Softwaretechnik

Design Patterns

Stephan Arlt

University of Freiburg

SS 2011

Design Patterns

- Gamma, Helm, Johnson, Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.[1]
- 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, # objects
- alternative approaches and combinations possible
- task: which (combination of) pattern(s) is best
- class-based ↔ object-based patterns
- inheritance ↔ delegation

[1] Gang of Four

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

Object composition

Inheritance = White-box reuse

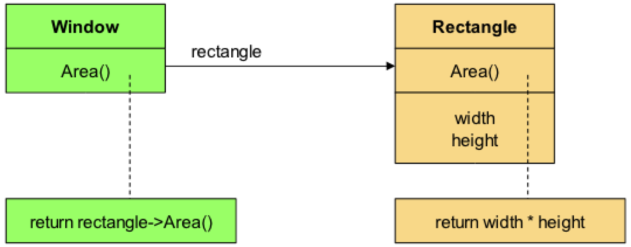
- Reuse by inheritance
- Inheritance is static
- Internals of base classes are visible
- Inheritance breaks encapsulation

Composition = Black-box reuse

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

Delegation

- 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

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!

Indirection

Object creation

- 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

- Coupling
  - Hard wiring of method calls
  - No changes without compiling
- Decoupling
  - Objectification of methods
  - Replaceable at runtime
- Patterns: Chain of Responsibility, Command

Indirection
Implementation

- 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 implementaion
  - 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

Indirection
Complex algorithms

- 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

- 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

Indirection
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
  - 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

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)

Scope

Class static relationships between classes (inheritance)

Object dynamic relationships between objects

Standard Template

- Intent
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Creational Patterns

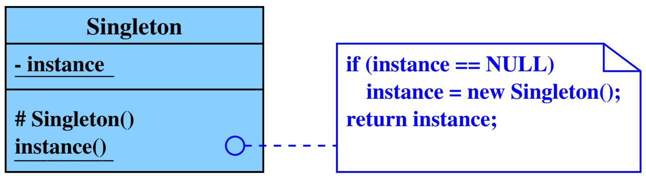
Pattern: Singleton object, creational

Intent

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

Motivation

- to create factories and builders



Structure

Applicability

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

Consequences

- access control on singleton
- structured address space (compared to global variables)

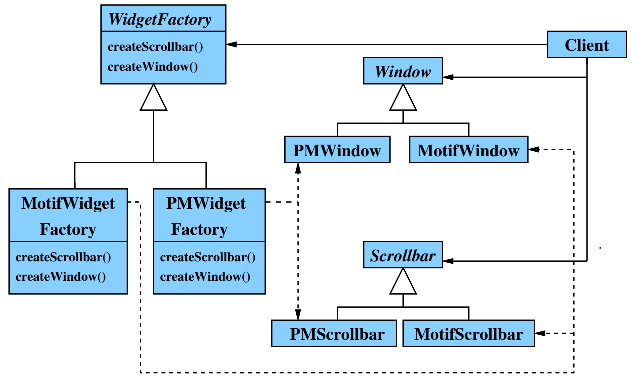
Pattern: Abstract Factory (Kit) object, creational

Intent

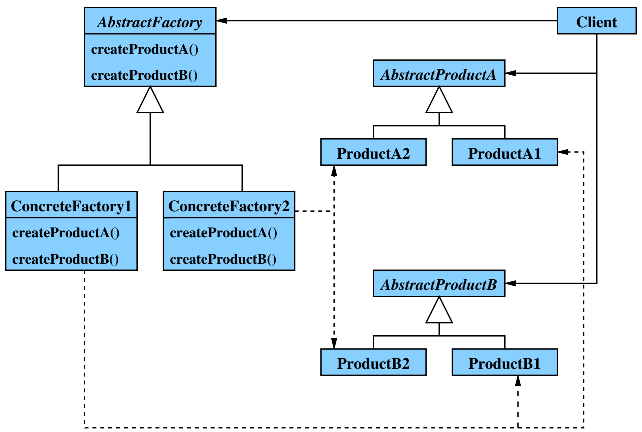
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes

Motivation

- user interface toolkit supporting multiple look-and-feel standards *e.g.*, Motif, Presentation Manager



Structure



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

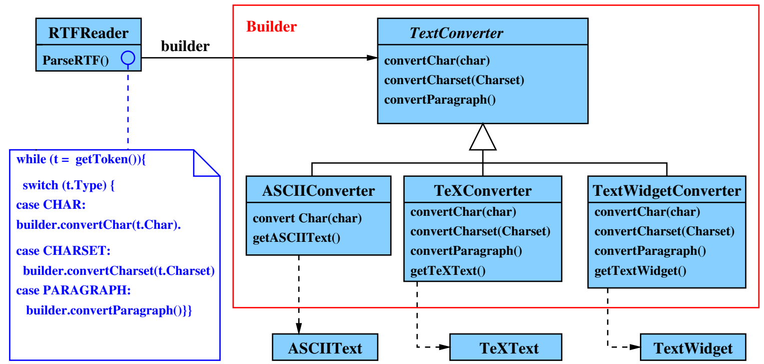
Pattern: Builder object, creational

Intent

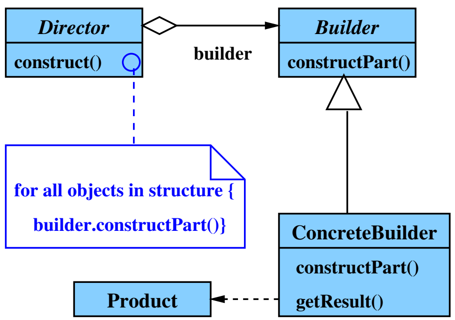
- Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Motivation

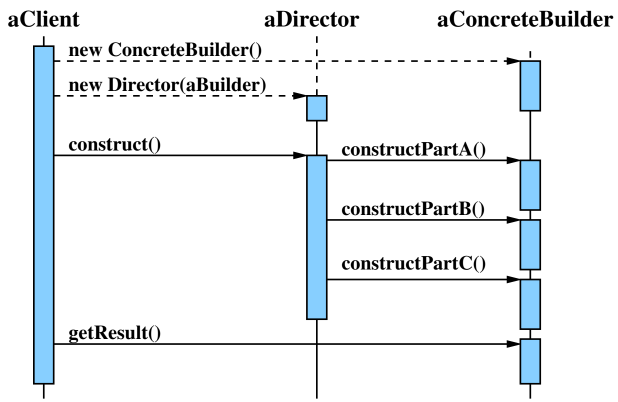
- read RTF and translate in different exchangeable formats



Structure



Interaction Diagram for 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 Patterns

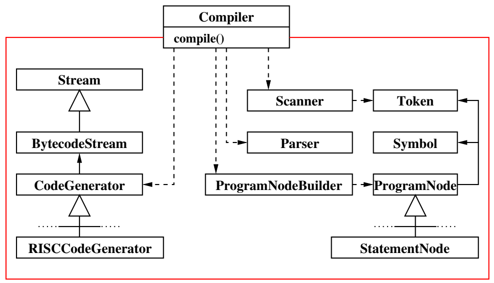
Pattern: Facade object, structural

Intent

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

Motivation

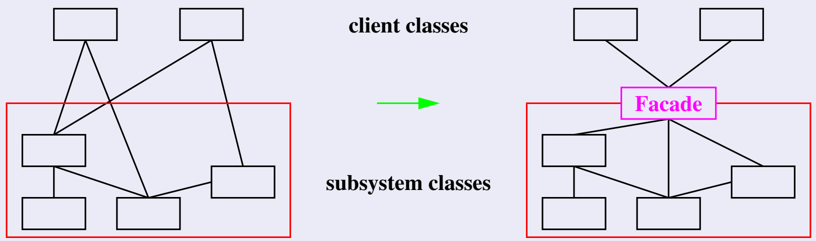
- compiler subsystem contains Scanner, Parser, Code generator, etc
- Facade combines interfaces and offers new `compile()` operation



Applicability

- simple interface to complex subsystem
- many dependencies between clients and subsystem → Facade reduces coupling
- layering

Structure



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

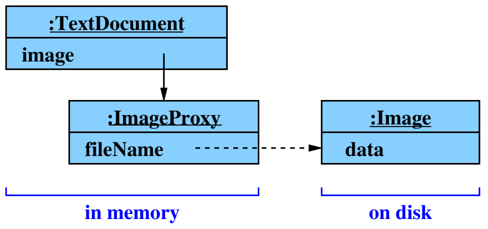
Pattern: Proxy (Surrogate) object, structural

Intent

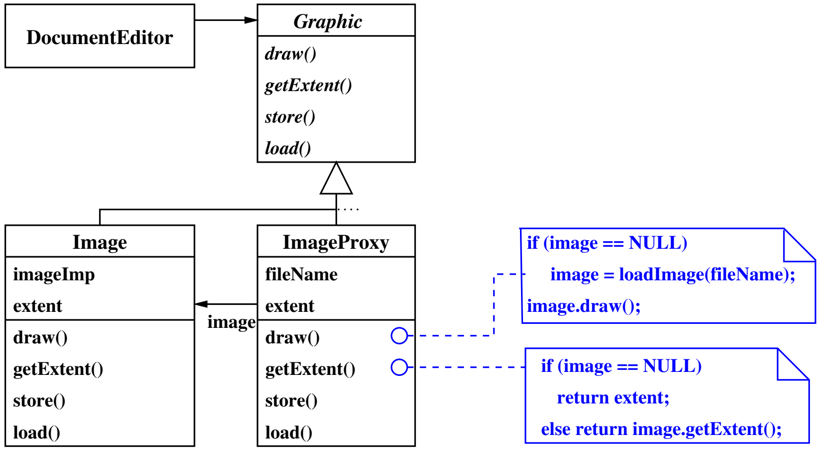
- control access to object

Motivation

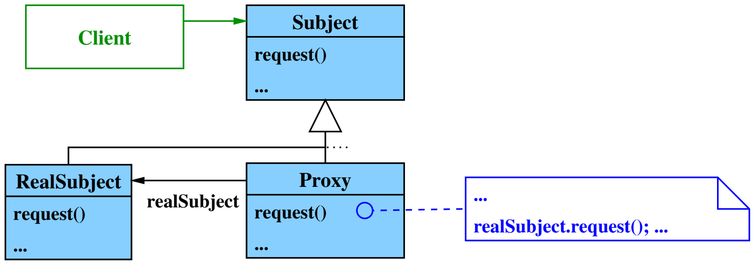
- multi-media editor loads images, audio clips, videos etc on demand
- represented by proxy in document
- proxy loads the "real object" on demand



Motivation (2)



Structure



Applicability

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

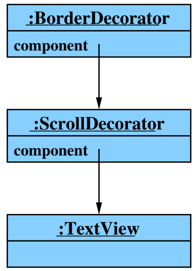
Pattern: Decorator (Wrapper) object, structural

Intent

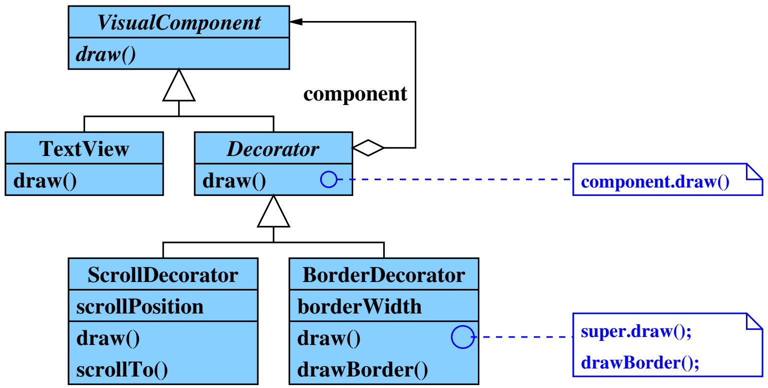
- extend object's functionality dynamically
- more flexible than inheritance

Motivation

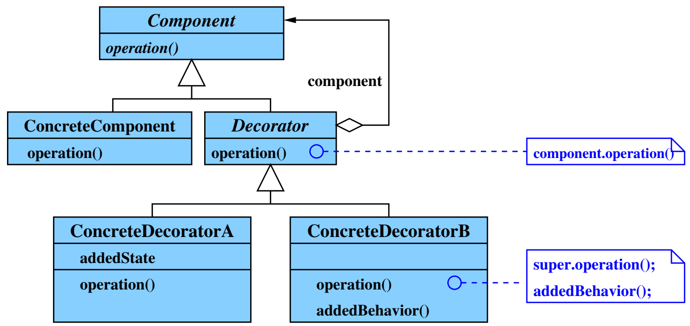
- graphical object can be equipped with border and/or scroll bar
- decorator object has same interface as the decorated object
- decorated forwards requests
- recursive decoration



Motivation (cont)



Structure



Applicability

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

Consequences

- more flexible than inheritance
- avoids feature-laden classes high up in the hierarchy
- decorator $\neq$ component
- lots of little objects $\rightarrow$ hard to learn and debug

Behavioral Patterns

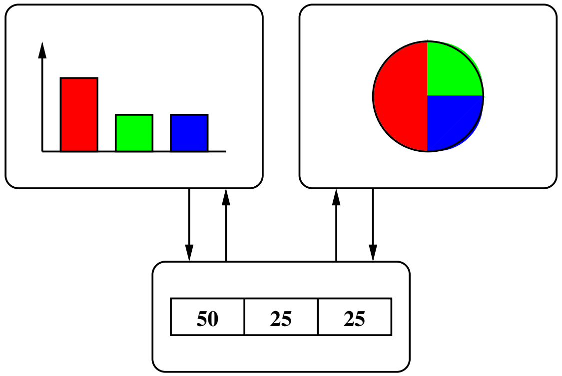
Pattern: Observer object, behavioral

Intent

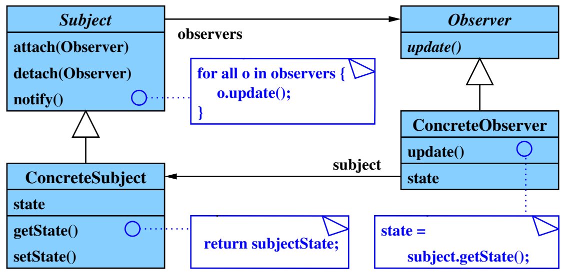
- define 1 : *n*-dependency between objects
- state-change of one object notifies all dependent objects

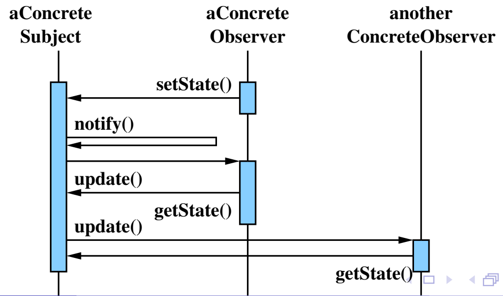
Motivation

- maintain consistency between internal model and external views



Structure





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()`

Pattern: Visitor object, behavioral

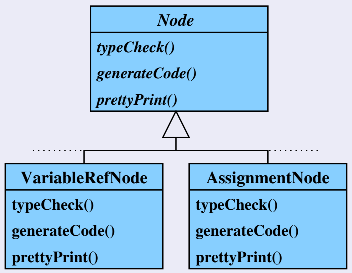
Intent

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

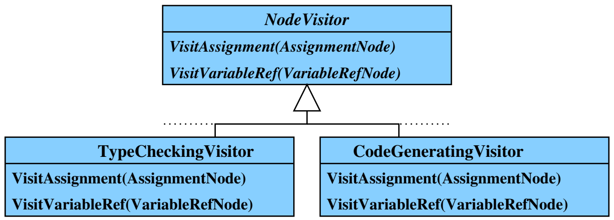
Motivation

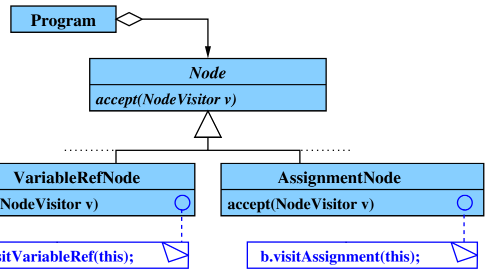
- processing of a syntax tree in a compiler: type checking, code generation, pretty printing, . . .
- naive approach: put operations into node classes → hampers understanding and maintainability
- here: realize each processing step by a visitor

without visitor

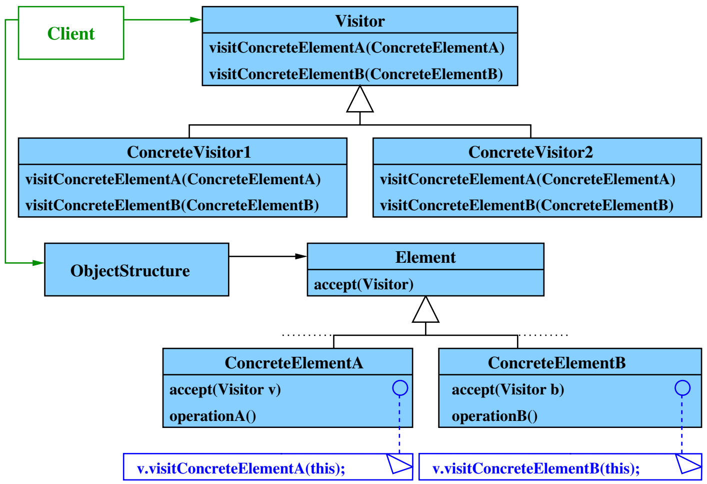


Syntax Tree with Visitors

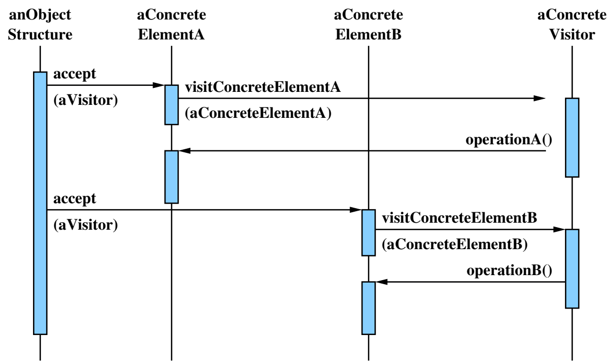




Structure



Visitor: Interaction Diagram



Applicability

- 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