Best Practices in Software Development: Introduction to Design Patterns

Ansgar Fehnker
Based on slides by: Arend Rensink, Christoph Bockisch, Mehmet Aksit
Patterns

- Patterns are used to represent a piece of knowledge to solve a particular problem
  - Concept introduced by Christopher Alexander, *The Timeless Way of Building*, 1979, Oxford University Press

A 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.

---

Christopher Alexander

*(the last bit is more specific to buildings!)*
An architectural pattern expresses a fundamental structural organization schema for software systems.
A design pattern provides a scheme for refining the architecture.

Expressed in a pattern format:
- Elements in the technology
- Relations among them
- Classes
- Inheritance
- Messages

solves a problem
Pattern discovery

This is a nice solution

This is the pattern

This is the definition

solving a problem
generalizing the solution into a pattern
recording the pattern
Pattern application

This is my problem

This is the pattern

This is the implementation

facing a problem and searching for a pattern

understanding the pattern

implementing the pattern
A pattern ..

- abstracts a recurring design structure
- comprises class and/or object, with dependencies, structures, interactions, and conventions
- names & specifies the design structure explicitly
- distills design experience
- should be language independent
- should be orthogonal to software development methodology or framework
Gang of Four

Goals

Codify good design
- distill & generalize experience
- aid to novices & experts alike

Give design structures explicit names
- common vocabulary
- reduced complexity
- greater expressiveness

Capture & preserve design information
- articulate design decisions succinctly
- improve documentation

Facilitate restructuring/refactoring
- patterns are interrelated
- additional flexibility.
Gang of Four

Categories

- Creational
  - Create objects for you, rather than having you instantiate objects
- Structural
  - These concern class and object composition
- Behavioral
  - Specifically concerned with communication between objects
Gang of Four

A pattern consists of

1. Name
2. Problem
3. Solution
4. Consequences & trade-offs of application
Gang of Four Template

Name
  succinct and expressive name

Intent
  short description of the pattern & its purpose

Also Known As
  Any aliases this pattern is known by

Motivation
  motivating scenario demonstrating pattern’s use

Applicability
  circumstances in which pattern applies

Structure
  graphical representation of the pattern using modified UML notation

Participants
  participating classes and/or objects & their responsibilities
Gang of Four Template (Continued)

Collaborations
how participants cooperate to carry out their responsibilities

Consequences
the results of application, benefits, liabilities

Implementation
pitfalls, hints, techniques, plus language-dependent issues

Sample Code
sample implementations in C++, Java, C#, Smalltalk, C, etc.

Known Uses
examples drawn from existing systems

Related Patterns
discussion of other patterns that relate to this one
Problem Description

- Many related classes differ only in details of their behavior
  - Different variants of an algorithm are needed
Example: Formatting Documents

- Formatting must break text into lines, lines into columns, and so on, taking into account the user's desires
  - User might want to vary margin widths, indentation, and tabulation; single or double space; and probably many other formatting constraints
  - Formatting algorithm must take all of these into account.

- Variety of formatting algorithms available
  - Different strengths and weaknesses

- Trade-offs
  - balance between formatting quality and formatting speed

- Strategy to choose may depend on context, e.g.:
  - Available resources
  - User preferences
Strategy Pattern: Overview

Client

Context

strategy

Strategy

ConcreteStrategyA
algorithm()

ConcreteStrategyB
algorithm()

ConcreteStrategyC
algorithm()
Solution 1: Using Inheritance

- Complicates Document’s code
  - No policy / mechanism separation
  - Document harder to understand, maintain and extend
  - Not possible to change algorithms at run-time

- Document hierarchy does not reflect an abstraction hierarchy
  - Classes differ only in the implementation of parts of their interfaces
Solution 2: Using Conditionals

- Bloats the implementation of the responsible class by implementing the various algorithm choices as individual methods
  - `formatWithSimpleAlgorithm()`
  - `formatWithTeXAlgorithm()`
- Use conditional statements to invoke the appropriate one.
Solution 3: Using the Strategy Pattern

- Isolate the algorithm and make it easily replaceable at the same time by encapsulating it in an object.

- We’ll define a separate class hierarchy for objects that encapsulate formatting algorithms.
  - The root of the hierarchy - Compositor - will define an interface that supports a wide range of formatting algorithms.
  - Each subclass will implement the interface to carry out a particular algorithm.
Formatting Documents Using the Strategy Pattern

```
DocElement
  Composite
    traverse()
    repair()

  composer.compose()

  composer
    Composite
      compose()

  SimpleCompositor
    compose()

  TeXCompositor
    compose()

  ArrayCompositor
    compose()
```
Demo
Applicability of Strategy pattern

- Possible indicators:
  - Many related classes differ only in their behaviour
  - You need different variants of an algorithm
  - An algorithm uses data that clients shouldn't know about
  - A class defines many behaviours, and these appear as multiple conditional statements in its operations
Advantages

- Hierarchies of **Strategy** classes define a family of algorithms
  - Inheritance can help factor out common functionality of the algorithms
- Strategies offer an alternative to subclassing, which
  - mixes the *algorithm* implementation with *Context*
  - makes *Context* harder to understand, maintain, and extend
  - can't vary the algorithm dynamically
  - leads to many related classes only differing in *algorithm*
  - hinders switching algorithm independent from *Context*
- Strategies eliminate conditional statements
- Strategies offer a choice of behaviours
Disadvantages

- Clients must be aware of different strategies and how they differ
  - Clients might be exposed to implementation issues.
  - Use Strategy only when the variation in behaviour is relevant to clients.

- **Strategy** pattern increases the number of objects
  - Can be reduced by implementing strategies as shared stateless objects
    - Any state is maintained by the context, which passes it in each request to the Strategy object
    - Shared strategies should not maintain state across invocations
  - See Flyweight pattern
Implementation Alternatives

How to provide data required by algorithm?

1. Pass the needed information as a parameter.
   - Pro: Context and Strategy are completely decoupled
   - Con: Communication overhead

2. Let Context pass itself as a parameter / let Strategy have a reference to it’s context.
   - Pro: Reduced communication overhead
   - Con: Context must define a more elaborate interface to its data, which couples Strategy and Context more closely.
UNIVERSITY OF TWENTE.

The State Pattern
Problem Description

- An object’s behaviour depends on its state, and it must change its behavior at run-time.

I am starving, give me something to eat

Sure

No, enough for today

do you want to drink coffee?
State Pattern

Typically the same name.

Client

Context

oper()

state.oper()

State

oper()

ConcreteStateA

oper()

ConcreteStateB

oper()
Collaborations

- Context delegates state-specific requests to current State object

- A Context may pass itself as an argument to the State object
  - Lets the State object access the context if necessary

- Context is the primary interface for clients
  - Clients can configure the context with State objects
  - Once a Context is configured, clients only deal with context

- Either Context, or the concrete State objects can decide which state succeeds another and under which circumstances
State Pattern

Context

oper

changeState

state.oper()

State

oper

Concrete State1

oper

Beste Practices in Software Development
Example

TCPConnection
- open
- close
- acknowledge

TCPState
- open
- close
- acknowledge

TCPEstablished
- open
- close
- acknowledge

TCPListen
- open
- close
- acknowledge

TCPClosed
- open
- close
- acknowledge
Demo
Applicability

- **Indicators**
  - Behavior depends on object's state
  - Must change its behavior at runtime
  - Operations have large multi-part conditional statements
  - That depend on object's state
  - Conditionals test enumerated constants
  - Several operations contain same conditional structure
Consequences

- Localized state-specific behavior
  - Partitioned for different states
  - Behavior associated with a particular state in one object
  - New states and transitions can be added

- Better way to structure state-specific code
  - Logic determining state transitions doesn't reside in monolithic if or switch statements
  - Encapsulating each state transition and action in a class elevates the idea of an execution state to full object status
  - Clear structure of code
  - Intent explicit in code
Consequences

- Explicit state transitions

- Protect the Context from inconsistent internal states
  - State transitions are atomic from the Context's perspective
  - Rebinding one rather than several variables
Consequences

- Behavior for different states distributed across several State subclasses.
- This increases the number of classes and is less compact than a single class.
- It is not explicit in the design that a State object is performing behavior on behalf of a Context object.
UNIVERSITY OF TWENTE.

Summary
Summary

Categories

- Creational
  - Create objects for you, rather than having you instantiate objects
- Structural
  - These concern class and object composition
- Behavioral
  - Specifically concerned with communication between objects
Creational patterns

Create objects for you, rather than having you instantiate objects

- More flexibility in deciding which objects need to be created

- **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.
Structural patterns

These concern class and object composition

- **Adapter**
  - Allows classes with incompatible interfaces to work together

- **Bridge**
  - Decouples an abstraction from its implementation

- **Composite**
  - Composes zero-or-more similar objects

- **Decorator**
  - Dynamically adds/overrides behaviour in an existing method

- **Facade**
  - Provides a simplified interface to a large body of code.

- **Flyweight**
  - Reduces the cost of manipulating a large number of similar objects

- **Proxy**
  - Provides a placeholder for another object
Behavioural patterns (1)

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 without exposing its representation.

- **Mediator**
  - Allows loose coupling between classes

- **Memento**
  - provides the ability to restore an object to its previous state (undo).
Behavioural patterns (2)

- **Observer**
  - is a publish/subscribe pattern

- **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

- **Template method**
  - defines the skeleton of an algorithm as an abstract class

- **Visitor**
  - separates an algorithm from an object structure
Promise

- reusable software
- reuse of design
- documentation
- communication
- teaching
- language design

→ patterns foster reusability
→ rather than code
→ information chunks
→ design vocabulary
→ passing on culture
→ high level languages
Observations

Patterns are applicable in all stages of the OO lifecycle
- analysis, design, & reviews
- realization & documentation
- reuse & refactoring

Patterns permit design at a more abstract level
- treat many class/object interactions as a unit
- often beneficial *after* initial design
- targets for class refactorings

Variation-oriented design
- consider what design aspects are variable
- identify applicable pattern(s)
- vary patterns to evaluate tradeoffs
- repeat
Reservations

Don’t apply them blindly
   Added indirection can yield increased complexity, cost
   Address a certain problem. Alternative might be to avoid the problem.

Resist branding everything a pattern
   Articulate specific benefits
   Demonstrate wide applicability
   Find at least three existing examples from code other than your own!

GoF has a 20 year history
   New useful patterns have been found
   Better implementations for some patterns have been found
   Languages have evolved
Criticism

The Show Trial at OOPSLA 1999 for *Crimes against Computer Science* (http://www.laputan.org/patterns/trial.html)

- *The Accused* have engaged in a usurpation of perfectly good English words and well-known technical terms for the purpose of establishing an arcane argot known only to a narrow circle of GoF initiates.

- *The Accused*, by making it possible to design object-oriented programs in C++, have inhibited the rightful growth of competing object-oriented languages such as Smalltalk, CLOS, and Java.

- *The Accused*, by elevating design from the realm of technical artifacts to a conceptual level, have encouraged a further estrangement of architecture from implementation.

- *The Accused* have promoted a cult of personality, and brought about the establishment of a cottage industry of consultants, trainers, and sundry acolytes to interpret their abstruse musings.
The Accused, by distilling hard-won design expertise into patterns, have encouraged novices to act like experts.

The Accused, by cataloging mere experience, rather than conducting novel research, have displayed an utter disregard for traditional standards of academic originality.

The Accused, by virtue of having used it in their catalog, have imposed a particularly awkward and odious pattern format on the software community, which has proven to be something of a Procrustean Bed.

The Accused, by dint of their influence, have encouraged a perception that patterns are good only for describing and designing object-oriented programs.
Conclusion

If you reuse code,
you'll save a load,
but if you reuse design,
your future will shine

Ralph E. Johnson