Introduction

What are Patterns?

Patterns are solution blueprints. They are independent of implementation, transferable and flexible. They document reusable concepts, abstracted from successful solutions to recurring problems. By abstracting the key aspects of a solution, it can be reapplied to solve similar problems in different contexts.

Patterns exist in a multitude of disciplines, including architecture and construction. In the world of computer science and software engineering, software design patterns offer tried and tested constructs to common software design problems. Whereas expert software developers can draw on their experience to side-step design pitfalls previously encountered, inexperienced developers can draw on the experience embodied in design patterns. Patterns afford a wealth of design wisdom in a simplified accessible form, so that inexperienced designers can learn how to build well-designed, correct software solutions.

Using the proven techniques recorded in design patterns, developers can build systems that are more:

- Flexible
- Reusable
- Maintainable
- Elegant
- Extensible
- Portable

In object-oriented software this is achieved by applying the fundamental concepts and principles described in chapters 2 and 3. For example, by abstracting and encapsulating the changeable parts of a system, the system is more flexible, cohesive and decoupled. Likewise, many patterns exploit the Open Closed Principle to create extensible, maintainable and reusable solutions. All patterns catalogued within this book are described with reference to these concepts and principles.

The Makeup of a Pattern

A pattern is comprised of four basic elements: A name, a description of the problem, a description of a solution, and an evaluation of its consequences.
A pattern has a unique name that is concise, but informative. This name is a proxy for the reusable solution the pattern records. A key affordance of any pattern catalogue is the common, shared vocabulary of pattern names it produces – the Pattern Language. Using this language, designs can be conveyed with minimal effort at higher, abstracted levels, allowing focus to be placed on components of significance.

**Problem**

Each pattern describes the problem it aims to alleviate or solve. In doing so, the goal of the pattern is also defined. The problem description may suggest prerequisites that should be met for a pattern to be considered appropriate.

**Solution**

The solution to a problem is defined in terms of general design elements, their relationships, functions and interactions. The general nature of the solution makes it transferable to other contexts.

**Consequences**

Patterns introduce considerable benefits to the design of a system, but at a cost. This element of the pattern impartially evaluates the pros and cons of applying the pattern. When designing a system it is imperative that the consequences of a pattern be assessed with respect to the constraints imposed by the specific context.

**Types of Software Design Patterns**

There are many forms of software design patterns. Subsequent to the initial Gang of Four object-oriented design patterns catalogue, specialised catalogues have been recorded on subjects such as: security, databases, enterprise architecture, user interfaces – even software project management.

This book focuses on both object-oriented design patterns and enterprise application architecture patterns. These are the most established, exercised and pertinent patterns for practicing software developers today. By collating both catalogues into a single book, you will be well equipped to deal with the vast majority of design problems you encounter.
Object-Oriented Design Patterns

Object-oriented design patterns are concerned with the responsibilities, roles and interactions of components at the object level. They aim to solve design issues that affect the classes and objects of an object-oriented application. Their concepts are however highly applicable to the overall architecture of a system. For example, the concepts involved in designing a simplified interface to a complex subsystem, as in the Façade pattern, are highly relevant to designing a well-defined API to an entire system component or layer, as in the Service Layer pattern.

Enterprise Application Architecture Patterns

Enterprise application architecture patterns ("enterprise patterns") are concerned with designing object-oriented enterprise applications. Enterprise software is defined as software used on a large or significant scale within an organisation, such as a blue chip company. For example, an accounting system, trading platform, or even grid computing service could all be regarded as enterprise solutions for a business. The unifying theme of these functionally diverse systems is their value to a company as a whole.

Enterprise patterns focus on the bigger picture: the overall architecture of a system or number of systems within an organisation. They are concerned with the responsibilities, roles and interactions of more course-grained portions of a system than object-oriented patterns. None-the-less, many of the principles introduced at the object level are evident at this elevated level.

The Power of Patterns

Whilst leading the development of a fair value loan pricing service for an investment bank client, I encountered an interesting software design problem. The challenge was not so much what to do in order to achieve a goal or overcome a challenge at design time, but more how to rectify an issue caused by inflexible design in an implemented system. By describing this situation, I hope to offer a glimpse of the benefits design patterns afford, as well as demonstrate their applicability in diverse settings.

The development of the system was divided between an IT development team and a business analytics team. Analytics implemented the pricing logic in Java. IT wrapped the pricing logic and exposed it as a service, provided a build environment and tooling for the business, and proposed design recommendations and guidelines for their pricing component.

Problem

As development progressed, the service throughput dropped dramatically. Profiling the application highlighted some peculiar behaviour within the pricing component. When calculating the price of a single product, product
type checking methods such as `isProductTypeA()` and `isProductTypeB()` were invoked several hundred thousand times. In addition, the `String equals()` method was invoked over 1 million times.

By supporting an increasing range of products, the calculation logic within the pricing component had deteriorated to spaghetti code. Each new product introduced a few more special cases. Conditional logic surrounded each special case, invoking the product type checking methods to establish whether the product-specific logic should be executed.

Each product type checking method compared a `String` member field of the product object (its type) against a `String` constant – a technique made unnecessary by Java 5’s `enum`.

Calculating the fair value of a product required calculating its price at multiple points in time throughout the maturity of the product. As a result, the calculation logic that contained complex and inefficient conditional logic was invoked many times.

Two aspects of this design concerned me. The first being that the product type checking logic was not type safe and that its string comparison was inefficient relative to the type safe `enum` comparison available. The second, more significant issue was the complexity of the calculation logic, and in particular the spaghetti structure of the control flow it contained.

The first concern is an implementation issue and a question of effective programming using best practices (language-specific programming patterns of a lower level than are addressed in this book).

The second concern however is definitely one of design. The calculation of a single price was fundamentally a sequence of steps: some that should be performed for a given product, some that should not, and some that should vary per product. The calculation had to confirm the product type before executing any product specialisation. However, the overall algorithm – i.e. the fundamental steps entailed in calculating a price – never changed.

**Name & Solution**

Enter the Template Method pattern. This pattern defines the skeleton of an algorithm in a base class, deferring the implementation of some steps of the algorithm to subclasses. Subclasses can also redefine some of the steps as necessary, but the overall algorithm cannot be altered.

We introduced the Template Method pattern to the calculation logic, abstracting the calculation algorithm and common steps to a base class. Product-specific steps were encapsulated in subclasses of this base class – one class per product type. If support for an additional product were required, a new subclass containing only the specialised logic need be
created. The overall calculation logic should never need to be altered.

**Consequences**

By introducing the *Template Method* pattern the spaghetti structure of the calculation logic was eliminated. The new design did not reduce product type checking – it eradicated it completely. There was no longer a need to check what product was being priced because it was implicit at runtime. Only code appropriate to the product would ever be executed.

As a result of the refactoring process, the pricing logic was easier to maintain and extend, simpler, more elegant, and 5 to 10 seconds faster.

**Pattern Fear**

Developers often avoid using design patterns in settings atypical to those originally described for fear that they are inappropriate. Ultimately, patterns are transferable concepts of structures, relationships and interactions. Although the term “algorithm” in the Template Method Pattern might suggest limited usefulness outside academic computer science, in truth it merely formalises a design encapsulating a sequence of steps used to achieve an outcome. It is commonly used to build frameworks and is prevalent in GUI design, database connection logic, and driver management.

The price service project illustrated how patterns may not only be used at an infrastructure level (e.g. database connection logic), but also within application business logic to structure it in a meaningful, elegant and extensible way.

**Pattern Fever**

While some developers are overly cautious of using patterns, there are also those who are overindulgent. Patterns are tools to solve recurring problems, but used in the wrong context or for the wrong reason they can have an adverse effect on a design. The most common symptom of this “pattern fever” is *Over Engineering* – the practice of introducing unnecessary flexibility or functionality into a system for which there is no need.

Patterns offer numerous benefits, but normally at some cost. For example, the *State* pattern introduces an additional class for every state of an object. This is sometimes referred to as “class explosion” due to the significant increase in the size of the code base.

A good software designer is a pragmatic designer. Applying design patterns to a problem is a trade-off that requires unbiased comparison of the inherent advantages and disadvantages. Each pattern in this book is evaluated in this regard, and quick reference Pros and Cons lists are provided to aid your
decision.

The History of Patterns

In 1977 architect Christopher Alexander published *A Pattern Language: Towns/Buildings/Construction*, a book that popularised the concept of a pattern language and was to have a profound effect on the computer science community in years to come. Alexander identified that common elements of architectural design could be abstracted from a given scenario and reapplied to other projects of similar requirements. He referred to this reusable element as a Pattern.

Nearly twenty years later, the concept of reusable patterns within an engineering discipline was transferring into the world of computer science as Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (the Gang of Four) published *Design Patterns: Elements of Reusable Object-Oriented Software*. Regarded as the seminal work in this field, the book not only offered a new perspective on efficient and effective software design, but also further illustrated the benefits that object-oriented programming languages offered to the software engineering community.

Building on the success of the Gang of Four’s Design Patterns, Martin Fowler, a leader in the field of object-oriented design, published *Patterns of Enterprise Application Architecture* in 2002. Fowler extended the concept of software design patterns into the enterprise, cataloguing reusable system components at a higher level of abstraction.

Using This Book

This book is intended as a quick reference pattern catalogue, offering examples from current real-world design problems found in a range of industry sectors.

The first section of the book outlines key object-oriented software design principles that are core to the design patterns that follow.

Using these principles as a benchmark, each pattern is presented in turn throughout the remainder of the book forming the pattern catalogue. Each pattern is documented using a consistent template containing the following sections: Name, Goal, Motivation, How It Works, Scenario, Structure, Sample Code, Pros & Cons, Known Uses, Also Known As and Related.

The Unified Modelling Language (UML) 2.0 is used to illustrate the actors involved in each solution. A UML key is provided on the front and back inside cover for your ease.

Sample code is provided to illustrate how a solution could be implemented.
The sample code alternates between Java and C#, as appropriate to the available language features. Code samples provided in both Java and C# are available for download from the book’s accompanying website.

The pattern catalogue is divided into two main sections: Object-Oriented Design Patterns, followed by Enterprise Application Architecture Patterns.

Although this book is intended as a quick reference, it is both concise and complete. New and interesting scenarios are described throughout, making the book engaging and readable from start to finish.

Further Reading

For an excellent introduction to object-oriented design patterns refer to *Head First Design Patterns* by Freeman et al., published by O’Reilly.

More in-depth explanations of object-oriented design patterns can be found in *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma et al., published by Addison Wesley. The first section of this book explains how to use the book and its patterns to solve design problems. The second section is an exhaustive pattern catalogue.

For a comprehensive description of all enterprise application architecture patterns, refer to *Patterns of Enterprise Application Architecture* by Fowler, published by Addison Wesley. The first section of the book offers a tutorial on designing and developing enterprise applications. The second section is a detailed pattern catalogue.

Further information on the version of UML notation used in this book can be found in *UML 2.0 in a Nutshell* by Pilone, published by O’Reilly.
Object-Oriented Design Patterns

Scope
Object-oriented design patterns solve general design problems regarding the roles and interactions of classes and objects. These general solutions can then be customised to solve specific problems in a particular context.

Goal
Object-oriented patterns aim to produce code that is:

• Extensible  • Reusable
• Decoupled   • Cohesive

Categories
Object-oriented patterns can be grouped in a variety of ways based on their structure, interrelationships, or most usefully their purpose. Categorising the patterns by purpose produces three main categories:

• Creational
  Patterns that solve design problems concerning the creation of objects.

• Structural
  Patterns that solve design problems concerning the composition of objects.

• Behavioural
  Patterns that solve design problems concerning the way in which objects or classes interact and the separation of their responsibilities and concerns.
Behavioural Patterns
Granularity: Object-Oriented

Scope
Behavioural patterns endeavour to solve design problems concerning the way in which objects or classes interact, and the separation of their concerns and responsibilities.

Behavioural Class Patterns
Class patterns solve behavioural problems using inheritance, producing a design that is fixed at compile time. The following patterns are primarily applied to classes:

- Interpreter
- Template Method

Behavioural Object Patterns
Object patterns solve behavioural problems by dividing aspects of the desired behaviour as responsibilities among a group of objects, and coordinating their interactions. The resulting designs are more dynamic than those produced by class patterns, as they can be changed at runtime.

The following patterns are primarily applied to objects:

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor
Chain of Responsibility

**Motivation**

It is sometimes necessary to allow one of a number of objects to process a request. It is unrealistic to require that the caller keep stock of every possible receiver of its request, as this would be inflexible, costly to maintain and would introduce tight coupling. Using the *Chain of Responsibility*, the caller can instead depend on one receiver — the head of the chain. The head may not be the receiver that handles the request. In fact, the caller need never know the ultimate receiver.

**How It Works**

Each receiver, or “handler”, implements a common interface on which the caller depends. This interface defines a single `handle()` method. Each handler also has a reference to its successor handler. If a handler cannot process a request, it passes the request to its successor by calling the successor’s `handle()`. When a handler can process a request it does so within `handle()`, then either ceases further processing or, if multiple handlers are permitted to handle the request, passes the request to its successor for further processing.

The handlers form a chain through which a request is passed until it is processed. If no handler processes the request, it will fall off the end of the chain. By programming to interfaces, chain members (and therefore behaviour) can be dynamically configured and swapped at runtime. Furthermore, the caller is decoupled from its many potential receivers.

**Scenario:** “An SOA Borrower Data Service in an Investment Bank”

Consider a Service Oriented Architecture system affording access to a variety of data in any combination. Rather than support statically defined *a priori* combinations, clients can *decorate* their requests to ask for additional data items.

Within the service, the vast number of possible request combinations is supported using a chain of handlers, through which the decoratable command is passed. Each handler in the chain retrieves a specific data item, if requested, and populates the response object accordingly.
Pros & Cons

- **Decouples** design: caller does not know receiver.
- **Simplifies** caller: caller knows nothing of the chain structure.
- Can **attach behaviour** to existing classes easily.
- **Flexible** and easy to change chain members and ordering.
- **Can attach behaviour** to existing classes easily.
- **Flexible** and easy to change chain members and ordering.

**Cons**

- Behaviour is not known until runtime.
- No compile-time checking of correctness.
- Trivial configuration bugs can be hard to find.
- Request is **not guaranteed** to be handled.

Structure

```
public abstract class Handler {
    protected Handler successor;
    public Handler(Handler successor) {
        this.successor = successor;
    }
    public void handle(Request request) {
        successor.handle(request); // Pass to successor by default
    }
}
public class HandlerA extends Handler {
    public HandlerA(Handler successor) {
        super(successor);
    }
    public void handle(Request request) {
        if (request instanceof RequestA) {
            /* Handle it */
        } else {
            successor.handle(request); // Pass it on
        }
    }
}
public class HandlerB extends Handler {
    public HandlerB(Handler successor) {
        super(successor);
    }
    public void handle(Request request) {
        /* Handle it */
    }
}
```

The abstract base `Handler` class provides default behaviour of forwarding the request to its successor. `HandlerA` overrides the default behaviour, handling the request if it can, forwarding if it cannot. `HandlerB` uses the default forwarding behaviour only.

Sample Code

```
public abstract class Handler {
    protected Handler successor;
    public Handler(Handler successor) {
        this.successor = successor;
    }
    public void handle(Request request) {
        successor.handle(request); // Pass to successor by default
    }
}
public class HandlerA extends Handler {
    public HandlerA(Handler successor) {
        super(successor);
    }
    public void handle(Request request) {
        if (request instanceof RequestA) {
            /* Handle it */
        } else {
            successor.handle(request); // Pass it on
        }
    }
}
public class HandlerB extends Handler {
    public HandlerB(Handler successor) {
        super(successor);
    }
    public void handle(Request request) {
        /* Handle it */
    }
}
```

Known Uses

Logging appenders, event handling, email filtering, context-sensitive help, implementing a workflow of some kind, SOA command/request processing.

Related

*Composite*: attach behaviour (e.g. context-specific help) to UI widgets.
Enterprise Application Architecture
Design Patterns

Scope
Enterprise patterns are concerned with high-level architectural issues that affect systems and subsystems, rather than objects and classes. None-the-less, every pattern is still implemented using these fundamental elements. As such, enterprise patterns extend many of the basic object-oriented concepts and principles central to object-oriented patterns.

The essential difference between object-oriented patterns and enterprise patterns is scope and purpose. Enterprise patterns affect the architecture of large parts of a system and solve problems that may only be encountered when developing enterprise software, such as object-relational mapping.

Goal
Enterprise patterns aim to produce systems and subsystems that are:

• Scalable
• Flexible
• Reusable
• Robust

Categories
Enterprise patterns can be grouped by purpose as follows:

• Domain Logic
  Patterns that organise and manage the roles and interactions of domain-specific components.

• Data Source Architectural
  Patterns that offer flexible and decoupled designs to connect an application to a data source.

• Object-Relational Behavioural
  Patterns that solve design problems regarding how and when associated objects are persisted and loaded from a relational data source.
• Object-Relational Structural
  Patterns that solve design problems concerning the composition of objects within a relational data source.

• Object-Relational Metadata Mapping
  Patterns that solve problems caused by the disparity between the object-oriented model and relational model of a system.

• Web Presentation
  Patterns that solve behavioural and structural problems that occur within the web presentation layer of a system.

• Distribution
  Patterns that solve design problems that affect distributed systems or components and their interactions.

• Offline Concurrency
  Patterns that solve issues of offline concurrency management and resolution within a system.

• Session State
  Patterns that solve problems concerned with state management within a multi-tier system.

• Base
  Primitive patterns upon which the preceding patterns build and depend.
Object-Relational Behavioural Patterns
Granularity: Enterprise

Scope
Object-relational behavioural patterns endeavour to solve design problems concerning how and when associated objects are persisted and loaded from a relational data source.

Patterns
The following patterns can be classified as object-relational behavioural patterns:

• Unit of Work
• Identity Map
• Lazy Load
Keep track of the objects affected by a business transaction, coordinate their persistence and resolve concurrency problems.

Motivation
When accessing data in a data source and modifying it, a record must be kept of what changes have been made so that they may be persisted. Changes could be persisted as they occur, but this would result in a lot of data source access. Instead, all changes should be gathered and committed at one time, but what is the best way to record these changes?

The Unit of Work (UoW) provides a global point were all changes that occur within a business transaction can be registered. The UoW can then coordinate the persistence of changes in an efficient, controlled and consistent manner.

How It Works
Create a Unit of Work class that maintains lists of objects affected by a business transaction. The UoW provides public methods to register an object as new, dirty (i.e. modified) or removed (i.e. deleted).

Either the caller or the objects themselves register changes with the UoW. In object registration, the object obtains a session-specific UoW and registers changes as they occur throughout the lifetime of the object.

When changes are complete, the caller requests that the UoW commit the recorded changes to the database (e.g. using a DataMapper) in a controlled and ordered fashion, managing any concurrency issues.

Scenario: “Withdrawing Money from a bank Account.”
When money is withdrawn from an account, a new Drawing object is created and attached to its parent Account. The Account’s balance is also modified. A Unit of Work records the creation of the new Drawing and the change to the Account. When committed, all changes are persisted.

Pros & Cons
- ✔ Faster than persisting each change individually.
- ✔ Minimises the duration of open system transactions.
- ✔ Caller does not contain data access logic or references.
- ✔ Concurrency issues can be resolved more easily.
- ✔ Referential integrity is managed by the order of updates.
- ✔ Deadlocks can be minimised by using a global order for updates.
- ✗ Developers may forget to register changes to an object.
- ✗ Scope of the Unit of Work must be managed per session.
Structure

Sample Code

Known Uses

Creating a resource-independent business transaction boundary. `UoW` is implemented in ADO.NET DataSet and many persistence frameworks.

Related

Layer Supertype, Data Mapper, Registry (to globalise the Unit Of Work).
Design Patterns Pocket Guide

Design Patterns are a crucial tool for every practicing software developer. By providing a catalogue of transferable concepts abstracted by peers and predecessors, their lessons learned can be harnessed to avoid repeating the same design blunders. Design Patterns offer tried and tested best-practices that, when used diligently, will increase your productivity throughout the lifecycle of a software development project and improve the quality of the end system.

Bringing together the Gang of Four’s object-oriented Design Patterns and Martin Fowler’s Patterns of Enterprise Application Architecture into a single condensed quick reference volume, Design Patterns Pocket Guide is the ideal companion to these seminal works.

The book begins by briefly outlining the key principles of successful software design, providing a useful benchmark for each pattern. The book then presents a quick reference catalogue of all 74 object-oriented and enterprise patterns. Each concise catalogue entry contains the following sections: Goal, Scenario, How It Works, Structure (UML), Sample Code (Java/C#), Pros & Cons checklist, Known Uses, amongst others.

Central to the book is the use of contemporary real-world scenarios from a wide range of interesting industrial sectors, revitalizing the world of patterns and transporting it into present day.

Never before has such a comprehensive guide to designing object-oriented software solutions been available to the software engineering community in a small form factor.

Graham R. Williamson is an independent software developer and the founder of Figment Software. He received his BSc. in Computing Science from the University of Glasgow and now lives in Surrey, UK, with his wife and golden retriever. Graham has worked as an IT professional in a multitude of industries, including Investment Banking, Insurance, Web and Electronic Payments. He has a passion for large-scale distributed systems and designing enterprise solutions.