Five Fundamental Patterns

Working with Classes and Interfaces

Jim Fawcett

CSE776 – Design Patterns

Summer 2004

Reference

- Working with Classes and Interfaces, five fundamental patterns, Dirk Riehle, C++ Report, March 2000.

A Pattern Language for OOD

- Small patterns
  - Design idioms
  - Good practice
- Pattern language
  - Coordinated set of patterns
  - One focused topic area
  - Individual patterns often stated in abbreviated style

Simple Class Pattern

- Problem:
  - You need to design and implement a concept.
- Context:
  - One implementation is sufficient, no other is needed.
  - Changes to implementation may affect clients.
  - You want to make it as simple as possible, but not simpler.
- Solution:
  - Implement the concept as a single class

Design by Primitives

- Problem:
  - You need to implement a class.
- Context:
  - You expect to evolve the class.
  - You want it to be easy to add new member functions.
  - You want to avoid a fragile class in which changes to a function affect many other functions.
  - You want to make it as simple as possible, but not simpler.
- Solution:
  - Separate more complex non-primative member functions from primitive member functions.
  - Determine the primitive member functions that best help implement the class.
  - Implement non-primitive member functions using primitive member functions.

Interface Class

- Problem:
  - You need to design and implement a concept with different implementations.
- Context:
  - You want to give clients freedom of choice:
    - for selecting an implementation.
    - to not care about the implementation.
  - You want to change implementations without affecting clients.
  - You want to introduce new implementations without making clients notice.
  - You want to separate implementations from their clients.
  - You want to make it as simple as possible, but not simpler.
- Solution:
  - Determine the functionality of the concept separately from its implementations.
  - Represent the functionality as an interface class (only pure virtual functions).
  - Make implementation classes inherit and implement the interface class.

Abstract Base Class

- Problem:
  - You need to ensure identical behavior of concept implementations where functionality is identical, and provide different behavior, where functionality is different.
- Context:
  - You want to avoid redundant code.
  - You want to ease adding other implementations.
  - You want to make it as simple as possible, but not simpler.
- Solution:
  - Separate variant functionality of the implementations from invariant functionality.
  - Implement the invariant part as shared functionality in an abstract base class.
  - Declare the variant part in the abstract base class using pure virtual functions.
  - Make implementations subclasses of the abstract base class that implement the variant part.

Narrow Interface Class

- Problem:
  - You need to minimize effort to introduce new subclasses of an abstract base class.
- Context:
  - You are using an abstract base class with many pure virtual member functions.
  - You expect existing subclasses to evolve and new subclasses to enter the system.
  - You want to make it as simple as possible, but not simpler.
- Solution:
  - Reduce the number of pure virtual member fuctions to it minimum by using design by primitives.
  - Provide default implementations of primitives were possible.
  - Implement all non-primitive member functions using primitives.