Design Patterns in Modern C++

Reusable Approaches for Object-Oriented Software Design

—

Dmitri Nesteruk

Apress®

Design Patterns in Modern C++

Reusable Approaches for Object-Oriented Software Design

Dmitri Nesteruk

Apress®

Design Patterns in Modern C++: Reusable Approaches for Object-Oriented Software Design

Dmitri Nesteruk
St. Petersburg, Russia

ISBN-13 (pbk): 978-1-4842-3602-4 ISBN-13 (electronic): 978-1-4842-3603-1
https://doi.org/10.1007/978-1-4842-3603-1

Library of Congress Control Number: 2018940774

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484236024. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Dmitri Nesteruk is a quantitative analyst, developer, course and book author, and an occasional conference speaker. His professional interests lie in software development and integration practices in the areas of computation, quantitative finance, and algorithmic trading. His technological interests include C# and C++ programming as well high-performance computing using technologies such as CUDA and FPGAs. He has been a C# MVP since 2009.

About the Technical Reviewer

Massimo Nardone has more than 24 years of experiences in Security, Web/Mobile development, Cloud, and IT Architecture. His true IT passions are Security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master of Science degree in Computing Science from the University of Salerno, Italy.

He has worked as a Project Manager, Software Engineer, Research Engineer, Chief Security Architect, Information Security Manager, PCI/SCADA Auditor, and Senior Lead IT Security/Cloud/SCADA Architect for many years.

Technical skills include: Security, Android, Cloud, Java, MySQL, Drupal, Cobol, Perl, Web and Mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, etc.

He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas).

He currently works as Chief Information Security Office for Cargotec Oyj and he is a member of ISACA Finland Chapter Board.

Massimo has reviewed more than 45 IT books for different publishers, and coauthored *Pro JPA 2 in Java EE 8* (Apress, 2018) and *Pro Android Games* (Apress, 2015).

CHAPTER 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 that aren't even properly OOP! So why another book on it?

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 modules, among other things.)

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.

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

D. Nesteruk, *Design Patterns in Modern C++*, https://doi.org/10.1007/978-1-4842-3603-1_1

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, that is, 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 coworkers), it's often not feasible.

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 will 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 want 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:

```
struct Foo : SomeBase<Foo>
{
  ...
}
```

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 Chapter 9.

Mixin Inheritance

In C++, a class can be defined to inherit from its own template argument, for example:

```
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 Chapter 9.

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 person;
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 five 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 lost 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!

I 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;
  }
};
```

This is precisely what is meant 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 antipattern that 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), 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 following code and recompile:

```
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;
}
```

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 preceding 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 preceding example, 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 standardize the notion of an enumeration or collection, something that exists in other programming languages (e.g., .NET's IEnumerable).

Based on the preceding, 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 preceding 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);
  }
};
```

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 preceding 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