**My Reading Notes from Design Patterns: Elements of Reusable Object-Oriented Software**

[[](https://danblevins.medium.com/?source=post_page-----fc813ae05802--------------------------------)](https://danblevins.medium.com/?source=post_page-----fc813ae05802--------------------------------)

[Dan Blevins](https://danblevins.medium.com/?source=post_page-----fc813ae05802--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fb7081a9d2646&operation=register&redirect=https%3A%2F%2Fdanblevins.medium.com%2Fmy-reading-notes-from-design-patterns-elements-of-reusable-object-oriented-software-fc813ae05802&user=Dan+Blevins&userId=b7081a9d2646&source=post_page-b7081a9d2646----fc813ae05802---------------------post_header-----------)

12 min read

·

Sep 9, 2021

6

Visit [Amazon.com](https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612) to learn more about Design Patterns.

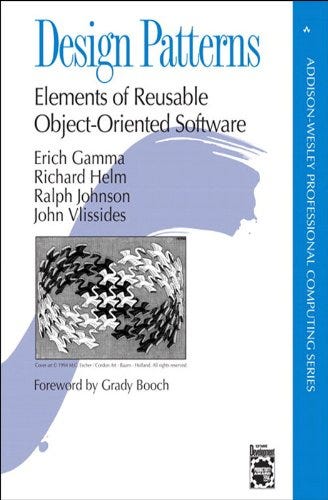


Image from: <https://images-na.ssl-images-amazon.com/images/I/51kuc0iWoKL.jpg>

**Chapter 1: Introduction**

1**. One thing expert designers know *not* to do is solve every problem from first principle**. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again.

2. In general, a pattern has four essential elements.

* Pattern name: Is a handle we can use to describe a design problem, its solutions, and consequences. Naming a pattern immediately increases our design vocabulary.
* Problem: Describes when to apply the pattern. It explains the problem and its context. It might describe design problems such as how to represent algorithms as object or describe classes/ structures.
* Solution: Describes the elements that make up the design, their relationships, responsibilities, and collaboration. The solution doesn’t describe a particular concrete design or implementation.
* Consequences: Are results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits.

3. There are two benefits to manipulating objects solely in terms of the interface defined by abstract classes:

* Clients remain unaware of the specific types of objects they use.
* Clients remain unaware of the classes that implement these objects.

4. **The First principle of Object-oriented Design:** Program to an interface, not an implementation. Commit only to an interface defined by an abstract class.

5. The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition:

* Class inheritance lets you define the implementation of one class in terms of another’s. Referred to as “white-box reuse” because with inheritance the internals of parent classes are often visible to subclasses.
* New functionality using object composition is obtained by assembling or composing objects to get more complex functionality. This is called “black-box reuse” because internal details are not visible.

6. **The Second Principle of Object-Oriented Design:**Favor object composition over class inheritance.

7. Delegation is a way of making composition as powerful for reuse as inheritance. Delegation is analogous to subclasses deferring requests to parent classes. It shows that you can always replace inheritance with object composition as a mechanism for code reuse.

* The main advantage of delegation: It makes it easy to compose behaviors at run-time.

8. The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing systems so that they can evolve accordingly. Here are **some common causes of redesign**:

* *Creating an object by specifying a class explicitly.*Specifying a class name when you create an object commits you to a particular implementation instead of a particular index. To avoid this, create objects indirectly.
* *Dependence on specific operations.* When you specify a particular operation, you commit to one way of satisfying a request, so it’s best to avoid hard-coded requests.
* *Dependence on hardware and software.* External operating system interfaces and APIs are different on different hardware and software platforms. So, it’s important to design your system to limit its platform dependencies.
* *Dependence on object representations or implementations.* Hiding object representation, storage, and location from clients keep changes from cascading.
* *Algorithmic dependencies.*Algorithms are often extended or replaced during development and reuse. Therefore, algorithms that are likely to change should be isolated.
* *Tight coupling*. Classes that are tightly coupled are hard to reuse in isolation. Loose coupling increases the probability that a class can be reused.

9. If you’re building an application program (such as a document editor or spreadsheet), then internal reuse, maintainability, and extension are high priorities.

10. **How to Select a Design Pattern:**

* Consider how design patterns solve design problems.
* Scan the problems intent.
* Study the relationships of different patterns. Then, study patterns of similar purpose.
* Examine a cause for any future redesign.
* Consider what should be variable in your design.

11. **How to Use a Design Pattern:**

* Read the pattern once through for an overview.
* Study and understand the pattern’s structure, participants, and collaborations.
* Review concrete examples of your pattern.
* Choose names for pattern participants that are meaningful in the application context.
* Define the classes.
* Define application-specific names for operations in the pattern.
* Implement the operations to carry out the responsibilities and collaborations in the pattern.

**Chapter 2: Case Study for Designing a Document Editor**

1. Recursive composition: Entails building increasingly complex elements out of simpler ones. This gives us a way to compose a document out of simple graphical elements.

2. The Composite Pattern: Composes objects into tree structures to represent part-whole hierarchies. Composites let clients treat individual objects and compositions of objects uniformly. Use the Composite Pattern when:

* You want to represent part-whole hierarchies of objects.
* You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat objects in the composite structure uniformly.

3. The Strategy Pattern: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm very independently from clients that use it. Use the Strategy Pattern when:

* Many related classes differ only in their behavior.
* You need different variations of an algorithm.
* An algorithm uses data that clients shouldn’t know about.
* A class defines many behaviors.

4. The Decorator Pattern: Provides a flexible alternative to subclassing for extending functionality. Use the Decorator Pattern:

* To add responsibilities to individual objects dynamically and transparently, without affecting other objects.
* When extension by subclassing is impractical.

5. The Abstract Factory Pattern: Captures how to create families of related product objects without instantiating classes directly. It’s most appropriate when the number and general kinds of product objects stay constant. Use the Abstract Factory Pattern when:

* A system should be independent of how its products are created.
* A family of related product objects is designed to be used together, and you need to enforce this constraint.
* You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

6. The Bridge Pattern: Allows separate class hierarchies to work together even as they evolve independently. Use the Bridge Pattern when:

* Both abstractions and their implementations should be extensible by subclassing.
* Changes in implementation of an abstraction should have no impact on clients aka the code should not have to be recompiled.

7. The Command Pattern: Prescribes a uniform interface for issuing requests that lets you configure clients to handle different requests. Use the Command Pattern to:

* Specify, queue, and execute requests at different times.
* Parameterize objects by an action to perform with a callback function that’s registered somewhere to be called at a later point.

8. The Iterator Pattern: Captures techniques for supporting access and traversal over object structures. It abstracts the traversal algorithm and shields clients from the internal structure of the objects they traverse. Use the Iterator Pattern to:

* Access an aggregate object’s contents without exposing its internal representation.
* Support multiple traversals of aggregate objects.
* Provide a uniform interface for traversing different aggregate structures.

9. The Visitor Pattern: Captures the technique we’ve used to allow an open-ended number of analytical capabilities without complicating the document structure’s implementation. Use the Visitor Pattern when:

* An object structure contains many classes of objects with differing interfaces.
* Many distinct and unrelated operations need to be performed on objects in an object structure.
* The classes defining the object structure rarely change, but you often want to define new operations over the structure.

**Chapter 3: Creational Patterns**

1. Creational Patterns become more important as systems evolve to depend more on object composition than class inheritance. Emphasis shifts away from hardcoding fixed behaviors toward defining a smaller set of fundamental behaviors.

2. Creational Patterns have two recurring themes:

* They encapsulate knowledge about which concrete classes the system uses.
* They hide how instances of these classes are created and put together.

3. Prototype Patterns specify the kinds of objects to create using an instance, which then can create new objects by copying the prototype. Use the Prototype Pattern when a system should be independent of how its products are created, composed, and represented.

4. Builder Patterns separate the construction of a complex object from its representation so that the same construction process can create different representations.

5. Singleton Patterns ensure a class only has one instance, and provides a global point of access to it. Singleton Patterns should be used when:

* There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
* When clients should be able to use an extended instance without modifying their code.

6. Factory Method Pattern defines an interface for creating an object, but let subclasses decide which classes to instantiate. Factory Method lets a class defer instantiation to subclasses. Use the Factory Method Pattern when:

* A class can’t anticipate the class of objects it must create.
* A class wants its subclasses to specify the objects it creates.

7. There are two common ways to parameterize a system by the classes of objects it creates:

* One way is to subclass the class that creates the objects (similar to the Factory Method Pattern). The main drawback here is that it can require creating a new subclass just to change the class of the product.
* The other way relies on defining an object that’s responsible for knowing the class of the product objects, and make it a parameter of the system (similar to the Abstract Factory Pattern, Builder Pattern, and Prototype Pattern).

8. The Factory Method Pattern makes a design more customizable and only a little more complicated. People often use the Factory Method as the standard way to create objects.

9. Designs that use the Abstract Factory, Prototype, or Builder Patterns are even more flexible, but they’re often more complex.

**Chapter 4: Structural Patterns**

1. Structural Patterns are concerned with how classes and objects are composed to form larger structures.

2. Rather than composing interfaces or implementations, structural patterns describe ways to compose objects to realize new functionality. This added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

3. The Composite Pattern composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. Use the Composite Pattern when:

* You want to represent a part-whole hierarchy.
* You want clients to be able to ignore the difference between compositions of objects and individual objects.

4. The Proxy Pattern acts as a convenient surrogate or placeholder for another object. A Proxy Pattern can be used in many ways. It can:

* Act as a local representative for an object in a remote address.
* Represent a large object that should be loaded on-demand.
* Protect access to a sensitive object.

5. The Flyweight Pattern defines a structure for sharing objects. Flyweight focuses on sharing for space efficiency. However, applications that use lots of objects must pay careful attention to the cost of each object. Overall, the Flyweight pattern’s effectiveness depends heavily on how and where it’s used.

6. The Facade Pattern shows how to make a single object represent an entire subsystem. A facade is a representative for a set of objects. The facade carries out its responsibilities by forwarding messages to the objects it represents.

7. The Bridge Pattern separates an object’s abstraction from its implementation so that you can vary them independently. Use the Bridge Pattern when:

* You want to avoid a permanent binding between an abstraction and its implementation.
* Both the abstractions and their implementations should be extensible by subclassing.
* Changes in the implementation of an abstraction should have no impact on clients.

8. The Decorator Pattern describes how to add responsibilities to objects dynamically. The Decorator is a structural pattern that composes objects recursively to allow an open-ended number of additional responsibilities. Decorators provide a flexible alternative to subclassing for extending functionality. Use Decorator:

* To add responsibilities to individual objects dynamically and transparently (without affecting other objects).
* When extension by subclassing is impractical

9. Structural Patterns rely on the same small set of language mechanisms for structuring code and objects: single and multiple inheritance for class-based patterns, and object composition for object patterns.

**Chapter 5: Behavioral Patterns**

1. Behavioral Patterns are concerned with algorithms and the assignment of responsibilities between objects.

2. Behavioral Patterns describe patterns of objects/classes and the patterns communication between them.

3. There are two patterns that use inheritance to distribute behavior between classes: the Template Method and the Interpreter.

4. The Template Method defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. It lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure. The Template Method Pattern should be used:

* To implement the invariant parts of an algorithm once and leave subclasses to implement the behavior.
* To control subclasses extensions.

5. The Interpreter Pattern represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of classes. Use the Interpreter Pattern when there is a language to interpret and you can represent statements in the language as abstract syntax trees.

6. The Mediator Pattern defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects referring to each other explicitly , and it allows you to vary their interaction independently. Use the Mediator Pattern when:

* reusing an object is difficult because it refers to and communicates with many other objects.
* a behavior that’s distributed between several classes should be customizable without a lot of subclassing.

7. Chain of Responsibility lets you send requests to an object implicitly through a chain of candidate objects. You can chain the receiving objects and pass the request along the chain until an object handles it. Use Chain of Responsibility when:

* You want to issue a request to one of several objects without specifying the receiver explicitly.
* The set of objects that can handle a request should be specified dynamically.

8. The Observer Pattern is the classic example of Model/View/Controller, where all views of the model are notified whenever the model’s state changes. Use the Observer Pattern when:

* An abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
* An object should be able to notify other objects without making assumptions about who these objects are.

9. The Strategy Pattern defines a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. Use the Strategy Pattern when:

* Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
* You need different variants of an algorithm.
* A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

10. The Command Pattern encapsulates a request in an object so that it can be passed as a parameter, stored on a history list, or manipulated in other ways. Use the Command Pattern when you want to:

* Parameterize objects by an action to perform. You can express parameterization in a procedural language with a callback function.
* Specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request.
* Support undo. The Command’s Execution operation can store state for reversing its effects in the command itself.

11. The State Pattern encapsulates the states of an object can change its behavior when its state object changes. Use the State Pattern in either of the following cases:

* An object’s behavior depends on its state, and it must change its behavior at run-time depending on that state.
* The State Pattern puts each branch of the conditional in a separate class. This lets you treat the object’s state as an object in its own right that can vary independently from other objects.

12. The Visitor Pattern represents an operation to be performed on the elements of an object structure. It lets you define a new operation without changing the classes of the elements on which it operates. Use the Visitor Pattern when:

* An object structure contains many classes o the objects with differing interfaces, and you want to perform operations on the objects that depend on their concrete classes.
* Many distinct and unrelated operations need to be performed on objects in an object structure.
* The classes defining the object structure rarely change, but you often want to define new operations over the structure.

**Chapter 6: Conclusion**

* Design patterns provide a common vocabulary for designers to communicate, document, and explore design alternatives, which makes a system seem less complex.
* Once your software has reached adolescence and is put into service, its evolution will then be governed by two conflicting needs: 1) The software must satisfy more requirements and 2) The software must be more reusable.
* The Refactoring Cycle: Prototyping > More Requirements > Expansion > More Reuse > Consolidation > Prototyping
* Design patterns provide targets for the refactoring cycle.
* The best designs will use many design patterns that dovetail and intertwine to produce a greater whole.