Gamma, Helm, Johnson, Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

Solutions for specific problems in object-oriented software design

- Catalog
- Architectural style for software development

Specific description or template to solve problems

- Recurring problems
- Special cases

Relationships and interactions between classes or objects

- Without specifying the final application, classes, objects

---

1Gang of Four
Intent
- Recurring patterns of collaborating objects
- Practical knowledge from practicians (best practices)
- Developer’s vocabulary for communication
- Structuring of code (architectures)

Goals
- Flexibility
- Maintainability
- Communication
- Reuse

Aspects
- Flexibility–Overhead
- Class-based–Object-based patterns
- Inheritance–Delegation

Alternative approaches and combinations possible
- Which (combination of) pattern(s) is best
1. Do program against an interface, not again an implementation
   - Many interfaces and abstract classes beside concrete classes
   - Generic frameworks instead of direct solutions

2. Do prefer object composition instead of class inheritance
   - Delegate tasks to helper objects

3. Decoupling
   - Objects less interdependent
   - Indirection as an instrument
   - Additional helper objects
Inheritance = White-box reuse

- Reuse by inheritance
- Inheritance is static
- Internals of base classes are visible
- Inheritance breaks encapsulation

Composition = Black-box reuse

- Reuse by object composition
- Needs well-formed interfaces for all objects
- Internals of base classes are hidden
Object composition is mighty as inheritance

Usage of delegation (indirection)

- More objects involved
- Explicit object references
- No this-pointers

Dynamic approach, hard to comprehend, maybe inefficient at runtime
A recurring pattern found in all design patterns

- List \( x = \) new ArrayList(); // direct example
- List \( x = \) aListFactory.createList(); // indirect example

Indirection

- Object creation
- Method calls
- Implementation
- Complex algorithms
- Excessive coupling
- Extension of features

Do spend additional objects!
Object creation
- Coupling
  - List \( x = \) new ArrayList();
- Decoupling
  - List \( x = \) aListFactory.createList();

Method calls
- Coupling
  - Hard wiring of method calls
  - No changes without compiling
- Decoupling
  - Objectification of methods
  - Replaceable at runtime

Implementation
- Dependencies on hardware and software platforms
  - Platform-independent systems

Complex algorithms
- Fixedness though hard-wiring
  - Conditional choices by classes instead of if, then, else
- Decouple parts of algorithm that might change in the future
- Excessive coupling
  - Single objects can’t be used isolated

- Decoupling
  - Additional helper objects

- Extension of features (coupling in class hierarchies)
  - Through inheritance
  - Implementing a sub class needs knowledge of base class
  - Isolated overriding of a method not possible
  - Too many sub classes
  - Decoupling by additional objects

- When a class can’t be changed...
  - No source code available
  - Changes have to many effects
Classification of Design Patterns

Albert-Ludwigs-Universität Freiburg

Purpose

**Creational Patterns** deal with object creation
e.g. Singleton, Abstract Factory, Builder

**Structural Patterns** composition of classes or objects
e.g. Facade, Proxy, Decorator, Composite, Flyweight

**Behavioral Patterns** interaction of classes or objects
e.g. Observer, Visitor, Command, Iterator

Scope

**Class** static relationships between classes (inheritance)

**Object** dynamic relationships between objects
Intent
Motivation
Applicability
Structure
Participants
Collaborations
Consequences
Implementation
Sample Code
Known Uses
Related Patterns
Intent

- Class with exactly one object (global variable)
- No further objects are generated
- Class provides access methods

Motivation

- To create factories and builders

```java
Singleton
#
instance()
if (instance == NULL)
    instance = new Singleton();
return instance;
```
Creational Pattern: Singleton

Structure
Albert-Ludwigs-Universität Freiburg

Applicability
- Exactly one object of a class required
- Instance globally accessible

Consequences
- Access control on singleton
- Structured address space (compared to global variables)
Creational Pattern: Abstract Factory

Albert-Ludwigs-Universität Freiburg

Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- User interface toolkit supporting multiple look-and-feel standards
e.g., Motif, Presentation Manager
Motivation
Albert-Ludwigs-Universität Freiburg
Creational Pattern: Abstract Factory

Structure

Albert-Ludwigs-Universität Freiburg

AbstractFactory
createProductA()
createProductB()

ConcreteFactory1
createProductA()
createProductB()

ConcreteFactory2
createProductA()
createProductB()

AbstractProductA

ProductA2
ProductA1

AbstractProductB

ProductB2
ProductB1

Client
Creational Pattern: Abstract Factory

Applicability
Albert-Ludwigs-Universität Freiburg

- Independent of how products are created, composed, and represented
- Configuration with one of multiple families of products
- Related products must be used together
- Reveal only interface, not implementation

Consequences

- Product class names do not appear in code
- Exchange of product families easy
- Requires consistency among products
Intent

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - read RTF and translate in different exchangeable formats
Creational Pattern: Builder

Motivation

Albert-Ludwigs-Universität Freiburg

RTFReader
ParseRTF()

Builder

TextConverter
convertChar(char)
convertCharset(Charset)
convertParagraph()

ASCIIConverter
convertChar(char)
getASCIItext()

TeXConverter
convertChar(char)
convertCharset(Charset)
convertParagraph()
getTeXText()

TextWidgetConverter
convertChar(char)
convertCharset(Charset)
convertParagraph()
getTextWriter()

ASCIIText
TeXText
TextWidget

while (t = getToken()){ switch (t.Type) {
    case CHAR: builder.convertChar(t.Char).
    case CHARSET: builder.convertCharset(t.Charset)
    case PARAGRAPH: builder.convertParagraph()}}
Creational Pattern: Builder

Structure

Albert-Ludwigs-Universität Freiburg

ConcreteBuilder
Builder
construct() constructPart()
constructPart()
for all objects in structure {
    builder.constructPart()}
getResult()

Director
construct()

builder

for all objects in structure {
    builder.constructPart()}

ConcreteBuilder
constructPart()
getResult()

Product
Creational Pattern: Builder

Consequences

Albert-Ludwigs-Universität Freiburg

- Reusable for other directors (e.g. XMLReader)

Difference to Abstract Factory

- Builder assembles a product step-by-step (parameterized over assembly steps)
- Abstract Factory returns complete product
Intent

- Provide a unified interface to a set of interfaces in a subsystem

Motivation

- Compiler subsystem contains Scanner, Parser, Code generator, etc
- Facade combines interfaces and offers new compile() operation
Structural Pattern: Facade

Motivation (2)

Albert-Ludwigs-Universität Freiburg

```
compile()
Compiler
Scanner
Parser
Token
Symbol
ProgramNode
StatementNode
ProgramNodeBuilder
Stream
BytecodeStream
CodeGenerator
RISCCodeGenerator
```
Simple interface to complex subsystem
Many dependencies between clients and subsystem—Facade reduces coupling
Layering
Shields clients from subsystem components

Weak coupling: improves flexibility and maintainability

Often combines operations of subsystem to new operation

With public subsystem classes: access to each interface
Intent

- Use sharing to support large numbers of fine-grained objects efficiently

Motivation

- Document editor represents images, tables, etc by objects
- But not individual characters!
- Reason: high memory consumption
- Objects would provide more flexibility and uniform handling of components
- One Flyweight Object is shared among many “equal” characters
Structural Pattern: Flyweight

Motivation

Albert-Ludwigs-Universität Freiburg
Structural Pattern: Flyweight

Structure
Albert-Ludwigs-Universität Freiburg

```java
FlyweightFactory flyweightFactory;

if (flyweightFactory.containsKey(key)) {
    return flyweightFactory.get(key);
} else {
    // Create new flyweight
    UnsharedConcreteFlyweight unsharedConcreteFlyweight;
    ConcreteFlyweight concreteFlyweight;

    // Initialize intrinsic state
    intrinsicState

    // Initialize extrinsic state
    extrinsicState

    // Operation on extrinsic state
    operation(extrinsicState)

    // Add to pool of flyweights
    flyweightFactory.add(key, unsharedConcreteFlyweight);
    return unsharedConcreteFlyweight;
}
```
Structural Pattern: Flyweight

Applicability

Many similar objects
Memory consumption too high for "full objects"
State decomposable in intrinsic and extrinsic state
Identity of objects does not matter

Consequences

Decreased memory consumption
Potentially increased time due to passing of extrinsic state
Behavioral Pattern: Observer
Albert-Ludwigs-Universität Freiburg

Intent

- Define 1: \( n \)-dependency between objects
- State-change of one object notifies all dependent objects
Maintain consistency between internal model and external views
Behavioral Pattern: Observer

Structure

Albert-Ludwigs-Universität Freiburg

```java
return subjectState;
getState()
setState()
for all o in observers {
    state = subject.getState();
o.update();
state
}
```

**Subject**
- attach(Observer)
- detach(Observer)
- notify()

**ConcreteSubject**
- state
- getState()
- setState()

**Observer**
- update()

**ConcreteObserver**
- update()
- state

```
for all o in observers {
    o.update();
}
```

Matthias Keil
Design Patterns
6. Mai 2013 32 / 42
Behavioral Pattern: Observer

Applicability

Albert-Ludwigs-Universität Freiburg

- Objects with at least two mutually dependent aspects
- Propagation of changes
- Anonymous notification

Consequences

- Subject and Observer are independent (abstract coupling)
- Broadcast communication
- Observers dynamically configurable
- Simple changes in Subject may become costly
- Granularity of update()
Behavioral Pattern: Visitor
Albert-Ludwigs-Universität Freiburg

**Intent**
- Represents operations on an object structure by objects
- New operations without changing the classes
Behavioral Pattern: Visitor

Motivation

Albert-Ludwigs-Universität Freiburg

- Processing of a syntax tree in a compiler: type checking, code generation, pretty printing, ...
- Naive approach: put operations into node classes → hampers understanding and maintainability
- Here: realize each processing step by a visitor
Behavioral Pattern: Visitor
Syntax Tree with Visitors
Albert-Ludwigs-Universität Freiburg

NodeVisitor
- VisitAssignment(AssignmentNode)
- VisitVariableRef(VariableRefNode)

TypeCheckingVisitor
- VisitAssignment(AssignmentNode)
- VisitVariableRef(VariableRefNode)

CodeGeneratingVisitor
- VisitAssignment(AssignmentNode)
- VisitVariableRef(VariableRefNode)

Program

Node
- accept(NodeVisitor v)

VariableRefNode
- accept(NodeVisitor v)
  v.visitVariableRef(this);

AssignmentNode
- accept(NodeVisitor v)
  b.visitAssignment(this);
Behavioral Pattern: Visitor

Structure

Albert-Ludwigs-Universität Freiburg

Client

Visitor

visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)

ConcreteVisitor1

visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)

ConcreteVisitor2

visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)

ObjectStructure

Element

accept(Visitor)

ConcreteElementA

accept(Visitor v)
operationA()

ConcreteElementB

accept(Visitor b)
operationB()

v.visitConcreteElementA(this);

v.visitConcreteElementB(this);
Behavioral Pattern: Visitor

Applicability

Albert-Ludwigs-Universität Freiburg

- Object structure with many differing interfaces; processing depends on concrete class
- Distinct and unrelated operations on object structure
- Not suitable for evolving object structures

Consequences

- Adding new operations easy
- Visitor gathers related operations
- Adding new ConcreteElement classes is hard
- Visitors with state
- Partial breach of encapsulation
Intent

- Sequential access to components of a container object
- Representation of object hidden
Behavioral Pattern: Iterator (Cursor)

Motivation
Albert-Ludwigs-Universität Freiburg

Client

AbstractList
createIterator()
count()
insert(Item)
remove(Item)
...

Iterator
first()
next()
isDone()
currentItem()

Tree

List

List–Iterator

Tree–Iterator

Matthias Keil
Design Patterns
6. Mai 2013

ConcreteIterator administers current object and determines subsequent object(s)
Behavioral Pattern: Iterator (Cursor)

Applicability
Albert-Ludwigs-Universität Freiburg

- Access objects “contents” without exposing representation
- Support multiple traversals
- Uniform interface for traversing different containers

Consequences

- Easy switching between different styles of traversal
- Simplifies Aggregate’s interface
- More than one pending traversal
- Control of iteration (internal vs. external)
- Traversal algorithm (Iterator vs. Aggregate)
- Robustness (are modifications visible?)