

Four (More) Patterns You Should Know

Four More... And Then Some

Kevlin Henney

kevin@curbralan.com

Agenda

- Intent
 - ◆ Expand design vocabulary based on patterns
- Content
 - ◆ About Patterns
 - ◆ Manager
 - ◆ Interceptor
 - ◆ Enumeration Method
 - ◆ Context Object
 - ◆ In Closing

(Not So) Hidden Agenda



ACCU, Silicon Valley, 2007-03-21

3

About Patterns

*The enormous success of design patterns is a testimonial to the commonality seen by object programmers. The success of the book *Design Patterns*, however, has stifled any diversity in expressing these patterns.*

Kent Beck

ACCU, Silicon Valley, 2007-03-21

4

Patterns

- A pattern documents a recurring problem-solution pairing within a given context
 - ◆ A pattern is more than either the problem or the solution structure
 - ◆ A pattern contributes to design vocabulary
- A problem is considered with respect to forces and a solution that gives rise to consequences
 - ◆ The full form in which a pattern is presented should emphasise forces and consequences, also stating the essential problem and solution clearly

Pattern Communities

- Patterns can be used in isolation with some degree of success
 - ◆ Represent foci for discussion or point solutions
 - ◆ Offer localised design ideas
- However, patterns are, in truth, gregarious
 - ◆ They're rather fond of the company of patterns
 - ◆ To make practical sense as a design idea, patterns inevitably enlist other patterns for expression and variation, where they compete and cooperate

Forms of Pattern Arrangement

- Patterns form a vocabulary, so it is important to understand how the vocab can be arranged
 - ◆ Pattern complements highlight competitive and cooperative groupings of patterns
 - ◆ Pattern sequences capture the underlying narrative when using many patterns
 - ◆ Pattern stories animate sequences with specifics
 - ◆ Pattern compounds capture commonly recurring subcommunities of patterns
 - ◆ Pattern languages capture a broad set of sequences

Manager

1. *Generally management of many is the same as management of few. It is a matter of organization.*
2. *And to control many is the same as to control few. This is a matter of formations and signals.*

Sun Tzu

static Interference and Clutter

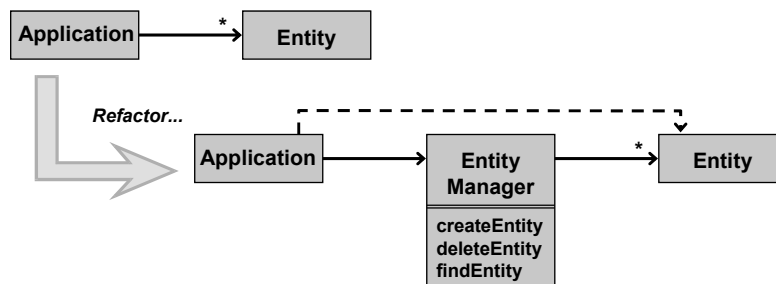
- How can a group of instances be managed so that management is minimally intrusive?
 - ♦ A common intrusive approach is to add *static* state and operations to a class so that the class becomes responsible for managing its instances – this dilutes the cohesion of the class
 - ♦ Another approach is for the user of the instances, such as a root application object, to manage the instances – this clutters such controller objects with incidental management detail

The Manager Pattern

- Introduce a manager object that handles the responsibility for managing instances
 - ♦ Responsibilities typically include creation (e.g. the Factory Method pattern), disposal or recycling (e.g. the Disposal Method pattern), remembering and finding
 - ♦ Clients access the instances via the manager object
 - ♦ A manager tends to simplify and clarify the roles of both client and target objects, decluttering objects higher up in the control hierarchy

A Typical Manager Configuration

- Details of implementation depend on language and application specifics
 - ♦ E.g. in C++ consider using containers for object storage rather than using the heap directly

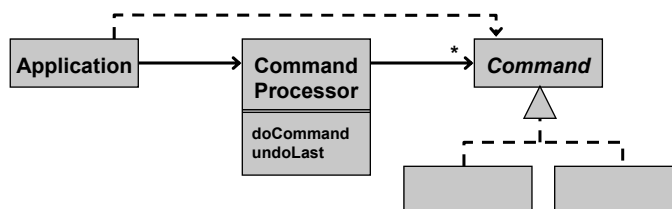


ACCU, Silicon Valley, 2007-03-21

11

The Command Processor Pattern

- A context-narrowed application of the Manager pattern is the Command Processor pattern
 - ♦ Simplifies management of Command objects, e.g. sidesteps the need for subtle solutions for undo, such as a *static* collection or linking Commands



ACCU, Silicon Valley, 2007-03-21

12

Interceptor

The perfect bridge is not perfectly rigid, nor is the perfect building. Rigidity works when nothing moves or changes. Flexibility is required when the ground is prone to shake or the wind to blow.

Richard P Gabriel

Extra-Functional Extension

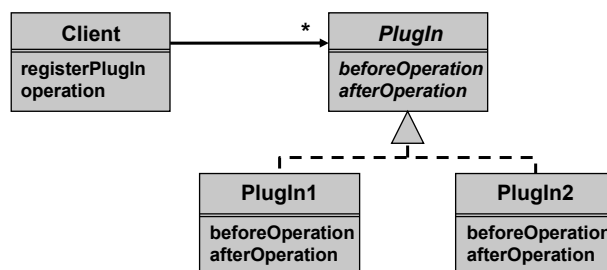
- There is often a need to add extra-functional behaviour to an object or framework
 - ◆ E.g. instrumentation, logging and filtering
- However, many solutions are too hardwired or can be difficult to work with
 - ◆ E.g. a Template Method approach is strongly intrusive with respect to class hierarchy, as is the speculative practice of NVI (Non-Virtual Interfaces)
 - ◆ E.g. Decorator is also intrusive and requires external effort to set up and manage

The Interceptor Pattern

- An object's or framework's basic behaviour can be extended by adding event-driven plug-ins
 - ♦ A loosely coupled, minimally intrusive, testable and easily extensible pluggable approach
 - ♦ Can use chaining or Observer-style notification within the Interceptor
- Callback plug-ins can be expressed in a variety of ways, depending on language and context
 - ♦ Function pointers, delegates, function objects, Strategy objects, closure objects, etc.

A Typical Interceptor Configuration

- Callbacks to plug-ins are made before, after or before and after corresponding events
 - ♦ Can be used to observe, manipulate or veto events



Less Is More

```
class command
{
public:
    void execute()
    {
        do_execute();
    }
    void rollback()
    {
        do_rollback();
    }
    virtual ~command();
private:
    virtual void do_execute() = 0;
    virtual void do_rollback() = 0;
};
```

```
class command_processor
{
public:
    void execute(command *);
    void rollback();
    ...
};
```

```
class command
{
public:
    virtual void execute() = 0;
    virtual void rollback() = 0;
    virtual ~command();
};
```

```
class command_processor
{
public:
    void execute(command *);
    void rollback();
    ...
};
```

Approaches such as NVI offer little benefit to the developmental quality of the code. Leaving out such wrapping keeps things simple and more open.

And Now for Interception

- To extend the capabilities of a Command extend the Command Processor
 - ◆ So first of all ensure that you have a Command Processor to extend!
 - ◆ Extension is orthogonal to the Command hierarchy

```
class command_filter
{
public:
    virtual void before_execute(command *) = 0;
    virtual void after_execute(command *) = 0;
    virtual void before_rollback(command *) = 0;
    virtual void after_rollback(command *) = 0;
    ...
};
```

```
class command_processor
{
public:
    void register_filter(command_filter *);
    void execute(command *);
    void rollback();
    ...
};
```

Enumeration Method

Substance doesn't change. Method contains no permanence. Substance relates to the form of the atom. Method relates to what the atom does. In technical composition a similar distinction exists between physical description and functional description. A complex assembly is best described first in terms of its substances: its subassemblies and parts. Then, next, it is described in terms of its methods: its functions as they occur in sequence. If you confuse physical and functional description, substance and method, you get all tangled up and so does the reader.

Robert M Pirsig

Encapsulated Iteration

- Traversal over object collection contents should preserve the encapsulation of the collection
 - ◆ But it should also reflect the environment of use of the collection – design is sensitive to context
- There are a number of solutions that range from distinct to constructively complementary
 - ◆ E.g. Iterator, Enumeration Method, Batch Method, Collecting Parameter, Combined Iterator, Batch Iterator
 - ◆ The detail of realisation varies with environment

The Classic Iterator Pattern

- A conventional and common form of iteration over an encapsulated target
 - ◆ Separate the responsibility for iteration from that of collection into separate objects
 - ◆ Encapsulates position of traversal, but not traversal
- Interface details vary according to context
 - ◆ E.g. in C++ the STL provides a pointer-based perspective on Iterators
 - ◆ E.g. in Java and C# the *for*-each and *foreach* loops are integrated with the library Iterator model

The Enumeration Method Pattern

- The collection receives a Command, which it then applies to its elements
 - ◆ An inversion of the basic Iterator design
- Iteration is encapsulated within the collection
 - ◆ Therefore, the client is not responsible for loop housekeeping
 - ◆ Additional iteration-related policies, such as synchronisation or rollback, can be encapsulated
 - ◆ Often used in conjunction with Visitor

Enumeration Method in Context

- This pattern is applicable in any language that supports a form of callback
 - ♦ Which, as well as the usual approach to expressing Command objects, includes C function pointers, function objects in C++, delegates in C#, etc.
- But its sweet spot is in languages where some form of closures is syntactically simple
 - ♦ E.g. anonymous methods in C# and blocks in Smalltalk and Ruby... but not quite Java inner classes

A Note on Naming and Separation

- Enumeration Method is also known as Internal Iterator (and sometimes just Iterator)
 - ♦ Internal Iterator was used to describe the pattern as a variant of Iterator (in contrast to External Iterator)
- However, the pattern described is a separate pattern and not simply a variant of Iterator
 - ♦ They resolve slightly different but overlapping problems, where appropriateness is also defined by language context, and have completely different solution structures and consequences

Context Object

The skill of writing is to create a context in which other people can think.

Edwin Schlossberg

Matters of Context

- How can objects in different parts of a system gain access to common facilities?
 - ♦ E.g. logging, configuration, security credentials
- This is a problem of accessing context that in effect surrounds the objects in a system
 - ♦ For a solution, keep in mind the desirable goal of loose coupling, which supports extensibility, testability, comprehensibility, etc.
- What about Singleton?
 - ♦ For a solution, keep in mind the desirable goal of...

The Epicycle Problem

- What is the problem with Singleton?
 - ♦ It is normally used by coincidence, it introduces a centralised point of coupling, it complicates testing and it comes with various lifecycle problems
- What are the alternatives?
 - ♦ Focus on making essential object relationships explicit, i.e. program to an interface, not an instance
- What about the Monostate pattern?
 - ♦ This is also known as the Borg pattern, which tells you everything you need to know

The Context Object Pattern

- Pass execution context for a component (whether object or layer) as an object
 - ♦ Avoids tedium and instability of long argument lists
 - ♦ Avoids explicit or implicit global services, e.g. Singletons or other common (ab)uses of *static*
- The essence of this pattern can be characterised as "parameterize from above"
 - ♦ As opposed to "... from below", which introduces transitive dependencies across layers in a system

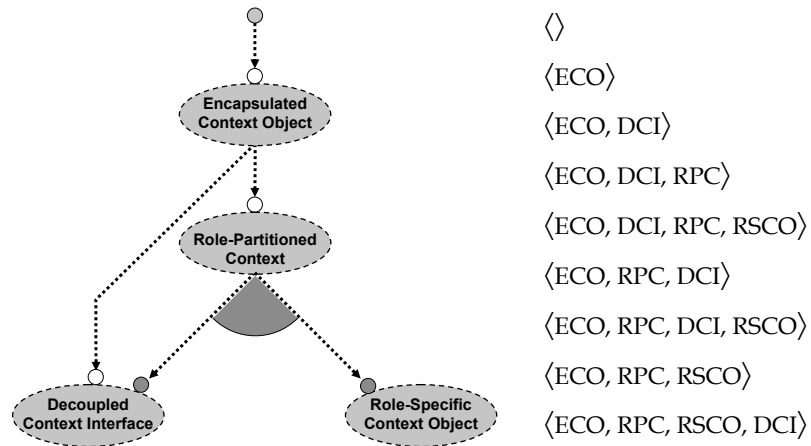
A Brief History of Context Object

- This pattern has been documented many times recently, but has a longer history
 - ♦ Can be seen in Scheme's *eval* procedure
 - ♦ Can be seen in the Interpreter pattern, which is best considered a pattern compound or the short pattern sequence ⟨Command, Context Object, Composite⟩
 - ♦ Documented as an individual pattern named Context Object and Encapsulated Context
 - ♦ Elaborated as the Context Encapsulation pattern language

The Context Encapsulation Language

- The Encapsulated Context Object pattern
 - ♦ Pass execution context for a component as an object
- The Decoupled Context Interface pattern
 - ♦ Decouple the usage of context from its concrete type
- The Role-Partitioned Context pattern
 - ♦ Subdivide a context interface to be cohesive with respect to usage
- The Role-Specific Context Object pattern
 - ♦ Support disjoint objects against context roles

Pattern Relationships and Sequences



ACCU, Silicon Valley, 2007-03-21

31

$\langle \text{ECO} \rangle$

```

public final class ExecutionContext
{
    public void writeLog(String message) ...
    public void writeConsole(String message) ...
    public boolean containsVariable(String name) ...
    public String valueOfVariable(String name) ...
    ...
}

public void configure(ExecutionContext context)
{
    String serverName = context.valueOfVariable("server");
    ...
}
public void start(ExecutionContext context)
{
    try ...
    catch(RuntimeException caught)
    {
        context.writeLog("Failed to start: " + caught);
        context.writeConsole("Error: " + caught);
        throw caught;
    }
}

```

ACCU, Silicon Valley, 2007-03-21

32

⟨ECO, DCI⟩

```
public interface ExecutionContext
{
    void writeLog(String message);
    void writeConsole(String message);
    boolean containsVariable(String name);
    String valueOfVariable(String name);
    ...
}
```

```
public class EnvironmentalContext implements ExecutionContext
{
    public void writeLog(String message) ...
    public void writeConsole(String message) ...
    ...
}
```

```
public class MockContext implements ExecutionContext
{
    public void writeLog(String message) ...
    public void writeConsole(String message) ...
    ...
}
```

⟨ECO, DCI, RPC⟩

```
public interface Reporting
{
    void writeLog(String message);
    void writeConsole(String message);
    ...
}
```

```
public interface Configuration
{
    boolean containsVariable(String name);
    String valueOfVariable(String name);
    ...
}
```

```
public class EnvironmentalContext implements Reporting, Configuration
{
    public void writeLog(String message) ...
    public void writeConsole(String message) ...
    public boolean containsVariable(String name) ...
    public String valueOfVariable(String name) ...
    ...
}
```

⟨ECO, DCI, RPC, RSCO⟩

```
public class NullReporting implements Reporting
{
    ...
}
public class FileBasedConfiguration implements Configuration
{
    ...
}

public void configure(Configuration config)
{
    String serverName = config.valueOfVariable("server");
    ...
}
public void start(Reporting reporter)
{
    try ...
    catch(RuntimeException caught)
    {
        reporter.writeLog("Failed to start: " + caught);
        reporter.writeConsole("Error: " + caught);
        throw caught;
    }
}
```

ACCU, Silicon Valley, 2007-03-21

35

In Closing

A witty saying proves nothing.

Voltaire

ACCU, Silicon Valley, 2007-03-21

36

Developer Summary

- There are some key patterns that are worth knowing in addition to the usual suspects
 - ◆ Manager reifies a responsibility that is often overlooked and mixed up in other code
 - ◆ Interceptor supports an orthogonal, plug-in style of extension
 - ◆ Enumeration Method is a familiar idiom to users of some languages, but is a novel approach for others
 - ◆ Context Object provides a means of capturing, controlling and manipulating execution context