CSE 70: Design Patterns
Strategy, Template Method, Proxy

Ingolf Krueger

Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093-0114, USA

California Institute for Telecommunications
and Information Technologies
La Jolla, CA 92093-0405, USA
Learning Goals for Today
Learning Goals

• Understand the structure, behavior and use of the following Design Patterns:
  – Strategy
  – Template Method
  – Proxy

• Be able to identify basic patterns in given design problems

• Be able to apply basic patterns to improve given designs
What are Patterns?

A pattern ... describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.¹

What are Patterns?

- Describes *one* proven solution for a recurrent design problem
- Defines the context for the solution’s applicability

Diagram:

```
Pattern

Architectural Pattern  Design Pattern  Idiom
```
What are Patterns?

• Describes *one* proven solution for a recurrent design problem

• Defines the context for the solution’s applicability

Pattern

Architectural Pattern  Design Pattern  Idiom

course  granularity  fine

adapted from [POSA96]
Patterns – Schema

• Name:
  - Concise identifier for the pattern

• Context:
  - Under what circumstances is the pattern applicable?

• Problem:
  - What problem does the pattern solve?
  - What are the tradeoffs?

• Solution:
  - Structure
  - Behavior
  - Implementation

• Application examples, variants, consequences
Strategy
Strategy

- Context & Problem
  - For the same problem multiple competing solutions exist or emerge over time
  - The optimal choice among alternatives is dependent on the runtime context
  - Necessity to swap algorithms/implementations of algorithms dynamically
  - Tight coupling between clients and the algorithms leads to high maintenance cost
  - Inheritance on the client-side leads to a complex-to-maintain class hierarchy

- Solution:
  - Decouple algorithm from its clients
  - Introduce an abstract behavior interface for each variation point in the behavior of the client
  - Introduce a class hierarchy for the different implementations of an abstract interface behavior
  - The client reference to the implementation can then be changed dynamically
Strategy: Structure

Client
  clientInterface()

«interface»
Strategy
  algorithm ()

ConcreteStrategy1
  algorithm ()

ConcreteStrategy2
  algorithm()
Template Method
Example: Process Chat Transcripts

• Goal: Develop a chat system capability allowing administrators to post-process chat transcripts.

• Requirements engineering reveals basic structure of such a process:
  1. Fetch transcript
  2. Filter transcript for privacy
  3. Perform analysis
  4. Store analysis result

• Context conditions:
  - Must ensure that privacy filtering always precedes analysis
  - Privacy filtering and analysis can differ vastly from one analysis type to another

• How to avoid code duplication while catering to these context conditions?
Template Method

• Context & Problem
  - You want to define a family of algorithms that all follow a similar structure, but differ in (some of) the concrete steps
  - You want to maximize reuse, and minimize design/code duplication
  - You want to control what parts of the algorithm family are fixed, and what parts can vary

• Solution:
  - Specify the fixed flow of the algorithm as a final method of an (abstract) superclass
  - Use method calls to represent the individual steps of the algorithm
  - Label methods that are allowed to vary as abstract in the superclass
  - Label methods that are not allowed to vary as final in the superclass
  - Derive classes that implement instances of the algorithm from the superclass, then implement the abstract methods
Template Method: Structure

```
final void algorithm() {
    step1();
    step2();
    step3();
    step4();
}
```
Proxy
Example: Mock Objects
Example: RMI
Proxy

• Context & Problem
  - Access to target component from several other components required
  - Direct access to the target is impossible due to security, efficiency, or distribution requirements

• Solution:
  - Define a placeholder (a proxy) that offers the same interface as the target component
  - Make only the proxy known to the environment
  - Let the proxy access the target, observing security, efficiency and distribution requirements
Proxy: Structure
Proxy: Behavior

:Client

:ServiceProxy

:ServiceProvider

service()

pre()

post()

service()
• Consequences:
  - Decoupling of service provider and service users
  - Location of service provider is transparent to service users
    • Proxy handles communication with service provider
    • Applicable for decoupling of address spaces
  - Increased communications overhead can reduce performance
What have you learned today?
Learning Goals

• Understand the structure, behavior and use of the following Design Patterns:
  – Strategy
  – Template Method
  – Proxy

• Be able to identify basic patterns in given design problems
• Be able to apply basic patterns to improve given designs