Design Patterns

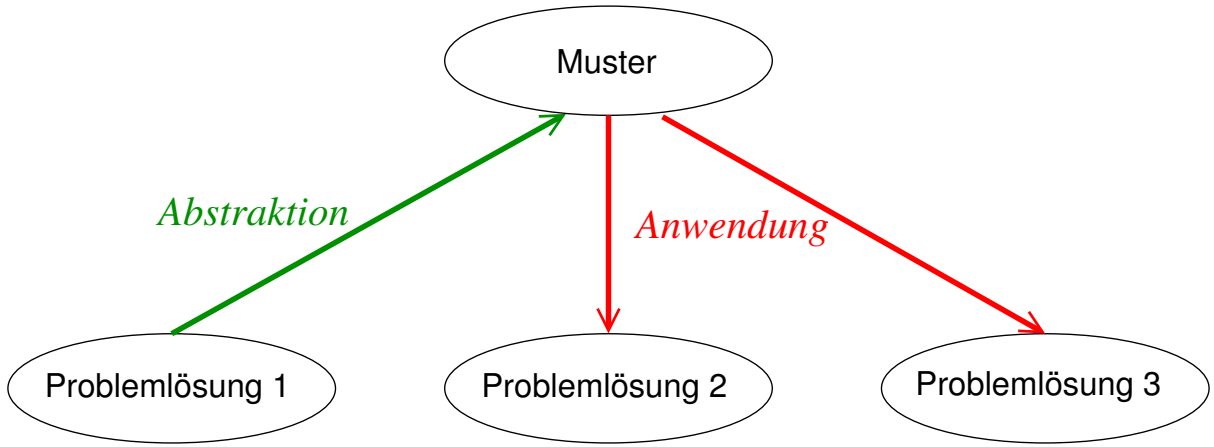
Prof. Mirco Tribastone, Ph.D.

22.11.2011

Introduction

Basic Idea

The same (well-established) schema can be reused as a solution to similar problems.



Advantages

- Reusing tried and tested solution principles (quality, cost savings)
- abstract documentation of designs
- common vocabulary for communication among developers

Morale

Do not reinvent the wheel!

History

- 1977 Alexander: Architecture patterns for buildings and urban development
- 1980 Smalltalk's MVC principle (Model View Controller)
- Since 1990 Object-oriented patterns in software engineering
- 1995 Design Pattern catalogue of Gamma, Helm, Johnson, Vlissides (GoF "Gang of Four")
- Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad (1996). Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons. ISBN 0-471-95869-7
- Martin Fowler (2002). Patterns of Enterprise Application Architecture. Addison-Wesley. ISBN 978-0321127426.
- Gregor Hohpe, Bobby Woolf (2003). Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley. ISBN 0-321-20068-3.
- Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra (2004). Head First Design Patterns. O'Reilly Media. ISBN 0-596-00712-4.

Essential Elements of a Software Design Pattern

- Name of the pattern
- Description of the class of problems the pattern is applicable to
- Description of an example of use
- Description of the solution (structure, responsibilities, ...)
- Description of the consequences (cost-benefit analysis)

Design Pattern Catalogue (GoF)

Description Schema for a Design Pattern

- **Pattern Name and Classification**
  The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.
- **Intent**
  A short statement that answers the following question: What does the design pattern do? What is its rational and intent? What particular design issue or problem does it address?
- **Also Known As**
  Other well-known names for the pattern, if any.
- **Motivation**
  A scenario that illustrates a design problem and how the class and oject structures in the pattern solve the problem. The scenario will help you understand the more abstract descriptions of the pattern that follows.

- **Applicability**
  What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?
- **Structure**
  A graphical representation of the classes in the pattern using a notation based on the Object Modelling Technique (OMT). We also use interaction diagrams to illustrate sequences of requests and collaborations between objects.
- **Participants**
  The classes and/or objects participating in the design pattern and their responsibilities.
- **Collaborations**
  How the participants collaborate to carry out their responsibilities.
- **Consequences**
  How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspects of system structure does it let you vary independently?

- **Implementation**
  What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?
- **Sample Code**
  Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.
- **Known Uses**
  Examples of the pattern found in real systems. We include at least two examples from different domains.
- **Related Patterns**
  What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

Classification of Design Patterns

- **Creational Patterns** (concern the creation of objects)
  - **Abstract Factory** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - **Builder** Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - **Factory Method** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
  - **Prototype** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - **Singleton** Ensure a class only has one instance, and provide a global point of access to it.
- **Structural Patterns** (concern the structural composition of classes or objects)
  - **Adapter** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn‘t otherwise because of incompatible interfaces.
  - **Bridge** Decouple an abstraction from its implementation so that the two can vary independently.
  - **Composite** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- **Structural Patterns** (continued)
  - **Decorator** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
  - **Facade** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
  - **Flyweight** Use sharing to support large numbers of fine-grained objects efficiently.
  - **Proxy** Provide a surrogate or placeholder for another object to control access to it.

- **Behavorial Patterns** (concern the interaction of objects and the distribution of responsibilities)
  - **Chain of Responsibility** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
  - **Command** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
  - **Interpreter** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
  - **Iterator** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- **Mediator** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Observer** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State** Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.
- **Strategy** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Template Method** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Visitor** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the element on which it operates.

Overview

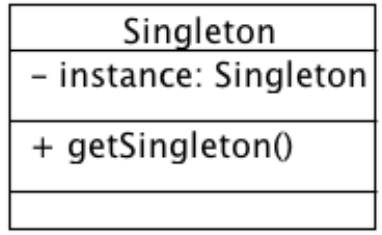
1. Singleton (creational)
2. Abstract Factory (creational)
3. Composite (structural)
4. Proxy (structural)
5. Iterator (behavioural)
6. Observer (behavioural pattern)
7. State (behavioural pattern)

Example 1: **Singleton (Creational Pattern)**

Intent

Ensures that a class has only one instance; provides a global access point to it.

Structure



Known uses: `java.lang.Runtime`; `org.eclipse.core.runtime.Plugin`.

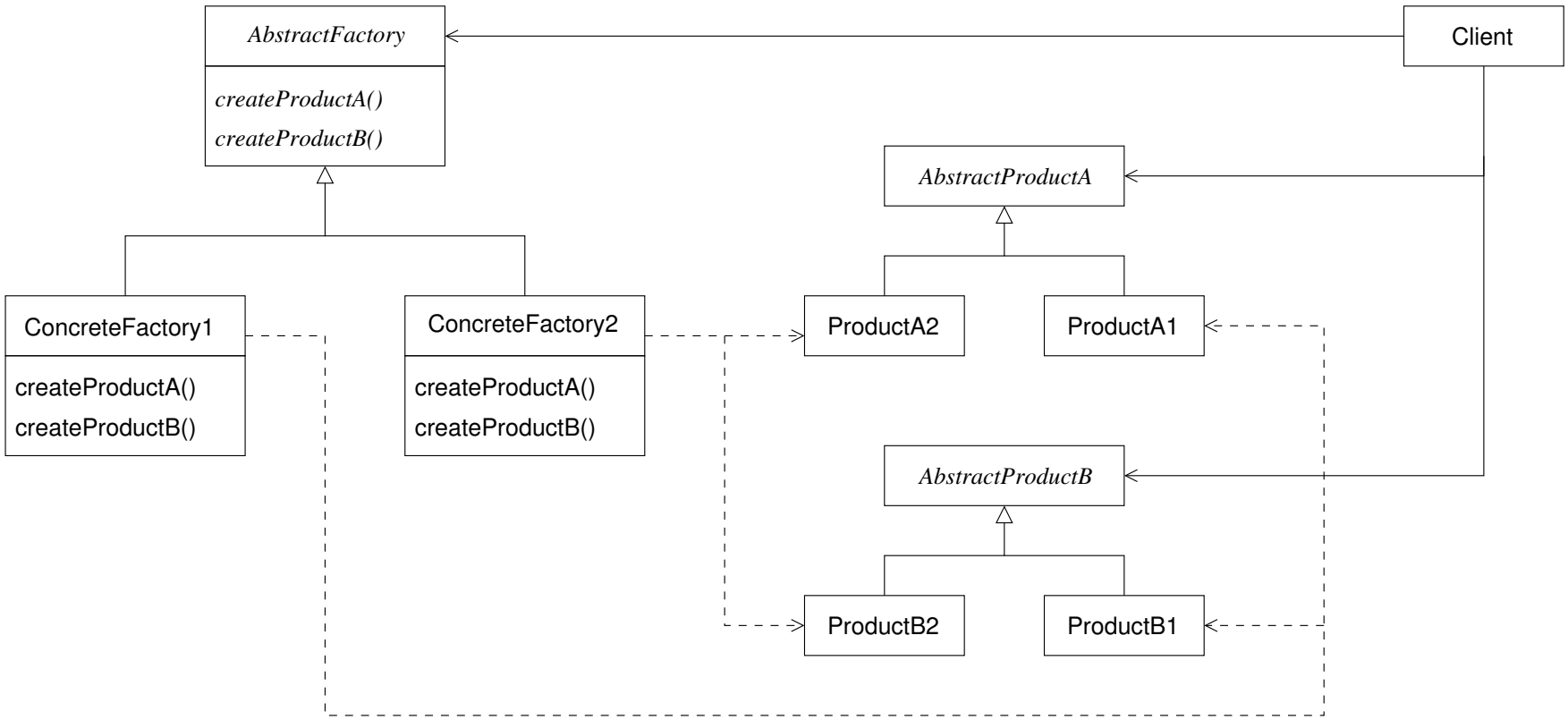
Example: . . .

Example 2: **Abstract Factory (Creational Pattern)**

Intent

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Structure



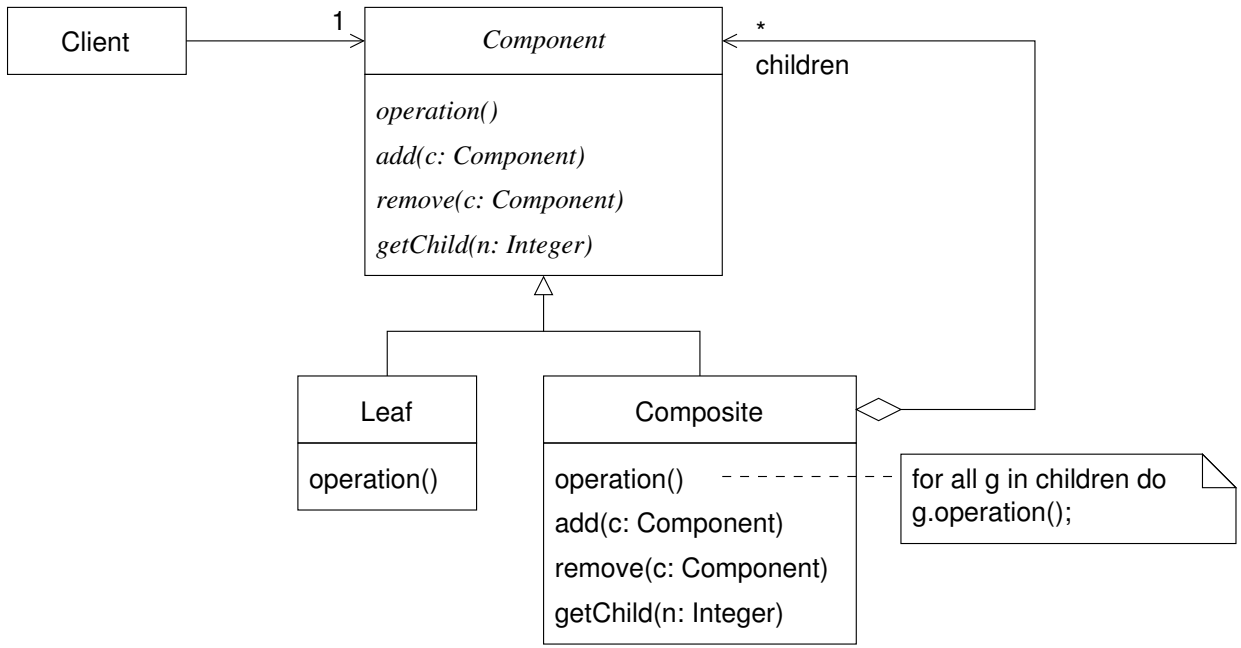
Known uses: `Toolkit` in AWT.

Example 3: **Composite (Structural Pattern)**

Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Structure



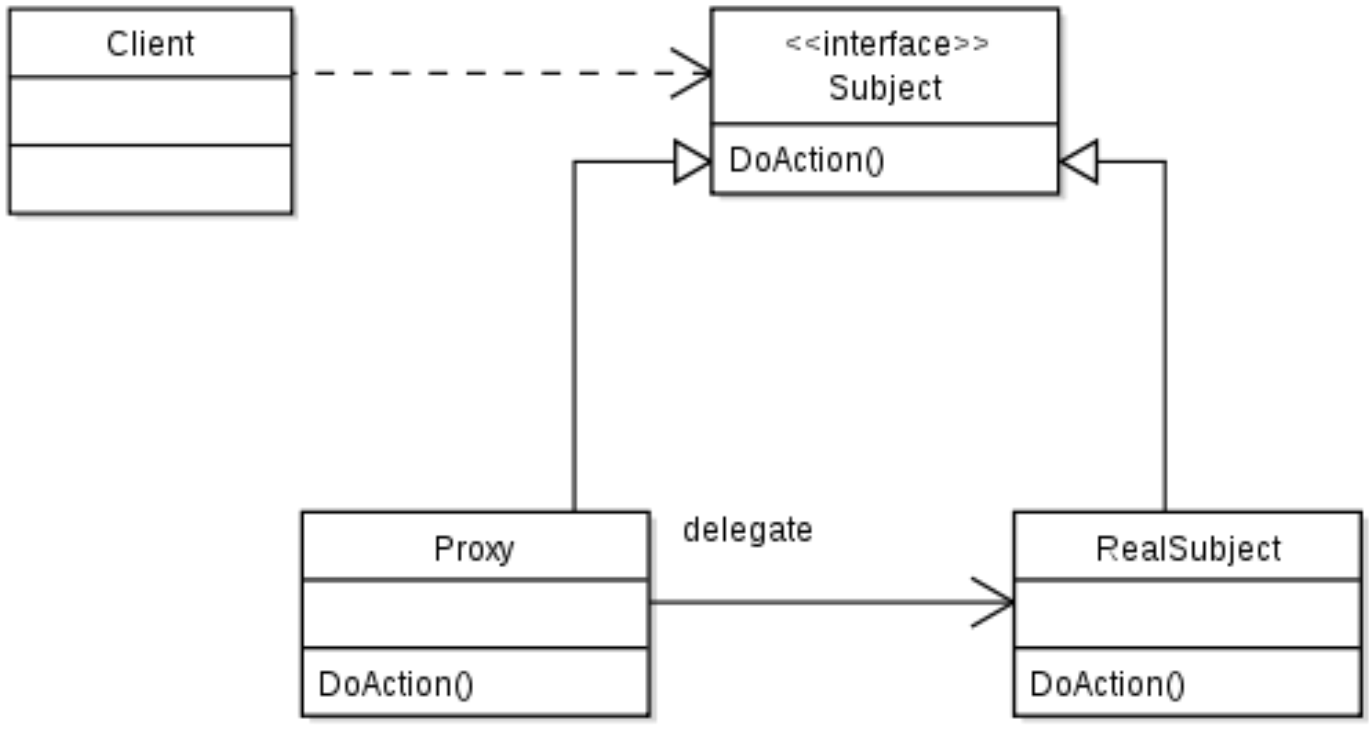
Known uses: `Composite` and `Control` in SWT; geometric figures (tutorial).

Example 4: **Proxy (Structural Pattern)**

Intent

Provide a surrogate or placeholder for another object to control access to it.

Structure



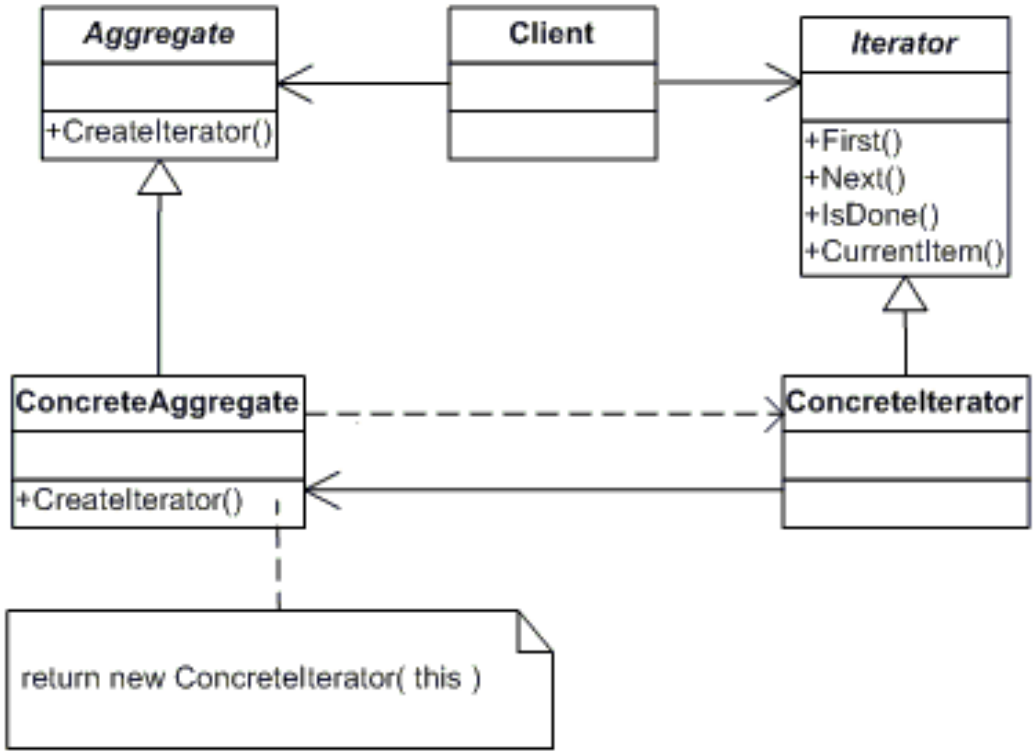
Known uses: Plug-in mechanisms.

Example 5: **Iterator (Behavioural Pattern)**

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Structure



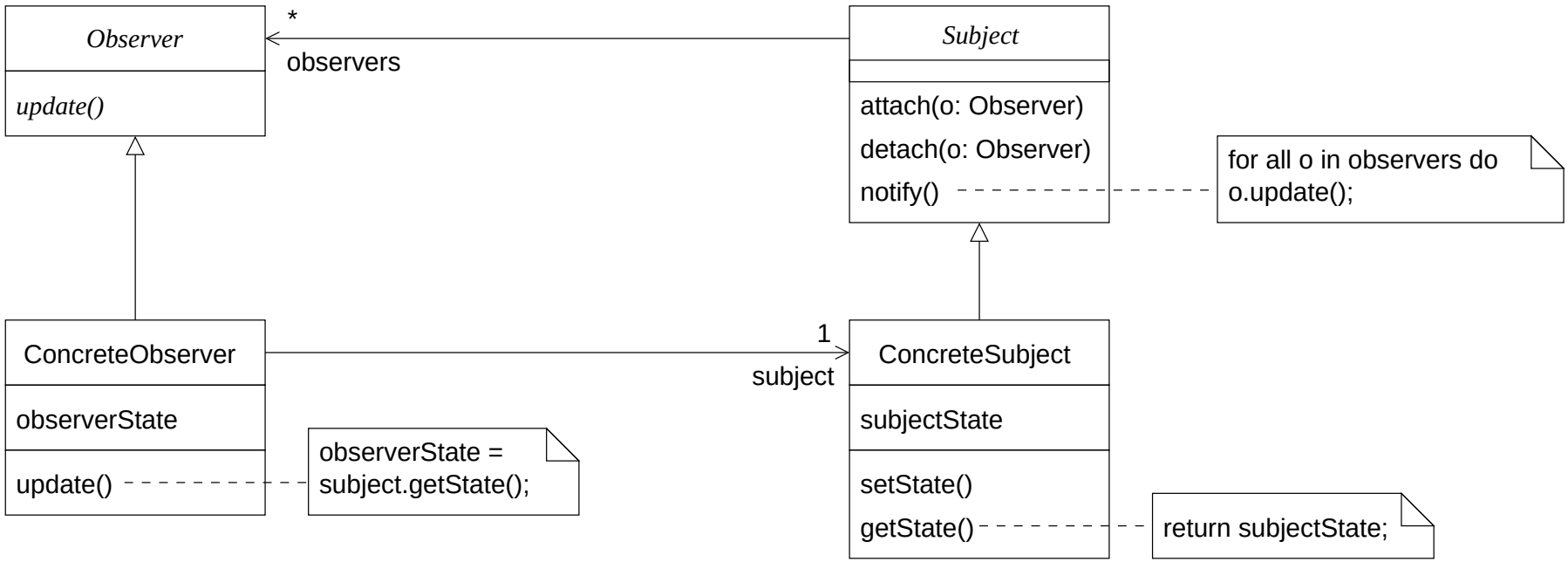
Known uses: Java API.

Example 6: **Observer (Behavioural Pattern)**

Intent

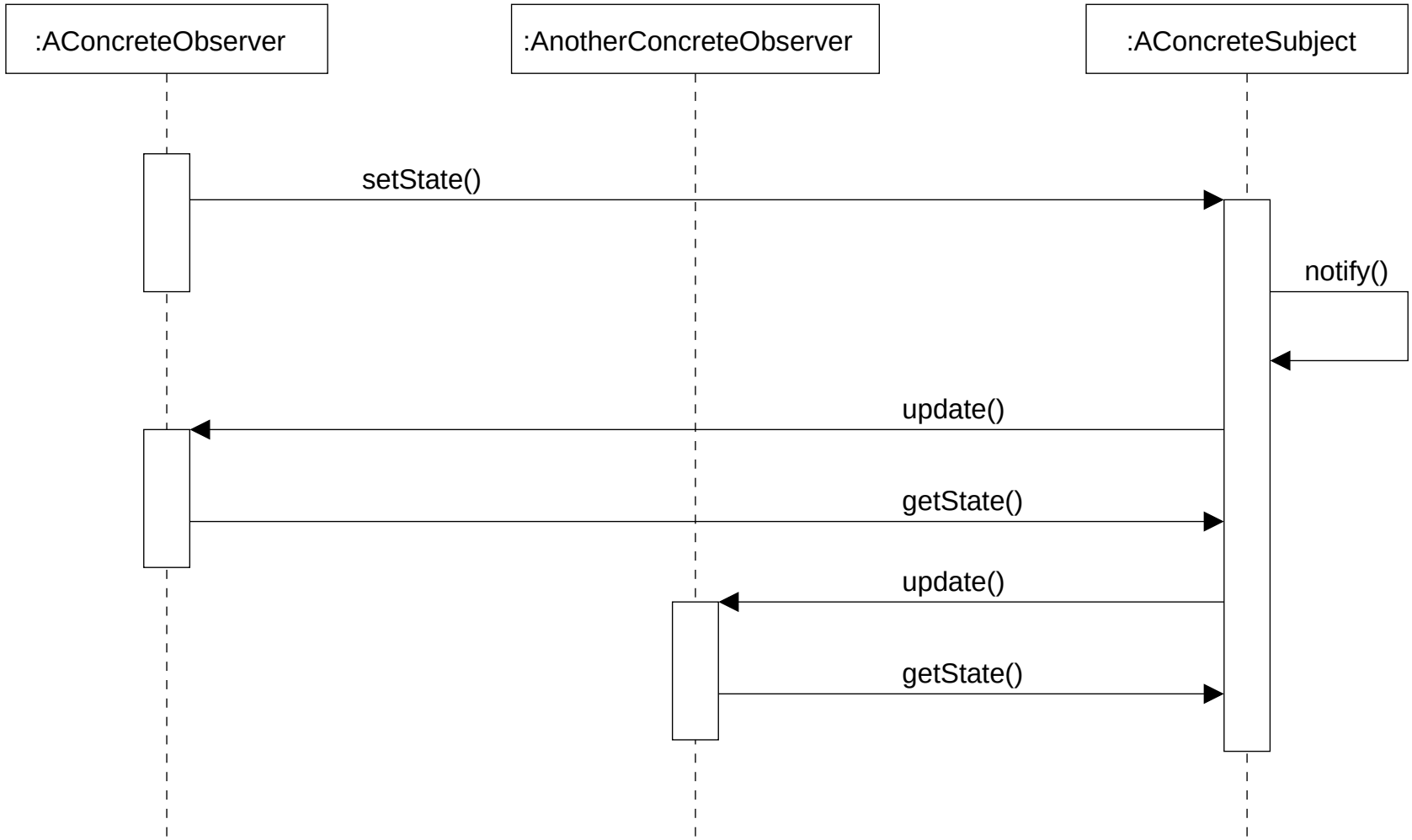
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Structure



Known uses: Event listeners in user interfaces (SWT).

Interactions

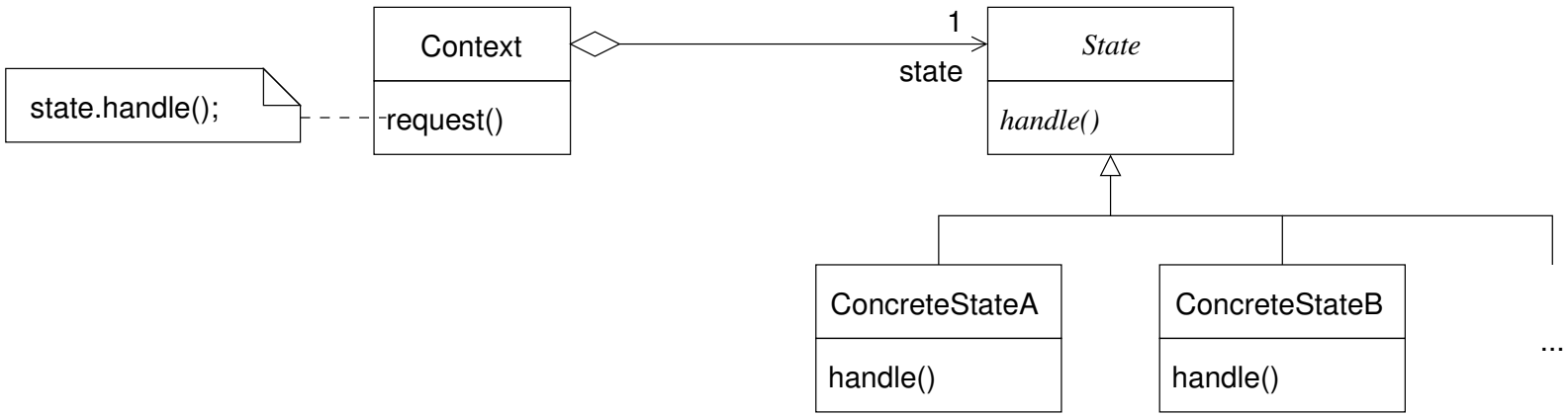


Example 7: **State (Behavioural Pattern)**

Intent

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

Structure



Example of use:

Realization of state diagrams by state objects.

MVC Architecture