Design Patterns

Software Engineering

Sergio Feo-Arenis
slides by:
Matthias Keil

Institute for Computer Science
Faculty of Engineering
University of Freiburg

30.06.2014

Design Patterns

Literature

- Gamma, Helm, Johnson, Vlissides: *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- McConell, Steve: *Code Complete*. Microsoft Press, 2004

- 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

Fundamentals of Design Patterns

- Intent
  - Recurring patterns of collaborating objects
  - Practical knowledge from practicians (best practices)
  - Developer's vocabulary for communication
  - Structuring of code (architectures)
- Goals
  - Flexibility
  - Maintainability
  - Code reuse
  - Improved communication between developers
- Aspects
  - Tradeoff between Flexibility–Overhead
  - There are class-based–object-based patterns
  - Inheritance vs. Delegation
- Alternative approaches and combinations possible
  - Which (combination of) pattern(s) is best

Principles of Design Patterns

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

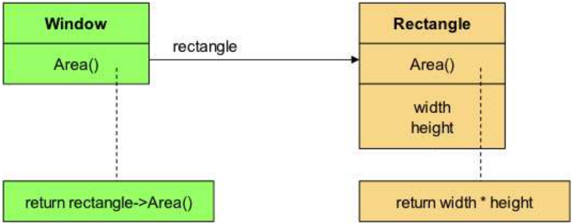
Composition = Black-box reuse

- Reuse by object composition
- Needs well-formed interfaces for all objects
- Internals of base classes are hidden

Principles of Design Patterns

Delegation

- Object composition is as powerful 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

Principles of Design Patterns

Indirection

- 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!

Principles of Design Patterns

Indirection (cont)

- 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 through hard-wiring
    - Conditional choices by classes instead of if, then, else
  - Decouple parts of algorithm that might change in the future

Principles of Design Patterns

Indirection (cont)

- Excessive coupling
  - Single objects can't be used isolated
- Decoupling
  - Additional helper objects
- Extension of features (coupling in class hierarchies)
  - Through inheritance
  - Implementing a subclass needs knowledge of base class
  - Isolated overriding of a method not possible
  - Too many subclasses
  - Decoupling by additional objects
- When a class can't be changed...
  - No source code available
  - Changes have too many effects

Classification of Design Patterns

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

Standard Template

- Intent
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- (Sample Code)

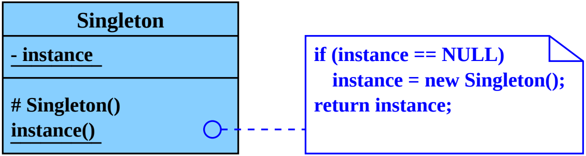
Creational Pattern: Singleton

Intent

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

Motivation

- To create factories and builders



```
if (instance == NULL)
   instance = new Singleton();
return instance;
```

Creational Pattern: Singleton

Structure

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: Singleton

Code

```java
public class Singleton {
    private static final Singleton INSTANCE =
                                  new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

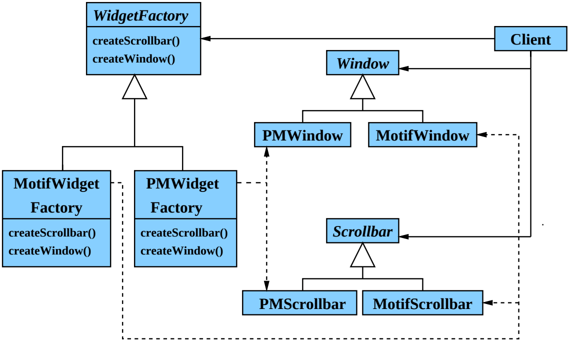
Creational Pattern: Abstract Factory

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

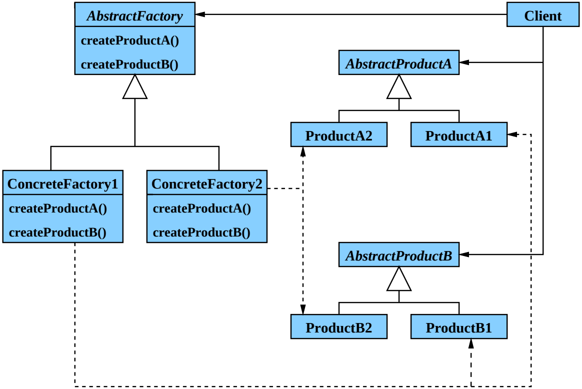
Creational Pattern: Abstract Factory

Motivation



Creational Pattern: Abstract Factory

Structure



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

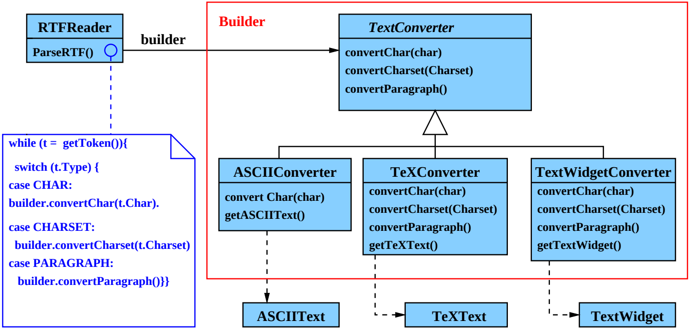
Creational Pattern: Builder

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

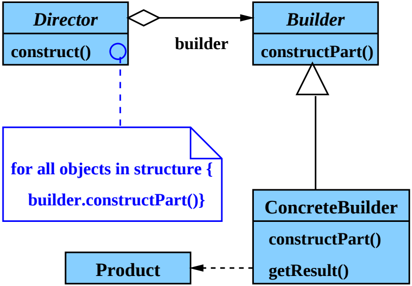
Motivation



```
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



Creational Pattern: Builder

Consequences

- 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

Structural Pattern: Façade

Intent

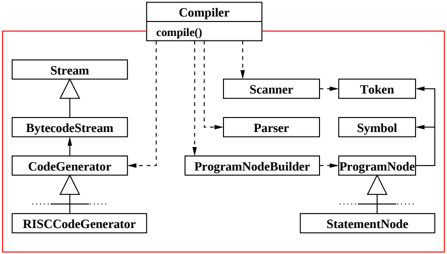
- Provide a unified interface to a set of interfaces in a subsystem

Motivation

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

Structural Pattern: Façade

Motivation (2)

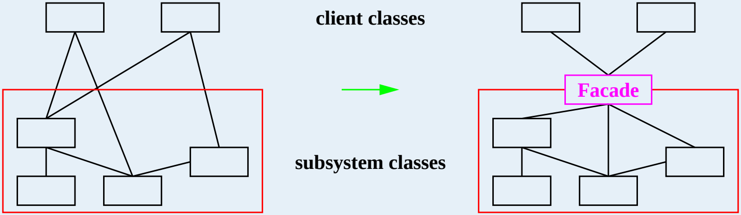


Structural Pattern: Façade

Applicability

- Simple interface to complex subsystem
- Many dependencies between clients and subsystem–Facade reduces coupling
- Layering

Structure



Structural Pattern: Façade

Consequences

- 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

Structural Pattern: Flyweight

Intent

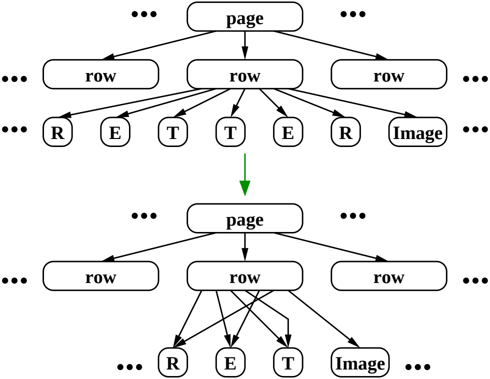
- Use sharing to support large numbers of fine-grained objects efficiently

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

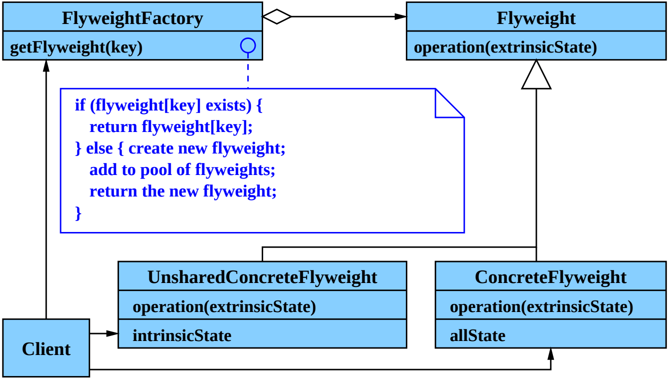
Structural Pattern: Flyweight

Motivation



Structural Pattern: Flyweight

Structure



FlyweightFactory
getFlyweight(key)
Flyweight
operation(extrinsicState)
UnsharedConcreteFlyweight
operation(extrinsicState)
intrinsicState
ConcreteFlyweight
operation(extrinsicState)
allState
Client

```
if (flyweight[key] exists) {
   return flyweight[key];
} else { create new flyweight;
   add to pool of flyweights;
   return the new flyweight;
}
```

Structural Pattern: Flyweight

Applicability

- Many similar objects
- Memory consumption to 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

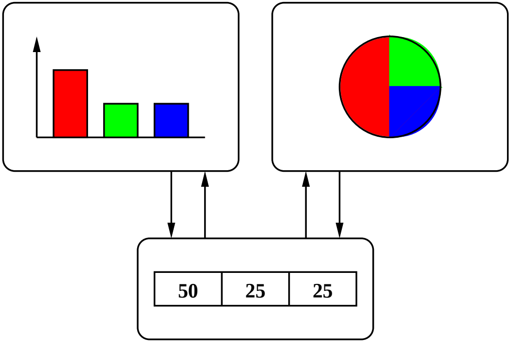
Intent

- Define 1 : *n*-dependency between objects
- State-change of one object notifies all dependent objects

Behavioral Pattern: Observer

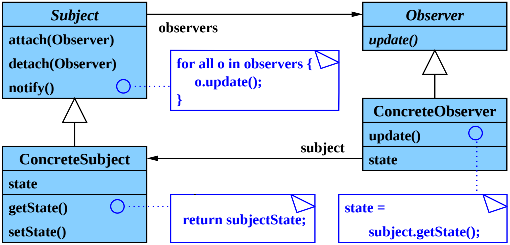
Motivation

- Maintain consistency between internal model and external views



Behavioral Pattern: Observer

Structure



Behavioral Pattern: Observer

Applicability

- 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

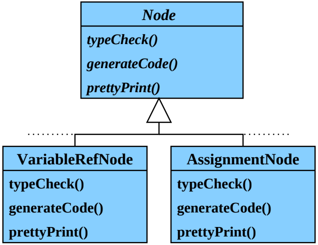
Intent

- Represents operations on an object structure by objects
- Is a way of separating an object structure from an algorithm that operates on it
- Allows adding new operations without changing the classes

Behavioral Pattern: Visitor

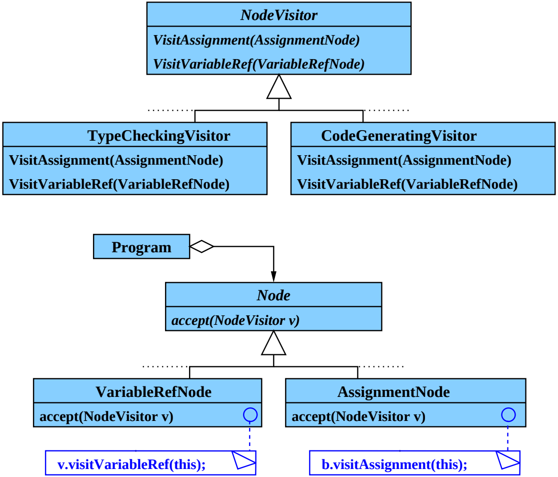
Motivation

- 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

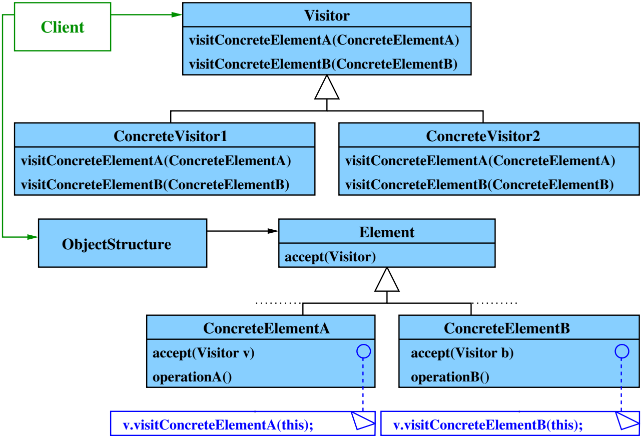
Syntax Tree with Visitors



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);

Behavioral Pattern: Visitor

Structure


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

- Object structure with many different 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

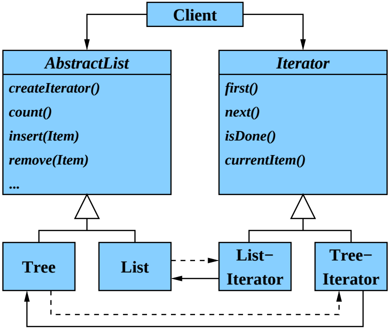
Behavioral Pattern: Iterator (Cursor)

Intent

- Sequential access to components of a container object
- Representation of object hidden

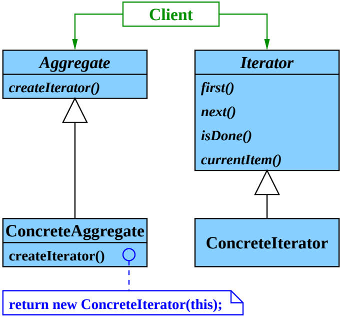
Behavioral Pattern: Iterator (Cursor)

Motivation



Behavioral Pattern: Iterator (Cursor)

Structure



- `ConcreteIterator` administers current object and determines subsequent object(s)

Behavioral Pattern: Iterator (Cursor)

Applicability

- 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?)