Reflection Pattern

Reflection Introduction

- The Reflection architectural pattern provides a mechanism for changing structure and behavior of software systems dynamically.
- It supports the modification of fundamental aspects such as type structures and function call mechanisms.
- In this pattern, an application is split into two parts. A meta level provides information about selected system properties and makes the software self-aware.
- A base level includes the application logic. Its implementation builds on the meta level.
- Changes to information kept in the meta level affect subsequent base-level behavior.

Problem

- How to build systems that support unanticipated changes.

Context

- Support for variation is the key to sustainable architectures for long-lived applications.
  - Over time they must respond to evolving and changing technologies, requirements, and platforms.
- However, it is hard to forecast what can vary in an application and when it must respond to a specific variation request.
- The need for variation can occur at any time, specifically while the application is in productive use.
- Variations can also be of any scale, ranging from local adjustments of an algorithm to fundamental modifications of distribution infrastructure.
- The complexity associated with particular variations should be hidden from maintainers, and there should be a uniform mechanism for supporting different types of variation.

Context (2)

- Designing a system that meets a wide range of different requirements *a priori* can be an overwhelming task.
- A better solution is to specify an architecture that is open to modification and extension.
- The resulting system can then be adapted to changing requirements on demand.
- In other words, we want to design for change and evolution.

Forces

- Changing software is tedious, error prone, and often expensive.
  - Wide-ranging modifications usually spread over many components and even local changes within one component can affect other parts of the system.
  - Every change must be implemented and tested carefully.
  - Software which actively supports and controls its own modification can be changed more effectively and more safely.
- Adaptable software systems usually have a complex inner structure.
  - Aspects that are subject to change are encapsulated within separate components.
  - The implementation of application services is spread over many small components with different interrelationships.
  - To keep such systems maintainable, we prefer to hide this complexity from maintainers of the system.

Forces (2)

- The more techniques that are necessary for keeping a system changeable, such as parameterization, subclassing, or even copy and paste, the more awkward and complex its modification becomes.
  - A uniform mechanism that applies to all kinds of changes is easier to use and understand.
- Changes can be of any scale, from providing shortcuts for commonly-used commands to adapting an application framework for a specific customer.
- Even fundamental aspects of software systems can change, for example the communication mechanisms between components.

Solution

- Encapsulate information about properties and variant aspects of the application's structure, behavior, and state into a set of meta-objects.
- Separate the meta-objects from the core application logic via a two-layer architecture:
  - The ***meta level*** contains the meta-objects
  - The ***base level*** contains the application logic.
- Base-level objects consult an appropriate meta-object before they execute behavior or access state that potentially can vary.

Solution (2)

- The meta level provides a self-representation of the software to give it knowledge of its own structure and behavior, and consists of so-called meta-objects.
- Meta-objects encapsulate and represent information about the software. Examples include type structures, algorithms, or even function call mechanisms.
- The base level defines the application logic. Its implementation uses the meta-objects to remain independent of those aspects that are likely to change.
  - For example, in a distributed application, base-level components might only communicate with each other via a meta-object that implements a specific user-defined messaging mechanism.
  - Changing this meta-object changes the way in which base-level components communicate, but without modifying the base-level code.

Meta-Object Protocol

- The meta level also implements a meta-object protocol (MOP), which is a specialized interface that administrators, maintainers, or even other systems can use to *dynamically* configure and modify the meta-objects in well defined way.
- Since the base-level implementation explicitly builds upon information and services provided by meta-objects, changing them has an immediate effect on the subsequent behavior of the base level.

Meta-Object Protocol (2)

- When *extending* the software, you pass the new code to the meta level as a parameter of the meta-object protocol.
- The meta-object protocol itself is responsible for integrating all change requests.
  - It performs modifications and extensions to meta-level code, and if necessary re-compiles the changed parts and links them to the application while it is executing.
- This provides a reflective application with explicit control over its own modification.

Reflection Benefits

- No explicit modification of source code.
  - You do not need to touch existing code when modifying a reflective system. Instead, you specify a change by calling a function of the meta-object protocol.
- Changing a software system is easy.
  - The meta-object protocol provides a safe and uniform mechanism for changing software.
  - It hides all specific techniques such as the use of visitors, factories and strategies from the user.
  - It also hides the inner complexity of a changeable application. The user is not confronted with the many meta-objects that encapsulate particular system aspects.
  - The meta-object protocol also takes control over every modification. A well-designed and robust meta-object protocol helps prevent undesired changes of the fundamental semantics of an application.

Reflection Benefits (2)

- Support for many kinds of change.
  - Meta-objects can encapsulate every aspect of system behavior, state and structure.
  - An architecture based on the Reflection pattern thus potentially supports changes of almost any kind or scale.
  - Even fundamental system aspects can be changed, such as function call mechanisms or type structures.
  - With the help of reflective techniques it is also possible to adapt software to meet specific needs of the environment or to integrate customer-specific requirements.

Reflection Liabilities

- Modifications at the meta level may cause damage.
  - Even the safest meta-object protocol does not prevent users from specifying incorrect modifications.
  - Such modifications may cause serious damage to the software or its environment.
  - Examples of dangerous modifications include changing a database schema without suspending the execution of the objects in the application that use it, or passing code to the meta-object protocol that includes semantic errors.

Reflection Liabilities (2)

- Not all potential changes to the software are supported.
  - Although a Reflection architecture helps with the development of changeable software, only changes that can be performed through the meta-object protocol are supported.
  - As a result, it is not possible to integrate easily all unforeseen changes to an application, for example changes or extensions to base-level code.
- Not all languages support refection.
  - A Reflection architecture is hard to implement in some languages which offer little or no support for reflection.
  - In such languages it may be impossible to exploit the full power of reflection, such as adding new methods to a class dynamically.

Reflection Liabilities (3)

- Increased Complexity.
  - A meta-object protocol is non-trivial to design and implement.
  - It may happen that a reflective software system includes even more meta-objects than base level components.
- Lower Efficiency.
  - The communication between the two levels decreases the overall performance of the system.
  - You can partly reduce this performance penalty by optimization techniques, such as injecting meta-level code directly into the base level when compiling the system.

> *The heavyweight measures of a Reflection architecture only pay off if there are similarly heavyweight flexibility requirements that justify these measures.*