GoF Design Patterns Introduction

Table of Contents

Table of Contents

GoF Design Patterns Introduction

1 *Design Patterns: Elements of Reusable Object-Oriented Software*

Design Patterns: Elements of Reusable Object-Oriented Software - October 21, 1994 (ISBN 0-201-63361-2) is a software engineering book describing recurring solutions to common problems in software design.

The book's authors (without the foreword author Grady Booch) are often referred to as the ***Gang of Four***, or ***GoF***. They are

- **Erich Gamma** - (born 1961 in Zürich) is Swiss computer scientist and co-author of the influential Software engineering textbook, Design Patterns: Elements of Reusable Object-Oriented Software. He co-wrote the **JUnit** software testing framework with Kent Beck and **led the design of the Eclipse platform**'s Java Development Tools (JDT). Erich has a Ph.D. in Computer Science from University of Zurich.
- **Richard Helm** - recently rejoined IBM to start the Australian branch of the Object Technology Practice. Prior to that, he was a technology consultant with **DMR Group**, an international information technology consulting firm. There he actively applied design patterns to the design of commercial systems. Prior to DMR, Richard was in the Software Technology department at **IBM T.J. Watson Research Center** investigating object-oriented design and reuse and visualization. Richard has numerous international publications, writes regularly in **Dr. Dobb's Journal**, and is a past OOPSLA program committee member. Richard has a Ph.D. in Computer Science from the University of Melbourne, Australia.
- **Ralph Johnson** - has been studying object-oriented technology and how it changes the way that software is developed for the past 10 years. He has been involved in the development of an object-oriented operating system (Choices), compiler (Typed Smalltalk), graphics editor framework (HotDraw), music synthesis system (Kyma), and is currently working on a framework for accounting. He is on the faculty of the **Department of Computer Science** at the **University of Illinois** and has helped organize several OOPSLA's, including OOPSLA'93 as program chair. He got his PhD from Cornell.
- **John Vlissides** - (August 2, 1961 - November 24, 2005) passed away November 24th, 2005. He was a researcher at the **IBM T.J. Watson Research Center**. His research interests included object-oriented design tools and techniques, application frameworks and builders, and program visualization. Before IBM, John was at the Computer Systems Laboratory at Stanford University. There he co-developed InterViews, a popular object-oriented system for developing graphical applications. John received his Ph.D. in electrical engineering from Stanford University.

GoF Design Patterns Introduction

The book is divided into two parts:

- with the **first two chapters** exploring the capabilities and pitfalls of object-oriented programming,
- and the **remaining chapters describing 23 classic software design patterns**.

The book includes examples in C++ and Smalltalk. It won a Jolt productivity award, and Software Development productivity award in 1994.

The original publication date of the book was **October 21, 1994** with a **1995** copyright, and as of April 2007, the book was in its 36th printing. The book was first made available to the public at OOPSL A meeting held in Portland, Oregon in October 1994.

It has been highly influential to the field of software engineering and is regarded as an important source for object-oriented design theory and practice. More than 500,000 copies have been sold in English and in 13 other languages.

2 Introduction Chapter 1

Chapter 1 is a discussion of object-oriented design techniques, based on the authors' experience, which they believe would lead to good object-oriented software design, including:

1 Interfaces Vs. Implementations

"**Program to an 'interface', not an 'implementation'**." The authors claim the following as advantages of interfaces over implementation:

- clients remain unaware of the specific types of objects they use, as long as the object adheres to the interface
- clients remain unaware of the classes that implement these objects; clients only know about the abstract class(es) defining the interface
- Use of an interface also leads to dynamic binding and polymorphism, which is consequentially important to object-oriented programming.

2 Composition Instead of Inheritance

"**Favor 'object composition' over 'class inheritance'.**"

- The authors refer to inheritance as ***white-box*** *reuse*, with white-box referring to visibility, because the internals of parent classes are often visible to subclasses.
- In contrast, the authors refer to object composition (in which objects with well-defined interfaces are used dynamically at runtime by objects obtaining references to other objects) as ***black-box*** *reuse* because no internal details of composed objects need be visible in the code using them
- The authors discuss the tension between inheritance and encapsulation at length and state that in their experience, designers overuse inheritance (Gang of Four 1995:20). The danger is stated as follows: "Because inheritance exposes a subclass to details of its parent's implementation, it's often said that 'inheritance breaks encapsulation'". (Gang of Four 1995:19)

- They warn that the implementation of a subclass can become so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclass to change. Furthermore, they claim that a way to avoid this is to inherit only from abstract classes—but then, they point out that there is minimal code reuse.
- Using inheritance is recommended mainly when adding to the functionality of existing components, reusing most of the old code and adding relatively small amounts of new code.
- To the authors, 'delegation' is an extreme form of object composition that can always be used to replace inheritance. Delegation involves two objects: a 'sender' passes itself to a 'delegate' to let the delegate refer to the sender. Thus the link between two parts of a system are established only at runtime, not at compile-time. The Callback article has more information about delegation.

3 Parametrized Types (Generics)

The authors also discuss so-called **parameterized types**, which are also known as **generics**. These allow any type to be defined without specifying all the other types it uses—the unspecified types are supplied as 'parameters' at the point of use.

The authors admit that delegation and parameterization are very powerful but add a warning: "Dynamic, highly parametrized software is harder to understand and build than more static software." (Gang of Four 1995:21)

4 Aggregation and Association

- The authors further distinguish between '**Aggregation**', where one object 'has' or 'is part of' another object (implying that an aggregate object and its owner have identical lifetimes)
- and **acquaintance (association)**, where one object merely 'knows of' another object. Sometimes acquaintance is called '**association**' or the 'using' relationship. Acquaintance objects may request operations of each other, but they aren't responsible for each other. Acquaintance is a weaker relationship than aggregation and suggests much **looser coupling** between objects, which can often be desirable for maximum maintainability in a design.

3 Case study, Chapter 2

Chapter 2 is a step-by-step case study on "the design of a 'What-You-See-Is-What-You-Get' (or 'WYSIWYG') document editor called **Lexi**."

The chapter goes through seven problems that must be addressed in order to properly design Lexi, including any constraints that must be followed.

Each problem is analyzed in-depth, and solutions are proposed. Each solution is explained in full, including pseudo-code and Unified Modeling Language where appropriate.

Finally, each solution is associated directly with one or more **design patterns**. It is shown how the solution is a direct implementation of that design pattern.

The seven problems (including their constraints) and their solutions (including the pattern(s) referenced), are as follows:

GoF Design Patterns Introduction

1 Document Structure

The document is an arrangement of basic graphical elements such as characters, lines, other shapes, etc, that "capture the total information content of the document". The structure of the document contains a collection of these elements, and each element can in turn be a substructure of other elements.

Problems and Constraints

1. Text and graphics should be treated the same way (that is, graphics aren't a derived instance of text, nor vice versa)
2. The implementation should treat complex and simple structures the same way. It should not have to know the difference between the two.
3. Specific derivatives of abstract elements should have specialized analytical elements.

Solution and Pattern (composite)

- A *recursive composition* is a hierarchical structure of elements, that builds "increasingly complex elements out of simpler ones".
- Each node in the structure knows of its own children and its parent. If an operation is to be performed on the whole structure, each node calls the operation on its children (recursively).
- This is an implementation of the **composite pattern**, which is a collection of nodes. The node is an abstract base class, and derivatives can either be leaves (singular), or collections of other nodes (which in turn can contain leaves or collection-nodes). When an operation is performed on the parent, that operation is recursively passed down the hierarchy.

2 Formatting

Formatting differs from structure. Formatting is a method of constructing a particular instance of the document's physical structure. This includes breaking text into lines, using hyphens, adjusting for margin widths, etc.

Problems and Constraints

1. Balance between (formatting) quality, speed and storage space
2. Keep formatting independent (uncoupled from) the document structure.

Solution and Pattern (Strategy)

- A *Compositor* class will encapsulate the algorithm used to format a composition. Compositor is a subclass of the primitive object of the document's structure. A Compositor has an associated instance of a Composition object. When a Compositor runs its `Compose()`, it iterates through each element of its associated Composition, and rearranges the structure by inserting Row and Column objects as needed.
- The Compositor itself is an abstract class, allowing for derivative classes to use different formatting algorithms (such as double-spacing, wider margins, etc.)
- The Strategy Pattern is used to accomplish this goal. A Strategy is a method of encapsulating multiple algorithms to be used based on a changing context. In this case, formatting should be different, depending on whether text, graphics, simple elements, etc, are being formatted.

3 Embellishing the User Interface

The ability to change the graphical interface that the user uses to interact with the document.

Problems and Constraints

1. Demarcate a page of text with a border around the editing area
2. Scroll bars that let the user view different parts of the page
3. User interface objects should not know about the embellishments
4. Avoid an "explosion of classes" that would be caused by subclassing for "every possible combination of embellishments" and elements

Solution and Pattern (Decorator)

- The use of a *transparent enclosure* allows elements that augment the behavior of composition to be added to a composition. These elements, such as Border and Scroller, are special subclasses of the singular element itself. This allows the composition to be augmented, effectively adding state-like elements. Since these augmentations are part of the structure, their appropriate `Operation()` will be called when the structure's `Operation()` is called. This means that the client does not need any special knowledge or interface with the structure in order to use the embellishments.
- This is a **Decorator** pattern, one that adds responsibilities to an object without modifying the object itself.

4 Supporting Multiple Look-And-Feel Standards

Look-and-feel refers to platform-specific UI standards. These standards "define guidelines for how applications appear and react to the user".

Problems and Constraints

1. The editor must implement standards of multiple platforms so that it is portable
2. Easily adapt to new and emergent standards
3. Allow for run-time changing of look-and-feel (ie: No hard-coding)
4. Have a set of abstract elemental subclasses for each category of elements (ScrollBar, Buttons, etc)
5. Have a set of concrete subclasses for each abstract subclass that can have a different look-and-feel standard. (ScrollBar having MotifScrollBar and PresentationScrollBar for Motif and Presentation look-and-feels)

Solution and Pattern (Abstract Factory)

- Since object creation of different concrete objects cannot be done at runtime, the object creation process must be abstracted. This is done with an abstract guiFactory, which takes on the responsibility of creating UI elements. The abstract guiFactory has concrete implementations, such as MotifFactory, which creates concrete elements of the appropriate type (MotifScrollBar). In this way, the program need only ask for a ScrollBar and, at run-time, it will be given the correct concrete element.
- This is an Abstract Factory. A regular factory creates concrete objects of one type. An abstract factory creates concrete objects of varying types, depending on the concrete implementation of the factory itself. Its ability to focus on not just concrete objects, but entire *families* of concrete objects "distinguishes it from other creational patterns, which involve only one kind of product object"

5 Supporting Multiple Window Systems

Just as look-and-feel is different across platforms, so is the method of handling windows. Each platform displays, lays out, handles input to and output from, and layers windows differently.

Problems and Constraints

1. The document editor must run on as many of the "important and largely incompatible window systems" that exist
2. An Abstract Factory cannot be used. Due to differing standards, there will not be a common abstract class for each type of widget.
3. Do not create a new, nonstandard windowing system

Solution and Pattern (Bridge)

- It is possible to develop "our own abstract and concrete product classes", because "all window systems do generally the same thing". Each window system provides operations for drawing primitive shapes, iconifying/de-iconfiying, resizing, and refreshing window contents.
- An abstract base `Window` class can be derived to the different types of existing windows, such as application, iconified, dialog. These classes will contain operations that are associated with windows, such as reshaping, graphically refreshing, etc. Each window contains elements, whose `Draw()` functions are called upon by the `Window`'s own draw-related functions.
- In order to avoid having to create platform-specific Window subclasses for every possible platform, an interface will be used. The `Window` class will implement a `Window` implementation (`WindowImp`) abstract class. This class will then in turn be derived into multiple platform-specific implementations, each with platform-specific operations. Hence, only one set of `Window` classes are needed for each type of `Window`, and only one set of `WindowImp` classes are needed for each platform (rather than the Cartesian product of all available types and platforms). In addition, adding a new window type does not require any modification of platform implementation, or vice-versa.
- This is a **Bridge** pattern. `Window` and `WindowImp` are different, but related. `Window` deals with windowing in the program, and `WindowImp` deals with windowing on a platform. One of them can change without ever having to modify the other. The Bridge pattern allows these two "separate class hierarchies to work together even as they evolve independently".

6 User Operations

All actions the user can take with the document, ranging from entering text, changing formatting, quitting, saving, etc.

Problems and Constraints

1. Operations must be accessed through different inputs, such as a menu option and a keyboard shortcut for the same command
2. Each option has an interface, which should be modifiable
3. Operations are implemented in several different classes
4. In order to avoid coupling, there must not be a lot of dependencies between implementation and user interface classes.
5. Undo and redo commands must be supported on most document changing operations, with no arbitrary limit on the number of levels of undo

6. Functions are not viable, since they don't undo/redo easily, are not easily associated with a state, and are hard to extend or reuse.
7. Menus should be treated like hierarchical composite structures. Hence, a menu is a menu item that contains menu items which may contain other menu items, etc.

Solution and Pattern (Command)

- Each menu item, rather than being instantiated with a list of parameters, is instead done with a *Command* object.
- Command is an abstract object that only has a single abstract `Execute()` method. Derivative objects extend the `Execute()` method appropriately (i.e., the `PasteCommand.Execute()` would utilize the content's clipboard buffer). These objects can be used by widgets or buttons just as easily as they can be used by menu items.
- To support undo and redo, `Command` is also given `Unexecute()` and `Reversible()`. In derivative classes, the former contains code that will undo that command, and the latter returns a boolean value that defines if the command is undoable. `Reversible()` allows some commands to be non-undoable, such as a Save command.
- All executed `Commands` are kept in a list with a method of keeping a "present" marker directly after the most recently executed command. A request to undo will call the `Command.Unexecute()` directly before "present", then move "present" back one command. Conversely, a `Redo` request will call `Command.Execute()` after "present", and move "present" forward one.
- This **`Command`** history is an implementation of the **Command pattern**. It encapsulates requests in objects, and uses a common interface to access those requests. Thus, the client can handle different requests, and commands can be scattered throughout the application.

7 Spell Check and Hyphenation

This is the document editor's ability to textually analyze the contents of a document. Although there are many analysis that can be performed, spell check and hyphenation-formatting are the focus.

Problems and Constraints

1. Allow for multiple ways to check spelling and identify places for hyphenation
2. Allow for expansion for future analysis (e.g., word count, grammar check)
3. Be able to iterate through a text's contents without access to the text's actual structure (e.g., array, linked list, string)
4. Allow for any manner of traversal of document (beginning to end, end to beginning, alphabetical order, etc.)

Solution and Pattern (Iterator)

- Removing the integer-based index from the basic element allows for a different iteration interface to be implemented. This will require extra methods for traversal and object retrieval. These methods are put into an abstract `Iterator` interface. Each element then implements a derivation of the `Iterator`, depending on how that element keeps its list (`ArrayIterator`, `LinkListIterator`, etc.).
- Functions for traversal and retrieval are put into the abstract Iterator interface. Future Iterators can be derived based on the type of list they will be iterating through, such as Arrays or Linked Lists. Thus, no matter what type of indexing method any implementation of the element uses, it will have the

appropriate Iterator.

- This is an implementation of the Iterator pattern. It allows the client to traverse through any object collection, without needing to access the contents of the collection directly, or be concerned about the type of list the collection's structure uses.
- Now that traversal has been handled, it is possible to analyze the elements of a structure. It is not feasible to build each type of analysis into the element structure themselves; every element would need to be coded, and much of the code would be the same for similar elements.
- Instead, a generic `CheckMe()` method is built into the element's abstract class. Each Iterator is given a reference to a specific algorithm (such as spell check, grammar check, etc.). When that Iterator iterates through its collection, it calls each element's `CheckMe`, passing the specified algorithm. `CheckMe` then passes a reference to its element back to said algorithm for analysis.
- Thus, to perform a spell check, a front-to-end iterator would be given a reference to a `SpellCheck` object. The iterator would then access each element, executing its `CheckMe()` method with the `SpellCheck` parameter. Each `CheckMe` would then call the `SpellCheck`, passing a reference to the appropriate element.
- In this manner, any algorithm can be used with any traversal method, without hard-code coupling one with the other. For example, Find can be used as "find next" or "find previous", depending on if a "forward" iterator was used, or a "backwards" iterator.
- In addition, the algorithm themselves can be responsible for dealing with different elements. For example, a `SpellCheck` algorithm would ignore a `Graphic` element, rather than having to program every `Graphic`-derived element to not send themselves to a `SpellCheck`.

4 GoF Design Patterns

1 Creational patterns

These patterns have to do with class instantiation. They can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation to get the job done.

- **Abstract Factory** groups object factories that have a common theme.
- **Builder** constructs complex objects by separating construction and representation.
- **Factory Method** creates objects without specifying the exact class to create.
- **Prototype** creates objects by cloning an existing object.
- **Singleton** restricts object creation for a class to only one instance.

2 Structural patterns

These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

- **Adapter** allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- **Bridge** decouples an abstraction from its implementation so that the two can vary independently.
- **Composite** composes zero-or-more similar objects so that they can be manipulated as one object.
- **Decorator** dynamically adds/overrides behaviour in an existing method of an object.
- **Facade** provides a simplified interface to a large body of code.
- **Flyweight** reduces the cost of creating and manipulating a large number of similar objects.
- **Proxy** provides a placeholder for another object to control access, reduce cost, and reduce complexity.

3 Behavioral patterns

Most of these design patterns are specifically concerned with communication between **objects**.

- **Chain of responsibility** delegates commands to a chain of processing objects.
- **Command** creates objects which encapsulate actions and parameters.
- **Interpreter** implements a specialized language.
- **Iterator** accesses the elements of an object sequentially without exposing its underlying representation.
- **Mediator** allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- **Memento** provides the ability to restore an object to its previous state (undo).
- **Observer** is a publish/subscribe pattern which allows a number of observer objects to see an event.

- **State** allows an object to alter its behavior when its internal state changes.
- **Strategy** allows one of a family of algorithms to be selected on-the-fly at runtime.
- **Template method** defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- **Visitor** separates an algorithm from an object structure by moving the hierarchy of methods into one object.

5 Motivation

- DP's aim is to catalog common object structures/ interactions that programmers have found useful.
- Learn to recognize problems that fall into familiar categories.
- Rely on literature such as *Design Patterns - Elements of Reusable Software* Gamma et-al.
- **Less re-inventing & re-thinking: -** Become familiar with common solutions and trade-offs associated with them.
- **Humility -** learn from the accumulated experience of others.
- **flexible programs -** Use it to create sophisticated, easy-to-maintain
- **A common vocabulary -** amongst guild members.
- When maintaining code written by others, learn to recognize (and perhaps criticize) the **design choices of other programmers.**