Patterns in C

By Adam Tornhill

Patterns in C

Patterns, Idioms and Design Principles

Adam Tornhill

This book is for sale at http://leanpub.com/patternsinc

This version was published on 2015-05-24

Tweet This Book!

Please help Adam Tornhill by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought #PatternsInC by @AdamTornhill

The suggested hashtag for this book is #patternsinc.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#patternsinc

Contents

Patterns in C

Patterns in software

Software development is hard. It's an act of balancing different forces while trying to close the gap between the problem domain and a solution model. Worse yet, we have to start at a point where our understanding of these models is still incomplete. We constantly have to juggle with and tame complexity on multiple levels. And we need to be extremely precise and correct in expressing our solutions. The computer is an unforgiving host.

Given this view, I consider patterns as a valuable tool in any software developers skill set. While patterns are valuable, they're also one of the most misunderstood concepts within our profession. They receive blame and ridicule. Much criticism is valid. Patterns have been used speculatively in many designs, perhaps most often within the Java community. The resulting context is layers of abstractions detouring far from the actual problem to solve. I mean, maintaining an *AbstractFactorySingletonDecoratorBuilder-PrototypeFlyweight* isn't why I went into programming.

Other parts of the criticism may stem from the early years of patterns. Patterns sprung from the work of architect Christopher Alexander. Since the dawn of large-scale programming projects, the software community has borrowed both terminology and design philosophy from the discipline of architecture. Sometimes the parallels have been stretched beyond the limits of both use and reason. This was the case with much of the secondary literature on design patterns that arose out of the excitement over this fascinating work. Writing about patterns rather than actual patterns lead to a gap; it wasn't always clear how patterns applied to the actual programs we were writing. How do they actually solve any interesting problem? The result was the patterns were viewed as something abstract. Nothing could be further from Alexander's original intent.

The patterns found in Alexander's books are simple and elegant formulations on how to solve recurring problems in different contexts. Alexander's pattens refer to the physical world. They are about the nature of making towns and buildings. Alexander's philosophy is about making those buildings come alive. His work is a praise of collaborative construction guided by a shared language – a pattern language. To Alexander, such a language is a generative, non-mechanical construct. It's a language with the power to evolve and grow. As such, patterns are more of a communication tool than technical solutions.

The cognitive view

In his classic talk at the Turing awards 1972, Dijkstra remarked that computer programming is an "intellectual challenge which will be without precedent in the cultural history of mankind" (Dijkstra, 1972). What is it that makes software development so hard, in Dijkstra's terms, "without precedent" to anything else we've constructed? Dijkstra himself gave the answer by concluding that a "competent programmer is fully aware of the strictly limited size of his own skull". It's a reference made to the great cognitive challenges of software development. Programming stretches our cognitive abilities to a maximum and we need to counter with effective design strategies to handle all the complexity inherent in software.

One of the main factors limiting the size of our skull is working memory. Working memory is a theoretical construct and understood as the system that allows us to hold information in our mind, integrate different parts, reason about them and manipulate them. Working memory is what we use when we try to decipher a function in Lisp, understand the relationship between two Java objects, or find a way to express a certain domain rule in Haskell. It's the conscious workbench of the mind.

Working memory is vital to our reasoning, problem solving, and decision making. It's also strictly limited in its capacity. Back in 1956, George Miller made the first quantification of our working memory capacity. Miller arrived at the now well-known heuristic of seven items, plus minus two.

Given the mere seven items we can hold in working memory simultaneously, it's no wonder that programming is hard; any interesting programming problem has a multitude of parameters, implications and possible alternatives. One way around this limitation is a process known as *chunking*. Chunking is an encoding strategy where individual elements are grouped into higher-level groups, *chunks*. While the limit on the number of units still apply, each unit now holds more information.

Patterns are a sophisticated form of chunking. Instead of describing a design as "well, I have this function that takes another function as argument. The function given as argument expresses the variability in the algorithm. That lets me extend the behavior without modifying the algorithm itself. I just write a new function fulfilling the contract" we could simply say "STRATEGY". Here patterns serve as a handle to sophisticated knowledge stored in our long-term memory. When it comes to software maintenance, a significant part of the budget for any successful product, documented patterns are an economical advantage.

Patterns have psychological value beyond chunking. Just like Alexander intended, patterns allow us to build and share a common vocabulary. It simplifies the communication. It's a also a powerful reasoning tool. Instead of reasoning about individual design elements and coding constructs, patterns provide a way to group these concepts into a larger unit

Patterns go C

The pattern format has gained tremendous popularity as a format for capturing experience. One of the reasons for this popularity is the unique success of the classic book Design Patterns by the Gang of Four. The Design Patterns book served the community by spreading the word about patterns. But it was a double edged sword. The unique success of Design Patterns lead many developers to believe that patterns are limited to object-orientation.

Todays patterns in the software industry aren't limited to design; there exists a broad range of patterns, covering analysis patterns, patterns for organizations, patterns for testing, etc.

When it comes to the actual implementations of patterns, most patterns are still described in the context of an object oriented design. By browsing a popular online bookstore, I noticed a lot of language specific pattern literature: design patterns in Java, C#, Smalltalk and other popular object oriented languages. The C language was sadly absent in the pattern literature.

Given the remaining popularity of C, where are the books targeting the unique implementation constraints and techniques for the language? Isn't it possible to use patterns in the development of C programs or doesn't it add any benefits?

The purpose of this book is to answer those questions. It is my intent to convey an idea of how known patterns may be applied naturally in C. I discuss the actual implementation techniques and trade-offs.

About this book

This patterns in this book are based on an article series I wrote back in 2005. Some years earlier I had returned to programming in C after some years spent with object-oriented languages. The relative simplicity of the C language has always attracted me. What I lacked was the guiding design principles common in object-oriented communities.

I soon realized that a lot of that knowledge applied to programs in C as well. The problem was the papers and books discussed them using completely different language elements. To a C programmer, inheritance and polymorphism aren't necessary parts of the design vocabulary. C has different approaches to solving problems.

As I started to apply design principles from my object-oriented background I was forced to balance the implementations with the way C worked. I wanted to avoid emulating object-oriented constructs.

Instead, I looked to leverage the existing design elements of C into higher-level constructs. An important thing to realize about patterns is that they are neither a blueprint of a design, nor are they tied to any particular implementation. My starting point was that it would be possible to find mechanisms fitting the paradigm of C and thus letting C programmers benefit from the experience captured by patterns.

Once the project was finished I took some time to document my findings. Through this book, I wish to share my findings with you. I've gone through the original writings, corrected where necessary, but largely left the original content unedited. Instead, I provide a new reflection on each chapter. With the advantage of seven more years of studies and experiences I hope to shed some new light on the patterns, the principles behind and their applicability.

Let the chapters ahead be the starting point for discussing patterns within the context of the ever popular C language.

Scope of the book

What you will experience

It is my belief that C programmers can benefit from the growing catalogue of patterns. This series will focus on the following areas:

- *Implementation techniques.* I will present a number of patterns and demonstrate techniques for implementing them in the context of the C language. In case I'm aware of common variations in the implementation, they will be discussed as well. The implementations included should however not by any means be considered as a final specification. Depending on the problem at hand, the implementation trade-offs for every pattern has to be considered.
- *Problem solved.* Patterns solve problems. Without any common problem, the "pattern" may simply not qualify as a pattern. Therefore I will present the main problem solved by introducing the pattern and provide examples of problem domains where the pattern can be used.
- *Consequences on the design.* Every solution implies a set of trade-offs. Therefore each pattern will discuss the consequences on the quality of the design in the resulting context.

...and what you won't

- *Object oriented feature emulation.* The pattern implementations will *not* be based on techniques for emulating object oriented features such as inheritance or C++ virtual functions. In my experience, these features are better left to a compiler; manually emulating such techniques are obfuscating at best and a source of hard to track down bugs at worst. Instead, it is my intent to present implementations that utilizes the strengths of the abstraction mechanisms already included in the C language.
- *In depth discussion of patterns.* As the focus in this book is on the implementation issues in C, each pattern should be seen as a complement to the original pattern descriptions. By those means, this series will *not* include exhaustive, in depth treatment of the patterns. Instead I will provide a high-level description of the pattern and reference existing work, where a detailed examination of the pattern is found.

Pattern Categories

This book covers the following patterns:

Pattern	Category	Purpose
First-Class ADT	Idiom	Improves encapsulation, manages dependencies.
State	Design	Models state-specific behavior.
Strategy	Design	Encapsulates families of algorithms, makes a design open-closed.
Observer	Design	A notification mechanism between loosely coupled entities.
Reactor	Architecture	Decouples responsibilities in event-driven applications.
Expressions	Idioms	A collection of idioms for expressiveness and robustness.

The patterns span the following categories:

- *Architectural patterns.* Frank Buschmann defines such a pattern as "a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them" [Buschmann, et al].
- *Design patterns.* A design pattern typically affects the subsystem or component level. Most patterns described in this book are from this category, including the patterns drawn from the classic Design Patterns [Gamma, et al] book.
- *Language level patterns.* This is the lowest level of the pattern-categories, also known as idioms. A language level pattern is, as its name suggests, mainly unique to one particular programming language. One simple, classic example is the strcpy version from Kernighan and Ritchie [Kernighan & Ritchie].

```c
void strcpy(char *s, char *t)
{
    while(*s++ = *t++);
}
```

Sample code

The source code included in the examples is available on GitHub: https://github.com/adamtornhill/PatternsInC

About Adam Tornhill

Adam is a programmer that combines degrees in engineering and psychology. He's the author of Your Code as a Crime Scene[1], has written the popular Lisp for the Web[2] tutorial and self-published a book on

[1]https://pragprog.com/book/atcrime/your-code-as-a-crime-scene
[2]https://leanpub.com/lispweb

Patterns in C. Adam also writes open-source software in a variety of programming languages. His other interests include modern history, music and martial arts.

Credits

My deepest thanks to my beautiful Jenny Tornhill for all her support and motivation. Jenny[3] also designed the cover of this book.

Drago Krznaric, Andre Saitzkoff, Tord Andersson and Magnus Adamsson all read early drafts of the original series. Thanks for your feedback! I would also like to thank ACCU[4] for publishing the original articles back in 2005.

Finally, thanks to all who have sent encouraging comments and feedback on the original writings over the years. This book is for you.

[3]http://www.jennytornhill.se

[4]http://accu.org/

The STATE Pattern

Reflections on STATE

Over the years, the STATE pattern has been at the end of the road for several successful refactorings. The main problem with the STATE pattern is its name. The name is ill-chosen. After all, state is all imperative programming is about. With such a dominant name, we run the risk of getting the emphasis on THE State Pattern. It's quite a complex pattern. I tend to balance the alternatives carefully. In most cases, simpler alternatives will do. Other patterns, like Methods for States, are often easier to reason about. In simple cases, explicit conditional logic is often just fine; we just have to watch out for possible duplications. As far as the name of the pattern is concerned, Design Patterns itself suggests a better name: Objects For State. The alternative name is better fit. It expresses both the intent and resulting structure in a much better way.

My adventures in functional programming languages have taught me the value of minimizing state and side-effects. Introducing state in our programs has implications for our ability to reason about them. Not only do we have to consider the algorithm in the light of its input arguments; we must also determine the current state and how it will impact the computation.

To complicate matters more, most of these states are hidden. They're often implicit in the design, expressed as combinations of different variables. In many cases, these states are better expressed as state machines in our code. By organizing and partitioning the algorithms into explicit states we get a different view of the problem. Its a simplification that allows us to asses the impact of potential changes. While all good designs try to keep side-effects to a minimum, mutable state is a complicated aspect inherent in imperative programming. We need conceptual tools to successfully express it.

Implementing the STATE Pattern

Every non-trivial program passes through a number of different states during its lifecycle. Describing this lifecycle as a finite state machine is a simple and useful abstraction. This chapter investigates different strategies for implementing state machines. The goal is to identify mechanisms that let the code communicate the intent of expressing the problem as a finite state machine.

Traditional Solution with Conditionals

Consider a simple, digital stop-watch. In its most basic version, it has two states: started and stopped. A traditional and direct way to implement this behavior in C is with conditional logic in the shape of switch/case statements and/or if-else chains.

The digital stop-watch in this example is implemented as a First-Class ADT.

```c
typedef enum
{
    stopped,
    started
} State;

struct DigitalStopWatch
{
    /* Let a variable hold the state of our object. */
    State state;
    TimeSource source;
    Display watchDisplay;
};

void startWatch(DigitalStopWatchPtr instance)
{
    switch(instance->state)
    {
        case started:
           /* Already started -> do nothing. */
           break;
        case stopped:
           instance->state = started;
           break;
        default:
           error("Illegal state");
           break;
    }
}

void stopWatch(DigitalStopWatchPtr instance)
{
    switch(instance->state)
    {
        case started:
            instance->state = stopped;
            break;
        case stopped:
            /* Already stopped -> do nothing. */
            break;
        default:
            error("Illegal state");
            break;
    }
}
```

This is a superficially simple solution. While the coding construct may be simple, the approach introduces several potential problems:

1. *It doesn't scale.* In large state machines the code may stretch over page after page of nested conditional logic. Imagine the true maintenance nightmare of changing large, monolithic segments of conditional statements.
2. *Duplication.* The conditional logic tends to be repeated, with small variations, in all functions that access the state variable. As always, duplication leads to error-prone maintenance. For example, simply adding a new state implies changing several functions.
3. *No separation of concerns.* When using conditional logic for implementing state machines, there is no clear separation between the code of the state machine itself and the actions associated with the various events. This makes the code hide the original intent (abstracting the behaviour as a finite state machine) and thus making the code less readable.

A Table-based Solution

The second traditional approach to implement finite state machines is through transition tables. Using this technique, our original example now reads as follows.

```c
typedef enum
{
    stopped,
    started
} State;

typedef enum
{
    stopEvent,
   startEvent
} Event;

#define NO_OF_STATES 2
#define NO_OF_EVENTS 2

static State TransitionTable[NO_OF_STATES][NO_OF_EVENTS] =
{
    { stopped, started },
    { stopped, started }
};

void startWatch(DigitalStopWatchPtr instance)
{
    const State currentState = instance->state;

    instance->state = TransitionTable[currentState][startEvent];
}

void stopWatch(DigitalStopWatchPtr instance)
{
    const State currentState = instance->state;

    instance->state = TransitionTable[currentState][stopEvent];
}
```

The choice of a transition table over conditional logic solved the previous problems:

1. *Scales well.* Independent of the size of the state machine, the code for a state transition is just one, simple table- lookup.
2. *No duplication.* Without the burden of repetitive switch/case statements, modification comes easily. When adding a new state, the change is limited to the transition table; all code for the state handling itself goes unchanged.
3. *Easy to understand.* A well structured transition table serves as a good overview of the complete lifecycle.

Shortcomings of Tables

As appealing as table-based state machines may seem at first, they have a major drawback: it is very hard to add actions to the transitions defined in the table. For example, the watch would typically invoke a function that starts to tick milliseconds upon a transition to state started. As the state transition isn’t explicit, conditional logic has to be added in order to ensure that the tick-function is invoked solely as the transition succeeds. In combination with conditional logic, the initial benefits of the table-based solution soon decrease together with the quality of the design.

Other approaches involve replacing the simple enumerations in the table with pointers to functions specifying the entry actions. Unfortunately, the immediate hurdle of trying to map state transitions to actions in a table based solution is that the functions typically need different arguments. This problem is possible to solve, but the resulting design loses, in my opinion, both in readability as well as in cohesion as it typically implies either giving up on type safety or passing around unused parameters. None of these alternatives seem attractive.s

Transition tables definitely have their use, but when actions have to be associated with state transitions, the STATE pattern provides a better alternative.

Enter STATE Pattern

In its description of the STATE pattern, Design Patterns [Gamma, et al] defines the differences from the table-based approach as “the STATE pattern models state-specific behavior, whereas the table-driven approach focuses on defining state transitions”. When applying the STATE pattern to our example, the structure in Illustration 1 emerges.

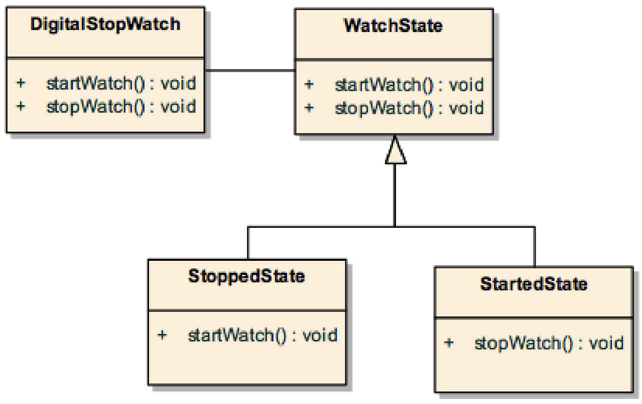

Illustration 1: Canonical structure of the STATE pattern.

This diagram definitely looks like an object oriented solution. But please don’t worry – we will not follow the temptation of the dark side and emulate inheritance in C. However, before developing a concrete implementation, let’s explain the involved participants.

- *DigitalStopWatch*: Design Patterns defines this as the context. The context has a reference to one of our concrete states, without knowing exactly which one. It is the context that specifies the interface to the clients.
- *WatchState*: Defines the interface of the state machine, specifying all supported events.
- *StoppedState* and *StartedState*: These are concrete states and each one of them encapsulates the behavior associated with the state it represents.

The main idea captured in the STATE pattern is to represent each state as an object of its own. A state transition simply means changing the reference in the context (*DigitalStopWatch*) from one of the concrete states to the other.

Implementation Mechanism

Which mechanism may be suitable for expressing this, clearly object oriented idea, in C? Returning to our example, we see that we basically have to switch functions upon each state transition. Luckily, the C language supplies one powerful feature, pointers to functions, that serves our needs perfectly by letting us change the behavior of an object at run-time. Using this mechanism, the interface of the states would look as:

Listing 1: The state interface in WatchState.h

```c
/* An incomplete type for the state representation itself. */
typedef struct WatchState* WatchStatePtr;

/* Simplify the code by using typedefs for
   the function pointers. */
typedef void (*EventStartFunc)(WatchStatePtr);
typedef void (*EventStopFunc) (WatchStatePtr);

struct WatchState
{
    EventStartFunc start;
    EventStopFunc stop;
};
```

Breaking the Dependency Cycle

After getting used to the scary syntax of pointers to functions, the interface above looks rather pleasant. However, with the interface as it is, a dependency cycle will evolve.

Consider the pointers in the WatchState structure. Every concrete state has to define the functions to be pointed at. This implies that each time an event is added to the interface, all concrete states have to be updated. The resulting code would be error-prone to maintain and not particularly flexible.

The good news is that breaking this dependency cycle is simple and the resulting solution has the nice advantage of providing a potential error-handler. The trick is to provide a default implementation, as illustrated in the listing below.

Listing 2: Extend the interface in WatchState.h

```c
/* ..previous code as before.. */

void defaultImplementation(WatchStatePtr state);
```

Listing 3: Provide the default implementations in WatchState.c

```c
static void defaultStop(WatchStatePtr state)
{
    /* We'll get here if the stop event isn't supported
       in the concrete state. */
}

static void defaultStart(WatchStatePtr state)
{
    /* We'll get here if the start event isn't supported
       in the concrete state. */
}

void defaultImplementation(WatchStatePtr state)
{
    state->start = defaultStart;
    state->stop = defaultStop;
}
```

Concrete States

The default implementation above completes the interface of the states. The interface of each state itself is minimal; all it has to do is to declare an entry function for the state.

Listing 4: Interface of a concrete state, StoppedState.h

```c
#include "WatchState.h"

void transitionToStopped(WatchStatePtr state);
```

Listing 5: Interface of a concrete state, StartedState.h

```c
#include "WatchState.h"

void transitionToStarted(WatchStatePtr state);
```

The responsibility of the entry functions is to set the pointers in the passed WatchState structure to point to the functions specifying the behavior of the particular state. As we can utilize the default implementation, the implementation of the concrete states is straightforward; each concrete state only specifies the events of interest in that state.

Listing 6: StoppedState.c

```c
#include "StoppedState.h"
/* Possible transition to the following state: */
#include "StartedState.h"

static void startWatch(WatchStatePtr state)
{
    transitionToStarted(state);
}

void transitionToStopped(WatchStatePtr state)
{
    /* Initialize with the default implementation before
       specifying the events to be handled in the stopped
       state. */
    defaultImplementation(state);
    state->start = startWatch;
}
```

Listing 7: StartedState.c

```c
#include "StartedState.h"
/* Possible transition to the following state: */
#include "StoppedState.h"

static void stopWatch(WatchStatePtr state)
{
    transitionToStopped(state);
}

void transitionToStarted(WatchStatePtr state)
{
     /* Initialize with the default implementation before
        specifying the events to be handled in the started
        state. */
     defaultImplementation(state);
     state->stop = stopWatch;
}
```

Client Code

The reward for the struggle so far comes when implementing the context, i.e. the client of the state machine. All the client code has to do, after the initial state has been set, is to delegate the requests to the state.

```c
struct DigitalStopWatch
{
    struct WatchState state;
    TimeSource source;
    Display watchDisplay;
};

DigitalStopWatchPtr createWatch(void)
{
    DigitalStopWatchPtr instance = malloc(sizeof *instance);

    if(NULL != instance)
    {
        /* Specify the initial state. */
        transitionToStopped(&instance->state);

        /* Initialize the other attributes here... */
    }

    return instance;
}

void destroyWatch(DigitalStopWatchPtr instance)
{
    free(instance);
}
```

```c

void startWatch(DigitalStopWatchPtr instance)
{
    instance->state.start(&instance->state);
}

void stopWatch(DigitalStopWatchPtr instance)
{
    instance->state.stop(&instance->state);
}
```

A Debug Aid

In order to ease debugging, the state structure may be extended with a string holding the name of the actual state:

```c
void transitionToStopped(WatchStatePtr state)
{
    defaultImplementation(state);
    state->name = "Stopped";
    state->start = startWatch;
}
```

Utilizing this extension, it becomes possible to provide an exact diagnostic in the default implementation. Returning to our implementation of WatchState.c, the code now looks like:

```c
static void defaultStop(WatchStatePtr state)
{
    /* We'll get here if the stop event isn't supported
       in the concrete state. */
    logUnsupportedEvent("Stop event", state->name);
}
```

Extending the State Machine

One of the strengths of the STATE pattern is that it encapsulates all state-specific behavior making the state machine easy to extend.

- *Adding a new event.* Supporting a new event implies extending the *WatchState* structure with a declaration of another pointer to a function. Using the mechanism described above, a new default implementation of the event is added to WatchState.c. This step protects existing code from changes; the only impact on the concrete states is on the states that intend to support the new event, which have to implement a function, of the correct signature, to handle it.
- *Adding a new state.* The new, concrete state has to implement functions for all events supported in that state. The only existing code that needs to be changed is the state in which we'll have a transition to the new state. Please note that the STATE pattern preserves one of the benefits of the table-based solution: client code, i.e. the context, remains unchanged.

Stateless States

The states in the sample code are stateless, i.e. the *WatchState* structure only contains pointers to re-entrant functions. Indeed, this is a special case of the STATE pattern described as “*If State objects have no instance variables [...] then contexts can share a State object*” [Gamma, et al]. However, before sharing any states, I would like to point to Joshua Kerievsky’s advice that “*it’s always best to add state-sharing code after your users experience system delays and a profiler points you to the state-instantiation code as a prime bottleneck*” [Kerievsky].

In the C language, states may be shared by declaring a static variable representing a certain state inside each function used as entry point upon a state transition. As the variables now have permanent storage, the signature of the transition functions is changed to return a pointer to the variable representing the particular state.

Listing 8: Stateless entry function, StartedState.c

```c
WatchStatePtr transitionToStarted(void)
{
    static struct WatchState startedState;
    static int initialized = 0;

    if(0 == initialized)
    {
        defaultImplementation(&startedState);
        startedState.stop = stopWatch;
        initialized = 1;
    }

    return &startedState;
}
```

The client code has to be changed from holding a variable representing the state to holding a pointer to the variable representing the shared state. Further, the context has to define a callback function to be invoked as the concrete states request a state transition.

Listing 9: Client code for changing state

```c
void changeState(DigitalStopWatchPtr instance,
                 WatchStatePtr newState)
{
    /* Provides a good place for controls and
      trace messages (all state transitions have to
      go through this function). */
    instance->state = newState;
}
```

The stateless state version comes closer to the State described in Design Patterns as a state transition, in contrast with the previous approach, implies changing the object pointed to by the context instead of just swapping its behavior.

Listing 10: State transition in StoppedState.c

```
static void startWatch(DigitalStopWatchPtr context)
{
    changeState(context, transitionToStarted());
}
```

A good quality of the stateless approach is that the point of state transitions is now centralized in the context. One obvious drawback is the need to pass around a reference to the context. This reference functions as a memory allowing the new state to be mapped to the correct context.

Another drawback is the care that has to be taken with the initialization of the static variables if the states are going to live in a multithreaded world.

Consequences

The main consequences of applying the STATE pattern are:

1. *Reduces duplication introduced by complex, state-altering conditional logic.* As illustrated in the example above, solutions based upon large segments of conditional logic tends to contain duplicated code. The STATE pattern provides an appealing alternative by removing the duplication and reducing the complexity.
2. *A clear expression of the intent.* The context delegates all state dependent operations to the state interface. Similar to the table-based solution, the STATE pattern lets the code reflect the intent of abstracting the problem as a finite state machine. With complex, conditional logic, that intent is typically less explicit.
3. *Encapsulates the behavior of each state.* Each concrete state provides a good overview of its behavior including all events supported in that very state. This encapsulation makes it easy both to identify as well as updating the relevant code when changes to a certain state are to be done.
4. *Implicit error handling.* The solutions based on conditional logic, as well as the table-based one, requires explicit code to ensure that a given combination of state and event is valid. Using the technique described above of initializing with a default implementation, the controls are built into the solution.
5. *Increases the number of compilation units.* The code typically becomes less compact with the STATE pattern. As Design Patterns says "*such distribution is actually good if there are many states*". However, for a trivial state machine with few, simple states, the STATE pattern may introduce an unnecessary complexity. In that case, if it isn't known that more complex behavior will be added, it is probably better to rely on conditional logic in case the logic will be easy to follow.

Summary

The STATE pattern lets us express a finite state machine, making the intent of the code clear. The behavior is partitioned on a per-state-basis and all state transitions are explicit.

The STATE pattern may serve as a valuable tool when implementing complex state-dependent behavior. On the other hand, for simple problems with few states, conditional logic is probably just right.

End of the free sample

Dear reader, you've reached the end of the free sample. I hope you like what you've read so far and found it valuable.

To continue reading, check out the complete book at: Patterns in C[5].

[5]https://leanpub.com/patternsinc