Design Patterns in Modern C++20

Reusable Approaches for Object-Oriented Software Design

—

Second Edition

—

Dmitri Nesteruk

Apress®

Design Patterns in Modern C++20

Reusable Approaches for Object-Oriented Software Design

Second Edition

Dmitri Nesteruk

Apress®

Design Patterns in Modern C++20: Reusable Approaches for Object-Oriented Software Design

Dmitri Nesteruk
St. Petersburg, Russia

ISBN-13 (pbk): 978-1-4842-7294-7 ISBN-13 (electronic): 978-1-4842-7295-4
https://doi.org/10.1007/978-1-4842-7295-4

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 by Clark van der Beken on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. 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 booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail 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/9781484272947. 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 as high-performance computing using technologies such as CUDA and FPGAs. He has been a C# MVP since 2009.

About the Technical Reviewers

David Pazmino has been developing software applications for 20 years in Fortune 100 companies. He is an experienced developer in front-end and back-end development who specializes in developing machine learning models for financial applications. David has developed many applications in C++, STL, and ATL for companies using Microsoft technologies. He currently develops applications in Scala and Python for deep learning neural networks. David has a degree from Cornell University, a masters from Pace University in Computer Science, and a masters from Northwestern in Predictive Analytics.

Massimo Nardone has more than 25 years of experience 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 CISO, CSO, security executive, IoT executive, project manager, software engineer, research engineer, chief security architect, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years. His 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, and more.

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 is currently working for Cognizant as head of cyber security and CISO to help both internally and externally with clients in areas of information and cyber security, like strategy, planning, processes, policies, procedures, governance, awareness, and so forth. In June 2017 he became a permanent member of the ISACA Finland Board.

Massimo has reviewed more than 45 IT books for different publishing companies and is the co-author of *Pro Spring Security: Securing Spring Framework 5 and Boot 2-based Java Applications* (Apress, 2019), *Beginning EJB in Java EE 8* (Apress, 2018), *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 dull, 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, you probably have a limited amount of time to decide whether this book is worth the investment.

The main reason why this book exists is that C++ is "great again." After a long period of stagnation, it's now evolving and growing, and, despite the fact that it has to contend with backward C compatibility, good things are happening – they may not always happen at the pace we'd all like, but this is a byproduct of the way the evolution of the C++ language standard is structured.

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. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley

D. Nesteruk, *Design Patterns in Modern C++20*,
https://doi.org/10.1007/978-1-4842-7295-4_1

Design patterns are also a fun investigation of how a particular 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 solve the problem *and* impress your coworkers), it's often not feasible in the real world of time and budgeting constraints.

Who This Book Is For

This book is intended 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.[2]

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., concepts) make difficult problems a lot easier to solve.

[2] To be fair, the Pharo variety of Smalltalk has some interesting ideas that I have since borrowed and adapted to other programming languages. One idea, which I managed to successfully transplant, is the idea of input-output matching. It works like this: you give the software desired input and output values, say, `abc` and `3`, and a piece of software uses combinatorial analysis to derive the expression `x.length()` for taking you from one to another.

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 a `struct` instead of a `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 be sure I'm referring to `std::string`.
- I will avoid adding virtual destructors, whereas, in real life, it might make sense to add them in certain places.
- In some 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 often 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. Feature-completing a type is quite often a separate challenge, somewhat unrelated to the topic at hand.
- There will be plenty of cases where I will omit `const`, whereas, under normal circumstances, it would actually make sense to use it. 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++ 14, 17, 20, and beyond) and generally use the latest C++ language features that are available to developers at the time of writing. 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,[3] you'll need to find workarounds.

At certain points in time, I will be referring to other programming languages such as C# or Kotlin. It is often 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, such as Clang, GCC, and MSVC. I make the general assumption that you are using the latest compiler version that is available and thus 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. Naturally, if I use any experimental language features, they might not work in all compilers until they catch up to the necessary level of C++ language support.

As far as developer tools are concerned, this book does not focus 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

[3] Plenty of compilers, such as the Intel C++ Compiler, do not make it their goal to support all features of a particular C++ standard as quickly as possible. Nevertheless, these compilers do have their own loyal followings because they shine in areas other than feature-completeness such as optimization.

single `.cpp` files, but some examples that involve complex dependencies or static initialization are spread across several files. Regardless, I'd like to take this opportunity to remind you that quality developer tools such as 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.

Preface to the Second Edition

The world is changing. Some of those changes, such as the pandemic that we're currently experiencing worldwide, are a bit frightening. On the other hand, some changes are good: the C++20 standard has finally been ratified, and C++20 language features such as modules and concepts are making an appearance in popular C++ compilers.

We are, of course, far from having a complete implementation in any given compiler. For example, even if we are able to use modules in our own code, we still need to wait in order to have modularized implementations of the Standard Library, Boost, and other popular libraries. But what we have right now is already changing the way design patterns are implemented. For example, if, in the past, we wanted to ensure a template argument implemented some interface, we would use a `static_assert`. But now, with C++20, we can leverage concepts, which are reusable (avoiding cut and paste) and self-descriptive.

With the never-ending evolution of C++, we can all feel as if we were on a never-ending journey that keeps getting better and better. The only challenge is to learn how to leverage all the new functionality, a challenge for which I hope this book can become a useful tool. Enjoy!

Important Concepts

Before we begin, I wanted to briefly mention some of the key concepts of the C++ world that will be referenced in this book. None of them are particularly advanced, and most of them will be familiar to experienced C++ developers.

Curiously Recurring Template Pattern

I don't know if it qualifies to be listed as a separate *design* pattern, but the curiously recurring template pattern (CRTP) is certainly a pattern of sorts in the C++ world. The idea is simple: an inheritor passes *itself* as a template argument to its base class.

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

Why would one 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`, rather than inside the inheritor class? Intuition suggests that you cannot, because `SomeBase` itself does not provide a `begin()/end()` interface. But if you use CRTP, a derived class can pass information about *itself* to the base class:

```
struct MyClass : SomeBase<MyClass>
{
  class iterator {
```

```
    // your iterator defined here
  }
  iterator begin() const { ... }
  iterator end() const { ... }
}
```

This means that, inside the base class, you can cast `this` to a derived class type:

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

When calling `foo()` on an instance of `MyClass`, `this` pointer gets cast from `SomeBase*` to `MyClass*`. We then dereference the pointer and iterate on it using a range-based `for` loop which, of course, calls `MyClass::begin()` and `MyClass::end()` behind the scenes.

For a concrete example of this approach, check out Chapter 8, "Composite."

Mixin Inheritance

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

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

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

Mixin inheritance is particularly useful together with Concepts because it allows us to put constraints on the type our mixin inherits from and lets us deterministically use the base type features without relying on compile-time errors to tell us we are doing something wrong.

For a concrete example of this approach, check out Chapter 9, "Decorator."

Old-Fashioned Static Polymorphism

Imagine you want to build an alert system that notifies someone about an event by different means: email, SMS, Telegram, etc. Under the CRTP paradigm, you could implement a base `Notifier` class similar to the following:

```
template <typename TImpl>
class Notifier {
public:
```

```
  void AlertSMS(string_view msg)
  {
    impl().SendAlertSMS(msg);
  }
  void AlertEmail(string_view msg)
  {
    impl().SendAlertEmail(msg);
  }
private:
  TImpl& impl() { return static_cast<TImpl&>(*this); }
  friend TImpl;
};
```

Since `TImpl` is a template argument, we can call methods on it with impunity, knowing that, even though we're not explicitly specifying that `TImpl` must inherit from `Notifier` (we'll do this soon enough), the compiler will check that the methods `AlertSMS()` and `AlertEmail()` do actually exist.

This allows us to define a method which sends an alert on all channels:

```
template <typename TImpl>
void AlertAllChannels(Notifier<TImpl>& notifier, string_view
msg)
{
  notifier.AlertEmail(msg);
  notifier.AlertSMS(msg);
}
```

Now all that remains is to construct implementations of `Notifier`. For example, you can build a no-op (see the Null Object pattern) notifier for testing:

```
struct TestNotifier: public Notifier<TestNotifier>
{
  void SendAlertSMS(string_view msg){}
  void SendAlertEmail(string_view msg){}
};
```

And you can use this to do absolutely nothing!

```
TestNotifier tn;
AlertAllChannels(tn, "testing!"); // just testing!
```

While this is a workable approach, it has deficiencies, namely:

- We end up having two parallel APIs, that is, `AlertSMS()`/`SendAlertSMS()`. We cannot call those methods the same because then one would hide another (and your IDE will complain).
- The whole `impl()` thing is weird and feels unnecessary. You'd expect the alert methods to be virtual in base class and overriding in the implementing class.
- There's no explicit enforcement that `TImpl` has any particular interface; we try to call things to check them at runtime, but the implementer is not informed about what we call and where. Concepts can help with this.

Static Polymorphism with Concepts

The solution here is to introduce a concept that requires the presence of relevant member functions:

```
template <typename TImpl>
concept IsANotifier = requires(TImpl impl) {
  impl.AlertSMS(string_view{});
  impl.AlertEmail(string_view{});
};
```

Now, we no longer need the base `Notifier` class: we can simply construct the `AlertAllChannels` method that expects some type that has all the `AlertXxx()` methods:

```
template <IsANotifier TImpl>
void AlertAllChannels(TImpl& impl, string_view msg)
{
  impl.AlertSMS(msg);
  impl.AlertEmail(msg);
}
```

In this function, the `TImpl` template argument is required to support the `IsANotifier` concept. We can make a class that conforms to this requirement:

```
struct TestNotifier
{
  void AlertSMS(string_view msg) {}
  void AlertEmail(string_view msg) {}
};
```

And continue to use it as before. As you can see, we avoid the notion of a base class altogether.

Properties

Properties are a topic worth mentioning even though they are not part of the C++ standard. Despite the fact that properties have already proven themselves over and over in other programming languages, many C++ programming purists continue to believe that they have no business being part of C++ and are best implemented as a library solution – something that doesn't work particularly well, to be honest.

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 non-standard 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;
};
```

What happens here is, within `__declspec(property(...))` field declaration, you specify the getter and the setter using the keywords `get` and `put`. This then becomes a virtual field – it doesn't result in any memory allocations, but attempts to access this field or write to it are replaced by the compiler with calls to the getter and setter, respectively.

This can be used as follows:

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

Those not fond of C++ language extensions typically expose properties as a combination of getter and setter methods, often by keeping the field private and exposing a pair of identically named (overloaded) methods with the same name as the field they expose:

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

Why is this discussion relevant? In and of themselves, getters and setters may seem useless: if you have a field that you want people to modify, expose it as public and be done with it! If, however, you want to perform additional actions – for example, notifying subscribers that a field has changed – then the setter is exactly the place where some of the code should go. This is what we'll encounter when we talk about the Observer design pattern.

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.[4] 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 to see them), we're going to do a brief recap of what the SOLID principles are all about.

[4] https://blog.cleancoder.com/

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: a situation that leads us 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 we'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 we use will actually let us perform a refactoring and have the change propagate everywhere. But when we need to completely rework an interface… well, this 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)
```