Patterns in C

By Adam Torn

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

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

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].

```
1  void strcpy(char *s, char *t)
2  {
3      while(*s++ = *t++);
4  }
```

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 FIRST-CLASS ADT Pattern

Reflections on FIRST-CLASS ADT

A common criticism of the book Design Patterns is that the patterns only apply to message-passing object oriented languages like C++ or Java. The patterns are seen as workarounds for absent language features. The need for patterns, the reasoning goes, is taken as an indication that the language is somehow less powerful.

There is indeed some truth to this statement. Design Patterns itself states it clearly: "the choice of programming language is important because it influences one's point of view" (Gamma, et al). The book goes on to state that its pattern catalogue assumes "Smalltalk/C++ level language features". If we go for a higher-level language than C++, like Common Lisp, we find that many design patterns don't apply; Common Lisp has a completely different design space. Many patterns are internalized in the language itself (like STRATEGY through first-class functions and VISITOR through the generic function model in Common Lisp's object system).

But a language's power always depend on context. There are problems where Common Lisp, despite its power, isn't necessarily the best fit. Likewise, there may be cases where C isn't the best choice. The reasons may be technical, strategical, social or cultural.

For us, the interesting part is considering patterns from a C horizon. Design Patterns notes that a procedural language might typically include patterns like encapsulation. First-Class ADT is exactly such a pattern.

To me, encapsulation is a tool for managing complexity. It's a way to separate aspects of a problem. With encapsulation we essentially get two views of a design: the external behavior and the internal details. Depending on my role I can view the design from any of this perspectives. In a successful design, these two views are orthogonal and may vary independently.

Like any other pattern, it is one to use with care and good taste. I wouldn't design all my modules as First-Class ADT. Rather, I often find the pattern useful when erecting boundaries between different layers in an application. The First-Class ADT may serve as a technical firewall between different development teams, allowing one of them to publish an interface while hiding the implementations details. Such a design encourages work on different, but related, parts to proceed in parallel.

Implementing the First-Class ADT Pattern

Our journey through the patterns will start with a language level pattern that decouples interface from implementation, thus improving encapsulation and providing loose dependencies.

This pattern will lay the foundation for many of the sub-sequent patterns in this book.

FIRST-CLASS ADT Pattern

It's getting close to the project deadline as the project manager rushes into your office. "They found some problem with your code", he says with a stressed voice. "According to the test-team, you cannot register more than 42 orders for a given customer. Sounds strange, doesn't it?"

Darn. You knew it. Those hardcoded limits. "Oh, I'll have a look at it", you reply in a soft voice. "Fine, I expect the problem to be solved tomorrow", the manager mumbles as he leaves you office.

"No problem", you reply, well confident that the design of the customer routines are highly modular and clearly implemented (after all, you've implemented it yourself).

You launch your favorite code-editor and open a file with the following content:

Listing 1 : Interface in Customer.h

```
/* Include guards and include files omitted. */
#define MAX_NO_OF_ORDERS 42

/* Internal representation of a customer. */
typedef struct
{
    const char* name;
    Address address;
    size_t noOfOrders;
    Order orders[MAX_NO_OF_ORDERS];
} Customer;

void initCustomer(Customer* theCustomer,
                  const char* name,
                  const Address* address);

void placeOrder(Customer* customer,
                const Order* order);
/* A lot of other related functions... */
```

A quick glance reveals the problem. Simply increasing *MAX_NO_OF_ORDERS* would do, wouldn't it? But what's the correct value for it? Is it 64, 128, maybe even 2048 or some other magic number? Should customers with one, single order allocate space for, let's say, 2047 non-existing orders?

As you think of it, you realize that the current solution doesn't scale well enough. Clearly, you need another algorithm. You recall that a linked list exists in the company's code library. A linked list must do the trick. However, this means changing the internal structure of the Customer.

No problem, it looks like you thought of everything; the clients of the customer module simply use the provided functions for all access of the customer structure. Updating those functions should be enough, shouldn't it?

Information hiding

Well, in an ideal world the change would be isolated to the one, single module. Given the interface above, clients depend upon the internal structure in at least one way.

At worst, the clients alter the internals of the data structure themselves leading to costly changes of all clients.

This can be prevented by frequent code-inspections and programmer discipline. In any case, we still have the compile-time dependencies and after changes, a re-compile of all clients is required and the compilation time may be significant in large systems.

The FIRST-CLASS ADT pattern will eliminate both dependency problems. The pattern provides us with a method of separating interface from implementation, thus achieving true information hiding.

Definition of a FIRST-CLASS ADT

ADT stands for Abstract Data Type and it is basically a set of values and operations on these values. The ADT is considered first-class if we can have many, unique instances of it.

Sounds close to the interface listed in the introductory example above, doesn't it? However, the data type in the example is not abstract as it fails to hide its implementation details. In order to make it truly abstract, we have to utilize a powerful feature of C – the ability to specify incomplete types.

Incomplete Types

The C standard [C 1999] allows us to declare objects of incomplete types in a context where there sizes aren't needed. In our example implementation, we are interested in one property of incomplete types – the possibility to specify a pointer to an incomplete type (please note that the pointer itself is not of an incomplete type).

Listing 2 : Pointer to an incomplete type

```
/* A pointer to an incomplete type (hides the
   implementation details). */
typedef struct Customer* CustomerPtr;
```

Instances of this pointer will serve as a handle for the clients of a FIRST-CLASS ADT. This mechanism enforces the constraint on clients to use the provided interface functions because there is no way a client can access a field in the Customer structure (the C language does not allow an incomplete type to be de-referenced).

The type is considered complete as soon as the compiler detects a subsequent specifier, with the same tag, and a declaration list containing the members.

Listing 3 : Completion of an incomplete type

```
/* The struct Customer is an incomplete type. */
typedef struct Customer* CustomerPtr;

/* Internal representation of a customer. */
struct Customer
{
    const char* name;
    Address address;
    size_t noOfOrders;
    Order orders[42];
};
/* At this point, struct Customer is considered complete. */
```

Object Lifetime

Before we dive into the implementation of an ADT, we need to consider object creation and destruction.

As clients only get a handle to the object, the responsibility for creating it rests upon the ADT. The straightforward approach is to write one function that encapsulates the allocation of an object and initializes it. In a similar way, we define a function for destructing the object.

Listing 4 : Interface to the ADT, Customer.h

```
/* Includes and include guards as before... */

/* A pointer to an incomplete type (hides the
   implementation details). */
typedef struct Customer* CustomerPtr;

/* Create a Customer and return a handle to it. */
CustomerPtr createCustomer(const char* name,
                           const Address* address);

```

```
/* Destroy the given Customer.
   All handles to it will be invalidated. */
void destroyCustomer(CustomerPtr customer);
```

Listing 5 : Implementation of the ADT in Customer.c

```
#include "Customer.h"
#include <stdlib.h>

struct Customer
{
    const char* name;
    Address address;
    size_t noOfOrders;
    Order orders[42];
};

CustomerPtr createCustomer(const char* name,
                           const Address* address)
{
     CustomerPtr customer = malloc(sizeof * customer);

     if(customer)
     {
          /* Initialize each field in the customer... */
     }
     return customer;
 }

void destroyCustomer(CustomerPtr customer)
{
    /* Perform clean-up of the customer internals,
       if necessary. */
    free(customer);
}
```

The example above uses **malloc** to obtain memory. In many embedded applications, this may not be an option. However, as we have encapsulated the memory allocation completely, we are free to choose another approach. In embedded programming, where the maximum number of needed resources is typically known, the simplest allocator then being an array.

Listing 6 : Example of a static approach to memory allocation

```
#define MAX_NO_OF_CUSTOMERS 42
static struct Customer objectPool[MAX_NO_OF_CUSTOMERS];
static size_t numberOfObjects = 0;

CustomerPtr createCustomer(const char* name,
                           const Address* adress)
{
    CustomerPtr customer = NULL;

    if(numberOfObjects < MAX_NO_OF_CUSTOMERS)
    {
        customer = &objectPool[numberOfObjects++];
       /* Initialize the object... */
    }
    return customer;
 }
```

In case de-allocation is needed, an array will still do, but a more sophisticated method for keeping track of "allocated" objects will be needed. However, such an algorithm is outside the scope of this book.

Copy Semantics

As clients only use a handle, which we have declared as a pointer, to the ADT, the issue of copy semantics boils down to pointer assignment. Whilst efficient, in terms of run-time performance, copies of a handle have to be managed properly; the handles are only valid as long as the real object exists.

In case we want to copy the real object, and thus create a new, unique instance of the ADT, we have to define an explicit copy operation.

Dependencies managed

With the interface above, the C language guarantees us that the internals of the data structure are encapsulated in the implementation with no possibility for clients to access the internals of the data structure.

Using the FIRST-CLASS ADT, the compile-time dependencies on internals are removed as well; all changes of the implementation are limited to, well, the implementation, just as it should be. As long as no functions are added or removed from the interface, the clients do not even have to be re-compiled.

Consequences

The main consequences of applying the FIRST-CLASS ADT pattern are:

1. *Improved encapsulation.* With the FIRST-CLASS ADT pattern we decouple interface and implementation, following the recommended principle of programming towards an interface, not an implementation.
2. *Loose coupling.* As illustrated above, all dependencies on the internals of the data structure are eliminated from client code.
3. *Controlled construction and destruction.* The FIRST-CLASS ADT pattern gives us full control over the construction and destruction of objects, providing us with the possibility to ensure that all objects are created in a valid state. Similarly, we can ensure proper de-allocation of all elements of the object, provided that client code behaves correctly and calls the defined destroy-function.

4. *Extra level of indirection.* There is a slight performance cost involved. Using the FIRST-CLASS ADT pattern implies one extra level of indirection on all operations on the data structure.
5. *Increased dynamic memory usage.* In problem domains where there may be potentially many instances of a quantity unknown at compile- time, a static allocation strategy cannot be used. As a consequence, the usage of dynamic memory tends to increase when applying the FIRST-CLASS ADT pattern.

Examples of use

The most prominent example comes from the C language itself or, to be more precise, from the C Standard Library – *FILE*. True, *FILE* isn't allowed by the standard to be an incomplete type and it may be possible to identify its structure, buried deep down in the standard library. However, the principle is the same since the internals of *FILE* are implementation specific and programs depending upon them are inherently non-portable.

Sedgewick[Sedgewick] uses FIRST-CLASS ADT to implement many fundamental data structures such as linked-lists and queues.

This pattern may prove useful for cross-platform development. For example, when developing applications for network communication, there are differences between Berkeley Sockets and the Winsock library. The FIRST-CLASS ADT pattern provides the tool for abstracting away those differences for clients. The trick is to provide two different implementations of the ADT, both sharing the same interface (i.e. include file).

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.

```
1  typedef enum
2  {
3      stopped,
4      started
5  } State;
6
7  struct DigitalStopWatch
8  {
9      /* Let a variable hold the state of our object. */
10     State state;
11     TimeSource source;
```

```
12      Display watchDisplay;
13  };
14
15  void startWatch(DigitalStopWatchPtr instance)
16  {
17      switch(instance->state)
18      {
19          case started:
20             /* Already started -> do nothing. */
21             break;
22          case stopped:
23             instance->state = started;
24             break;
25          default:
26             error("Illegal state");
27             break;
28      }
29  }
30
31  void stopWatch(DigitalStopWatchPtr instance)
32  {
33      switch(instance->state)
34      {
35          case started:
36              instance->state = stopped;
37              break;
38          case stopped:
39              /* Already stopped -> do nothing. */
40              break;
41          default:
42              error("Illegal state");
43              break;
44      }
45  }
```

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.

```
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.

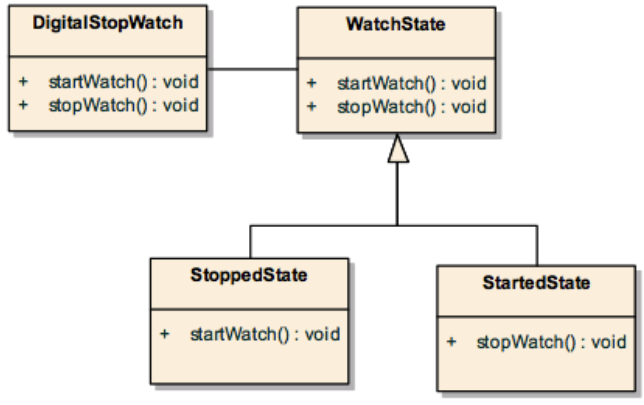

Illustration 1: Canonical structure of the STATE pattern.

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

```
/* 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

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

void defaultImplementation(WatchStatePtr state);
```

Listing 3: Provide the default implementations in WatchState.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

```
#include "WatchState.h"

void transitionToStopped(WatchStatePtr state);
```

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

```
#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

```
#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

```
#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.

```
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);
}
```

```
27
28 void startWatch(DigitalStopWatchPtr instance)
29 {
30     instance->state.start(&instance->state);
31 }
32
33 void stopWatch(DigitalStopWatchPtr instance)
34 {
35     instance->state.stop(&instance->state);
36 }
```

A Debug Aid

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

```
1 void transitionToStopped(WatchStatePtr state)
2 {
3     defaultImplementation(state);
4     state->name = "Stopped";
5     state->start = startWatch;
6 }
```

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:

```
1 static void defaultStop(WatchStatePtr state)
2 {
3     /* We'll get here if the stop event isn't supported
4        in the concrete state. */
5     logUnsupportedEvent("Stop event", state->name);
6 }
```

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

```
1  WatchStatePtr transitionToStarted(void)
2  {
3      static struct WatchState startedState;
4      static int initialized = 0;
5
6      if(0 == initialized)
7      {
8          defaultImplementation(&startedState);
9          startedState.stop = stopWatch;
10         initialized = 1;
11     }
12
13     return &startedState;
14 }
```

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

```
1  void changeState(DigitalStopWatchPtr instance,
2                   WatchStatePtr newState)
3  {
4      /* Provides a good place for controls and
5        trace messages (all state transitions have to
6        go through this function). */
7      instance->state = newState;
8  }
```

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.

The STRATEGY Pattern

Reflections on STRATEGY

Of all the design patterns in this book, STRATEGY is the one I've been using the most. The pattern is fundamental. It's a shift in mindset. Applying STRATEGY shifts the focus of an algorithm from _how to *what. Rather than trying to control how a function performs its task we parameterize it with a high-level behavior encapsulating what to do.*

Within a software design, excessive coupling is one of the most damaging properties of a program. And the worst form of coupling is control coupling. In the degenerate case of control coupling, a client passes a flag to some function to control its behavior. It's a common coding style that leads to excessive conditional logic. Excessive conditional logic implies complexity in itself. But it gets worse.

Control coupling, as discussed in my original article, breaks the encapsulation. Suddenly, the client is exposed to the inner workings of a function its using. Barriers are teared down, modules melt together. Such code is often hard to extend. It's also an invitation to bugs. The hardest bugs I've found are when different features interact. That is precisely the case with the control coupling approach. Instead, a good design separates different features and variations. It lets them vary independently (features often have different change rates). Good design is about orthogonality.

STRATEGY isn't an object-oriented pattern. One of the main problems I have with the Gang of Four book is its prominent use of class diagrams. Open any pattern in the book. First thing we see is a class diagram. Often, a similar diagram is repeated in the "Structure" section. This has lead many developers to confuse the diagram with the actual pattern. It's not. At best, it's one possible way to implement the pattern in a certain language. Nothing more. A pattern is a dynamic, generative entity. Depending on context, the applications of a pattern may look radically different each time. It's my firm belief that much of the harm done to the design patterns movement could have been avoided had the Gang of Four just excluded the section on structure.

Implementing STRATEGY

Identifying and exploiting commonality is fundamental to software design. By encapsulating and re-using common functionality, the quality of the design rises above code duplication and dreaded anti-patterns like copy-paste. This chapter dives into a design pattern that adds flexibility to common software entities by letting clients customize and extend them without modifying existing code.

Control coupled customers

To attract and keep customers, many companies offer some kind of bonus system. In our example, a customer is placed in either of three categories:

- *Bronze customers*: Get 2 % reduction on every item bought.
- *Silver customers*: Get 5 % reduction on every item bought.
- *Gold customers*: Get 10 % off on all items and free shipping.

A simple and straightforward way to implement a price calculation based upon this bonus system is through conditional logic.

```
typedef enum
{
    bronzeCustomer,
    silverCustomer,
    goldCustomer
} CustomerCategory;

double calculatePrice(CustomerCategory category,
                      double totalAmount,
                      double shipping)
{
    double price = 0;

    /* Calculate the total price depending on
      customer category. */
    switch(category) {
        case bronzeCustomer:
            price = totalAmount * 0.98 + shipping;
            break;
        case silverCustomer:
            price = totalAmount * 0.95 + shipping;
            break;
        case goldCustomer:
            /* Free shipping for gold customers.*/
            price = totalAmount * 0.90;
            break;
        default:
            onError("Unsupported category");
            break;
    }
    return price;
}
```

Before further inspection of the design, I would like to point out that representing currency as *double* may lead to marginally inaccurate results. Carelessly handled, they may turn out to be fatal to, for example, a banking system. Further, security issues like a possible overflow should never be ignored in business code. However, such issues have been deliberately left out of the example in order not to lose the focus on the problem in the scope of this book.

Returning to the code, even in this simplified example, it is possible to identify three serious design problems with the approach, that wouldn't stand up well in the real-world:

1. *Conditional logic.* The solution above basically switches on a type code, leading to the risk of introducing a maintenance challenge. For example, the above mentioned security issues have to be addressed on more than one branch leading to potentially complicated, nested conditional logic.
2. *Coupling.* The client of the function above passes a flag (category) used to control the inner logic of the function. Page-Jones defines this kind of coupling as "control coupling" and he concludes that control coupling itself isn't the problem, but that it "*often indicates the presence of other, more gruesome, design ills*" [Page-Jones]. One such design ill is the loss of encapsulation; the client of the function above knows about its internals. That is, the knowledge of the different customer categories is spread among several modules.

3. *It doesn't scale.* As a consequence of the design ills identified above, the solution has a scalability problem. In case the customer categories are extended, the inflexibility of the design forces modification to the function above. Modifying existing code is often an error-prone activity.

The potential problems listed above, may be derived from the failure of adhering to one, important design principle.

The Open-Closed Principle

Although mainly seen in the context of object oriented literature, the open-closed principle defines properties attractive in the context of C too. The principle is summarized as "*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*"[Martin].

According to the open-closed principle, extending the behavior of an ideal module is achieved by adding code instead of changing the existing source. Following this principle not only minimizes the risk of introducing bugs in existing, tested code but also typically raises the quality of the design by introducing loose coupling. Unfortunately, it is virtually impossible to design a module in such a way that it is closed against all kinds of changes. Even trying to design software in such a way would overcomplicate the design far beyond suitability. Identifying the modules to close, and the changes to close them against, requires experience and a good understanding of the problem domain.

In the example used in this chapter, it would be suitable to close the customer module against changes to the customer categories. Identifying a pattern that lets us redesign the code above in this respect seems like an attractive idea.

STRATEGY

The design pattern STRATEGY provides a way to follow and reap the benefits of the open-closed principle. Design Patterns [Gamma, et al] defines the intent of STRATEGY as "*Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it*". Related to the discussion above, the price calculations in the different customer categories are that "family of algorithms". By applying the STRATEGY pattern, each one of them gets fully encapsulated rendering the structure in Illustration 1.

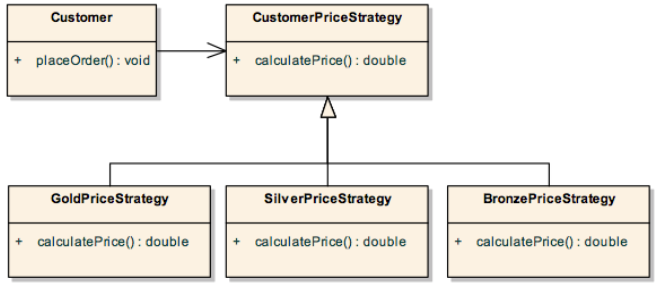

Illustration 1: The customer bonus system as a canonical STRATEGY.

The context, in this example the *Customer*, does not have any knowledge about the concrete categories anymore; the concrete strategy is typically assigned to the context by the application. All price calculations are delegated to the assigned strategy.

Using this pattern, the interface for calculating price adheres to the open closed principle; it is possible to add or remove categories without modifying the existing calculation algorithms or the *Customer* itself.

Implementation Mechanism

When implementing the STRATEGY pattern in C, without language support for polymorphism and inheritance, an alternative to the object oriented features has to be found for the abstraction. This problem is almost identical to the one faced when implementing the STATE pattern in C, indicating that the same mechanism may be used, namely pointers to functions. The possibility to specify pointers to functions proves to be an efficient technique for implementing dynamic behavior.

In a C-implementation, the basic structure in the diagram above remains, but the interface is expressed as a declaration of a pointer to function.

Listing 2: Strategy interface in CustomerStrategy.h

```
typedef double (*CustomerPriceStrategy)(double amount,
                                        double shipping);
```

The different strategies are realized as functions following the signature specified by the interface. The privileges of each customer category, the concept that varies, are now encapsulated within their concrete strategy. Depending on the complexity and the cohesion of the concrete strategies, the source code may be organized as either one strategy per compilation unit or by gathering all supported strategies in a single compilation unit. For the simple strategies used in this example, the latter approach has been chosen.

Listing 3: Interface to the concrete customer categories in CustomerCategories.h

```
double bronzePriceStrategy(double amount, double shipping);
double silverPriceStrategy(double amount, double shipping);
double goldPriceStrategy(double amount, double shipping);
```

Listing 4: Implementation of the concrete customer strategies in CustomerCategories.c

```
/* In production code, all input should be validated and
   the calculations secured upon entry of each function. */
double bronzePriceStrategy(double amount, double shipping)
{
    return amount * 0.98 + shipping;
}

double silverPriceStrategy(double amount, double shipping)
{
    return amount * 0.95 + shipping;
}

double goldPriceStrategy(double amount, double shipping)
{
    /* Free shipping for gold customers. */
    return amount * 0.90;
}
```

Using the strategies, the context code now delegates the calculation to the strategy associated with the customer.

Listing 5: Delegation to a strategy, Customer.c

```
#include "CustomerStrategy.h"
/* Other include files omitted... */

struct Customer
{
    const char* name;
    Address address;
    List orders;
   /* Bind the strategy to Customer: */
   CustomerPriceStrategy priceStrategy;
};

void placeOrder(struct Customer* customer, const Order* order)
{
    double totalAmount = customer->priceStrategy(order->amount,
                                                  order->shipping);
    /* More code to process the order... */
}
```

The code above solves the detected design ills. As the customer now only depends upon the strategy interface, categories can be added or removed without changing the code of the customer and without the risk of introducing bugs into existing strategies. The remaining open issue with the implementation is to specify how a certain strategy gets bound to the customer.

Binding the Strategy

The strategy may be supplied by the client of the customer and bound to the customer during its creation, or an initial strategy may be chosen by the customer itself. The former alternative is clearly more flexible as it avoids the need for the customer to depend upon a concrete strategy. The code below illustrates this approach, using a customer implemented as a FIRST-CLASS ADT.

Listing 6: Binding strategy upon creation, Customer.c

```
CustomerPtr createCustomer(const char* name,
                           const Address* address,
                           CustomerPriceStrategy priceStrategy)
{
    CustomerPtr customer = malloc(sizeof *customer);

    if(NULL != customer)
    {
        /* Bind the initial strategy supplied by
           the client. */
        customer->priceStrategy = priceStrategy;

       /* Initialize the other attributes of the
          customer here. */
    }
    return customer;
}
```

Listing 7: Client code specifying the binding

```
#include "Customer.h"
#include "CustomerCategories.h"

static CustomerPtr createBronzeCustomer(const char* name,
                                        const Address* address)
{
    return createCustomer(name, address, bronzePriceStrategy);
}
```

Depending on the problem at hand, a context may be re-bound to another strategy. For example, as a customer gets promoted to the silver category, that customer should get associated with the **silverPriceStrategy**. Using the technique of pointers to functions, a run-time change of strategy simply implies pointing to another function.

Listing 8: Rebinding a strategy, Customer.c

```
void changePriceCategory(CustomerPtr customer,
                         CustomerPriceStrategy newPriceStrategy)
{
    assert(NULL != customer);
    customer->priceStrategy = newPriceStrategy;
}
```

Yet another alternative is to avoid the binding altogether and let the client pass the different strategies to the context in each function invocation. This alternative may be suitable in case the context doesn't have any state memory. However, for our example, which uses first-class objects, the opposite is true and the natural abstraction is to associate the customer with a price-strategy as illustrated above.

Comparison of STRATEGY and STATE

When discussing the STRATEGY pattern, its relationship with the pattern preceding it in the Design Patterns [Gamma, et al] book deserves special mention. The design patterns STATE and STRATEGY are closely related. Robert C. Martin puts it this way: "*all instances of the State pattern are also instances of the Strategy pattern, but not all instances of Strategy are State*" [Martin].

This observation leads us to a recommendation by John Vlissides, co-author of Design Patterns, stating that "*Let the intents of the patterns be your guide to their differences and not the class structures that implement them*" [Vlissides]. And indeed, even though STATE and STRATEGY have a similar structure and use similar mechanisms, they differ in their intent. The STATE pattern focuses upon managing well-defined transitions between discrete states, whereas the primary purpose with STRATEGY is to vary the implementation of an algorithm.

Related to the example above, had the categories been based on the sum of money spent by a customer, STATE would have been a better choice; the categories would basically illustrate the lifecycle of a customer, as the transitions between categories depend upon the history of the customer object itself. The STATE pattern may be used to implement this behavior as a finite state machine.

On the other hand, in case the placement in a certain category is the result of a membership fee, STRATEGY is a better abstraction. It is still possible for a customer to wander between different categories, but a transition to a new category doesn't depend upon the history of that customer object.

Although not to be taken as a universal truth, a further observation relates to the usage of these two patterns and their relationships towards the client. STATE tends to be an internal concern of the context and the existence of STATE is usually encapsulated from the client. Contrasting this with STRATEGY, the client usually binds a concrete strategy to its context.

Consequences

The main consequences of applying the STRATEGY pattern are:

1. *The benefits of the open-closed principle.* The design pattern STRATEGY offers great flexibility in that it allows clients to change and control the behavior of an existing module by implementing their own, concrete strategies. Thus, new behavior is supported by adding new code instead of modifying existing code.
2. *Reduces complex, conditional logic.* Complex interactions may lead to monolithic modules littered with conditional logic, potentially in the form of control coupling. Such code tends to be hard to maintain and extend. By encapsulating each branch of the conditionals in a strategy, the STRATEGY pattern eliminates conditional logic.
3. *Allows different versions of the same algorithm.* The non-functional requirements on a module may typically vary depending on its usage. The typical trade-off between an algorithm optimized for speed versus one optimized with respect to memory usage is classical. Using the Strategy pattern, the choice of trade-offs may be delegated to the user ("*Strategies can provide different implementations of the same behavior*" [Gamma, et al]).
4. *An extra level of indirection.* The main issue with this consequence arises as data from the context has to be obtained in a strategy. As all functions used as strategies have to follow the same signature, simply adding potentially unrelated parameters lowers the cohesion. Implementing the context as a FIRST-CLASS ADT may solve this problem as it reduces the parameters to a single handle. With the FIRST-CLASS ADT approach, the knowledge about the data of interest is encapsulated within each strategy and obtained through the handle to the FIRST-CLASS ADT. A careful design should strive to keep the module implemented as a FIRST-CLASS ADT highly cohesive and avoid having it decay into a repository of unrelated data needed by different strategies (such a design ill typically indicates that the wrong abstraction has been chosen). Similarly, in case the flexibility of the STRATEGY pattern isn't needed and the problem may be solved by conditional logic that is easy to follow, the latter is probably a better choice.

Example of use

STRATEGY may prove useful for specifying policies in framework design. Using STRATEGY, clients may parameterize the implementation.

For example, error-handling is typically delegated to the application. Such a design may be realized by letting the client provide a strategy to be invoked upon an error. By those means, the error may be handled in an application specific manner, which may stretch between simply ignoring the error to logging it or even reboot the system.

The OBSERVER Pattern

Reflections on OBSERVER

As I wrote the original patterns articles I based the implementations on actual experience from production code. Over the precious years I had collected notes and sample implementations of several well-known patterns. My initial plan was to publish five articles, starting with the more general patterns first. That left me in the position where I had to choose and prioritize between the patterns. The first three (First-Class ADT, STATE and STRATEGY) were all given. And I knew I wanted to finish with a discussion of the REACTOR pattern. Clearly, some patterns had to go.

I finally selected OBSERVER. In retrospect, other patterns would probably have made a better candidate. The FACADE [Gamma, et al], for example, is straightforward and natural in C. In the end I went with the OBSERVER for two reasons: preparation and generality.

First, the implementation mechanisms I used in the OBSERVER are similar to the ones I tend to use for the REACTOR. As such, the OBSERVER implementation could serve as a gentle introduction to the more complex architectural REACTOR pattern.

Second, event notification is a central part of many applications. Decoupling the sender and receiver of a certain notification is crucial. Today, many programming languages have come to include native support for OBSERVER. C#, for example, provides a general version with its events mechanism. And of course Common Lisp has had it forever through the :before and :after methods that make OBSERVER implementations both trivial and expressive. In C, we have to go through more challenges to arrive at an implementation that fits. Done right, observers can still be a viable solution.

Implementing OBSERVER

Managing dependencies between entities in a software system is crucial to a solid design. In the previous chapter we had a look at the open-closed principle. Now we'll turn to another principle for dependency management and illustrate how both of these principles may be realized in C using the OBSERVER pattern.

Dependencies arise

Returning to the examples used for the STATE pattern, they described techniques for implementing the behavior of a simple digital stop-watch. Implementing such a watch typically involves the task of fetching the time from some kind of time-source. As the time-source probably will be tied to interactions with the operating system, it is a good idea to encapsulate it in an abstraction hiding the system specific parts in order to ease unit testing and portability. Similarly, the internals of the digital stop-watch should be encapsulated in a module of its own. As the digital stop-watch is responsible for fetching the time from the time-source (in order to present it on its digital display), it stays clear that there will be a dependency between the watch and the time-source. There are two obvious choices for the direction of this dependency.

Consider the Watch as a Client

It may seem rather natural to consider the watch as a client of the time-source. That is, letting the watch depend upon the time-source. Unfortunately, implementing the dependency in this direction introduces one obvious problem: how does the watch know if the time changes? The quick answer is: it doesn't. At least it doesn't unless it introduces some kind of polling mechanism towards the time-source. Just as important as it is to avoid premature optimization, one should also strive to avoid premature pessimization;

even if the direction of the dependency seems correct, this solution is very likely to be extremely CPU consuming. In case the application needs to do more than updating a display, the problem calls for another solution.

Let the Time-Source update the Watch

The potential capacity problem described above may easily be avoided by reversing the dependency. The time-source may simply notify the watch as soon as its time changes. This approach introduces a dependency from the time-source upon the watch.

Listing 1: Code for the time-source

```
#include "DigitalStopWatch.h"
#include "SystemTime.h"

static DigitalStopWatchPtr digitalWatch;
static SystemTime currentTime;

/* Invoked once by the application at start-up. */
void startTimeSource()
{
    digitalWatch = createWatch();

    /* Code for setting up handlers for interrupts, or
       the like, in order to get notified each millisecond
       from the operating system. */
}

/* This function is invoked each millisecond through an
   interaction with the operating system. */
static void msTick()
{
    /* Invoke a function encapsulating the knowledge about
       time representation. */
    currentTime = calculateNewTime();

    /* Inform the watch that another millisecond passed. */
    notifyChangedTime(digitalWatch, &currentTime);
}
```

The attractiveness of this approach lies in its simplicity. However, if scalability and flexibility are desired properties of the solution, the trade-offs are unacceptable. The potential problems introduced are best described in terms of the principles that this design violates.

The Open-Closed Principle

Having a quick recap on the open-closed principle, it is summarized as "*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*" [Martin]. The code for the time-source above clearly violates this principle. In its current shape it supports only a single watch of one type. Imagine supporting other types of watches, for example one with an analogue display. The code for the time-source would, due to its hard-coded notification, explode with dependencies in all directions on all types of watches.

The Stable Dependencies Principle

During software maintenance or incremental development changes to existing code are normally unavoidable; even when applying the open-closed principle the design is just closed against certain modifications based upon assumptions by the original designer (it is virtually impossible for a software entity to be completely closed against all kinds of changes). The stable dependencies principle tries to isolate the impact of changes to existing code by making the recommendation that software entities should "*depend in the direction of stability*" [Martin].

In the initial approach, the watch itself fetched the time from the time-source. With respect to the stable dependencies principle, this was a better choice. A time-source is typically a highly cohesive unit and the more stable entity; simply encapsulating it in a proper abstraction, which hides the system specific details, makes it a good candidate to be packaged in a re-usable lower-level domain layer.

By having the time-source depend upon the higher-level digital watch, we violate the stable dependencies principle. This violation manifests itself by making the code for the watch difficult to update. Changes may actually have impact upon the code of the time-source itself!

Combining the dependency direction of the first approach with the notification mechanism of the second would make up an ideal design and the design pattern OBSERVER provides the extra level of indirection necessary to achieve the benefits of both solutions without suffering from any of their drawbacks.

OBSERVER

The OBSERVER pattern may serve as a tool for making a design follow the open-closed principle. Design Patterns [Gamma, et al] captures the intent of OBSERVER as "*Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically*". Applying the OBSERVER pattern to our example, extended with a second type of watch, the "dependents" are the different watches. Design Patterns defines the role played by *TimeSource* as a "concrete subject", the object whose state changes should trigger a notification and update of the dependents, i.e. Observers.

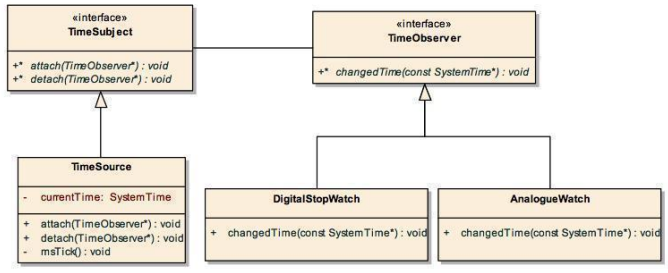

Illustration 1: Decouple the sender and receivers through OBSERVER.

Implementation Mechanism

In order to decouple the subject from its concrete observers and still enable the subject to notify them, each observer must correspond to a unique instance. The FIRST-CLASS ADT pattern provides a way to realize this. However, as seen from the subject, all observers must be abstracted as one, general type sharing a common interface. In the C language, without language support for inheritance, generality is usually spelled *void**; any pointer to an object may be converted to *void** and back to its original type again without any loss of information. This rule makes a common interface possible.

Listing 2: Interface of the observers, TimeObserver.h

```
typedef void (*ChangeTimeNotification)(void* instance,
                            const SystemTime* newTime);
typedef struct
{
    void* instance;
    ChangeTimeNotification notification;
} TimeObserver;
```

The interface above declares a pointer to a function, which proves to be an efficient technique for implementing dynamic behavior; a concrete observer attaches itself, together with a pointer to a function, at the subject. As the subject changes, it notifies the observer through the attached function.

By using *void** as abstraction, type-safety is basically traded for flexibility; the responsibility for the conversion of the observer instance back to its original type lies on the programmer. A conversion back to another type than the original may have disastrous consequences. Franz Kafka, although not very experienced in C programming, provided a good example of such behavior: "*When Gregor Samsa woke up one morning from unsettling dreams, he found himself changed in his bed into a monstrous vermin*" [Kafka]. Obviously, Gregor Samsa wasn't type-safe. In order to guard against such erroneous conversions, the *TimeObserver* structure has been introduced to maintain the binding between the observer and the function pointer. The implicit type conversion is encapsulated within the observer itself, as illustrated in the code below.

Listing 3: A concrete observer implemented as a FIRST-CLASS ADT

```
/* Include files omitted. */
struct DigitalStopWatch
{
    Display watchDisplay;
    /* Other attributes of the watch, e.g.
       digital display. */
};

/* Implementation of the function required by the
   TimeObserver interface. */
static void changedTime(void* instance,
                        const SystemTime* newTime)
{
    DigitalStopWatchPtr digitalWatch = instance;
    assert(NULL != digitalWatch);

    updateDisplay(digitalWatch, newTime);
}
```

Before an observer can get any notifications, it has to register for them. Listing 4 declares the interface required by the observers.

Listing 4: Interface to the subject, TimeSubject.h

```
#include "TimeObserver.h"

void attach(const TimeObserver* observer);

void detach(const TimeObserver* observer);
```

These functions are implemented by the concrete subject, the *TimeSource*, which has to keep track of all attached observers. The example below uses a linked-list for that task. Attaching an observer corresponds to adding a copy of the given *TimeObserver* representation to the list. Similarly, upon a call to *detach*, the node in the list corresponding to the given *TimeObserver* shall be removed and that observer will no longer receive any notifications from the subject. The code fragments below illustrate this mechanism.

Listing 5: Implementation of the subject, TimeSource.c

```
#include "TimeSubject.h"

struct ListNode
{
    TimeObserver item;
    struct ListNode* next;
};

static struct ListNode observers;
static SystemTime currentTime;

/* Local helper functions for managing the linked-list
  (implementation omitted). */
static void appendToList(const TimeObserver* observer)
{
    /* Append a copy of the observer to the linked-list. */
}

static void removeFromList(const TimeObserver* observer)
{
    /* Identify the observer in the linked-list and
       remove that node. */
}

/* Implementation of the TimeSubject interface. */
void attach(const TimeObserver* observer)
{
    assert(NULL != observer);
    appendToList(observer);
}

void detach(const TimeObserver* observer)
{
    assert(NULL != observer);
    removeFromList(observer);
}
```

```
37
38 /* Implementation of the original responsibility of the
39    TimeSource (code for initialization, etc omitted). */
40 static void msTick()
41 {
42     struct ListNode* node = observers.next;
43
44     /* Invoke a function encapsulating the knowledge
45        about time representation. */
46     currentTime = calculateNewTime();
47
48     /* Walk through the linked-list and notify every
49        observer that another millisecond passed. */
50     while(NULL != node)
51     {
52         TimeObserver* observer = &node->item;
53         observer->notification(observer->instance,
54                                &currentTime);
55         node = node->next;
56     }
57 }
```

The code above solves our initial problems; new types of watches may be added without any modification to the *TimeSource*, yet these watches, our observers, are updated in an efficient way without the need for expensive polling.

Observer Registration

An additional benefit of implementing the OBSERVER pattern as a FIRST-CLASS ADT is the combination of loose dependencies with information hiding. The client neither knows nor depends upon a subject. In fact, the client doesn't even know that the *DigitalStopWatch* acts as an observer because the functions for creating and destructing the ADT encapsulate the registration handling. The code below, which extends Listing 3, illustrates this technique:

Listing 6: Encapsulation of the registration handling

```
1  static void changedTime(void* instance,
2                          const SystemTime* newTime)
3  {
4      /* Implementation as before (Listing 3). */
5  }
6
7  DigitalStopWatchPtr createDigitalWatch(void)
8  {
9      DigitalStopWatchPtr watch = malloc(sizeof *watch);
10
11     if(NULL != watch)
12     {
13         /* Successfully created -> attach to
14            the subject. */
15         TimeObserver observer = {0};
```

```
16          observer.instance = watch;
17          observer.notification = changedTime;
18
19          attach(&observer);
20      }
21      return watch;
22  }
23
24  void destroyDigitalWatch(DigitalStopWatchPtr watch)
25  {
26      /* Before deleting the instance we have to detach
27         from the subject. */
28      TimeObserver observer = {0};
29      observer.instance = watch;
30      observer.notification = changedTime;
31
32      detach(&observer);
33      free(watch);
34  }
```

Subject – Observer Dependencies

The introduction of the OBSERVER pattern results in loose dependencies between the subject and its observers. However, no matter how loose, the dependencies cannot be ignored and an often overseen aspect of the OBSERVER pattern is to ensure correct behavior in case of changed registrations during a notification of the observers. The problem should be addressed in all OBSERVER implementations and is illustrated by investigating the notification loop coded earlier:

Listing 7: Code extracted from Listing 5

```
1  /* Walk through the linked-list and notify every
2     observer that another millisecond passed. */
3  while(NULL != node)
4  {
5      TimeObserver* observer = &node->item;
6      observer->notification(observer->instance,
7                             &currentTime);
8      node = node->next;
9  }
```

In case an observer decides to detach itself during a notification, the list containing the nodes may become corrupted. The solutions span between forbidding subject changes during notification and, at the other extreme, allow the subject to change and ensure it works.

The solution with forbidding subject changes during notification calls for a well-documented Subject interface. Further, the constraint may be checked at run-time using assertions.

Listing 8: Checking Subject constraints with assertions

```
static int isNotifying = 0;

void attach(const TimeObserver* observer)
{
    assert(0 == isNotifying);
   /* Code as before. */
}

void detach(const TimeObserver* observer)
{
    assert(0 == isNotifying);
    /* Code as before. */
}

static void msTick()
{
    struct ListNode* node = observers.next;

    /* Ensure that no changes are done to the
       subject during notification. */
    isNotifying = 1;

    while(NULL != node)
    {
        /* Loop through the observers as before. */
    }

    /* All observers notified, allow changes again. */
    isNotifying = 0;
}
```

The solution at the other side of the spectrum depends upon the actual data structure used to store the observers. The idea is to keep a pointer to the next observer to notify at file-scope. Attaching or detaching an observer now involves the possible adjustment of that pointer. By exclusively using this pointer in the notification-loop, the problem with unregistrations is solved. By choosing a strategy for new registrations, defining if the new observers are to be notified during the loop where they attach or not, the solution is complete. The extra complexity is rewarded with the flexibility of allowing observers to be added or removed during notification.

One pattern, two models

In the example above, the part of the subject that changed (in our example the system time) was given to the observers as an argument to the notification-function. This technique is known as the push-model. The advantage of this model is its simplicity and efficiency; the data that changed is immediately available to the observers in the notification. A potential problem is the logical coupling between a subject and its observers; in order to deliver the correct data, the subject has to know about the needs of its observers.

This potential problem is eliminated by the other model used in OBSERVER implementations. This model, known as the pull-model, does not send any detailed information about what changed at the subject; the observers have to fetch that data themselves from the subject. In case the subject contains several large data structures, the efficiency of the pull-model may be improved by introducing an update

protocol. Such a protocol specifies what changed while still putting the responsibility on the observers to fetch the actual data. A simple *enum* may serve well as an update protocol.

Consequences

The main consequences of applying the OBSERVER pattern are:

1. *Introduces loose dependencies.* As the subject only knows its observers through the Observer interface, the code conforms to the open- closed principle; by avoiding hard-coded notifications, any number and any types of observers may be introduced as long as they support the Observer interface. New behavior, in the form of new types of observers, is added without modifying existing code. The loose dependencies provide a way to communicate between layers in a sub-system. Design Patterns [Gamma, et al] recognizes this potential: "*Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system. A lower-level subject can communicate and inform a higher-level observer, thereby keeping the system's layering intact*". This property may serve as a tool for minimizing the impact of modifications by following the stable dependencies principle and yet enable a bidirectional communication between layers.
2. *Potentially complex management of object lifetimes.* As illustrated above, the FIRST-CLASS ADT pattern simplifies the management by encapsulating the interaction with the subject. However, in case the subject is also implemented as a first-class object, the dependencies have to be resolved on a higher level and the client has to ensure that the subject exists as long as there are observers attached to it.
3. *May complicate a design with cascades of notifications.* In case an observer plays a second role as subject for other observers, a notification may result in further updates of these observers leading to overly complex interactions. Another problem may arise if an observer, during its update, modifies the subject resulting in a new cascade of notifications.
4. *Lowers the cohesion of the subject.* Besides serving its central purpose (in our example being a time-source) a subject also takes on the responsibility of managing and notifying observers. By merging two responsibilities in one module, the complexity of the subject is increased. This extra complexity is acceptable in case the loose dependencies, gained by introducing the OBSERVER pattern, provide significant benefits. Further, the subject may be simplified in cases where, during the life of the subject, there isn't any need to detach observers. In such a case, the *detach* function is simply omitted.
5. *Trades type-safety for flexibility. _The gist of the OBSERVER pattern is that the subject should be able to notify its dependents without making any assumptions about who they are. I.e. it must be possible to have observers of different types. The solution here uses _void** as the common abstraction of an observer. The potential problem arises as the subject notifies its observers and passes them as *void** to the notification functions. When converting a void-pointer back to a pointer of a concrete observer type, the compiler doesn't have any way to detect an erroneous conversion. This problem may be prevented by defining a unique notification function for each different type of observer in combination with using a binding such as the *TimeObserver* structure introduced above.

Summary

This chapter illustrates how the loose coupling introduced by the OBSERVER pattern may serve as a way to implement modules following the open- closed principle. We can add new observers without modifying the subject. In fact, observers may even be replaced at runtime.

Further, as OBSERVER reverses the dependencies it allows lower-level modules to notify modules at a higher abstraction level without compromising layering. This property makes it possible to implement

modules following the stable dependencies principle in cases where the lower layers in a system are the more stable and yet need to communicate with the higher layers.

However, when hard-coded notifications are enough, by all means stick to them; in case the loose coupling and extra level of indirection isn't needed, the OBSERVER pattern may just overcomplicate a design.

The REACTOR Pattern

Reflections on REACTOR

So far the patterns in this book have addressed general, recurring design problems. The kind of problems present in most non-trivial programs. The REACTOR pattern is more limited in its scope. It's a pattern that operates on the architectural level of design. The main reason I included it was due to the elegant separations of concerns in the REACTOR. As such, the pattern is a nice example on good design principles that have value in themselves.

Every design decision involves trade-offs. The REACTOR pattern is no exception. It sure is a non-trivial pattern to implement. Yet, when used in the right context, a REACTOR provides an elegant programming model. Over the years I've worked on two systems where the REACTOR proved to be a perfect fit. These were both systems running on hardware with limited capabilities. Both systems received stimuli from multiple sources. The systems were based on asynchronous messages sent between independent Linux processes. The messages were distributed through a third-party message-oriented middleware. At the same time, many processes had to interface external systems over the network. By providing the REACTOR as a service through a library, a great deal of work could be accomplished in a relative short amount of programming time since the communication challenges were solved. Much of the messaging, independent of the type of receiver, could be encapsulated. When supporting a new feature, the application code could focus on the actual problem and rely on the REACTOR implementation to route the I/O.

The REACTOR is a pattern for synchronous I/O. Over the last years, asynchronous I/O has become much more common. In part because asynchronous I/O provides an opportunity for programs to take advantage of multi-core machines. If you happen to work in that context, I recommend you to have a look at a close relative to the REACTOR – the PROACTOR pattern. A PROACTOR specifically targets asynchronous read and write operations.

Implementing REACTOR

In this chapter we'll step outside the domain of standard C and investigate a pattern for event-driven applications. The REACTOR pattern decouples different responsibilities and allows applications to demultiplex and dispatch events from potentially many clients.

The case of many Clients

In order to simplify maintenance of large systems, the diagnostics of the individual subsystems are gathered in one, central unit. Each subsystem connects to the diagnostics server using TCP/IP. As TCP/IP is a connection-oriented protocol, the clients (the different subsystems) have to request a connection at the server. Once a connection is established, a client may send diagnostics messages at any time.

The brute-force approach is to scan for connection requests and diagnostics messages from the clients one by one as illustrated in the activity diagram in Illustration 1.

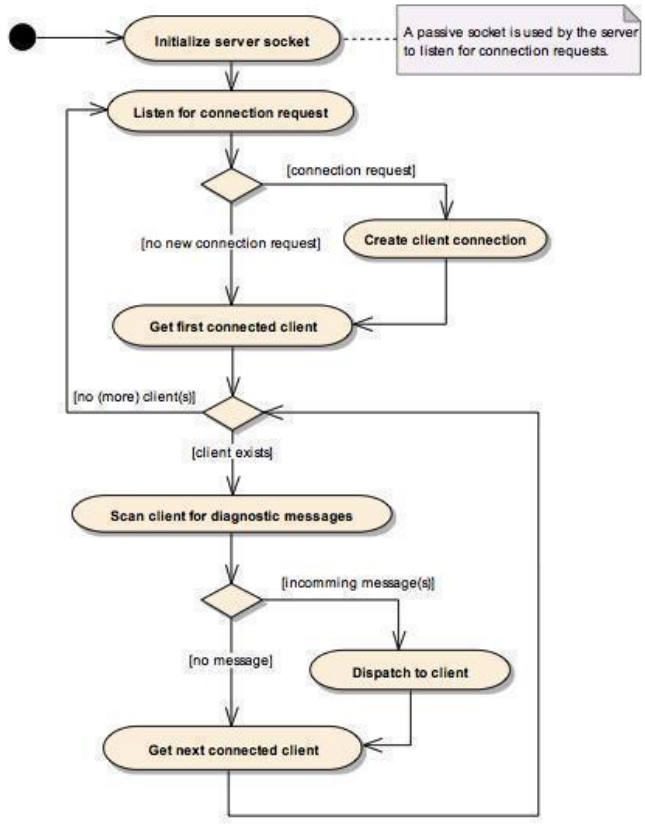

Illustration 1: An eternal loop to scan for events.

Even in this strongly simplified example, there are several potential problems. By intertwining the application logic with the networking code and the code for event dispatching, several unrelated responsibilities have been built into one module. Such a design is likely to lead to serious maintenance, testability, and scalability problems by violating a fundamental design principle.

The Single Responsibility Principle

The single responsibility principle states that "*a class should have only one reason to change*" [Martin]. The goal of this principle is close to that of the open-closed principle treated below: both strive to protect existing code from modifications. When violating the single responsibility principle, a module gets more reasons to change and modifications to it become more likely. Worse, the different responsibilities absorbed by a single module may become coupled and interact with each other making modifications and testing of the module more complicated.

The single responsibility principle is basically about cohesion. It is useful and valuable on many levels of abstraction, not at least in a procedural context; simply replacing the word "class" with "function" enables us to analyze algorithms like the one above with respect to this principle.

Violation of the Open-Closed Principle

By violating the single responsibility principle, the module in the example above will be hard to maintain; it is code that one never wants to dig into in the future. Unfortunately, on collision course with that wish is the fact that the event loop above violates the open-closed principle; new functionality cannot be added without modifying existing code. Related to our example, a diagnostics server typically allows a technician to connect and query stored information. Introducing that functionality would double the

logic in the event loop. Clearly, this design makes the code expensive to modify.

From a Performance Perspective

To make things worse, the solution above fails to scale in terms of performance as well. As all events are scanned serially, even in case timeouts are used, valuable time is wasted doing nothing.

The potential performance problem above may be derived from the failure of taking the concurrent nature of the problem into account. One approach to address this problem is by introducing multiple threads. The diagnostics server keeps its event loop, but the loop is now reduced to scan for connection requests. As soon as a connection is requested, a new thread is allocated exclusively for handling messages on that new connection.

The multithreading approach fails to address the overall design issue as it still violates both the single responsibility principle and the open-closed principle. Although the code for scanning and dispatching diagnostics messages is moved out of the event loop, adding a new server-port still requires modifications to existing code.

From a design perspective threads didn't improve anything. In fact, even with respect to performance, this solution may due to context switches and synchronization actually perform worse than the initial single-threaded approach.

The sheer complexity inherent in designing and implementing multithreaded applications is a further argument for discarding this solution.

Problem Summary

Summarizing the experience so far, the design fails as it assumes three different responsibilities. This problem is bound to be worse as the design violates the open-closed principle, making modifications to existing code more likely.

Summarizing the ideal solution, it should scale well, encapsulate and decouple the different responsibilities, and be able to serve multiple clients simultaneously without introducing the liabilities of multithreading. The REACTOR pattern realizes this solution by encapsulating each service of the application logic in event handlers and separating out the code for event demultiplexing.

The REACTOR Pattern

The intent of the REACTOR pattern is: "The REACTOR architectural pattern allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients" [Schmidt et al].

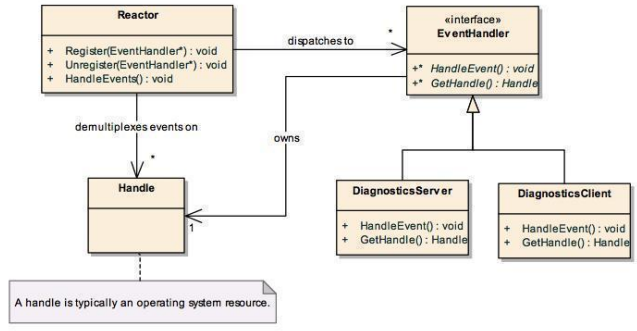

Illustration 2: Demultiplex requests to decoupled event handlers.

The roles of the involved participants are:

- *EventHandler*: An *EventHandler* defines an interface to be implemented by modules reacting to events. Each *EventHandler* own its own **Handle**.
- *Handle*: An efficient implementation of the REACTOR pattern requires an OS that supports handles (examples of Handles include system resources like files, sockets, and timers).
- *DiagnosticsServer* and *DiagnosticsClient*: These two are concrete event handlers, each one encapsulating one responsibility. In order to be able to receive event notifications, the concrete event handlers have to register themselves at the *Reactor*.
- *Reactor*: The *Reactor* maintains registrations of *EventHandlers* and fetches the associated *Handles*. The *Reactor* waits for events on its set of registered *Handles* and invokes the corresponding *EventHandler* as a *Handle* indicates an event.

Event Detection

In its description of REACTOR, Pattern-Oriented Software Architecture [Schmidt et al] defines a further collaborator, the Synchronous Event Demultiplexer. The Synchronous Event Demultiplexer is called by the *Reactor* in order to wait for events to occur on the registered *Handles*.

A synchronous event demultiplexer is often provided by the operating system. This example will use *poll()* (*select()* and Win32's *WaitForMultipleObjects()* are other functions available on common operating systems), which works with any descriptor.

The code interacting with *poll()* will only be provided as a sketch, because the POSIX specific details are outside the scope of this article. The complete sample code, used in this article, is available from my homepage [Petersen].

Implementation Mechanism

The collaboration between an *EventHandler* and the *Reactor* is similar to the interaction between an observer and its subject in the design pattern OBSERVER [Gamma, et al]. This relationship between these two patterns indicates that the techniques used to realize OBSERVER in C may serve equally well to implement the REACTOR.

In order to decouple the Reactor from its event handlers and still enable the Reactor to notify them, each concrete event handler must correspond to a unique instance. In our OBSERVER implementation, the FIRST-CLASS ADT pattern was put to work to solve this problem. As all concrete event handlers have to be abstracted as one, general type realizing the *EventHandler* interface, *void** is chosen as "the general type" to be registered at the Reactor (please refer to the previous part in this series for the rationale and technical reasons behind the *void** abstraction). These decisions enable a common interface for all event handlers:

Listing 1 : Interface of the event handlers, EventHandler.h

```
1  /* The type of a handle is system specific --
2     this example uses UNIX I/O handles, which are
3     plain integer values. */
4  typedef int Handle;
5
6  /* All interaction from Reactor to an event
7     handler goes through function pointers with the
8     following signatures: */
9     typedef Handle (*getHandleFunc)(void* instance);
10    typedef void (*handleEventFunc)(void* instance);
11
12 typedef struct
```

```
{
    void* instance;
    getHandleFunc getHandle;
    handleEventFunc handleEvent;
} EventHandler;
```

Having this interface in place allows us to declare the registration functions of the Reactor.
Listing 2 : Registration interface of the Reactor, Reactor.h

```
#include "EventHandler.h"

void Register(EventHandler* handler);

void Unregister(EventHandler* handler);
```

The application specific services have to implement the *EventHandler* interface and register themselves using the interface above in order to be able to react to events.
Listing 3 : Implementation of a concrete event handler, DiagnosticsServer.c

```
#include "EventHandler.h"

struct DiagnosticsServer
{
    Handle listeningSocket;
    EventHandler eventHandler;
    /* Other attributes here... */
};

/* Implementation of the EventHandler interface. */

static Handle getServerSocket(void* instance)
{
    const DiagnosticsServerPtr server = instance;
    return server->listeningSocket;
}

static void handleConnectRequest(void* instance)
{
    DiagnosticsServerPtr server = instance;
    /* The server gets notified as a new connection
       request arrives. Add code for accepting the
       new connection and creating a client here... */
}

DiagnosticsServerPtr createServer(unsigned int tcpPort)
{
    DiagnosticsServerPtr newServer = malloc(sizeof *newServer);

    if(NULL != newServer)
    {
```

```
32        /* Code for creating the server socket here.
33           The real code should look for a failure, etc. */
34        newServer->listeningSocket = createServerSocket(tcpPort);
35
36        /* Successfully created -> register the
37           listening socket: */
38        newServer->eventHandler.instance = newServer;
39        newServer->eventHandler.getHandle = getServerSocket;
40        newServer->eventHandler.handleEvent = handleConnectRequest;
41
42        Register(&newServer->eventHandler);
43      }
44      return newServer;
45  }
46
47  void destroyServer(DiagnosticsServerPtr server)
48  {
49      /* Before deleting the server we have to
50         unregister at the Reactor. */
51      Unregister(&server->eventHandler);
52      free(server);
53  }
```

Reactor Registration Strategy

When implementing the concrete event handlers as a FIRST-CLASS ADT, the functions for creating and destructing the ADT serves well to encapsulate the registration handling. The advantage is the combination of loose dependencies with information hiding as a client does not even have to know about the usage and interactions with the Reactor.

Another attractive property is that the internals of the server, in our example the handle, is encapsulated within the *getServerSocket* function. Sure, we are giving the Reactor a way to fetch it, but the Reactor is considered a well-trusted collaborator and we are actively giving it access by registering our event handler. There is no way for any other module to mistakenly fetch the handle and corrupt the associated resource.

Reactor Implementation

The details of the Reactor implementation are platform specific as they depend upon the available synchronous event demultiplexers. In case the operating system provides more than one synchronous event demultiplexer (e.g. *select()* and *poll()*), a concrete Reactor may be implemented for each one of them and the linker used to chose either one of them depending on the problem at hand. This technique is referred to as *link-time polymorphism*.

Each Reactor implementation has to decide upon the number of reactors required by the surrounding application. In the most common case, the application can be structured around one, single Reactor. In this case, the interface in Listing 2 (Reactor.h) will serve well. An application requiring more than one Reactor should consider making the Reactor itself a FIRST-CLASS ADT. This second variation complicates the clients slightly as references to the Reactor ADT have to be maintained and passed around in the system.

Independent of the system specific demultiplexing mechanism, a Reactor has to maintain a collection of registered, concrete event handlers. In its simplest form, this collection may simply be an array. This approach serves well in case the maximum number of clients is known in advance.

Listing 4: Implementation of a Reactor using poll(), PollReactor.c

```
#include "Reactor.h"
#include <poll.h>
/* Other include files omitted... */

/* Bind an event handler to the struct used to
   interface poll(). */
typedef struct
{
    EventHandler handler;
    struct pollfd fd;
} HandlerRegistration;
```

static HandlerRegistration registeredHandlers[MAX_NO_OF_HANDLES];

```
/* Add a copy of the given handler to the first free
   position in registeredHandlers. */
static void addToRegistry(EventHandler* handler);

/* Identify the event handler in the registeredHandlers
    and remove it. */
static void removeFromRegistry(EventHandler* handler);

/* Implementation of the Reactor interface used for
   registrations.*/
void Register(EventHandler* handler)
{
    assert(NULL != handler);
    addToRegistry(handler);
}

void Unregister(EventHandler* handler)
{
    assert(NULL != handler);
    removeFromRegistry(handler);
}
```

Invoking the Reactor

The reactive event loop is the core of the Reactor and its responsibilities are to control the demultiplexing and dispatch the detected events to the registered, concrete event handlers. The event loop is contained within the *HandleEvents()* function and is typically invoked from the *main()* function.

Listing 5: Client code driving the reactor

```
int main(void)
{
    const unsigned int serverPort = 0xC001;
    DiagnosticsServerPtr server = createServer(serverPort);

    if(NULL == server)
    {
        error("Failed to create the server");
    }

    /* Enter the eternal reactive event loop. */
    for(;;)
    {
        HandleEvents();
    }
}
```

Before investigating the implementation, which include file should provide the declaration of the *HandleEvents()* function? Unfortunately, adding it to the file in Listing 2 (Reactor.h) would clearly make that interface less cohesive; the registration functions models a different responsibility than the event loop. A solution is to create a separate interface for the event loop. This interface is intended solely for the compilation unit invoking the event loop.

Listing 6: Interface to the event loop, ReactorEventLoop.h

```
void HandleEvents(void);
```

Despite its simplicity, this separation solves the cohesiveness problem and shields clients from functions they do not use. This technique of providing separate interfaces to separate clients is known as the interface-segregation principle.

Implementing the Event Loop

With the interface in place, we can move on and implement the event loop itself. The listing below extends Listing 4.

Listing 7: Example of a reactive event loop, PollReactor.c

```
#include "ReactorEventLoop.h"

/* The code from Listing 4 go here (omitted)... */

/* Add a copy of all registered handlers to the
   given array. */
static size_t buildPollArray(struct pollfd* fds);

/* Identify the event handler corresponding to
   the given descriptor in the registeredHandlers. */
static EventHandler* findHandler(int fd);

static void dispatchSignalledHandles(const struct pollfd* fds,
                                     size_t noOfHandles)
```

```
{
     /* Loop through all handles.
        Upon detection of a handle signaled by poll,
        its corresponding event handler is fetched
        and invoked. */
     size_t i = 0;
     for(i = 0; i < noOfHandles; ++i)
     {
          /* Detect all signalled handles and
             invoke their corresponding event handlers. */
          if((POLLRDNORM | POLLERR) & fds[i].revents)
          {
            EventHandler* signalledHandler =
                           findHandler(fds[i].fd);

            if(NULL != signalledHandler)
            {
               signalledHandler-> handleEvent(
                                  signalledHandler->instance);
            }
          }
      }
 }

/* Implementation of the reactive event loop. */
void HandleEvents(void)
{
    /* Build the array required to interact
       with poll(). */
    struct pollfd fds[MAX_NO_OF_HANDLES] = {0};
    const size_t noOfHandles = buildPollArray(fds);

    /* Invoke the synchronous event demultiplexer.*/
    if(0 < poll(fds, noOfHandles, INFTIM))
    {
        /* Identify all signalled handles and
           invoke the event handler associated with
           each one. */
        dispatchSignalledHandles(fds, noOfHandles);
    }
    else
    {
        error("Poll failure");
    }
}
```

The example above lets each element in the collection maintain a binding between the registered event handler and the structure used to interact with *poll()*. One alternative approach is to keep two separate lists and ensure consistency between them. Pattern-Oriented Software Architecture [Schmidt et al] describes

another, system specific alternative: in a UNIX implementation using *select()*, the "*array is indexed by UNIX I/O handle values, which are unsigned integers ranging from 0 to FD_SETSIZE-1*".

Returning to the example, by grouping the registration and the poll-structure together, the array used to interact with *poll()* has to be built each time the reactive event loop is entered. In case the performance penalty is acceptable, this is probably a better choice as it enables a simpler handling of registrations and unregistrations during the event loop.

Handling new registrations

In my previous article, I discussed strategies for managing changed registrations during the notification of observers. The alternative of forbidding changed registrations is, unlike the OBSERVER pattern, not an option for a REACTOR implementation. In the example used in this article, the server reacts to the notification by creating a new client, which must be registered shall it ever be activated again. This leaves only one option for a REACTOR implementer: ensure that it works.

One solution is to maintain a separate array to interact with the synchronous event demultiplexer as illustrated above. This array is never modified in the event loop. However, this solution has the consequence that handles unregistered during the current event loop may be marked as signaled in the separate array. The code simply has to check for this case and ignore such handles, as illustrated by the function *dispatchSignalledHandles* in Listing 7 above.

The code uses the handle alone as identification. In cases resources are disposed and created during the same event loop, there is, depending on platform, a possibility that the handle ID's are re-used; a signaled handle in the copy may belong to an unregistered event handler, but due to a new registration using the re-cycled handle ID, the new event handler may be erroneously invoked. If this is an issue, the problem may be prevented by introducing further book-keeping data. For example, a second array containing the identities of the handles unregistered during the current event loop makes it possible to identify the case described above and thus avoid it.

More than one Type of Event

The design in the example above does only allow applications to register for one type of event (read-events). The event type is even hardcoded in the Reactor and it is a simple solution sufficient for applications without any need for further event detection. The REACTOR pattern, however, is not limited to one type of event. The pattern scales well to support different types of events.

Pattern-Oriented Software Architecture [Schmidt et al] describes two general strategies for dispatching event notifications:

• *Single-method interface*: An event handler is notified about all events through one, single function. The type of event (typically in the form of an *enum*) is passed as a parameter to the function. The disadvantage of this approach is that it sets the stage for conditional logic, which soon gets hard to maintain.

• *Multi-method interface*: In this case, the event handler declares separate functions for each supported event (e.g. *handleRead*, *handleWrite*, *handleTimeout*). As the Reactor has the knowledge of what event occurred, it invokes the corresponding function immediately, thus avoiding placing the burden on the event handler to re-create the event from a parameter.

Comparison of REACTOR and OBSERVER

Although the mechanisms used to implement them are related, there are differences between these two patterns. The main difference is in the notification mechanism. As a Subject changes its state in an OBSERVER implementation, all its dependents (observers) are notified. In a REACTOR implementation,

this relationship is one to one – a detected event leads the *Reactor* to notify exactly one dependent (*EventHandler*).

One typical liability of the OBSERVER pattern is that the cohesion of the subject is lowered; besides serving its central purpose, a subject also takes on the responsibility of managing and notifying observers. With this respect, a Reactor differs significantly as its whole raison d'être is to dispatch events to its registered handlers.

Consequences

The main consequences of applying the REACTOR pattern are:

1. *The benefits of the single-responsibility principle.* Using the REACTOR pattern, each of the responsibilities above is encapsulated and decoupled from each other. The design results in increased cohesion, which simplifies maintenance and minimizes the risk of feature interaction. As the platform dependent code for event detection is decoupled from the application logic, unit testing is greatly simplified (it is straightforward to simulate events through the *EventHandler* interface).
2. *The benefits of the open-closed principle.* The design now adheres to the open-closed principle. New responsibilities, in the form of new event handlers, may be added without affecting the existing event handlers.
3. *Unified mechanism for event handling.* Even if the REACTOR pattern is centered on handles, it may be extended for other tasks. Pattern-Oriented Software Architecture [Schmidt et al] describes different strategies for integrating the demultiplexing of I/O events with timer handling. Extending the Reactor with timer support is an attractive alternative to typical platform specific solutions based upon signals or threads. This extension builds upon the possibility to specify a timeout value when invoking the synchronous event demultiplexer (for example, *poll()* allows a timeout to be specified with a resolution of milliseconds). Although it will possibly not suit a hard real-time system, a Reactor based timer mechanism is easier to implement and use than a signal or thread based solution as it tends to avoid re-entrance problems and race-conditions.
4. *Provides an alternative to multithreading.* Using the REACTOR pattern, blocking operations in the concrete event handlers can typically be avoided and consequently also multithreading. As discussed above, a multithreaded solution does not only add significant complexity; it may also prove to be less efficient in terms of run-time performance. However, as the Reactor implies a non pre-emptive multitasking model, each concrete event handler must ensure that it does not perform operations that may starve out other event handlers.
5. *Trades type-safety for flexibility.* All concrete event handlers are abstracted as *void**. When converting a void-pointer back to a pointer of a concrete event handler type, the compiler doesn't have any way to detect an erroneous conversion This potential problem was faced in the implementation of the OBSERVER pattern and the solution is the same for the REACTOR: define unique notification functions for each different type of event handler and bind the functions and event handler together using an *EventHandler* structure as described in Listing 1.

Summary

The REACTOR pattern simplifies event-driven applications by decoupling the different responsibilities. The responsibilities are now encapsulated in separate modules.

There is much more to the REACTOR pattern than described in this article. Particularly several variations that all come with different benefits and trade-offs. For an excellent in-depth treatment of the REACTOR and other patterns in the domain, I recommend the book Pattern-Oriented Software Architecture, volume 2 [Schmidt et al].

Idiomatic Expressions

Reflections on the Idioms

After I'd finished the original patterns series I took some time off on a winter vacation to document some of the idioms I had applied in the implementations of the patterns. This was a response to some direct questions from my readers on why I used certain code constructs or preferred a specific style. These were idioms that I had picked up over the years. Either by reading other peoples code or sometimes through discussions in the comp.lang.c newsgroup I frequented at that time. Often, a change in style came about after I'd made a mistake. The idioms I applied were the results of a learning process. A painful bug due to an unintended assignment lead me to adapt CONSTANTS TO THE LEFT. My experiences with maintaining a large legacy code base taught me to appreciate NAMED PARAMETERS and MAGIC NUMBERS AS VARIABLES.

Patterns exist at all levels of scale in our designs. At their lowest level they are referred to as idioms. Idioms depend upon the implementation technology. Sometimes we find that the idioms we use in one language isn't applicable in a higher-level language. It may be that the problem it intends to prevent isn't present in the new language. INITIALIZE COMPOUND TYPES WITH {0} is one such example; many modern languages provide sensible default initializations for all variables. CONSTANTS TO THE LEFT is a non-issue in functional programming languages with immutability as a guiding principle. It might also be the case that the language has first-class support for the idea behind the idiom. NAMED PARAMETERS is one such example. It is supported naturally by languages such as Python and modern versions of C#. But these higher-level languages grow their own idioms too. One challenge of learning a new language is to get accustomed with its idiomatic ways of problem solving. By documenting the idioms we can provide guidance and ease the learning curve for newcomers to our technologies.

I'm happy that I took the opportunity to write down the idioms I had collected during my decade as a C programmer. My original idea was to write up more patterns. But life had it others. I switched jobs and found myself working with quite different technologies. The C implementations got abandoned. Writing them up without actual working experience didn't feel right. I believe in fresh knowledge. Idiomatic expressions turned out to be my last writings on C. I do hope to return some day. When I do, I'll pick up a copy of this final chapter as a starting point to guide my coding style.

Idiomatic Expressions in C

The Evolution of Idioms

As a language evolves, certain efficient patterns arise. These patterns get used with such a high frequency and naturalness that they almost grow together with the language and generate *an idiomatic usage* for the practitioner of the language. The resulting idiomatic expressions often seem peculiar to a newcomer of the language. This holds equally true for both natural and computer languages; even a programmer not using the idioms has to know them in order to be efficient at reading other peoples' code.

The idioms at the lowest levels are useful in virtually every non-trivial C program. Unfortunately idioms at this level are seldom described in introductory programming books and more advanced literature already expects the reader to be familiar with the idioms. The intention of this chapter is to capture some of these common idiomatic expressions.

Idiom Description Form

Because of their relative simplicity, devoting a chapter using a full-blown pattern form to a single one of these idioms would be to complicate it. On the other hand, simply listing the idioms would overemphasize the solution aspect. Therefore, each idiom will be introduced with a code sketch that illustrates a certain problem context, where after the idiom is applied to the code in order to solve the problem. This before-after approach also identifies certain code constructs as candidates for refactoring towards the idiomatic construct.

The idioms presented here are of two, overlapping categories:

- *Idioms for robustness*: The idioms in this category arose to avoid common pitfalls in the C language. For example, the idiom INITIALIZE COMPOUND TYPES WITH {0} provides a tight and portable way to initialize structs and arrays.
- *Idioms for expressiveness*: Writing code that communicates its intent well goes hand in hand with robustness; making the code easier to understand simplifies maintenance and thereby contributes to the long-term robustness of the program. For example, the idiom ASSERTION CONTEXT makes assertions self descriptive.

SIZEOF TO VARIABLES

Problem Context

In the C language, generality is usually spelled *void**. This is reflected in the standard library. For example, the functions for dynamic memory allocation (*malloc*, *calloc*, and *realloc*) return pointers to allocated storage as *void** and do not know anything about the types we are allocating storage for. The client has to provide the required size information to the allocation functions. Here's an example with *malloc*:

```
1  HelloTelegram* telegram = malloc(sizeof(HelloTelegram));
```

Code like this is sneaky to maintain. As long as the *telegram* really stays as a *HelloTelegram* everything is fine. The sneakiness lies in the fact that the *malloc* usage above contains a subtle form of dependency; the size given must match the size of the type on the left side of the assignment. Consider a change in telegram type to a *GoodByeTelegram*. With the code above this means a change in two places, which is at least one change too much:

```
1  /* We must change the type on both sides of the assignment! */
2  GoodByeTelegram* telegram = malloc(sizeof(GoodByeTelegram));
```

A failure to update both places may have fatal consequences, potentially leaving the code with undefined behavior.

Applying the Idiom

By following the idiom of SIZEOF TO VARIABLES the dependency is removed. The size follows the pointer type being assigned to and the change is limited to one place. The original example now reads:

```
1  HelloTelegram* telegram = malloc(sizeof *telegram);
```

But wait! Isn't the code above dereferencing an invalid pointer? No, and the reason that it works is that *sizeof* is an operator and not a function; *sizeof* doesn't evaluate its argument and the statement *sizeof *telegram* is computed at compile time. Better yet, if the type of *telegram* is changed, say to a *GoodByeTelegram*, the compiler automatically calculates the correct size to allocate for this new type.

As *telegram* is of pointer type, the unary operator * is applied to it. The idiom itself is of course not limited to pointers. To illustrate this, consider functions such as *memset* and *memcpy*. These functions achieve their genericity by using *void** for the pointer arguments. Given only the *void** pointer, there is of course no way for the functions to know the size of the storage to operate on. Exactly as with *malloc*, it is left to the client to provide the correct size.

```
uint32_t telegramSize = 0;
memcpy(&telegramSize, binaryDatastream, sizeof telegramSize);
```

With SIZEOF TO VARIABLES applied as above, the size information automatically matches the type given as first argument. For example, consider a change in representation of the telegram size from 32-bits to 16-bits; the *memcpy* will still be correct.

INITIALIZE COMPOUND TYPES WITH {0}

Problem Context

Virtually every coding standard bans uninitialized variables and that with very good reasons. Initializing a basic type such as *int* or *char* is straightforward, but what is the correct way of initializing a compound type like an array or struct? A dangerous but unfortunately common practice is shown below:

```
struct TemperatureNode
{
    double todaysAverage;
    struct TemperatureNode* nextTemperature;
};

struct TemperatureNode node;
memset(&node, 0, sizeof node);
```

The problem is that *memset*, as its name indicates, sets the bits to the given value and it does so without _any _knowledge of the underlying type. In C, all bits zero do not necessarily represent floating-point zero or a *NULL* pointer constant. Initializations using *memset* as above result in undefined behavior for such types.

The alternative of initializing the members of the struct one by one is both cumbersome and risky. Due to its subtle duplication with the declaration of the struct, this approach introduces a maintenance challenge as the initialization code has to be updated every time a member is added or removed.

Applying the Idiom

Luckily, the portable solution provided by the INITIALIZE COMPOUND TYPES WITH {0} does not only ensure correctness; it also requires less typing. The code below guarantees to initialize all members (including floating-points and pointers) of the structure to zero. The compiler guarantees to do so in a portable way by automatically initializing to the correct representation for the platform.

```
struct TemperatureNode node = {0};
```

At the expense of creating a zero-initialized structure, *memcpy* may be used to reset an array or members of a structure later. Because it works by copying whole bytes, possible padding included, *memcpy* does not suffer from the same problem as *memset* and may safely operate on the structure in our example.

```
const struct TemperatureNode zeroNode = {0};
struct TemperatureNode node = {0};

/* Perform some operations on the node. */
...
/* Reset the node (equal to node = zeroNode; ) */
memcpy(&node, &zeroNode, sizeof node);
```

Using *memcpy* for zeroing out a struct sure isn't the simplest possible way. After all the last line above could be rewritten as *node = zeroNode* while still preserving the same behavior. Instead the strength of this idiom is brought out when applied to an array. It helps us avoid an explicit loop over all elements in the array as *memcpy* now does the hard and admittedly boring task of resetting the array.

```
const double zeroArray[NO_OF_TEMPERATURES] = {0};
double temperatures[NO_OF_TEMPERATURES] = {0};

/* Store some values in the temperatures array. */
...
/* Reset the array. */
memcpy(temperatures, zeroArray, sizeof temperatures);
```

ARRAY SIZE BY DIVISION

Problem Context

The C language itself does not provide much support for handling its built-in arrays. For example, when passing a certain array to a function, the array decays into a pointer to its first element. Without any specific convention and given only the pointer, it simply isn't possible to tell the size of the array. Most APIs leave this book-keeping task to the programmer. One example is the *poll()* function in the POSIX API. *poll()* is used as an event demultiplexer scanning handles for events. These handles, which refer to platform specific resources like sockets, are stored in an array and passed to *poll()*. The array is followed by an argument specifying the number of elements.

```
struct pollfd handles[NO_OF_HANDLES] = {0};
/* Fill the array with handles to poll,
   code omitted... */
result = poll(handles, NO_OF_HANDLES, INFTIM);
```

The problem with this approach is that there is nothing tying the constant NO_OF_HANDLES to the possible number of elements except the name. Good naming does matter, but it only matters to human readers of the code; the compiler couldn't care less.

Applying the Idiom

By calculating ARRAY SIZE BY DIVISION, we are guaranteed that the calculated size always matches the actual number of elements. The calculation below is done at compile time by the compiler itself.

```
struct pollfd handles[NO_OF_HANDLES] = {0};
const size_t noOfHandles = sizeof handles / sizeof handles[0];
```

This idiom builds upon taking the size of the complete array and dividing it with the size of one of its elements (*sizeof handles[0]*).

MAGIC NUMBERS AS VARIABLES

Problem Context

Experienced programmers avoid magic numbers. Magic numbers do not communicate the intent of the code very well and may confuse anyone trying to read or modify it. Traditionally, some numbers like 0, 1, 3.14 and 42 are considered less magic than others. Consider a variable that is initialized to 0. A reader immediately expects this to be a default initialization and the variable will probably be assigned another value later. Similarly, culturally acquainted people know that 42 is the answer to the ultimate question of life, universe, and indeed everything.

The problem with all these not-so-magic numbers is that they build upon assumptions and expectations. These may be broken. One example is *struct tm*, which is used to represent the components of a calendar time in standard C. For historical reasons and in grand violation of the principle of least astonishment, assigning 0 to its *tm_year* member does not represent year 0; *tm_year* holds the years since 1900. Sigh.

Pure magic numbers are of course even worse. Consider the code fragment below:

```
1  startTimer(10, 0);
```

Despite a good function name, it is far from clear what's going on. What's the resolution - is 10 a value in seconds or milliseconds? Is it a value at all or does it refer to some kind of id? And what about this zero as second parameter?

Applying the Idiom

A step towards self documenting code is to express MAGIC NUMBERS AS VARIABLES. By applying this idiom to the code construct above the questions asked do not even arise; the code is now perfectly clear about its intent.

```
1  const size_t timeoutInSeconds = 10;
2  const size_t doNotRescheduleTimer = 0;
3
4  startTimer(timeoutInSeconds, doNotRescheduleTimer);
```

Of course the whole approach may be taken even further. By writing

```
1  startTimer(tenSecondsTimeout, doNotRescheduleTimer);
```

the code gets even more clear. Or does it really? The problem is that to the compiler the variable *tenSecondsTimeout* is really just a name.

There is no guarantee that it really holds the value 10 as an evil maintenance programmer my have changed the declaration to:

```
1  /* Original value was 10 */
2  const size_t tenSecondsTimeout = 42;
```

Such things happen and now anyone debugging the program will be unpleasantly surprised about how long ten seconds really are. They may feel like, hmm, well, 42 actually.

An idiom cannot be blindly applied and MAGIC NUMBERS AS VARIABLES is no exception. My recommendation is to use it extensively but avoid coding any values or data types into the names of the variables. Values and types are just too likely to change over time and such code gets unnecessarily hard to maintain.

NAMED PARAMETERS

Problem Context

As we expressed MAGIC NUMBERS AS VARIABLES the code got easier to read. That is, as long as the names of the variables convey meaning and finding good names is hard. To illustrate this, let us return to the previous example where a timer was started and extend it to include the start of two timers. I am rather happy with the name *timeoutInSeconds* and would have a hard time finding a second name that helps me remember the purpose of the variable equally well. Instead of going down the dark path of naming by introducing communicative obstacles such as *timeout1* and *timeout2*, I try to reuse the existing variable by removing its const qualification and re-assign it for the second timer.

```
1  size_t timeoutInSeconds = 10;
2  const size_t doNotRescheduleTimer = 0;
3
4  notifyClosingDoor = startTimer(timeoutInSeconds,
5                                 doNotRescheduleTimer);
6
7  timeoutInSeconds = 12;
8  closeDoor = startTimer(timeoutInSeconds,
9                         doNotRescheduleTimer);
```

This is a tiny example, yet it's obvious that the code doesn't read as well as before. The extra timer is part of the story, but there's more to it. By re- using the *timeoutInSeconds* variable, a reader of the code has to follow the switch of value. As the right-hand sides of the expressions starting the timers look identical, the reader has to span multiple lines in order to get the whole picture.

Applying the Idiom

By naming parameters, it is possible to bind a certain value to a name at the immediate call site. C doesn't have language support for this construct, but it is possible to partly emulate NAMED PARAMETERS.

```
1  size_t timeoutInSeconds = 0;
2  const size_t doNotRescheduleTimer = 0;
3
4  notifyClosingDoor = startTimer(timeoutInSeconds = 10,
5                                 doNotRescheduleTimer);
6
7  closeDoor = startTimer(timeoutInSeconds = 12,
8                         doNotRescheduleTimer);
```

The *timeoutInSeconds* variable is still re-used to start the timers, but this time directly in the function call. A reader of the code is freed from having to remember which value the variable currently has, because the expression now reads perfectly from left to right.

As neat as this idiom may seem, I hesitated to include NAMED PARAMETERS in this collection. The first time I saw the idiom was in a Java program. It felt rather exciting as I realized it would be possible in C as well. Not only would it make my programs self-documenting; it would also bring me friends, money, and fame. All at once. After the initial excitement had decreased, I looked for examples and possible uses of the idiom. I soon realized that the thing is, it *looks* like a good solution. However, most often it's just deodorant covering a code smell. Most of the examples of its applicability that I could come up with would be better solved by redesigning the code (surprisingly often by making a function more cohesive or simply

renaming it). All this suggests that NAMED PARAMETERS are more of a cool trick than a true idiomatic expression.

That said, I still believe NAMED PARAMETERS have a use. There are cases where a redesign isn't possible (third-party code, published API's, etc). As I believe these cases are rare, my recommendation is to rethink the code first and use NAMED PARAMETERS as a last resort. Of course, comments could be used to try to achieve the same.

```
closeDoor = startTimer(/* Timeout in seconds */ 12,
                       /* Do not reschedule */ 0);
```

Because I believe that such a style breaks the flow of the code by obstructing its structure, I would recommend against it.

ASSERTION CONTEXT

Problem Context

Assertions are a powerful programming tool that must not be confused with error handling. Instead they should primarily be used to state something that due to a surrounding context is known to be true. I use assert this way to protect the code I write from its worst enemy, the maintenance programmer, which very often turns out to be, well, exactly: myself.

Validating function arguments is simply good programming practice and so is high cohesion. To simplify functions and increase their cohesion I often have them delegate to small, internal functions. When passing around pointers, I validate them once, pass them on and state the fact that I know they are valid with an assertion (please note that compilers prior to C99 take the macro argument to assert as an integral constant, which requires programmers to write assert(NULL != myPointer); for well-defined behavior).

```
void sayHelloTo(const Address* aGoodFriend)
{
    if(aGoodFriend)
    {
        Telegram friendlyGreetings = {0};
        /* Add some kind words to the telegram,
           omitted here... */
        sendTelegram(&friendlyGreetings, aGoodFriend);
     }
     else
     {
         error("Empty address not allowed");
     }
}

static void sendTelegram(const Telegram* telegram,
                         const Address* receiver)
{
    /* At this point we know that telegram
       points to a valid telegram... */
    assert(telegram);
   /* ...and the receiver points to a
      valid address. */
```

```
24    assert(receiver);
25
26    /* Code to process the telegram omitted... */
27 }
```

In the example above *assert* is used as a protective mechanism decorated with comments describing the rationale for the assertions. As always with comments, the best ones are the ones you don't have to write because the code is already crystal clear. Specifying the intent of the code is a step towards such clearness and *assert* itself proves to be an excellent construct for that.

Applying the Idiom

The idiom ASSERTION CONTEXT increases the expressiveness of the code by merging the comment, describing the assertion context, with the assertion it refers to. This context is added as a string, which always evaluates to true in the assertion expression. The single-line assertion is now self describing in that it communicates its own context:

```
1 assert(receiver && "Is validated by the caller");
```

Besides its primary purpose, communicating to a human reader of this code that the receiver is valid, ASSERTION CONTEXT also simplifies debugging. As an assertion fires, it writes information about the particular call to the standard error file. The exact format of this information is implementation-defined, but it will include the text of the argument. With carefully-chosen descriptive strings in the assertions it becomes possible, at least for the author of the code, to make a qualified guess about the failure without even look at the source. As it provides rapid feedback, I found this feature particularly useful during short, incremental development cycles driven by unit-tests.

A drawback of ASSERTION CONTEXT is the memory used for the strings. In small embedded applications it may have a noticeable impact on the size of the executable. However, it is important to notice that *assert* is pure debug functionality and do not affect a release build; compiling with NDEBUG defined will remove all assertions together with the context strings.

CONSTANTS TO THE LEFT

Problem Context

The C language has a rich flora of operators. The language is also very liberal about their usage. How liberating the slogan "Trust the programmer" may feel, it also means less protection against errors and some errors are more frequent than others. It wouldn't be too wild a guess that virtually every C programmer in a moment of serious, head aching caffeine abstinence has erroneously written an assignment instead of a comparison.

```
1 int x = 0;
2
3 if(x = 0)
4 {
5     /* This will never be true! */
6 }
```

So, why not simply ban assignments in comparisons? Well, even if I personally avoid it, assignments in comparisons may sometimes actually make some sense.

```
Friend* aGoodFriend = NULL;
...
if(aGoodFriend = findFriendLivingAt(address))
{
    sayHelloTo(aGoodFriend);
}
else
{
    printf("I am not your friend");
}
```

How is the compiler supposed to differentiate between the first, erroneous case and the second, correct case?

Applying the Idiom

By keeping CONSTANTS TO THE LEFT in comparisons the compiler will catch an erroneous assignment. A statement such as:

```
if(0 = x) { }
```

is simply not valid C and the compiler is forced to issue a diagnostic. After correcting the if-statement, the code using this idiom looks like:

```
if(0 == x)
{
    /* We'll get here if x is zero -- correct! */
}
```

The idiom works equally well in assertions involving pointers.

```
assert(NULL == myNullPointer);
```

Despite its obvious advantage, the CONSTANTS TO THE LEFT idiom is not completely agreed upon. Many experienced programmers argue that it makes the code harder to read and that the compiler will warn for potentially erroneous assignments anyway. There is certainly some truth to this. I would just like to add that it is important to notice that a compiler is _not _required to warn for such usage. And as far as readability concerns, this idiom has been used for years and a C programmer has to know it anyway in order to understand code written by others.

Summary

The collection of idiomatic expressions in this chapter is by no means complete. There are many more idioms in C and each one of them solves a different problem.

Idiomatic expressions at this level are really just above language constructs. Thus, the learning curve is flat. Changing coding style towards using them is a small step with, I believe, immediate benefits.

References

- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., “POSA, A System of Patterns, Volume 1”, Wiley
- Dijkstra (1972). *The Humble Programmer*
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J, “Design Patterns”, Addison-Wesley
- ISO/IEC 9899:1999, The C standard
- Kafka, F. “The Metamorphosis”
- Kerievsky, J. “Refactoring to Patterns”, Addison-Wesley
- Kernighan, B., and Ritchie, D., “The C Programming Language”, Prentice Hall
- Martin, R. C. “Agile Software Development”, Prentice Hall
- Page-Jones, M. “The Practical Guide to Structured Systems Design”, Prentice Hall
- Petersen, A. _The complete REACTOR sample code used in this article, _http://www.adampetersen.se/
- Schmidt, Stal, Rohnert, Buschmann: “Pattern-Oriented Software Architecture, volume 2”, Wiley
- Sedgewick, R., “Algorithms in C, Parts 1-4”, Addison-Wesley
- Vlissides, J. “Pattern Hatching”, Addison-Wesley