Introduction

I wrote this book because there is a dire need for clean and simple design patterns in embedded firmware projects.

This dire need has been around for years. It manifests itself by projects continuously accumulating technical debt through poorly structured source code that gets progressively harder and harder to debug as the project evolves.

I can see how all of this technical debt can be easily avoided and how bits and pieces of complex multi-threaded firmware can easily fall into place if only a few simple rules were applied throughout the source code.

I have therefore put together this course that outlines the most essential design patterns that help large project get organized.

What you will learn

In this course you will learn valuable programming patterns that are designed to be used as templates for code structure.

The primary purpose of design patterns is clear and simple expectations. Thus, through the application of the patterns in this book, you will write code where code structure adheres to the obvious ways of how things should be.

Design patterns are not just for a particular programming language. Instead they are abstractions that we express in a programming language. All programming languages come with their own language constructs that make implementations of these patterns easier (or harder) but the pattern still always remains the same.

This book starts with the **`Object Pattern`** which is the single most important design pattern in C programming. Yet it is also the pattern that is heavily underused. The Object pattern, when applied to a disorganized code base, very quickly makes everything fall into place. It irons out dependencies, it forces sensible refactoring, it makes things clear, crisp and simple.

We then build on the Object pattern and look at how we can deal with memory allocation needed for implementing the **`Opaque Object`** pattern. This pattern helps us forcefully hide implementation details in order to encourage loose coupling. As in many cases of design patterns, we use them and enforce them precisely for the purpose of letting the pattern help us avoid mistakes. In the case of the opaque pattern, it is by applying the pattern that we are prevented from directly accessing structure data members outside of the C file that implements the structure.

After the Opaque Pattern, we continue to the **`Singleton Pattern`** which is a way of enforcing only one instance of a particular object. When you ensure that all of your developers on the team understand design patterns, it becomes simple to signal intent when you then use design patterns and developers quickly catch on the idea and the code generally remains in good shape. Singleton is another pattern that enforces an original intent of only having one instance.

We also cover the **`Factory Pattern`** as it is another pattern that belongs to the group we

call Creational Patterns. This pattern greatly simplifies our code when we use it correctly and can parameterize creation of complex data structures using structured data.

In the Structural Patterns section we begin with looking at the **`Callback Pattern`** which is absolutely essential pattern for all of embedded programming because without it our source code almost always deteriorates into a mess of data dependencies and code dependencies which are completely out of control. When you do not apply this pattern and use raw function calls or function pointers instead, the bad design choices tend to proliferate throughout the firmware, making your whole code base gradually decay.

After the callback pattern, we then briefly look at how we can deal with inheritance concepts in C in the section on **`Inheritance Pattern`**. The C language is rather limited in syntax to represent inheritance, so we mainly look at conceptual rules that we follow which keep inheritance relationships between objects in clear view so that developers can easily reason about structure of the firmware application.

The **`Virtual API Pattern`** is an absolute necessity in embedded systems where it is often the case that we want to have multiple implementations of drivers and algorithms that can all be treated just the same and accessed through the same API. In this pattern we look at how we can implement this in a type safe and memory efficient way without resorting to any kinds of unsafe practices such as generic (void) pointers.

Once we have virtual API covered, we are ready to build higher level abstractions because these depend on simple lower level abstractions. We now look at a more complex **`Bridge Pattern`** that makes heavy use of virtual API pattern but enables us to bridge two independent hierarchies of objects such as to make the interface very simple.

Then I cover the **`Return Value Pattern`** which is so important for having a clear convention around return values. It saves time and impacts all code because return values need to be always checked and if expectations are different each time then it opens up possibility for huge errors.

Then we move into the **`Concurrency Pattern`** where I cover the main overarching pattern of concurrency and how concurrency is designed to help you solve problems related to responsiveness.

The first synchronization pattern that we look at is the **`Spinlock Pattern`** which takes care of synchronization closest to hardware between application code and interrupt handlers as well as between code running on multiple CPUs.

The **`Semaphore Pattern`** describes a pattern of signaling from an interrupt handler to thread context. This is a very simple pattern that builds on the spinlock mechanics and adds a thread queue where we can park threads that are waiting for the semaphore to become available.

In the **`Mutex Pattern`** we look at how we implement mutual exclusion completely above the level of interrupts. The mutex pattern gives us means to control mutually exclusive access to data between multiple threads regardless of which CPU they are running on.

Finally we look at the **`Conditional Pattern`** where multiple threads can wait for an event

and then become unblocked all at the same time by an event that is triggered either from an interrupt handler or from another thread.

Who this training is for

This training may sound simple, but it is in fact extremely useful for you even if you are a professional engineer with many years of experience in embedded firmware development. You will likely several areas in your architecture which you can improve on. It will shine clarity on concepts you have direct use for in every day development.

If you are a beginner programmer then these patterns will help you build future software better and faster. If you just learn a programming language without establishing higher level rules and expectations about code structure then you will always continue to struggle with questions that relate to structuring your code efficiently.

If you are a manager of an embedded team then you absolutely must make sure that all your team members understand how to structure code effectively. You must adopt and create your own patterns that keep your code base clean and this training will give you ideas on how to do that.

When to introduce these patterns

If you are new to design patterns and have not been applying structure to your existing code then the best time to do it is when you can spend some time refactoring your code. You will have to untangle dependencies, refactor interfaces, clean up C files littered with static data and just generally bring your whole code base to the next level. When you have existing code to refactor it becomes much easier to see how the pattern improves your code so this is probably one of the best starting points - refactoring your existing code.

If you are starting a new project then do not worry about religiously following the patterns - but rather try to write some functionality first and then refactor it before you mark it as completed. This way you can split your learning into two parts. The first part where you focus on solving your specific problem and the second part where you focus on structuring the code right.

When you start applying the patters that is when you will start to see how useful they are for your whole software architecture.

Patterns are architectural building blocks. They are largely language agnostic and we must often find language features that are best suited for each pattern and use them. Some languages provide better language features for some of these patterns and in such cases these basic patterns may evolve into more intricate structures in your project. This evolution process you will have to do on your own.

Where these patterns are most useful

Patterns are structural templates. They provide the most value when they are duplicated across your software architecture so that the details of your implementation fall neatly into

place.

For example, if you apply the return value pattern to every single function that needs to return a status, then the details of your return values everywhere fall into place. You know what to expect and how to check the return values consistently each time you call a function. This is the power of the patterns. Clear expectations. Making mistakes obvious when you look at the code and making new mistakes harder to make.

It is generally not as valuable to have a pattern that you only use once. We use patterns to truly tighten the bolts on our architecture. The patterns should always be applied everywhere. This way you can minimize future refactoring and keep your code in a clean and maintainable state at all times.

How to study

Each section of this training consists of several parts:

- **Overview**: here we cover a quick overview of the pattern and what it is for.
- **Use Cases**: this section gives you clues as to where you would use a particular pattern.
- **Benefits**: here we cover benefits of the pattern in question. Sometimes we compare the pattern to other patterns and I try to clarify how the particular pattern is better and when.
- **Drawbacks**: this section outlines the drawbacks of a pattern compared to other, possibly better, patterns in some situations.
- **Implementation**: in this section we look at actual code and typical implementation of the pattern that covers all important aspects of the pattern. Mostly in C and in some sections I also use Rust along side of C.
- **Best Practices**: this section outlines a list of best practices to keep in mind when applying the pattern.
- **Common Pitfalls**: here we look at some common pitfalls to look out for when implementing the pattern.
- **Alternatives**: this section briefly mentions alternative patterns.
- **Quiz**: the quiz checks your understanding of each pattern.

Getting help

- **Join discord server**: https://swedishembedded.com/community
- **Email**: martin.schroder@swedishembedded.com
- **Connect with me on linkedin**: https://www.linkedin.com/in/martinschroder/
- **Become a member**: https://swedishembedded.com/#/portal/signup/monthly