Static Decorator chapter.

Static Polymorphism

The term *static polymorphism* hints at the idea of some sort of inheritance of `static` functions, something not possible in conventional C++. Why would anyone want this anyway?

Suppose you are using CRTP to specialize a base class with policies that either allow or disallow undo operations. Given the base class

```
template <class Derived> class Base
{
public:
  static bool IsUndoSupported()
  {
    return Derived::isUndoEnabled();
  }
}
```

a derived type that leveraged `Base` functionality but disallowed undo operations could subsequently be defined as follows:

```
struct DerivedNoUndo : Base<DerivedNoUndo>
{
  friend class Base<DerivedNoUndo>;
private:
  static bool isUndoEnabled()
  {
    return false;
  }
}
```

These equilibristics allow us to perform polymorphic static function calls, e.g.:

```
if (Base<DerivedNoUndo>::IsUndoSupported())
{
  // perform undo operation
}
```

Properties

A *property* is nothing more than a (typically private) field and a combination of a getter and a setter. In standard C++, a property looks as follows:

```
class Person
{
  int age;
public:
  int get_age() const { return age; }
  void set_age(int value) { age = value; }
};
```

Plenty of languages (e.g., C#, Kotlin) internalize the notion of a property by baking it directly into the programming language. While C++ has not done this (and is unlikely to do so anytime in the future), there is a nonstandard declaration specifier called `property` that you can use in most compilers (MSVC, Clang, Intel):

```
class Person
{
  int age_;
public:
  int get_age() const { return age_; }
  void set_age(int value) { age_ = value; }
  __declspec(property(get=get_age, put=set_age)) int age;
};
```

This can be used as follows:

```
Person p;
p.age = 20; // calls p.set_age(20)
```

The SOLID Design Principles

SOLID is an acronym which stands for the following design principles (and their abbreviations):

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)

- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

These principles were introduced by Robert C. Martin in the early 2000s – in fact, they are just a selection of 5 principles out of dozens that are expressed in Robert's books and his blog. These five particular topics permeate the discussion of patterns and software design in general, so before we dive into design patterns (I know you're all eager), we're going to do a brief recap of what the SOLID principles are all about.

Single Responsibility Principle

Suppose you decide to keep a journal of your most intimate thoughts. The journal has a title and a number of entries. You could model it as follows:

```
struct Journal
{
  string title;
  vector<string> entries;

  explicit Journal(const string& title) : title{title} {}
};
```

Now, you could add functionality for adding an entry to the journal, prefixed by the entry's ordinal number in the journal. This is easy:

```
void Journal::add(const string& entry)
{
  static int count = 1;
  entries.push_back(boost::lexical_cast<string>(count++)
    + ": " + entry);
}
```

And the journal is now usable as:

```
Journal j{"Dear Diary"};
j.add("I cried today");
j.add("I ate a bug");
```

It makes sense to have this function as part of the `Journal` class because adding a journal entry is something the journal actually needs to do. It is the journal's responsibility to keep entries, so anything related to that is fair game.

Now suppose you decide to make the journal persist by saving it in a file. You add this code to the `Journal` class:

```
void Journal::save(const string& filename)
{
  ofstream ofs(filename);
  for (auto& s : entries)
    ofs << s << endl;
}
```

This approach is problematic. The journal's responsibility is to *keep* journal entries, not to write them to disk. If you add the disk-writing functionality to `Journal` and similar classes, any change in the approach to persistence (say, you decide to write to the cloud instead of disk) would require lots of tiny changes in each of the affected classes.

I want to pause here and make a point: an architecture that leads you to having to do lots of tiny changes in lots of classes, whether related (as in a hierarchy) or not, is typically a *code smell* – an indication that something's not quite right. Now, it really depends on the situation: if you're renaming a symbol that's being used in a hundred places, I'd argue that's generally OK because ReSharper, CLion or whatever IDE you use will actually let you perform a refactoring and have the change propagate everywhere. But when you need to completely rework an interface... well, that can be a very painful process!

We therefore state that persistence is a separate concern, one that is better expressed in a separate class, for example:

```
struct PersistenceManager
{
  static void save(const Journal& j, const string& filename)
  {
    ofstream ofs(filename);
    for (auto& s : j.entries)
      ofs << s << endl;
  }
};
```

And this is precisely what we mean by *Single Responsibility*: each class has only one responsibility, and therefore has only one reason to change. `Journal` would need to change only if there's something more that needs to be done with respect to storage of entries – for example, you might want each entry prefixed by a timestamp, so you would change the `add()` function to do exactly that. On the other hand, if you wanted to change the persistence mechanic, this would be changed in `PersistenceManager`.

An extreme example of an anti-pattern which violates the SRP is called a *God Object*. A God Object is a huge class that tries to handle as many concerns as possible, becoming a monolithic monstrosity that is very difficult to work with.

Luckily for us, God Objects are easy to recognize and thanks to source control systems (just count the number of member functions), and the responsible developer can be quickly identified and adequately punished.

Open-Closed Principle

Suppose we have an (entirely hypothetical) range of products in a database. Each product has a color and size and is defined as:

```
enum class Color { Red, Green, Blue };
enum class Size { Small, Medium, Large };

struct Product
{
  string name;
  Color color;
  Size size;
};
```

Now, we want to provide certain filtering capabilities for a given set of products. We make a filter similar to the following:

```
struct ProductFilter
{
  typedef vector<Product*> Items;
};
```

Now, to support filtering products by color, we define a member function to do exactly that:

```
ProductFilter::Items ProductFilter::by_color(Items items, Color color)
{
  Items result;
  for (auto& i : items)
    if (i->color == color)
      result.push_back(i);
  return result;
}
```

Our current approach of filtering items by color is all well and good. Our code goes into production but, unfortunately, some time later, the boss comes in and asks us to implement filtering by size, too. So we jump back into `ProductFilter.cpp`, add the code below and recompile:

```
ProductFilter::Items ProductFilter::by_size(Items items, Size size)
{
  Items result;
  for (auto& i : items)
    if (i->size == size)
      result.push_back(i);
  return result;
}
```

This feels like outright duplication, doesn't it? Why don't we just write a general method that takes a predicate (some `function`)? Well, one reason could be that different forms of filtering can be done in different ways: for example, some record types might be indexed and need to be searched in a specific way; some data types are amenable to search on a GPU while others are not.

Our code goes into production but, once again, the boss comes back and tells us that now there's a need to search by both color *and* size. So what are we to do but add another function?

```
ProductFilter::Items ProductFilter::by_color_and_size(Items items,
  Size size, Color color)
{
  Items result;
  for (auto& i : items)
    if (i->size == size && i->color == color)
      result.push_back(i);
  return result;
}
```

What we want, from the above scenario, is to enfoce the *Open-Closed Principle* that states that a type is open for extension but closed for modification. In other words, we want filtering that is extensible (perhaps in a different compilation unit) without having to modify it (and recompiling something that already works and may have been shipped to clients).

How can we achieve it? Well, first of all, we conceptually separate (SRP!) our filtering process into two parts: a filter (a process which takes all items and only returns some) and a specification (the definition of a predicate to apply to a data element).

We can make a very simple definition of a specification interface:

```
template <typename T> struct Specification
{
  virtual bool is_satisfied(T* item) = 0;
};
```

In the above, type `T` is whatever we choose it to be: it can certainly be a `Product`, but it can also be something else. This makes the entire approach reusable.

Next up, we need a way of filtering based on `Specification<T>`: this is done by defining, you guessed it, a `Filter<T>`:

```
template <typename T> struct Filter
{
  virtual vector<T*> filter(
    vector<T*> items,
    Specification<T>& spec) = 0;
};
```

Again, all we are doing is specifying the signature for a function called `filter` which takes all the items and a specification, and returns all items that conform to the specification. There is an assumption that the items are stored as a `vector<T*>`, but in reality you could pass `filter()` either a pair of iterators or some custom-made interface designed specifically for going through a collection. Regrettably, the C++ language has failed to standardise the notion of an enumeration or collection, something that exists in other programming languages (e.g., .NET's `IEnumerable`).

Based on the above, the implementation of an improved filter is really simple:

```
struct BetterFilter : Filter<Product>
{
  vector<Product*> filter(
    vector<Product*> items,
    Specification<Product>& spec) override
  {
    vector<Product*> result;
    for (auto& p : items)
      if (spec.is_satisfied(p))
        result.push_back(p);
    return result;
  }
};
```

Again, you can think of a `Specification<T>` that's being passed in as a strongly-typed equivalent of an `std::function` that is constrained only to a certain number of possible filter specifications.

Now, here's the easy part. To make a color filter, you make a `ColorSpecification`:

```
struct ColorSpecification : Specification<Product>
{
  Color color;

  explicit ColorSpecification(const Color color) : color{color} {}

  bool is_satisfied(Product* item) override {
    return item->color == color;
  }
};
```

Armed with this specification, and given a list of products, we can now filter them as follows:

```
  Product apple{ "Apple", Color::Green, Size::Small };
  Product tree{ "Tree", Color::Green, Size::Large };
  Product house{ "House", Color::Blue, Size::Large };

  vector<Product*> all{ &apple, &tree, &house };

  BetterFilter bf;
  ColorSpecification green(Color::Green);

  auto green_things = bf.filter(all, green);
  for (auto& x : green_things)
    cout << x->name << " is green" << endl;
```

The above gets us "Apple" and "Tree" because they are both green. Now, the only thing we haven't implemented so far is searching for size *and* color (or, indeed, explained how you would search for size *or* color, or mix different criteria). The answer is that you simply make a *composite* specification. For example, for the logical AND, you can make it as follows:

```
template <typename T> struct AndSpecification : Specification<T>
{
  Specification<T>& first;
  Specification<T>& second;

  AndSpecification(Specification<T>& first, Specification<T>& second)
    : first{first}, second{second} {}

  bool is_satisfied(T* item) override
  {
    return first.is_satisfied(item) && second.is_satisfied(item);
  }
};
```

And now, you are free to create composite conditions on the basis of simpler `Specifications`. Reusing the `green` specification we made earier, finding something green and big is now as simple as:

```
SizeSpecification large(Size::Large);
ColorSpecification green(Color::Green);
AndSpecification<Product> green_and_large{ large, green };

auto big_green_things = bf.filter(all, green_and_big);
for (auto& x : big_green_things)
  cout << x->name << " is large and green" << endl;

// Tree is large and green
```

This was a lot of code! But keep in mind that, thanks to the power of C++, you can simply introduce an `operator &&` for two `Specification<T>` objects, thereby making the process of filtering by two (or more!) criteria extremely simple:

```
template <typename T> struct Specification
{
  virtual bool is_satisfied(T* item) = 0;

  AndSpecification<T> operator &&(Specification&& other)
  {
    return AndSpecification<T>(*this, other);
  }
};
```

If you now avoid making extra variables for size/color specifications, the composite specification can be reduced to a single line:

```
auto green_and_big =
  ColorSpecification(Color::Green)
  && SizeSpecification(Size::Large);
```

So let's recap what OCP principle is and how the above example enforces it. Basically, OCP states that you shouldn't need to go back to code you've already written and tested and change it. And that's exactly what's happening here! We made `Specification<T>` and `Filter<T>` and, from then on, all we have to do is implement either of the interfaces (without modifying the interfaces themselves) to implement new filtering mechanics. This is what is meant by "open for extension, closed for modification".

Liskov Substitution Principle

The Liskov Substitution Principle, named after Barbara Liskov, states that if an interface takes an object of type `Parent`, it should equally take an object of type `Child` without anything breaking. Let's take a look at a situation where LSP is broken.

Here's a rectangle; it has width and height and a bunch of getters and setters calculating the area:

```
class Rectangle
{
protected:
  int width, height;
public:
  Rectangle(const int width, const int height)
    : width{width}, height{height} { }

  int get_width() const { return width; }
  virtual void set_width(const int width) { this->width = width; }
  int get_height() const { return height; }
  virtual void set_height(const int height) { this->height = height; }

  int area() const { return width * height; }
};
```

Now let's suppose we make a special kind of `Rectangle` called a `Square`. This object overrides the setters to set both width *and* height:

```
class Square : public Rectangle
{
public:
  Square(int size): Rectangle(size,size) {}
  void set_width(const int width) override {
    this->width = height = width;
  }
  void set_height(const int height) override {
    this->height = width = height;
```

```
  }
};
```

This approach is *evil*. You cannot see it yet, because it looks very innocent indeed: the setters simply set both dimensions, what could possibly go wrong? Well, if we take the above, we can easily construct a function taking a `Rectangle` that would blow up when taking a square:

```
void process(Rectangle& r)
{
  int w = r.get_width();
  r.set_height(10);

  cout << "expected area = " << (w * 10)
    << ", got " << r.area() << endl;
}
```

The function above takes the formula `Area = Width × Height` as an invariant. It gets the width, sets the height, and rightly expects the product to be equal to the calculated area. But calling the above function with a `Square` yields a mismatch:

```
Square s{5};
process(s); // expected area = 50, got 25
```

The takeaway from this example (which I admit is a little contrived) is that `process()` breaks the LSP by being thoroughly unable to take a derived type `Square` instead of the base type `Rectangle`. If you feed it a `Rectangle`, everything is fine, so it might take some time before the problem shows up in your tests (or in production – hopefully not!).

What's the solution? Well, there are many. Personally, I'd argue that the type `Square` shouldn't even exist: instead, we can make a Factory (see the Factories chapter) that creates both rectangles and squares:

```
struct RectangleFactory
{
  static Rectangle create_rectangle(int w, int h);
  static Rectangle create_square(int size);
};
```

You might also want a way of detecting that a `Rectangle` is, in fact, a square:

```
bool Rectangle::is_square() const
{
  return width == height;
}
```

The nuclear option, in this case, would be to throw an exception in `Square`'s `set_width()/set_height()`, stating that these operations are unsupported and you should be using `set_size()` instead. This, however, violates the *principle of least surpise*, since you would expect a call to `set_width()` to make a meaningful change... am I right?

Interface Segregation Principle

Oh-kay, here is another contrived example that is nonetheless suitable for illustrating the problem. Suppose you decide to define a multifunction printer: a device that can print, scan and also fax documents. So you define it like so:

```
struct MyFavouritePrinter /* : IMachine */
{
  void print(vector<Document*> docs) override;
  void fax(vector<Document*> docs) override;
  void scan(vector<Document*> docs) override;
};
```

This is fine. Now, suppose you decide to define an interface that needs to be implemented by everyone who also plans to make a multifunction printer. So you could use the Extract Interface function in your favourite IDE and you'll get something like the following:

```
struct IMachine
{
  virtual void print(vector<Document*> docs) = 0;
  virtual void fax(vector<Document*> docs) = 0;
  virtual void scan(vector<Document*> docs) = 0;
};
```

This is a problem. The reason it is a problem is that some implementor of this interface might not need scanning or faxing, just printing. And yet, you are forcing them to implement those extra features: sure, they can all be no-op, but why bother with this?

So what the Interface Segregation Principle suggests is you split up interfaces so that implementors can pick and choose depending on their needs. Since printing and scanning are different operations (for example, a Scanner cannot print), we define separate interfaces for these:

```
struct IPrinter
{
  virtual void print(vector<Document*> docs) = 0;
};

struct IScanner
{
  virtual void scan(vector<Document*> docs) = 0;
};
```

Then, a printer or a scanner can just implement the required functionality:

```
struct Printer : IPrinter
{
  void print(vector<Document*> docs) override;
};

struct Scanner : IScanner
{
  void scan(vector<Document*> docs) override;
};
```

Now, if we really want an `IMachine` interface, we can define it as a combination of the aforementioned interfaces:

```
struct IMachine: IPrinter, IScanner /* IFax and so on */
{
};
```

And when you come to implement this interface in your concrete multifunction device, this is the interface to use. For example, you could use simple delegation to ensure that `Machine` reuses the functionality provided by a particular `IPrinter` and `IScanner`:

```
struct Machine : IMachine
{
  IPrinter& printer;
  IScanner& scanner;

  Machine(IPrinter& printer, IScanner& scanner)
    : printer{printer},
      scanner{scanner}
  {
  }

  void print(vector<Document*> docs) override {
    printer.print(docs);
  }

```

```
16    void scan(vector<Document*> docs) override
17    {
18      scanner.scan(docs);
19    }
20  };
```

So, just to recap, the idea here is to segregate parts of a complicated interface into separate interfaces so as to avoid forcing implementors to implement functionality that they do not really need. Anytime when you write a plugin for some complicated application and you're given an interface with 20 confusing functions to implement with various no-ops and `return nullptr`, more likely than not the API authors have violated the ISP.

Dependency Inversion Principle

The original definition of the Dependency Inversion Principle states the following[3]:

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

What this statement basically means is that, if you're interested in logging, your reporting component should not depend on a concrete `ConsoleLogger`, but can depend on an `ILogger` interface. In this case, we are considering the reporting component to be high-level (closer to the business domain), whereas logging, being a fundamental concern (kind of like file I/O or threading, but not quite) is considered a low-level module.

B. Abstractions should not depend on details. Details should depend on abstractions.

This is, once again, restating that dependencies on interfaces or base classes is better than dependencies on concrete types. Hopefully the truth of this statement is obvious, because such an approach supports better configurability and testability… provided you're using a good framework to handle these dependencies for you.

So now, the main question is: how do you actually implement all of the above? It surely is a lot more work, because now you need to explicitly state that, e.g., `Reporting` depends on an `ILogger`. The way you would express it is perhaps as follows:

[3] Martin, Robert C. (2003), *Agile Software Development, Principles, Patterns, and Practices*, Prentice Hall, pp. 127–131.

```
class Reporting
{
  ILogger& logger;
public:
  Reporting(const ILogger& logger) : logger{logger} {}
  void prepare_report()
  {
    logger.log_info("Preparing the report");
    ...
  }
};
}
```

Now the problem is that, to initialize the above class, you need to explicitly call `Reporting{ConsoleLogger{}}` or something similar. And what if `Reporting` is dependent upon 5 different interfaces? What if `ConsoleLogger` has dependencies of its own? You *can* manage this by writing a lot of code, but there is a better way.

The modern, trendy, fashionable way of doing the above is to use *Dependency Injection*: this essentially means that you use a library such as Boost.DI[4] to *automatically* satisfy the dependency requirements for a particular component.

Let's consider an example of a car which has an engine, but also needs to write to a log. As it stands, we can say that a car *depends on* both of these things. To start with, we may define an engine as:

```
struct Engine
{
  float volume = 5;
  int horse_power = 400;

  friend ostream& operator<< (ostream& os, const Engine& obj)
  {
    return os
      << "volume: " << obj.volume
      << " horse_power: " << obj.horse_power;
```

4At the moment, Boost.DI is not yet part of Boost proper, it is part of the `boost-experimental` GitHub repository.

```
  } // thanks, ReSharper!
};
```

Now, it's up to us to decide whether or not we want to extract an `IEngine` interface and feed it to the car. Maybe we do, maybe we don't, and this is typically a design decision. If you envision having a hierarchy of engines, or you foresee needing a `NullEngine` (see the Null Object pattern) for testing purposes then yes, you do need to abstract away the interfaces.

At any rate, we also want logging, and since this can be done in many ways (console, email, SMS, pigeon mail, ...), we probably want to have an `ILogger` interface...

```
struct ILogger
{
  virtual ~ILogger() {}
  virtual void Log(const string& s) = 0;
};
```

...as well as some sort of concrete implementation:

```
struct ConsoleLogger : ILogger
{
  ConsoleLogger() {}

  void Log(const string& s) override
  {
    cout << "LOG: " << s.c_str() << endl;
  }
};
```

Now, the car we're about to define depends on both the engine *and* the logging component. We need both, but it's really up to us how to store them: we can use a pointer, reference, a `unique_ptr/shared_ptr` or something else. We shall define both of the dependent components as constructor parameters:

```
struct Car
{
  unique_ptr<Engine> engine;
  shared_ptr<ILogger> logger;

  Car(unique_ptr<Engine> engine,
      const shared_ptr<ILogger>& logger)
    : engine{move(engine)},
      logger{logger}
  {
    logger->Log("making a car");
  }

  friend ostream& operator<<(ostream& os, const Car& obj)
  {
    return os << "car with engine: " << *obj.engine;
  }
};
```

Now, you're probably expecting to see `make_unique/make_shared` calls as we initialize the `Car`. But we won't do any of that. Instead, we'll use Boost.DI. First of all, we'll define a binding that binds `ILogger` to `ConsoleLogger`; what this means is, basically, "any time someone asks for an `ILogger` give them a `ConsoleLogger`":

```
auto injector = di::make_injector(
  di::bind<ILogger>().to<ConsoleLogger>()
);
```

And now that we've configured the injector, we can use it to create a car:

```
auto car = injector.create<shared_ptr<Car>>();
```

The above creates a `shared_ptr<Car>` that points to a *fully initialized* `Car` object, which is exactly what we wanted. The great thing about this approach is that, to change the type of logger being used, we can change it in a single place (the `bind` call) and every place where an `ILogger` appears can now be using some other logging component that we provide. This approach also helps us with unit testing, and allows us to use stubs (or the Null Object pattern) instead of mocks.

Time for Patterns!

With the understanding of the SOLID design principles, we are ready to take a look at the design patterns themselves. Strap yourselves in, it's going to be a long (but hopefully not boring) ride!

2. Builder

The Builder pattern is concerned with the creation of *complicated* objects, i.e., objects that cannot be built up in a single-line constructor call. These types of objects may themselves be composed of other objects and might involve less-than-obvious logic, necessitating a separate component specifically dedicated to object construction.

I suppose it's worth noting beforehand that, while I said the Builder is concerned with *complicated* objects, we'll be taking a look at a rather trivial example. This is done purely for the purposes of space optimization, so that the complexity of the domain logic doesn't interfere with the reader's ability to appreciate the actual implementation of the pattern.

Scenario

Let's imagine that we are building a component that renders web pages. To start with, we shall output a simple unordered list with two items containg the words *hello* and *world*. A very simplistic implementation might look as follows:

```
string words[] = { "hello", "world" };
ostringstream oss;
oss << "<ul>";
for (auto w : words)
  oss << "  <li>" << w << "</li>";
oss << "</ul>";
printf(oss.str().c_str());
```

This does in fact give us what we want, but the approach is not very flexible. How would we change this from a bulleted list to a numbered list? How can we add another item *after* the list has been created? Clearly, in this rigid scheme of ours, this is not possible.

We might, therefore, go the OOP route and define an `HtmlElement` class to store information about each tag:

```
struct HtmlElement
{
  string name;
  string text;
  vector<HtmlElement> elements;

  HtmlElement() {}
  HtmlElement(const string& name, const string& text)
    : name(name), text(text) { }

  string str(int indent = 0) const
  {
    // pretty-print the contents
  }
}
```

Armed with this approach, we can now create our list in a more sensible fashion:

```
string words[] = { "hello", "world" };
HtmlElement list{"ul", ""};
for (auto w : words)
  list.elements.emplace_back{HtmlElement{"li", w}};
printf(list.str().c_str());
```

This works fine and gives us a more controllable, OOP-driven representation of a list of items. But the process of building up each `HtmlElement` is not very convenient and we can improve it by implementing the Builder pattern.

Simple Builder

The Builder pattern simply tries to outsource the piecewise construction of an object into a separate class. Our first attempt might yield something like this:

```
struct HtmlBuilder
{
  HtmlElement root;

  HtmlBuilder(string root_name) { root.name = root_name; }

  void add_child(string child_name, string child_text)
  {
    HtmlElement e{ child_name, child_text };
    root.elements.emplace_back(e);
  }

  string str() { return root.str(); }
};
```

This is a dedicated component for building up an HTML element. The `add_child()` method is the method that's intended to be used to add additional children to the current element, each child being a name-text pair. It can be used as follows:

```
HtmlBuilder builder{ "ul" };
builder.add_child("li", "hello");
builder.add_child("li", "world");
cout << builder.str() << endl;
```

You'll notice that, at the moment, the `add_child()` function is `void`-returning. There are many things we could use the return value for, but one of the most common uses of the return value is to help us build a fluent interface.

Fluent Builder

Let's change our definition of `add_child()` to the following:

```
HtmlBuilder& add_child(string child_name, string child_text)
{
  HtmlElement e{ child_name, child_text };
  root.elements.emplace_back(e);
  return *this;
}
```

By returning a reference to the builder itself, the builder calls can now be chained. This is what's called a *fluent interface*:

```
HtmlBuilder builder{ "ul" };
builder.add_child("li", "hello").add_child("li", "world");
cout << builder.str() << endl;
```

The choice of references or pointers is entirely up to you. If you want to chain calls with the `->` operator, you can define `add_child()` like this:

```
HtmlBuilder* add_child(string child_name, string child_text)
{
  HtmlElement e{ child_name, child_text };
  root.elements.emplace_back(e);
  return this;
}
```

And then use it like this:

```
HtmlBuilder builder{"ul"};
builder->add_child("li", "hello")->add_child("li", "world");
cout << builder << endl;
```

Communicating Intent

We have a dedicated Builder implemented for an HTML element, but how will the users of our classes know how to use it? One idea is to simply *force* them to use the builder whenever they are constructing an object. Here's what you need to do:

```
struct HtmlElement
{
  string name;
  string text;
  vector<HtmlElement> elements;
  const size_t indent_size = 2;

  static unique_ptr<HtmlBuilder> build(const string& root_name)
  {
    return make_unique<HtmlBuilder>(root_name);
  }

protected: // hide all constructors
  HtmlElement() {}
  HtmlElement(const string& name, const string& text)
    : name{name}, text{text}
  {
  }
};
```

Our approach is two-pronged. First, we have hidden all constructors, so they are no longer available. We have, however, created a Factory Method (this is a design pattern we'll discuss later) for creating a builder right out of the `HtmlElement`. And it's a static method, too. Here's how one would go about using it:

```
auto builder = HtmlElement::build("ul");
builder.add_child("li", "hello").add_child("li", "world");
cout << builder.str() << endl;
```

But let's not forget that our ultimate goal is to build an `HtmlElement`, not just a builder for it! So the icing on the cake can be an implementation of `operator HtmlElement` on the builder to yield the final value:

```
struct HtmlBuilder
{
  operator HtmlElement() const { return root; }
  HtmlElement root;
  // other operations omitted
};
```

One variation on the above would be to return `std::move(root)`, but whether or not you want to do this is really up to you.

Anyways, the addition of the operator allows us to write the following:

```
HtmlElement e = HtmlElement::build("ul")
  .add_child("li", "hello")
  .add_child("li", "world");
cout << e.str() << endl;
```

Regrettably, there is no way of explicitly telling other users to use the API in this manner. Hopefully the restriction on constructors coupled with the presence of the static `build()` function get the user to use the builder, but, in addition to the operator, it might make sense to also add a corresponding `build()` function to `HtmlBuilder` itself:

```
HtmlElement HtmlBuilder::build() const
{
  return root; // again, std::move possible here
}
```

Groovy-style Builder

This example is a minor digression from dedicated builders since there is really no builder in sight. It is simply an alternative means of object construction.

Programming languages such as Groovy, Kotlin and others all try to show off how great they are at building DSLs by supporting syntactic constructs that make the process better. But why should C++ be any different? Thanks to initializer lists, we can effectively build an HTML-compatible DSL using ordinary classes.

First of all, we'll define an HTML tag:

```
struct Tag
{
  std::string name;
  std::string text;
  std::vector<Tag> children;
  std::vector<std::pair<std::string, std::string>> attributes;

  friend std::ostream& operator<<(std::ostream& os, const Tag& tag)
  {
    // implementation omitted
  }
};
```

So far, we have a `Tag` that can store its name, text, children (inner tags) and even HTML attributes. We also have some pretty-printing code that's too boring to show here.

Now we can give it a couple of `protected` constructors (because we don't want anyone to actually instantiate this directly). Our previous experiments have taught us that we have at least two cases:

- A tag initialized by name and text (e.g., a list item)
- A tag initialized by name and a collection of children

That second case is more interesting; we'll use a parameter of type `std::vector`:

```
struct Tag
{
  ...
  protected:
    Tag(const std::string& name, const std::string& text)
      : name{name}, text{text} {}


    Tag(const std::string& name, const std::vector<Tag>& children)
      : name{name}, children{children} {}
  };
```

Now we can inherit from this `Tag` class, but only for valid HTML tags (thereby constraining our DSL). Let's define two tags: one for a paragraph and another for an image:

```
struct P : Tag
{
  explicit P(const std::string& text)
    : Tag{"p", text} {}

  P(std::initializer_list<Tag> children)
    : Tag("p", children) {}

};

struct IMG : Tag
{
  explicit IMG(const std::string& url)
    : Tag{"img", ""}
  {
    attributes.emplace_back({"src", url});
  }
};
```

The above constructors further constrain our API. A paragraph, according to the constructors above, can only contain either text or a set of children. An image, on the other hand, can contain no other tag, but *must* have an attribute called `img` with the provided address.

And now, *the prestige* of this magic trick... thanks to uniform ininialization and all the constructors we've spawned, we can write the following:

```
std::cout <<

  P {
    IMG { "http://pokemon.com/pikachu.png" }
  }

  << std::endl;
```

Isn't this great? We've built a mini-DSL for paragraphs and images, and this model can easily be extended to support other tags. And there's no `add_child()` call in sight!

Composite Builder

We are going to finish off the discussion of Builder with an example where multiple builders are used to build up a single object. Let's say we decide to record some information about a person:

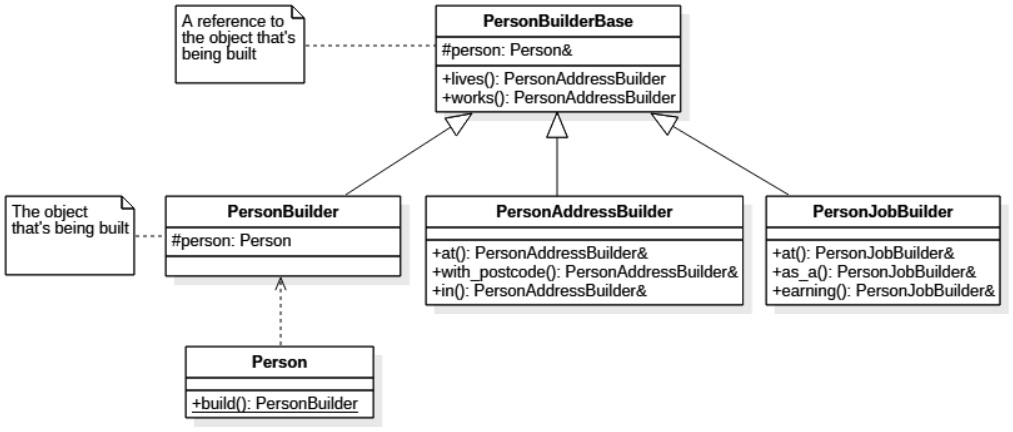
```
class Person
{
  // address
  std::string street_address, post_code, city;

  // employment
  std::string company_name, position;
  int annual_income = 0;

  Person() {}
};
```

There are two aspects to `Person`: their address and employment information. What if we want to have separate builders for each - how can we provide the most convenient API? To do this, we'll construct a composite builder. This construction is not trivial, so pay attention - even though we want separate builders for job and address information, we'll spawn no less than *four* distinct classes.

I know I promised no UML in this book, but this is the one case where a class diagram makes sense, so here is what we are actually going to build:



We'll call the first class `PersonBuilderBase`:

```
class PersonBuilderBase
{
protected:
  Person& person;
  explicit PersonBuilderBase(Person& person)
    : person{ person }
  {
  }
public:
  operator Person()
  {
    return std::move(person);
  }

  // builder facets

  PersonAddressBuilder lives() const;
  PersonJobBuilder works() const;
};
```

This is *much* more complicated than our simple Builder earlier, so let's discuss each member in turn:

- The reference `person` is a reference to the object that's being built. This may seem seriously weird, but it's done deliberately for the sub-builders. Note that the physical storage of `Person` is not present in this class. This is critical! The root class only holds a reference, not the constructed object.
- The reference-assigning constructor is `protected` so that only the inheritors (`PersonAddressBuilder` and `PersonJobBuilder`) can use it.
- `operator Person` is a trick that we've done before. I'm making the assumption that `Person` has a properly defined move constructor – ReSharper generates one with ease.
- `lives()` and `works()` are functions returning builder facets: those sub-builders that initialize the address and employment information separately.

Now, the only thing that is missing from the above base class is the actual object that's being constructed. Where is it? Well, it's actually stored in an inheritor that we'll call, ahem, `PersonBuilder`. That's the class that we expect people to actually use:

```
class PersonBuilder : public PersonBuilderBase
{
  Person p; // object being built
public:
  PersonBuilder() : PersonBuilderBase{p} {}
};
```

So this is where the built-up object is actually built. This class isn't meant to be inherited: it's only meant as a utility that lets us initiate the process of setting up a builder.[1]

To find out why exactly we ended up with different public and protected constructors, let's take a look at the implementation of one of the sub-builders:

[1] This approach to separating the hierarchy into two separate base classes so as to avoid duplication of `Person` instances was suggested by @CodedByATool on GitHub – thanks for the idea!

```
class PersonAddressBuilder : public PersonBuilderBase
{
  typedef PersonAddressBuilder self;
public:
  explicit PersonAddressBuilder(Person& person)
    : PersonBuilderBase{ person } {}

  self& at(std::string street_address)
  {
    person.street_address = street_address;
    return *this;
  }

  self& with_postcode(std::string post_code) { ... }

  self& in(std::string city) { ... }
};
```

As you can see, `PersonAddressBuilder` provides a fluent interface for building up a person's address. Note that it actually *inherits* from `PersonBuilderBase` (meaning it has acquired the `lives()` and `works()` member functions) and calls the base constructor, passing a reference. It doesn't inherit from `PersonBuilder` though – if it did, we'd create far too many `Person` instances, and truth be told, we only really need one.

As you can guess, `PersonJobBuilder` is implemented in identical fashion. Both of the classes, as well as `PersonBuilder` are declared as `friend` classes inside `Person` so as to be able to access its private members.

And now, the moment you've been waiting for: an example of these builders in action:

```
Person p = Person::create()
  .lives().at("123 London Road")
          .with_postcode("SW1 1GB")
          .in("London")
  .works().at("PragmaSoft")
          .as_a("Consultant")
          .earning(10e6);
```

Can you see what's happening here? We use the `create()` function to get ourselves a builder, use the `lives()` function to get us a `PersonAddressBuilder` but once we're done initializing the address information, we simply call `works()` and switch to using a `PersonJobBuilder` instead.

When we're done with the building process, we use the same trick as before to get the object being built-up as a `Person`. Note that, once this is done, the builder is unusable, since we moved the `Person` with `std::move()`.

Summary

The goal of the Builder pattern is to define a component dedicated entirely to piecewise construction of a complicated object or set of objects. We have observed the following key characteristics of a Builder:

- Builders can have a fluent interface that is usable for complicated construction using a single invocation chain. To support this, builder functions should return `this` or `*this`.
- To force the user of the API to use a Builder, we can make the target object's constructors inaccessible and then define a static `create()` function that returns the builder.
- A builder can be coerced to the object itself by defining the appropriate operator.
- Groovy-style builders are possible in C++ thanks to uniform initializer syntax. This approach is very general, and allows for the creation of diverse DSLs.
- A single builder interface can expose multiple sub-builders. Through clever use of inheritance and fluent interfaces, one can jump from one builder to another with ease.

Just to re-iterate something that I've already mentioned, the use of the Builder pattern makes sense when the construction of the object is a *non-trivial* process. Simple objects that are unambiguously constructed from a limited number of sensibly named constructor parameters should probably use a constructor (or dependency injection) without necessitating a Builder as such.