Patterns—Common Elements and Forms

A pattern is a literary form. Several forms have emerged and evolved over the years. This document summarizes the most popular pattern forms, drawing directly on their originators insights.

Common Elements: an explanation of the most common structural elements found in patterns.

Intent/ Question: The intent is a phrase or sentence that summarizes what the pattern does, describing the design issue or problem it addresses. As a designer scans patterns for solutions to a specific problem, the intents provide a road map into promising patterns and around irrelevant ones.

Problem: the problem section describes the problem to be solved. A concise problem statement helps the problem solver decide whether to read further, and it often serves as the primary index for pattern selection.

Context: Context includes a history of patterns that have been applied before the current pattern was considered, It also specifies size, scope, market, programming language, or anything else that, if changed, would invalidate the pattern. Pattern context is particularly crucial to the success of a pattern language, a collection of patterns that work together to solve system-level problems. Contexts weave patterns together into a pattern language. Context sections mature with experience: as designers find special situations that invalidate the pattern, the context grows to become more restrictive.

Forces: The forces should amplify and illustrate the problem statement because it is through the forces that one fully appreciates the problem. If we understand the forces in a pattern, then we understand the problem (because we understand the trade-offs) and the solution (because we know how it balances the forces). As patterns mature they move past purely technical and mechanical forces and take human forces into account.

Solution: A good solution has enough detail so the designer knows what to do, but it is general enough to address a broad context. Some patterns provide only partial solutions but open a path to other patterns that balance unresolved forces.

Sketch—(the purpose of the sketch is to indicate structure and indicate the relationship between parts) the meaning behind the word sketch is important here. The sketch associated with a pattern should not be taken as a graphical specification. Many readers will interpret a finely refined diagram to literally. Rough sketches are valuable because they refocus the solution back on to the context. Beautiful drawings can become ends in themselves.

Resulting Context—Each pattern is designed to transform a system in one context to a new context. The "resulting context" of one pattern is input to the patterns that follow. Context ties related patterns together into a pattern language. Simply put, the resulting context is the wrap up of the pattern.
It tells us:

- Which forces were resolved
- Which new problems may arise because of this pattern
- What related patterns may come next?

Five Forms: A discussion of the most common forms found across the pattern language community:

The five forms that are discussed below are as follows:

1. Alexanderian form
2. GOF form
3. Coplien Form
4. Portland Form
5. Canonical Form

Alexanderian Form:

The Alexanderian form, from Christopher Alexander's work, is the original pattern form. The sections of an Alexanderian pattern are not strongly delimited. The major syntactic structure is a *Therefore* immediately preceding the solution. Other elements of the form are usually present: a clear statement of the problem, a discussion of forces, the solution, and a rationale. Each Alexanderian pattern usually follows an introductory paragraph that enumerates the patterns that must already have been applied to make the ensuing pattern meaningful. The pattern itself starts with a name and a confidence designation of zero, one, or two stars. Patterns with two stars are the ones in which the authors have the most confidence because they have empirical foundations. Patterns with fewer stars may have strong social significance but are more speculative.

Alexander Patterns have five parts:
Name.
A short familiar, descriptive name or phrase, usually more indicative of the solution than of the problem or context. Examples include *Alcoves*, *Main entrance*, *Public outdoor room*, *Parallel roads*, *Density rings*, *Office connections*, *Sequence of sitting spaces*, and *Interior windows*.
Example.
One or more pictures, diagrams, and/or descriptions that illustrate prototypical application.
Context.

Delineation of situations under which the pattern applies. Often includes background, discussions of why this pattern exists, and evidence for generality.

Problem.

A description of the relevant forces and constraints, and how they interact. In many cases, entries focus almost entirely on problem constraints that a reader has probably never thought about. Design and construction issues sometimes themselves form parts of the constraints.

Solution.

Static relationships and dynamic rules (microprocess) describing how to construct artifacts in accord with the pattern, often listing several variants and/or ways to adjust to circumstances. Solutions reference and relate other higher- and lower-level patterns.

Here is Alexander s own description of his form:

"For convenience and clarity, each pattern has the same format. First, there is a picture, which shows an archetypal example of that pattern. Second, after the picture, each pattern has an introductory paragraph,which sets the context for the pattern, by explaining how it helps to complete certain larger patterns. Then there are three diamonds to mark the beginning of the problem. After the diamonds there is a headline, in bold type. This headline gives the essence of the problem in one or two sentences. After the headline comes the body of the problem. This is the longest section. In describes the empirical background of the pattern, the evidence for its validity, the range of different ways the pattern can be manifested in a building, and so on. Then, again in bold type, like the headline, is the solution the heart of the pattern which describes the field of physical and social relationships which are required to solve the stated problem, in the stated context. This solution is always stated in the form of an instruction so that you know exactly what you need to do, to build the pattern. Then, after the solution, there is a diagram, which shows the solution in the form of a diagram, with labels to indicate its main components. After the diagram, another three diamonds, to show that the main body of the pattern is finished. And finally, after the diamonds there is a paragraph which ties the pattern to all those smaller patterns in the language, which are needed to complete this pattern, to embellish it, to fill it out." (Alexander et al., 1977: pp.x xi)

Example:

Simply Understood Code:

At the lowest levels of a program are chunks of code. These are the places that need to be understood to confidently make changes to a program, and ultimately understanding a program thoroughly requires understanding these chunks. In many pieces of code the problem of disorientation is acute. People have no idea what each component of the code is for and they experience considerable mental stress as a result. Suppose you are writing a chunk of code that is not so complex that it requires

extensive documentation or else it is not central enough that the bother of writing such documentation is worth the effort, especially if the code is clear enough on its own. How should you approach writing this code?

People need to stare at code in order to understand it well enough to feel secure making changes to it. Spending time switching from window to window or scrolling up and down to see all the relevant portions of a code fragment takes attention away from understanding the code and gaining confidence to modify it. People can more readily understand things that they can read it their natural text reading order; for Western culture this is generally left to right, top to bottom.

If code cannot be confidently understood, it will be accidentally broken.

Therefore,
Arrange the important parts of the code so it fits on one page. Make that code understandable to a person reading it from top to bottom. Do not require the code to be repeatedly scanned in order to understand how data is used and how control moves about.

This pattern can be achieved by using the following patterns:

- **Local Variables Defined and Used on One Page,** which tries to keep local variables on
- one page;
- **Assign Variables Once,** which tries to minimize code scanning by having variables
- changed just once;
- **Local Variables Reassigned Above their Uses**, which tries to make a variable s
- value apparent before its value is used while scanning from top to bottom;
- **Make Loops Apparent**, which helps people understand parts of a program that are non-linear while retaining the ability to scan them linearly;
- **Use Functions for Loops**, which packages complex loop structure involving several state variables into chunks, each of which can be easily understood. (Gabriel, 1995)

The GOF Form:

The GOF (Gang of Four) Form was established in *Design Patterns* (Gamma et al., 1995). It has the following sections:

- **Pattern Name and Classification:** The pattern s name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary...
- **Intent**: A shorn statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?
- **Also Known As:** Other well-known names for the pattern, if any.
- **Motivation**: A scenario that illustrates a design problem and how the class

and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

- **Applicability**: What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?
- **Structure**: A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [Rumbaugh et al., 19911. We also use interaction diagrams (Jacobson et al., 1992; Booch, 1994) to illustrate sequences of requests and collaborations between objects...
- **Participants**: The classes and /or objects participating in the design pattern and their responsibilities.
- **Collaborations**: How the participants collaborate to carry out their responsibilities.
- **Consequences**: How does the pattern support its objectives?What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary ndependently?
- **Implementation**: What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues? Sample Code: Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.
- **Known Uses**: Examples of the pattern found in real systems. We include at least two examples from different domains.
- **Related Patterns:** What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?
- (Gamma et al., 1995: pp. 6 7)

Example :

Name--The Bridge Pattern
Intent
Decouple an abstraction from its implementation so that the two can vary independently.
Also Known As
Handle/Body
Motivation
When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance. An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways. But this approach isn't always flexible enough. Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently. The Bridge pattern addresses these

problems by punning the ... abstraction and its implementation in separate class hierarchies.

Applicability

- 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; that is, their code should not have to be recompiled.
- (C++) you want to hide the implementation of an abstraction completely from clients...
- You have a proliferation of classes... Such a class hierarchy indicates the need for splitting an object into two parts...
- You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client.

- A simple example is Coplien's String class [Coplien, 1992], in which multiple objects can share the same string representation (StringRep). (Gamma et al., 1995: pp. 151—153)

Name

Mediator

Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Motivation

Object-oriented design encourages the distribution of behavior among objects. Such distribution can result in an object structure with many connections between objects; in the worst case, every object ends up knowing about every other. Though partitioning a system into many objects generally enhances reusability, proliferating interconnections

Interconnections tend to reduce it again. Lots of interconnections make it less likely that an object can work without the support of others the system acts as though it were

monolithic. Moreover, it can be difficult to change the system s behavior in any significant way, since behavior is distributed among many objects. As a result, you may be forced to define subclasses to customize the system s behavior. You can avoid these problems by encapsulating collective behavior in a separate **mediator** object. A mediator

is responsible for controlling and coordinating the interactions of a group of objects. The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly. The objects only know the mediator, thereby reducing the number of interconnections.

Consequences

1. *It limits subclassing...*
2. *It decouples colleagues...*
3. *It simplifies object protocols...*
4. *It abstracts how objects cooperate...*
5. *It centralizes control*

The Portland Form:

Ward Cunningham maintains an on-line repository of patterns called the Portland Pattern Repository. Many of the patterns found in that forum follow the Portland Form, of which Cunningham writes: The repository prefers patterns written in the Portland Form, a form first adopted by three authors submitting papers to the Pattern Languages of Programs conference, PLoP 94. (All three were from Portland, Oregon, hence the name.) The form has been described as narrative, as opposed to the more outline like form of keyword templates first used by Peter Coad and made popular by Erich Gamma et al. The form is actually a fairly direct emulation of Alexander s form with some simplification in typesetting. We hope that the hypertext aspects of the repository will more than make up
for the omissions and simplifications of the Portland Form.

Language Document
Each document in the Portland Form contains a system of patterns that work together. Alexander calls such systems languages since he believes the human mind assembles the words of a natural language, namely, without much conscious thought. The Portland Form collects and connects patterns so that they will be studied and understood as a whole. Although we believe all patterns will be ultimately linked, we currently give authors the responsibility of defining a suitable whole consistent with their own Knowledge and their readers ability to absorb. This unit we call both a language and a document...

Pattern Paragraphs
Each pattern in the Portland Form makes a statement that goes something like: such and so forces create this or that problem, therefore, build a thing-a-ma-jig to deal with them. The pattern takes its name from the thing-a-ma-jig, the solution. Each pattern in the Portland Form also places itself and the forces that create it within the context of other forces, both stronger and weaker, and the solutions they require. A wise designer
resolves the stronger forces first, then goes on to address weaker ones. Patterns capture this ordering by citing stronger and weaker patterns in opening and closing paragraphs. The total paragraph structure ends up looking like:

- Having done so and so you now face this problem...
- Here is why the problem exists and what forces must be resolved...

Therefore:

- E Make something along the following lines. I'll give you the help I can...
- E Now you are ready to move on to one of the following problems...

Summary Screen
Long pattern languages find groups of patterns working around similar ideas. Portland Form introduces such groups with a summary section. This section explains the general problem under consideration and names the patterns that address it.

(Source: The Portland Pattern Repository, http:// c2.com/ppr/)

Example:

Early Development
These patterns talk about how you begin developing a system. How do you learn what you need to from the client without alienating them? How do you make sure your understanding is consistent with the needs of the program? How do you accept and take advantage of the inevitable changes in your client's thinking? The patterns are:

1. Story
2. Early Program
3. Architecture Prototype
4. Interface Prototype

1. Story

<What patterns are upstream of Story? Must be something about deciding to computerize something.>

How do you start development?

Unless you are already an expert in the field, as a software engineer you need to begin to understand your client's area of expertise before you can begin making decisions. Somehow, you need to learn enough to get going.

Some engineers begin development with a software mind-set. They begin diagramming or even programming as soon as the client begins talking. As you will see in Early Program (2), I believe in beginning to program early in the process, but the client has to drive the process at first. If you begin with the mapping of desires to possibilities too soon, you risk missing the real point of the desires. Often, your client will say subtle things in the first few minutes that can have enormous impact on the development. If you are too busy figuring out whether you have a 1-to-1 or 1-to-n relationship, you will overlook these nuggets.

Other engineers take the opposite tack. They want to gather all the requirements before development begins. I call this "use case paralysis". The notion that there are "requirements" that can be "gathered" has killed more software projects than any other. It's like trying to bake a cake before you've mixed the ingredients. The presence of the system, even the presence of the process of building the system, changes the client's perception of their world. Better to recognize the way the world works, and adapt to it, than to continue wasting time, money, and energy chasing a will-o-the-wisp.

Another factor that enters into the decision of how to get informed about the client's expertise is the natural human need to feel heard. Software development is often disenfranchises the client. Whatever they were doing before is about to change forever. But without the willing participation of your client, you can never be successful. How can you get someone who is about to be screwed to help you with enthusiasm?

Finally, recognize that you will never truly become an expert in your client's field, unless you stop looking at it through the eyes of a software engineer. No matter how well you learn to talk the talk, you still won't walk the walk. Fortunately, it is not important to be an expert to write effective software (already it is probably necessary to be an expert to write great software). You only need to know enough to map your client's desires to the capabilities of current computers. In the beginning, you only need to understand enough to get the evolution of the system rolling.

Therefore, ask your client to tell you a half a dozen stories about how the system will be used. Ask them to think about the moral or point of each story before they begin telling it. Record the stories for later transcription. Don't ask too many questions while you are listening, unless you are hopelessly lost.

You need Early Program (2) to test your understanding and begin the process of evolution.

---

2. Early Program

You have a handful of Stories (1) that describe how the system should feel.

What do you do to begin mapping what the client wants into software?

Some developers begin casting their client's desires as software in the abstract. They use a CASE tool to draw diagrams that represent the client's domain. Then they use a CASE tool to draw diagrams that represent the software. Finally, they allow the program to be cast in executable form.

I think that this abhorrence of running software comes from the early days of large scale software development. It wasn't far into the history of programming that it was obvious that just sitting down and programming wasn't enough. Programmers without the discipline to take a more structured approach were looked down on. "Oh, he's just a hacker."

What is the problem with early code? In the early days of programming styles, languages and environments, it was common for assumptions to permeate the entire

program. These assumptions formed enormous inertia against change. Thus, decisions made early in ignorance were extremely costly if they turned out to be wrong.

What is the problem with abstract representations of the client's desires and the software which realizes them? Simply, it is far too easy to bullshit about the quality of the software, the degree to which it pleases the client, and how far along it is. Thousands of pages of analysis and design diagrams can easily mask the fact that no one know how, or even if, the whole system works.

System development needs some objective measure of the progress. Diagrams have a strong role to play in communicating the structure and intent of the system at all phases. Relying solely on diagrams is begging for trouble.

But what about the problems with early programs? Aren't they written in ignorance of the eventual needs of the system, invoking the inordinate costs pointed about above? Many things have changed since the days when a program was a monolith to be altered only at the sacrifice of all you held dear. Programming languages have developed abstractions like polymorphism and garbage collection which effectively insulate parts of the program from changes in other parts. Programming environments with support for large system development ease the mechanics of making and verifying changes. The final piece of the puzzle of reducing the cost of code changes, the factor which makes early code more than worthwhile, in fact makes it essential, is pattern guided development. Programming style based on patterns ensures that all parts of the program keep an appropriate eye on future evolution, and the nothing will be done to gratuitously hinder changing the code.

The combination of these factors- language, environment, and patterns- creates code which is not expensive to change, even radically. Thus, as new insights come to light, they can be incorporated into the running system quickly and at minimal cost. The advantages of concrete, running software are overwhelming if there is no great price to pay.

Therefore, build concrete software that shows how the system executes the Stories.

If you are working in a group, you may want to build simulated software first with CRC (?). A technically literate client can understand an Architecture Prototype (3). A client who is more visually oriented might profit from an Interface Prototype (4).

---

3. Architecture Prototype

You need to write an Early Program (2).

What kind of program should you write early in development?

Early in a project the client and the developers are both looking at a distant object through fuzzy telescopes. The real question is, which telescope do you focus first?

There's lots to recommend prototyping the client's view of the system, the user interface. It is the one point in the system where the client and the developer are looking at the same thing. For clients and developers who aren't communicating well, prototyping the user interface can give them enough common ground to stand them through the rest of the project.

The major argument against user interface prototyping is that it prematurely raises the client's expecation of what is possible. As soon as they see something that looks like the system, they will assume the rest is done. Clients who aren't familiar with software development can get quite belligerent when told that even though the pictures look finished, the software is only 5% done.

The other important aspect of the system to decide on early is the architecture- the major components, their distribution of responsibility and flows of control. The architecture is the common ground for the whole development staff. Until the team shares a vision of the architecture, little real progress is made. Actually, code that is written in the absence of an architecture may slow further development.

Therefore, write a tiny system which communicates the important shared responsibilities to developers. Give each object no more than a handful of methods. Optimize readability over flexibility.

Begin the system with a User's Object (?). Document the prototype with a Literate Program (?).

---

4. Interface Prototype

You need to write an Early Program (2).

What code do you write early in development?

Some developers are so focused on how they are going to make the system work that they are unable to take a step back and understand the system from the client's point of view. They tuck what little understanding they have of the client's domain under their arm and run with it. Systems developed in this environment are often technically beautiful but useless, providing no payback for the development investment. The developers need to "walk a mile in the client's shoes".

Clients often have the opposite problem- they are so focused on the grand and glorious future that they are unable to focus on what to do first. Symptoms of this disease are system concepts that never get more specific than a handful of bullet items or architecture diagrams drawn by non-technical clients. Systems with unfocused clients go through wild thrashing early in their lives. The client needs to set priorities.

Therefore, write the user visible portion of the system required to support the most important Story (1).

You will need to build a User Interface (?).

The Coplien Form

The Coplien form also reflects the basic elements found in the Alexanderian form. It delineates pattern sections with section headings and includes:

- **The pattern name**: The Coplien form commonly uses nouns for pattern names, but short verb phrases can also be used. This follows from the Alexanderian form.
- **The problem**: The problem is often stated as a question or design challenge. This is analogous to the Alexanderian section that follows the first three diamonds.
- **The context**: A description of the context in which the problem might arise, and to which the solution applies. This is like Alexander's introductory paragraph that sets context.
- **The forces**: The forces describe pattern design trade-offs; what pulls the problem in different directions, toward different solutions? This is like Alexander's in-depth description of the problem, the longest part of the pattern.
- **The solution**: The solution explains how to solve the problem, just as in the emboldened section of an Alexanderian pattern. A sketch may accompany the solution—analogous to the second sketch of Alexander's patterns.
- **A rationale**: Why does this pattern work? What is the history behind the pattern? We extract this so it doesn't "clutter" the solution. As a section, it draws attention to the importance of principles behind a pattern; it is a source of learning, rather than action.
- **Resulting context:** This tells which forces the pattern resolves and which forces remain unresolved by the pattern, and it points to more patterns that might be the next ones to consider. This is like the Alexanderian section following the second set of three diamonds.

Example:

“the Log book”
URL: http://c2.com/cgi/wiki?LogBook

Type: SelfImprovementPatterns
Definition: A notebook (NonVirtualHardCopy?) where you log and probably describe and explain your activities while performing it. Not an agenda.
Problem:

- You want to stop forgetting about how you did things in the past, or what you were thinking at a certain moment.
- You want to organize your thinking while working.
- You want to keep legal records of your actions.
- You want to record your activities.

Context: A log book keeps track of knowledge acquired over time. It can be a record of data, thoughts, or activities. It answers the question "What did you know and when did you know it?" It documents the rationale for the actions you take.
Forces:

- Human LongTermMemory is unreliable.
- SecondaryStorage? is subject to crashes and malfunction.
- You can not be sure that you will be able to read disks and tapes after 10 years, for technological changes, and 6 months, due to media problems.
- Paper is the media with longer duration.
- Writing enforces LongTermMemory.
- You may be legally bound to keep a log.
- Writing helps thinking.

Solution: Keep a notebook besides you. Log everything you do, including the rationale you followed. This is called a LogBook and is current practice in many activities, like ExperimentalPhysics?.
Observation And required in many others (e.g. my better half - a biotechnology research scientist - is required to keep them). When they're full, they get stored in a controlled atmosphere and environment in a cave somewhere in southern England. Log books might provide evidence for later patent cases. *So, she should keep two, a second one for her.*
Resulting Context:
Through your work life you will have the LogBook, actually many LogBooks?, to access.
The act of writing itself enforces LongTermMemory.
The words you write may someday be read back to you in Court.
Design Rationale: Some thoughts:

- Start slowly. Any log is better that no log.
- Keep it where you can see.
- Keep it open.
- Keep it where you can use it.
- Must be easy to use.
- Must be easy to carry with you.

- Must be personal.
- Must be safe from crashes.
- Date pages.
- Number pages.
- Cross reference when you have the option.
- Use a consistent format, if you manage.
- Keep it readable.
- Leave spaces after entries, if you are not using it for legalpurposes.
- Use icons, drawings, colors.
- Don't spend time being beautiful, spend time being informative.
- It's more important to keep the log than to follow all of these practices.

Related Patterns:
ProgrammersNotebook
PhoneLog?
LaboratoryNotebook?
ProjectLogBook?
PersonalLogBook?
FieldNotebook?

Canonical Form

> "The section headings of the paragraphs which immediately follow, make up what is called "**canonical form**" (sometimes this too is called "Alexandrian form") and is the format used by *[POSA]*, AGCS, and many others (often with slight adaptations)."

Name
It must have a meaningful name. This allows us to use a single word or short phrase to refer to the pattern, and the knowledge and structure it describes. It would be very unwieldy to have to describe or even summarize the pattern every time we used it in a discussion. Good pattern names form a vocabulary for discussing conceptual abstractions. Sometimes a pattern may have more than one commonly used or recognizable name in the literature. In this case it is common practice to document these nicknames or synonyms under the heading of **Aliases** or **Also Known As**. Some pattern forms also provide a **classification** of the pattern in addition to its name.

Problem
A statement of the problem which describes its **intent**: the goals and objectives it wants to reach within the given context and forces. Often the forces oppose these objectives as well as each other (one might think of this as a "wicked problem" reminiscent of DeGrace and Stahl, in their book **Wicked Problems, Righteous Solutions**).

Context
The *preconditions* under which the problem and its solution seem to recur, and for which the solution is desirable. This tells us the pattern's **applicability**. It can be thought of as the initial configuration of the system before the pattern is applied to it.

Forces
A description of the relevant *forces* and constraints and how they interact/conflict with one another and with goals we wish to achieve (perhaps with some indication of their priorities). A concrete scenario which serves as the **motivation** for the pattern is frequently employed (see also **Examples**). Forces reveal the intricacies of a problem and define the kinds of *trade-offs* that must be considered in the presence of the tension or dissonance they create. A good pattern description should fully encapsulate all the forces which have an impact upon it. A list of prospective pattern forces for software may be found in the answer to question 11 of *Doug Lea's Patterns-Discussion FAQ*.

Solution
Static relationships and dynamic rules describing how to realize the desired outcome. This is often equivalent to giving instructions which describe how to construct the necessary work products. The description may encompass pictures, diagrams and prose which identify the pattern's **structure**, its **participants**, and their **collaborations**, to show how the problem is solved. The solution should describe not only *static structure* but also *dynamic behavior*. The static structure tells us the form and organization of the pattern, but often it is the behavioral **dynamics** that make the pattern "come alive". The description of the pattern's solution may indicate guidelines to keep in mind (as well as pitfalls to avoid) when attempting a concrete **implementation** of the solution. Sometimes possible **variants** or specializations of the solution are also described.

Examples
One or more sample applications of the pattern which illustrate: a specific initial context; how the pattern is applied to, and transforms, that context; and the resulting context left in its wake. Examples help the reader understand the pattern's use and applicability. Visual examples and analogies can often be especially illuminating. An example may be supplemented by a *sample implementation* to show one way the solution might be realized. Easy-to-comprehend examples from known systems are usually preferred (see also **Known Uses**).

Resulting Context
The state or configuration of the system after the pattern has been applied, including the **consequences** (both good and bad) of applying the pattern, and other problems and patterns that may arise from the new context. It describes the *postconditions* and *side-effects* of the pattern. This is sometimes called **resolution of forces** because it describes which forces have been resolved, which ones remain unresolved, and which patterns may now be applicable (see the answer to question 12 of *Doug Lea's Patterns-Discussion FAQ* for an excellent discussion of *resolution of forces*). Documenting the resulting context produced by one pattern helps you correlate it with the initial context of other patterns (a single pattern is often just one step towards accomplishing some larger task or project).

Rationale

A justifying explanation of steps or rules in the pattern, and also of the pattern as a whole in terms of how and why it resolves its forces in a particular way to be in alignment with desired goals, principles, and philosophies. It explains how the forces and constraints are orchestrated in concert to achieve a resonant harmony. This tells us how the pattern actually works, why it works, and why it is "good". The solution component of a pattern may describe the outwardly visible structure and behavior of the pattern, but the rationale is what provides insight into the *deep structures* and *key mechanisms* that are going on beneath the surface of the system.

Related Patterns

The static and dynamic relationships between this pattern and others within the same pattern language or system. Related patterns often share common forces. They also frequently have an initial or resulting context that is compatible with the resulting or initial context of another pattern. Such patterns might be predecessor patterns whose application leads to this pattern; successor patterns whose application follows from this pattern; alternative patterns that describe a different solution to the same problem but under different forces and constraints; and codependent patterns that may (or must) be applied simultaneously with this pattern.

Known Uses

Describes known occurrences of the pattern and its application within existing systems. This helps validate a pattern by verifying that it is indeed a *proven solution* to a *recurring problem*. Known uses of the pattern can often serve as instructional examples (see also **Examples**).