Design Patterns (1)
Albert-Ludwigs-Universität Freiburg

- solutions for specific problems in object-oriented software design
- 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

- Gamma, Helm, Johnson, Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.¹

¹Gang of Four
- recurring patterns of collaborating objects
- practical knowledge from practicians (best practices)
- developer’s vocabulary for communication
- structuring of code (microarchitectures)
- goals: flexibility, maintainability, communication, reuse
- each pattern emphasizes certain aspects
  - flexibility vs. overhead
- alternative approaches and combinations possible
- task: which (combination of) pattern(s) is best
- class-based ↔ object-based patterns
- inheritance ↔ delegation
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
**Object composition**

Albert-Ludwigs-Universität Freiburg

---

**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)

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

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

Do spend additional objects!
Indirection
Object creation
Albert-Ludwigs-Universität Freiburg

Coupling
- List x = new ArrayList();
- Implementation class is hard-wired
- Usage of implementation class instead interface
- Replacement of implementation class is hard

Decoupling
- List x = aListFactory.createList();
- Creates an object indirectly

Patterns: Abstract Factory, Factory Method, Prototype
Indirection
Method calls
Albert-Ludwigs-Universität Freiburg

- Coupling
  - Hard wiring of method calls
  - No changes without compiling

- Decoupling
  - Objectification of methods
  - Replaceable at runtime

- Patterns: Chain of Responsibility, Command
Dependencies on hardware and software platforms
- External OS-API’s may vary
- Platform-independent systems as possible
- Patterns: Abstract Factory, Bridge

Dependencies on object representation or implementation
- Clients know, how and where an object is represented, stored, implemented, etc.
- Clients must be changed, even if the interfaces don’t change
- Patterns: Abstract factory, Bridge, Memento, Proxy
Fixedness though hard-wiring

- Catching all cases of an algorithm
  - Many conditional choices (if, then, else)
  - Conditional choices by classes instead of if, then, else
- Changes, extensions, optimizations bring further conditional choices
- Decouple parts of algorithm that might change in the future

Flexibilization by decoupling additional algorithm objects

Patterns: Builder, Iterator, Strategy, Template Method, Visitor
Indirection
Excessive coupling
Albert-Ludwigs-Universität Freiburg

- Too close coupled objects
  - Leads to monolithic systems
  - Single objects can’t be used isolated

- Decoupling
  - Additional helper objects

- Patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer
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
- Patterns: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy

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

Albert-Ludwigs-Universität Freiburg

Purpose

**Creational Patterns** deal with object creation
Singleton, Abstract Factory, Builder, (and Factory Method, Prototype)

**Structural Patterns** composition of classes or objects
Facade, Proxy, Decorator (and Adapter, Bridge, Flyweight, Composite)

**Behavioral Patterns** interaction of classes or objects
Observer, Visitor, (and Command, Iterator, Memento, State, Strategy)

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

```cpp
class Singleton
{
    private:
        static Singleton* instance;

    public:
        static Singleton* instance()
        {
            if (instance == NULL)
                instance = new Singleton();
            return instance;
        }

    private:
};
```
Applicability

- exactly one object of a class required
- instance globally accessible

Consequences

- access control on singleton
- structured address space (compared to global variables)
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
Creational Pattern: Abstract Factory

Structure
Albert-Ludwigs-Universität Freiburg

AbstractFactory
createProductA()
createProductB()

Client

AbstractProductA

ConcreteFactory1
createProductA()
createProductB()

ConcreteFactory2
createProductA()
createProductB()

AbstractProductB

ProductA2
ProductA1

ProductB2
ProductB1
Creational Pattern: Abstract Factory

Applicability

- 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

ConcreteBuilder

Director

Builder

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

Product

ConcreteBuilder

getResult()
Creational Pattern: Builder

Interaction Diagram for Builder

Albert-Ludwigs-Universität Freiburg

aClient

new ConcreteBuilder()

new Director(aBuilder)

construct()

aDirector

constructPartA()

constructPartB()

constructPartC()

getResult()

aConcreteBuilder

Matthias Keil

Softwaretechnik

14. Juni 2012
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

Matthias Keil
Softwaretechnik
14. Juni 2012
- simple interface to complex subsystem
- many dependencies between clients and subsystem → Facade reduces coupling
- layering

**Structure**

![Facade Structure Diagram]

- **client classes**
- **Facade**
- **subsystem classes**
- 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

- control access to object

### Motivation

- multi-media editor loads images, audio clips, videos etc on demand
- represented by proxy in document
- proxy loads the “real object” on demand
Structural Pattern: Proxy

Motivation
Albert-Ludwigs-Universität Freiburg

if (image == NULL)
    image = loadImage(fileName);
    image.draw();

if (image == NULL)
    return extent;
else return image.getExtent();
Structural Pattern: Proxy

Structure

Albert-Ludwigs-Universität Freiburg
1. remote proxy communication with object on server (CORBA)
2. virtual proxy
   - creates expensive objects on demand
   - delays cost of creation and initialization
3. protection proxy controls access permission to original object
4. smart reference additional operations: reference counting, locking, copy-on-write
Intent

- extend object’s functionality dynamically
- more flexible than inheritance
- graphical object can be equipped with border and/or scrollbar
- decorator object has same interface as the decorated object
- decorated forwards requests
- recursive decoration

```
:BorderDecorator
  component
  
  :ScrollDecorator
  component
  
  :TextView
```
Structural Pattern: Decorator

Motivation (2)

Albert-Ludwigs-Universität Freiburg

VisualComponent
draw()

Decorator
draw()

component
drawBorder();

TextView
draw()

ScrollDecorator
scrollPosition
draw();
scrollTo();

BorderDecorator
borderWidth
draw()
drawBorder();
Structural Pattern: Decorator

Applicability

- dynamically add responsibilities to individual objects
- for withdrawable responsibilities
- when extension by inheritance is impractical
more flexible than inheritance

- avoids feature-laden classes high up in the hierarchy
- decorator ≠ component
- lots of little objects → hard to learn and debug
Behavioral Pattern: Observer

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 (1)
Albert-Ludwigs-Universität Freiburg

```java
Subject
attach(Observer)
detach(Observer)
notify()

ConcreteSubject
state
getState() setEstado()

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

Observer
update()

ConcreteObserver
update()
state

return subjectState;

state = subject.getState();
```
Behavioral Pattern: Observer
Structure (2)

Albert-Ludwigs-Universität Freiburg

A Concrete Subject

aConcrete Subject

setState()

notify()

update()

getState()

update()

A Concrete Observer

another ConcreteObserver

getState()
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()
Intent

- represents operations on an object structure by objects
- new operations without changing the classes
- 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

without visitor

![Diagram](image_url)
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)

AssignmentNode
accept(NodeVisitor v)

v.visitVariableRef(this);

b.visitAssignment(this);
Pattern: Visitor
Structure
Albert-Ludwigs-Universität Freiburg

Visitor
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)

ConcreteVisitor1
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)

ConcreteVisitor2
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)

ObjectStructure
accept(Visitor)

ConcreteElementA
accept(Visitor v)
operationA()

ConcreteElementB
accept(Visitor b)
operationB()

v.visitConcreteElementA(this);
v.visitConcreteElementB(this);
Pattern: Visitor
Interaction Diagram
Albert-Ludwigs-Universität Freiburg

anObject Structure

accept (aVisitor)

aConcrete ElementA

visitConcreteElementA (aConcreteElementA)

operationA()

aConcrete ElementB

visitConcreteElementB (aConcreteElementB)

operationB()

aConcrete Visitor

Matthias Keil
Softwaretechnik
14. Juni 2012 49 / 50
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