Pattern Connections

Putting Together the Pieces of the Design Jigsaw

Kevlin Henney
kevlin@curbralan.com

Agenda

• Intent
  • Present a number of pattern concepts, going from lone patterns to a more connected view of patterns

• Content
  • Overview of Pattern Concepts
  • Some Examples
  • From a Pattern to a Language
Shameless Plug

Overview of Pattern Concepts

- Intent
  - Present core pattern terminology and ideas
- Content
  - Patterns and pattern quality
  - Patterns of misunderstanding
  - Pattern communities
  - Pattern stories and sequences
  - Pattern compounds
  - Pattern languages
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

Kinds of Patterns

- There are many kinds of patterns, not just OO-focused design patterns
  - Patterns for designing user interfaces
  - Patterns for programmer testing
  - Patterns for organisational structure and development process
- However, the focus of this talk is on patterns that relate to the design of code
  - These focus on artefacts visible to the programmer
Pattern Quality

- Contrary to popular belief, a pattern is not by definition "good"
  - There are also poor patterns — dysfunctional designs recur, through either habit or fashion
  - And there are also poor applications of good patterns
- A poor pattern or pattern application can be characterised as being out of balance
  - Its consequences and forces do not adequately match up

Patterns of Misunderstanding

- There are other misconceptions concerning the pattern concept that are worth clearing up...
  - *Design Patterns* is a limited subset of design patterns and the pattern concept
  - Patterns are not frameworks, components, blueprints or parameter-based collaborations
  - Patterns are more than just a sample class diagram of the solution
  - Only language-independent patterns are language independent: patterns may be language specific
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

Pattern Stories and Sequences

- A pattern story brings out the sequence of patterns applied in a given design example
  - They capture the conceptual narrative behind a given piece of design, real or illustrative
  - Forces and consequences are played out in order
- More generally, pattern sequences describe specific ordered applications of patterns
  - A pattern story is to a pattern sequence as a pattern example is to an individual pattern
Pattern Compounds

- Pattern compounds capture commonly recurring subcommunities of patterns
  - In truth, most patterns are compound, at one level or another or from one point of view or other
  - Also known as *compound patterns* or — originally and confusingly — *composite patterns*

- We can see many pattern compounds as named pattern subsequences
  - They are commonly recurring design fragments that can be further decomposed, if desired

Pattern Languages

- A pattern language connects many patterns together to capture a broader range of paths
  - The intent of a language is to generate a particular kind of system or subsystem
  - A pattern language can describe vernacular design style, with general patterns incorporated into a language that is presented more specifically

- There may be many possible and practical sequences through a pattern language
  - In the limit, a sequence is a narrow language
Some Examples

- Intent
  - Illustrate some of the concepts discussed with specific examples
- Content
  - Lone patterns
  - Complementary patterns
  - Pattern compounds and sequences
  - Pattern languages

Something for Nothing

- Where a non-null reference is interpreted as an option and a null as its absence...
  - Code may be littered with guard if statements
- Polymorphism can replace the explicit decision
Null Object

- The *Null Object* pattern is a tactical design based on substitution of pluggable parts
  - It generalises beyond object orientation, although it is often described in those terms

if
- An object reference may optionally be null and
- This reference must be checked before every use and
- The result of a null check is to do nothing or use a default value
then
- Provide a class subclasses from object reference's type and
- Implement all of its methods to do nothing or provide default results and
- Use an instance of this class when the object reference would have been null

Modal Object Lifecycles

- Many objects can be characterised as having groups of states (modes)
  - Each mode defines a set of behaviour that is significant and different to that of other modes
  - Objects transition from mode to mode in response to certain events
- There are many patterns that deal with the expression of the modes and the transitions
  - There is more to state than *State*
Objects for States

- Reflect a hierarchical view of the state model in a class hierarchy
  - A context object delegates to a behavioural object whose class represents a mode of behaviour

Implementing Objects for States

- There are many considerations, some of which are language specific
  - In Java, inner classes can be used to simplify access of the context object's fields
  - In C++, the whole state-behaviour hierarchy can be fully encapsulated using a Cheshire Cat
### Methods for States

- Methods for States represents each state as a table or record of method references
  - Methods referenced are on the target object

![Diagram of Methods for States](image)

### Implementing Methods for States

- This pattern is only suitable for languages that support simple manipulation of methods
  - E.g. member function pointers in C++, delegates in C# and use of `send` for Pluggable Selector in Ruby

```cpp
class context {
    public:
        void function();
        ... // other public functions
    private:
        ... // private functions
        struct mode {
            const mode *behaviour;
            ... // other private data
        };
};

struct context::mode {
    void (context::*function)();
    ... // other 'public' functions
    static const mode first_mode;
    ... // other modes
};
```
Collections for States

- For objects managed collectively, objects can be collected together according to state
  - State is extrinsically represented by membership

Implementing Collections for States

- There are different ways of organising the collections, depending on the situation
  - For $N$ modal states, at least $N$ collections are needed, e.g. a collection for each mode
  - But more than $N$ can sometimes be useful, e.g. a collection for all objects plus a collection for each mode
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

Iterator and Batch Method

- *Iterator* presents the common and conventional design of iteration over an encapsulated target
  - Separate the responsibility for iteration from that of collection into separate
- *Batch Method* is an alternative that addresses the needs of remote or otherwise costly access
  - The repetition is provided in data structure rather than in control flow
  - The granularity of access is coarser, which reduces one aspect of access overhead
**Batch Iterator as a Pattern Compound**

- *Batch Iterator* is a compound resulting from combining both *Iterator* and *Batch Method*
  - Offers a compromise in granularity and control, allowing a caller to step through a collection in strides greater than one step but less than the whole

```
typedef sequence<any> many;
interface BatchIterator
{
    boolean next_n(in unsigned long how_many, out many result);
    boolean skip_n(in unsigned long how_many);
};
```

**Batch Iterator as a Pattern Sequence**

- Another take on *Batch Iterator* is that it is the result of...
  - First, introducing an *Iterator*
  - Second, expressing its interface with a *Batch Method*
- In other words, a (very) short pattern sequence
  - This can be named as a proper noun, e.g. *Batch Iterator* or *Chunky Iterator*
  - Or labelled with respect to its parts and process, i.e. ⟨*Iterator*, *Batch Method⟩⟩
Value-Based Programming

- Values express simple informational concepts, such as quantities
  - In programming, values are expressed as objects, but their object identity is considered transparent, with state governing behaviour and use
- A number of idiomatic practices go together to support value-based programming in Java
  - The *Patterns of Value* language is a work in progress that aims to capture these

(A Part of) *Patterns of Value*
Generic Programming

• Generic programming is characterised by an open, orthogonal and expressive style
  • It is an approach to program composition that emphasises algorithmic abstraction, loose coupling and a strong separation of concerns
• The approach that underpins the STL
  • More than just coding with templates in C++ — this is a common misunderstanding: the principle of generic programming is not actually about generics
  • Originated with Alex Stepanov and others

STL Patterns

Encapsulated Algorithm  Container  Stream
Encapsulated Algorithm
Function Object
Counted Iteration Range
Iterator
Half-Open Iteration Range
Past-the-End Value
Iterator
Tagged Category
Tagged Overload
Tagged Overload
Pointer Protocol
Nested Type
Nested Trait
Pluggable Type
Pluggable Object
Deduction Helper
Pluggable Object
Trait
Trait
Helper
Holder
Traits
Traits
Base Class
Trait
Lookup
Template
Trait
Lookup
Template
Copy Value
Copy Value
Container
Encapsulated Algorithm
Adaptor
From a Pattern to a Language

- Intent
  - Present the Context Encapsulation pattern language, starting from its root
- Content
  - Encapsulated Context Object
  - Decoupled Context Interface
  - Role-Partitioned Context
  - Role-Specific Context Object

〉

- Consider the context of a loosely coupled and extensible architecture
  - The extensibility can be per runtime, per release or per product configuration
- How can objects in different parts of this system gain access to common facilities?
  - Keeping in mind the goal of loose coupling, which supports extensibility, comprehensibility, testability, etc.
**Encapsulated Context Object**

- Pass execution context for a component — whether a layer or a lone object — as an object
  - Avoids tedium and instability of long argument lists of individual configuration parameters
  - Avoids explicit or implicit global services, e.g. *Singletons, Monostates* and other uses of *static*
- The context may include external configuration information and services, such as logging
  - But features should not be included arbitrarily

```java
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;
    }
}
```
Decoupled Context Interface

- Reduce the coupling of a component to the concrete type of the Encapsulated Context Object
  - Define its dependency in terms of an interface rather than the underlying implementation class
- This allows substitution of alternative implementations
  - E.g. Null Objects and Mock Objects
  - Also decouples context dependent from any changes in a single implementing class

```java
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) ...
    ...
}
```
Role-Partitioned Context

- Split uncohesive Encapsulated Context Objects into smaller more cohesive context interfaces
  - It is all too easy to end up with a bucket of arbitrary variables that have no genuine relation to one another, either in concept or in use
- Base the partitioning on usage role, i.e. features that are used together should stay together
  - Each partitioned piece of context can be expressed with a Decoupled Context Interface, or through a Role-Specific Context Object, or both

```java
〈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) ...
    ...
}
```
Role-Specific Context Object

- Multiple Role-Partitioned Contexts may be expressed at runtime as a single object per role
  - This allows independent parts of a context to be more loosely coupled and separately parameterized
- The Role-Partitioned Context may also be expressed with Decoupled Context Interfaces
  - Which also allows the context to be contained in a single object, offering an additional degree of parameterization freedom

```java
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;
    }
}
```
Context Encapsulation

\[
\begin{align*}
&\{\} \\
&\{\text{ECO}\} \\
&\{\text{ECO, DCI}\} \\
&\{\text{ECO, DCI, RPC}\} \\
&\{\text{ECO, DCI, RPC, RSCO}\} \\
&\{\text{ECO, RPC, DCI}\} \\
&\{\text{ECO, RPC, DCI, RSCO}\} \\
&\{\text{ECO, RPC, RSCO}\} \\
&\{\text{ECO, RPC, RSCO, DCI}\}
\end{align*}
\]

In Conclusion

- A pattern captures recurrence, structure and intention in design
  - But beware: not all that recurs is necessarily good
- Patterns inevitably combine to address more intricate problems than lone patterns can
  - A pattern compound captures common groupings
  - A pattern sequence represents a gradual process of stable transformation from one design to another
  - A pattern language describes connections between patterns that can yield many different paths