The GoF Patterns are now required learning for most developers. GoF (Gang of Four/Gamma et al.) Patterns are now part of software folklore, used daily in real-world development and mandatory education for students and practioners seeking certification, as in the Sun Certified Enterprise Architect (SCEA) exam. The tips on this page help you get a heads-up as quick as possible. They are based on the author's experiences in using the patterns, mentoring other developers, and lecturing at university.

**Tip: Learn Patterns In The Following Order**

Here's the most important tip. The original text groups patterns in three categories, and then alphabetically within each. All good for reference purposes, but lousy for learning. For example, the first pattern is Abstract Factory, one of the more sophisticated entries. A much simpler starting point, Facade, is buried in the middle. The alphabetical order is unnecessary, and even the categories are not especially meaningful, for learning purposes. So it's clear you need to learn the patterns in a different order. The problem is you don't know them yet, so how can a pattern newcomer decide on an order? The answer is to use an existing "learner's roadmap" - that's why I'm listing a suitable order below. Hopefully, any future revision of the book will do likewise. The order I recommend is based on three principles:

- Group similar patterns, as the commonalities will reduce the overall effort. You're effectively learning 16 or so patterns with some variations, rather than 23 individual patterns.
- Early on, focus on the simpler patterns.
- Early on, focus on the patterns you have probably already encountered.

With that in mind, here is the order I recommend. As in the book, page numbers are shown in parentheses.

**Easy GoF Patterns.** These easy patterns concern only one class, or a straightforward relationship among several classes. Incidentally, don't discount them as being ingenious; their ease of learning actually stands testimony to their cleverness. Learn in the following order ...

1. Facade (185) All the ingredients for a starting pattern. Conceptually clear, solves a problem many developers have experienced, many real-world metaphors. Good introduction to the world of patterns.
2. Singleton (127) Also very easy to follow.
4. Iterator (257) Conceptually simple problem and solution, Should be a breeze for anyone who has looped across a collection using the standard Java 1.0-1.4 iteration mechanism.
5. Strategy (315) The first pattern here to be centered around polymorphism, making it a bit more sophisticated. But it's still quite simple as it concerns a single class hierarchy.
6. Command (233) SIMILARITY: Command is almost identical to Strategy. That is, you call an "execute" method and the instance executes itself. The only difference here is Command "feels" more dynamic - it is, at some level, triggered by an external agent (e.g. an end-user). In contrast, a Strategy often represents a business rule, e.g. tax calculation. So the application may vary, but what actually happens is identical.
7. Builder (97) SIMILARITY: This is really just a special case of Strategy, applied specifically to the problem of creating an object. The name is misleading, because the most important logic is actually in the Director. The Director has a fixed high-level procedure, but the actual construction depends on what its building, and that construction process is encapsulated in a Strategy object called the Builder.
8. State (305) SIMILARITY: This pattern is quite similar to Strategy and Command - keep these patterns in mind. It is based on polymorphism, and once again is centered around a single superclass/interface with any number of subclasses. Unlike those patterns, it is not intended to perform a single action. Instead, is says how any number of actions are performed while in a certain state. Don't complicate things by worrying about state transitions; that's actually not the main point of State. Once you understand that State is about what happens in state X, the concept of state transitions follows naturally.
9. Template Method (325) Another pattern involving polymorphism. Still only a single class hierarchy. Also important to see how this is an alternative to Strategy.
10. Factory Method (107) SIMILARITY: This pattern is just a special case of Template Method. As stated in the text, it applies only to Template Methods which involve the creation of an object. Note that most people nowadays use a different definition: a public creation method, typically of the class (or subclass) that the method belongs to.
11. Memento (283) With its involvement in Undo functionality, this pattern is related to Command. It's easy to understand once you see that certain commands cause loss of information. That is, you can't reverse the effect of a command just by knowing the final state. For instance, the command "multiply by zero" will always lead to zero, so how can you reverse it to get the original number? The answer is to attach the original number to the command. The attached number is a memento
12. Prototype (117) Actually very simple, but is listed last here because it is not used very much in practice. Actually, the structure diagram shows a class hierarchy, but the pattern is essentially about a single class being cloned.
Intermediate GoF Patterns. Now that you've been exposed to the basic patterns, you are well equipped to read up on pattern theory in general. Then, when you're ready to proceed ...

13. **Proxy** (207) Should be familiar to anyone who has used a web proxy.

14. **Decorator** (175) SIMILARITY: Proxy is a special case of Decorator. It's worth learning Proxy first, because it's a more familiar situation. Once understood, it's not so hard to see how a Decorator could wrap classes to perform tasks other than those performed by Proxy. Proxy is usually used with regard to security and networking, and typically a class with a proxy cannot be accessed directly. In contrast, a Decorator is usually used to enhance an existing class's functionality. A client can freely choose whether to wrap a class with its Decorator(s). As with the Strategy-Command relationship, Decorator shares the same mechanism as Proxy, varying only in the application area and style of use.

15. **Adapter** (139) Learn along with Bridge (below).

16. **Bridge** (151) SIMILARITY: Similar to Adaptor, but with subtle differences, to Adapter. Good to learn these in tandem and most importantly, to understand the emphasis here is on up-front design, whereas Adapter is usually used to avoid changing existing classes. So a Bridge is usually designed when neither end is in place. But both ends (sometimes just one end) of an Adapter already exist before the Adapter is written to glue them together. Many Java frameworks are an example of Bridge, where a set of interfaces are listed and individual vendors can then provide their own implementations.

17. **Observer** (293) Makes some sophisticated use of object-oriented constructs, but not especially difficult as there are some great real-world metaphors, viz., publishing and broadcasting. Also, should be intimately familiar for anyone who has used a web proxy or subscription protocols like RSS.

Advanced GoF Patterns. Don't worry - they're all very accessible. A design pattern is supposed to be a distinct, simple, idea, and the GoF patterns have certainly adhered to this principle ...


19. **Interpreter** (243) SIMILARITY: Interpreter may seem a bit unfamiliar as you've probably not written a parser before, but it's actually just a special case of Composite applied to parsing. Note that this pattern is the odd one out in the overall GoF collection, because it is domain-specific - its applicability is limited to parsing.

20. **Chain Of Responsibility** (223) Best learned with regard to a Composite structure. Confronted with a request, children try to handle it, and if they can't, they pass it to their parents, and so on, until eventually some class along the chain can deal with it.

21. **Abstract Factory** (87) Often misinterpreted as being just a Factory, but actually it's more than that because it allows for creation of class families.

22. **Flyweight** (195) Think of a pool of expensive objects which are shared, such as EJBs or database connections. This pattern may not be familiar to everyone, hence this pattern is considered a bit more difficult.

23. **Visitor** (331) Alphabetical order lined up with complexity in this case, because this pattern does deserve its place at the end of the GoF text! It helps to see Visitor as an after-the-fact substitute for multiple inheritance.

**Tip: It's the Patterns, Stupid!**

The pattern format is simple enough to require no introduction. Read the introductory material, and any other info on pattern theory, only when you are confident with some of the patterns themselves.

**Tip: Only Read High-Level Pattern Details**

The great thing about patterns is their consistent, well-organised, format. Use this format to concentrate on high-level information and filter out the detail such as code segments. On first reading, you only need to familiarise yourself with the "Intent", then glance at the "Structure" diagram, and finally read just the "Motivation". Only read other sections if you don't understand anything here.

**Tip: Think of Real-World Metaphors**

I'm not talking about how you've used the patterns in existing software. I'm talking about patterns of social structures, building architecture, and so on, which are similar to the design patterns. For example, the Facade pattern is illustrated by an office receptionist, and the Proxy pattern is illustrated by a CEO's personal assistant.

**Tip: Therefore, BOOM!**

Here is a tip for authors of patterns which will also help you in learning them. A pattern should read as "Therefore, BOOM!". A clear problem that frustrates you, makes you think "how can I ever deal with this", and then BOOM! You slap yourself in the realisation that such a simple solution exists. The GoF patterns are cunning in their simplicity - if you understand the pattern, you should be able to state the problem in one sentence and the solution in another.