Coordination Patterns

Design Patterns and their relevance for Coordination

Oscar Nierstrasz

Software Composition Group
Institut für Informatik (IAM)
Universität Bern

oscar@iam.unibe.ch
http://iamwww.unibe.ch/~scg/

Overview

- ❑ What are Design Patterns?
  - ➪ Example: the Proxy pattern
- ❑ What problems do patterns solve?
  - ➪ Patterns are a form of *communication*
- ❑ What kinds of patterns exist?
  - ➪ Architectural styles, design patterns, idioms ...
- ❑ What patterns solve coordination problems?
  - ➪ Administrator/Worker, Pipes and Filters, Blackboard, ...
- ❑ What are the research opportunities?
  - ➪ Specify and classify coordination patterns
  - ➪ Develop tools and languages that make it easier to apply patterns

What are Design Patterns?

Patterns were first systematically catalogued in the domain of architecture:

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."*
>
> *Alexander, et al., A Pattern Language*

Software design patterns document standard solutions to common design problems:

> *"Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems. Our goal is to capture design experience in a form that people can use effectively."*
>
> *Gamma, et al., Design Patterns*

What Design Patterns are not ...

Algorithms are not design patterns

- ☞ algorithms solve computation problems, not design problems
- ☞ *merge-sort* is an algorithm; *divide and conquer* is a design pattern

Software components are not design patterns

- ☞ design patterns describe a *way* of solving a problem
- ☞ design patterns document pros and cons of different implementations
- ☞ software components may be implemented using design patterns

Frameworks are not design patterns

- ☞ a framework implements a generic software architecture using an object-oriented language
- ☞ a design pattern documents the solution to a *specific* design problem
- ☞ a framework may use and be documented with design patterns
- ☞ like frameworks, design patterns are drawn from experience with multiple applications solving related problems

How are Design Patterns Specified?

1. **Pattern Name and Classification:** should convey *essence* of pattern
   - ☞ *Also Known As:* other common names
2. **The Problem Forces:** describes when to apply the pattern
   - ☞ *Intent:* short statement of rationale and intended use
   - ☞ *Motivation:* a problem scenario and example solution
   - ☞ *Applicability:* in which situations can the pattern be applied
3. **The Solution:** abstract description of design elements
   - ☞ *Structure:* class and scenario diagrams
   - ☞ *Participants:* participating classes/objects and their responsibilities
   - ☞ *Collaborations:* how participants carry out responsibilities
4. **The Consequences:** results and trade-offs of applying the pattern
   - ☞ *Implementation:* pitfalls, hints, techniques, language issues
   - ☞ *Sample Code:* illustrative examples in C++, Smalltalk etc.
   - ☞ *Known Uses:* examples of the pattern found in real systems
   - ☞ *Related Patterns:* competing and supporting patterns

The Proxy Pattern

Intent

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

Also Known As

Surrogate

Motivating Example

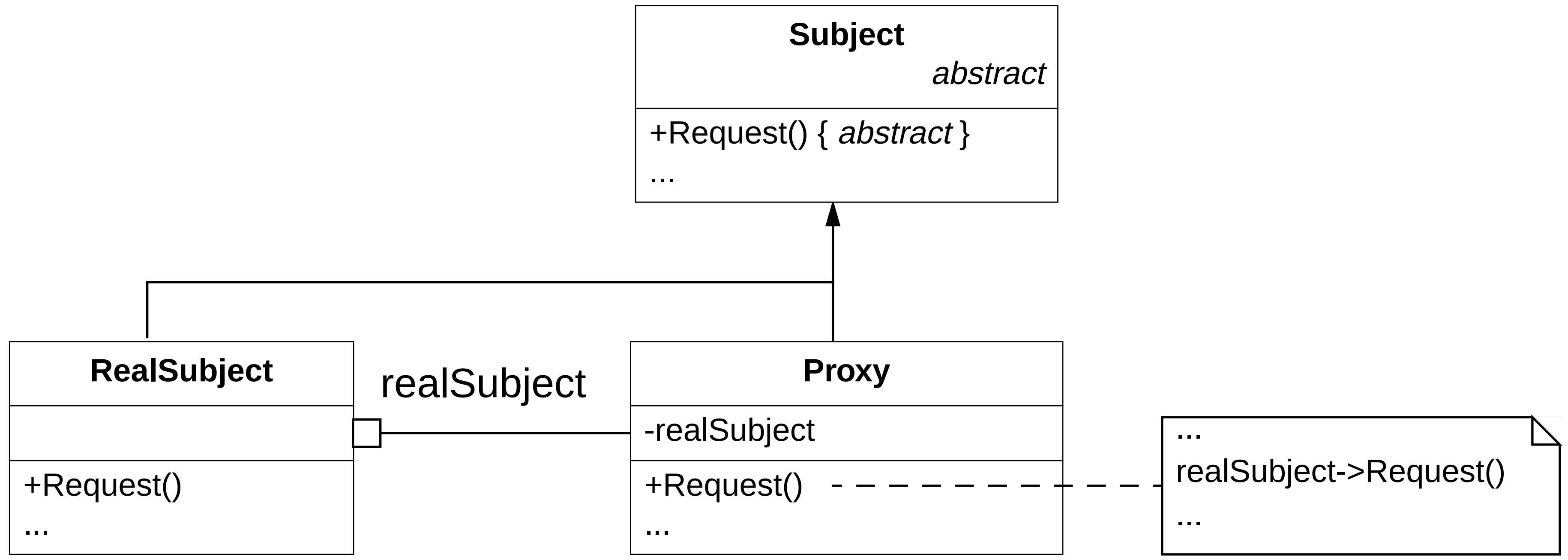
Speed up loading of a complex multimedia document by using proxies to represent images and other large components. The actual objects will be loaded by the proxies only when they need to be displayed.

Applicability

1. A *remote proxy* represents an object in a different address space.
2. A *virtual proxy* creates expensive objects on demand (e.g., multimedia).
3. A *protection proxy* controls access to the original object.
4. A *smart reference* looks like a pointer but performs additional actions when dereferenced, such as:
   - counting references so the object can be freed when the count is zero
   - loading a persistent object into memory when first referenced
   - checking an object is locked before it is accessed or modified

Proxy — Structure

Structure



Proxy — Participants and Collaborations

Participants

- ❑ Proxy:
  - maintains a reference to the real Subject
  - provides an identical interface to the Subject
  - controls access to the real subject, and may be responsible for creating and deleting it
  - other responsibilities depend on the kind of proxy ...
- ❑ Subject:
  - defines common interface so Proxy and RealSubject can be interchanged
- ❑ RealSubject:
  - defines the real object that the proxy represents

Collaborations

Proxy forwards requests to RealSubject when appropriate

Proxy — Consequences ...

Consequences

The Proxy introduces a level of indirection that can be used to do various things, such as hiding the real location of an object, delaying loading or initialization until an object is needed, or performing various housekeeping activities.

Known Uses

NEXTSTEP uses proxies to represent distributed objects. ...

Related Patterns

Adaptor provides a different interface to the object it adapts, in contrast to Proxy.

Decorators may be implemented in a similar way to Proxies, but the intent is different. Decorators add responsibilities, whereas Proxies control access.

Sample Design Patterns

The following design patterns are typical of those found in *Gamma, et al.*

Creational Patterns

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.

Structural Patterns

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.
Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Behavioural Patterns

Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
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 with changing the algorithm's structure.

What Problems do Design Patterns Solve?

Patterns document design experience:

- ❑ Patterns enable widespread reuse of software architecture
- ❑ Patterns improve communication within and across software development teams
- ❑ Patterns explicitly capture knowledge that experienced developers already understand implicitly
- ❑ Useful patterns arise from practical experience
- ❑ Patterns help ease the transition to object-oriented technology
- ❑ Patterns facilitate training of new developers
- ❑ Patterns help to transcend “programming language-centric” viewpoints

Schmidt, CACM Oct 1995

Authoring Patterns

- ❑ Pattern descriptions should contain concrete examples
- ❑ Pattern names should be chosen carefully and used consistently
- ❑ Resist the temptation to recast everything as a pattern
- ❑ Focus on developing patterns that are strategic to the domain and reuse existing tactical patterns
- ❑ Patterns are validated by experience rather than by testing
- ❑ Directly involve pattern authors with application developers and domain experts
- ❑ Pattern descriptions explicitly record engineering trade-offs and design alternatives that resolve non-functional forces
- ❑ Carefully document the contexts where patterns apply and do not apply

Schmidt, CACM Oct 1995

Common Design Techniques

Design patterns make use of many common design techniques:

- ❑ Class vs. Interface inheritance
  - ☞ Class inheritance supports sharing of implementation
  - ☞ Interface inheritance supports polymorphism
- ❑ Program to an interface, not an implementation!
  - ☞ Increase flexibility by declaring variables of abstract, not concrete classes
  - ☞ Localize knowledge concerning which concrete classes to instantiate
- ❑ Inheritance vs. Object Composition
  - ☞ Inheritance occurs statically, and exposes parent class implementation
  - ☞ Object composition occurs dynamically, and increases run-time flexibility
- ❑ Delegation vs. Inheritance
  - ☞ An object can “implement” a service by delegating it to another object
  - ☞ Delegation increases flexibility by allowing behaviour to change at run-time

Improving Design Flexibility

Many design problems are concerned with achieving flexibility:

- ❑ Varying which *classes* are instantiated
  - ☞ Create objects indirectly by delegating to a “Factory” or “Prototype” object
- ❑ Varying which *operations* are performed at run-time
  - ☞ Use polymorphism and delegation to dynamically select operations
- ❑ Varying hardware or software *platform*
  - ☞ Use polymorphism to hide implementation details from clients
- ❑ Varying object *representations* and implementations
  - ☞ Encapsulate dependencies to prevent changes from cascading
- ❑ Varying *algorithms*
  - ☞ Use polymorphism to substitute or parameterize algorithms
- ❑ *Decoupling* objects
  - ☞ Use object composition and delegation to avoid tight coupling
- ❑ Extending functionality in arbitrary ways
  - ☞ Prefer object composition and delegation to inheritance
- ❑ Adapting existing classes
  - ☞ Use object composition and delegation to hide and adapt them

Idioms

Most Design patterns make use of common idioms:

- ❑ **Handle/Body Classes:** separate classes into a handle (for controlling access) and a body (for implementation) [cf. Proxy, Adaptor, Decorator etc.]
- ❑ **Functors (Function Objects):** an alternative to using function pointers that supports reuse through inheritance, encapsulation of state, and dynamic changes of behaviour
- ❑ **Orthodox Canonical Form:** supports construction, destruction, assignment and copying of non-trivial classes
- ❑ ...

Kinds of Patterns

A Software Architecture defines a system in terms of computational components and interactions amongst those components.

An Architectural Style defines a family of systems in terms of a pattern of structural organization.

— cf. Shaw & Garlan, Software Architecture, pp. 3, 19

- ❑ Architectural patterns (styles)
  - ➪ "a fundamental structural organization schema for software systems"
- ❑ Design patterns
  - ➪ "a commonly-recurring structure of communicating components that solves a general design problem within a particular context"
- ❑ Idioms
  - ➪ "a low-level pattern specific to a programming language" *— or more generally: "an implementation technique"*

— cf. Buschmann et al., Pattern-Oriented Software Architecture, pp. 12-14

What is Coordination?

Coordination is managing dependencies between activities.

— Malone and Crowston, CACM, 26.1

Coordination Patterns

- Architectural styles:
  - Pipes and Filters [SA, POSA]
  - Blackboard; result/agenda/specialist parallelism [SA, POSA, HWPP]
  - Event-based implicit invocation [SA]
- Design Patterns:
  - Master/Slave; Administrator/Worker [POSA, HWPP]
  - Proxy [DP, POSA]
  - Active Object [Schmidt, PLoPD]
- Idioms:
  - Handle/Body [Coplien, *Advanced C++*]
  - Futures
  - RPC

Research Topics

Specifying Patterns:

- ❑ Identifying and classifying Coordination Patterns, Styles and Idioms

Developing better tools and languages

- ❑ Make architectures and designs explicit
  - ☞ Separate coordination from computation
  - ☞ Declarative vs. operational specification

- ❑ Provide more high-level coordination components and connectors that realize common design patterns and arch. styles
  - ☞ Make it easy to implement coordination idioms and patterns as components
  - ☞ Make it easier to reflect about coordination

Literature — Design Patterns

1. Christopher Alexander, Sara Ishakawa and Murray Silverstein, *A Pattern Language*, Oxford University Press, New York, 1977.
2. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stad, *Pattern-Oriented Software Architecture — A System of Patterns*, John Wiley, 1996.
3. Nicholas Carriero and David Gelernter, *How to Write Parallel Programs: a First Course*, MIT Press, cop. 1990, Cambridge, 1990.
4. James O. Coplien, *Pattern Languages of Program Design*, Addison-Wesley, 1995.
5. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.
6. Doug Lea, *Concurrent Programming in Java — Design principles and Patterns*, The Java Series, Addison-Wesley, 1996.
7. Douglas C. Schmidt, “Using Design Patterns to Develop Reusable Object-Oriented Communication Software,” *Communications of the ACM*, vol. 38, no. 10, October 1995, pp. 65-74.
8. Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.