

DESIGN PATTERNS & ANTIPATTERNS

"A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context."

— DesignPatternsBook (GoF)

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution."

—ChristopherAlexander

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

—ChristopherAlexander

"As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves. As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant."

— Ibid.

Canonical Pattern Language Form:

Name
Alias (optional)
Problem
Context
Forces
Solution
Example (optional)
Resulting Context
Rationale (optional)
Known Uses
Related Patterns

"Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves."

— Gabriel, "A Timeless Way of Hacking."

HISTORY

Christopher Alexander – architect; spatial view; takes into account configuration/relations of problems to contexts

· *A Pattern Language* [1977] and *A Timeless Way of Building* [1979]

· patternlanguage.com - urban planning

OOPSLA - Object-Oriented Programming, Systems, Languages, and Applications

Design Patterns: Elements of Reusable Object-Oriented Software [1995], Gang of Four

· ErichGamma

· JohnVlissides

· RichardHelm

· RalphJohnson

23 Patterns in the Book

· Creational

Abstract Factory, Builder, Factory Method, Prototype, Singleton

· Structural

Adaptor, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

· Behavioural

Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor

Architectural Software Engineering Patterns:

Basic structural Design

Model-View-Controller (MVC):

- X-Code and Cocoa, Java Swing API, Qt Toolkit
- separates data (model) from interface (view) and event-handler (controller)

Presentation-Abstraction-Control

- MVC in tree form

Client-Server

- client sends request to server, which processes and sends replies

Three-Tier

- 1) Presentation
- 2) Application
- 3) Data

- client never communicates directly with Data Tier
- front end, back end, middleware

Services-Oriented Architecture

- SOA, linking resources, applications, data, esp. of a distributed nature
- shows how resources interconnect without specifying the implementation details

Pipeline

- the general pattern behind the shell implementation
- "pipes and filters"

Implicit Invocation

- components are not explicitly called, but rather broadcast a state to which other components show "interest" in, then it's called
- the Hollywood Principle ("don't call us, we'll call you")
- see, Observer Pattern

Fundamental Patterns:

Delegation Pattern:

- an Object advertises some functionality (a method) but in implementation delegates that responsibility to another object
- composition and "has a"

Functional Design Pattern:

- demonstrated by Functional Programming Languages: ML, Haskell
- no side-effects and minimal coupling

Interface Pattern:

- Delegation, composite, and bridge patterns can also be interface patterns

Proxy Pattern:

- proxy: something that represents something else
- create proxies for an object, which you need multiple copies of but is too large to actually duplicate
- remote proxy, access protection proxy, copy-on-write proxy, synchronization proxy

Immutable Object Pattern

Marker Interface Pattern:

- marker interface allows the user to see an underlying property of the class that cannot be learned from looking at the methods
- design markers - a marker interface used to document design choices
 - often called "Explicit Programming"; JavaDoc is a good example of this

Creational Patterns:

controlling object creation

Abstract Factory Pattern:

- grouping/encapsulating factory methods with commonalities
- concrete factories inherit from the abstract factory, but through clever interfaces the client doesn't need to worry about what concrete class is being created;

Factory Method Pattern:

- OO Design pattern for defining methods that create objects
- overloaded constructors with descriptive names

Builder Pattern:

- very similar to Factory Pattern
 - factory is concerned with what is made, builder with how
 - factory with families of objects, build with procedural view of how to make it
- Lazy Initialization Pattern:
- delay the calculation, instantiation, etc, until the it is needed
 - cf. lazy evaluation
- Prototype Pattern:
- used when instantiating a class (with 'new') is too expensive
 - an abstract class is used as a marker with will clone other classes as needed
- Singleton Pattern:
- only one instantiation
 - in Java: use a private static instance
 - in C, you need to use mutex to provide a thread-safe solution

Concurrency Patterns

Active Object Pattern
 Double Checked Locking Pattern
 Guarded Suspension Pattern
 Read/Write Lock Pattern
 Scheduler Pattern
 Thread Pool Pattern

Structural Patterns:

identifying relationships between components

- Adapter Pattern:
- "wrapper" pattern that adapts an interface to an object so that it can be used by something which expects a different object
 - object adapters will contain instances of the class is wraps
 - class adapters will use inheritance to achieve the same goal
 (object adapter is more often used because true multiple inheritance is rare)
- Aggregate Pattern:
- bundling information so that it doesn't need to be repeated in the program
 - ex: going through the same information multiple times with separate iteration loops
- Bridge Pattern:
- "decouple an abstraction from its implementation so that the two can vary independently" - GoF
- Composite Pattern:
- "has-a"; composition
 - allows one to use a group of objects as a single instance
- Container Pattern:
- Objects to hold other objects: confer Proxy, Interface, and Delegation Patterns
 - The Standard Library for example: vectors, queues, stacks, etc.
- Decorator Pattern:
- another "wrapper" pattern; the wrapper in this case adds functionality onto the original object
- Extensibility Pattern:
- mechanisms for adding functionality to components later
- Façade Pattern:
- an object that is a simplified interface to more complex code
 - complex object -> façade -> simple object
- Flyweight Pattern:
- describes how to store/manipulate large numbers of complex objects by sharing common states
 - example: word-processing; 1000 characters in Arial 8 point
 - instead of storing each character and its relevant formatting data in an array, the character is stored along with a pointer to a another object that contains the formatting data
- Proxy Pattern: vide supra
 Pipeline Pattern: vide supra
 Private Class Data Pattern: encapsulation

Behavioural Patterns:

describes communication between objects

Command Pattern:

- objects represent actions
- in our Activity Diagrams: `WashClothes` would be a class, with attributes of `ClothesToBeWashed`, `WashCycle`, etc., and a method to begin the wash
- good for designing transactional behaviour, wizards/expert systems, networking, etc.

Chain of Responsibility Pattern:

- Command Pattern with processing objects
- the processing objects contain the logic to determine the next command object in the chain

Interpreter Pattern:

- a domain-specific language to solve a specialized class of problems
- say to implement RPN... or SQL... or communication protocols

Iterator Pattern:

- iterators allow for sequential traversal through the elements of a collection
- this is considered a pattern because it still hides the underlying representation of the aggregate objects

Mediator Pattern:

- makes the program easier to maintain by unifying the interfaces

Memento Pattern:

- provides "undo" functionality by using an "originator" and a "caretaker" class

Observer Pattern:

- used to observe the state of an object; often used with Implicit Invocation
- useful with event-driven programming

State Pattern:

- used to represent the state of an object

Strategy Pattern:

- determines the algorithm to use at run-time
- accomplished with either polymorphism or reflection (see Ruby, SmallTalk)
- a strategy interface communicates with concrete strategies

Template Method Pattern:

- similar to Strategy Pattern; algorithms are represented by abstract classes whose methods are overridden with concrete ones

Visitor Pattern:

Single-Serving Visitor Pattern:

Hierarchical Visitor Pattern:

Criticisms:

- Design Patterns are evidence of poor abstraction abilities in the language
 - a pattern is duplication, and duplication should be factored out
 - declarations of behaviour should appear one time (*once and only once*);
 - cf. with normalizing in a relational model
 - in other words, Design Patterns should be built-in features of a powerful language
 - Lisp is the common example here
 - many Design Patterns are workarounds to limitations of particular language
 - Builder Pattern for C++
 - Anonymous Subroutine Objects Pattern for Perl

"The obvious way to achieve what the BuilderPattern achieves is to have a base-class constructor which calls a bunch of virtual methods, which can be overridden by the sub-classes (or provided for the first time, if they were abstract on the base class). This doesn't work in C++ since function calls made within the construction phase of the base class are not polymorphic."

- no formal foundations
- difficulty in rewriting Design Patterns
 - they must be refactored for a specific problem

AntiPatterns

common bad solutions to common problems

a pattern that tells how to go from a problem to a bad solution

'Bad Ideas'

bad solutions can be attractive

Amelioration Pattern – a pattern that tells how to go from a bad solution to a good solution.

codes smells

- large method/class
- shotgun surgery - large codes changes to small system changes
- feature envy
- lazy class
- inappropriate intimacy

Listen to the Code

Smoke and Mirrors

- making a program appear to have functionality which hasn't been implemented yet

Ball of Mud

- a program or system with little or no architecture
- "held together with glue and rubber bands"
- also called "Spaghetti Code"

Database as an IPC

Input Kludge

- programs fails to handle invalid user input

Magic PushButton

- a result of graphical IDEs (Visual Studio, Borland Delphi, etc)
- the programmer designs the GUI first, and then fills in the code for what each button does

God Object

- an object that knows or does far too much

Cargo Cult Programming

- you poor ignorant savage

Race Hazard

- see threads and concurrency

Singletonitis

Busy spin

- using CPU resources while waiting for an event

Lava Flow

- keeping bad code because it's too expensive to do otherwise

Magic Numbers

De-Factoring

- Doc-Factoring

Copy-and-Paste Programming

- "Someone Else's Example" Pattern

Golden Hammer

Silver Bullet

- No Silver Bullet for accidental complexity