Kevlin Henney / khenney@qatraining.com

Patterns of Value

WHAT IS AN object worth? For that matter, what is a pattern worth? The answer to both of these questions is: not much. At least, not on their own. Object-oriented (OO) systems work because the behavior and information of the system is distributed across a network of connected objects, each responsible for a particular part of the system's behavior and/or information, their granularity ranging from the large to the small. The references connecting objects together may be held for the lifetime of an object, or for less than the duration of a method call. In execution the threads of control ripple through and affect this network.

A similar thing can be said of patterns: Patterns do not exist in isolation only solving individual design problems at a single level. Patterns can be collected together for common use, but more powerfully they can be connected together as a pattern language to describe how to build a particular kind of system or resolve a family of related problems.

Putting Patterns in Their Place

The Gang of Four (GoF) catalog of design patterns[1] collects a number of general purpose patterns for use in OO design. However, it is certainly not the last word on either patterns or design. Design embraces many levels of detail in a system, from its gross architecture right down to the use of language features; design must also relate to the purpose as well as the mechanism of the system.

Pattern-Oriented Software Architecture (POSA)[2] is another catalog that is in many ways GoF-like. One way in which it goes further than GoF is in classifying its patterns as belonging to one of three levels: architecture, by which the gross architecture is meant; design, by which detailed design at the same level of GoF is intended; and idioms, which focus on programming language-specific patterns.

Kevlin Henney is a Principal Technologist with QA Training in the UK.

Idioms

As anyone who has studied a language at school only to be left—literally— speechless when visiting somewhere it is spoken natively will know, understanding of any language goes beyond a by rote knowledge of syntax and semantics. Fluency in a language is also about embracing its idioms and expressing yourself appropriately in that language with intention, rather than by accident or by dogma. This is as true of programming languages as it is of natural languages, and Java presents a context of mechanisms and common practices for design to incorporate.

Many idioms can be seen to define conventions of style, e.g., class and method naming conventions. Others have a more direct relationship to patterns. Where patterns are considered to be solutions to problems occurring in a context, many idioms are language-level or technology-specific patterns[3]; that is, they have the language or technology as part of their context.

Parts of Java's context that affects how design decisions are taken include its strong typing, reflection, support for multithreading, garbage collection, and reference-based objects. This context leads Java developers down different routes than those taken by either C++ or Smalltalk developers. For instance, much of the detail describing C++ issues such as memory management in *Design Patterns*[1] is not relevant in Java. At the same time, there are Java issues of interest that are not explored. So, some idioms may adapt general design patterns to fit in more appropriately with the language, as was done with the *Design Patterns Smalltalk Companion*,[4] which expresses and discusses the GoF patterns in a more idiomatic form for Smalltalk programmers.

Idioms may also represent design decisions that exist only in that language. Note that not all lan-

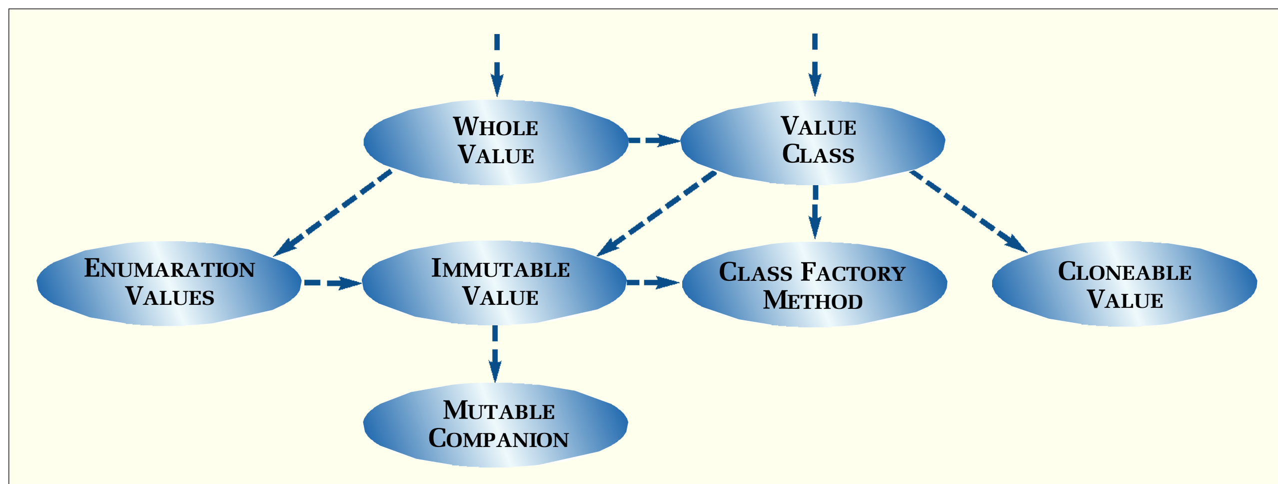

Figure 1. ***Patterns and their successors for supporting value-based programming in Java.***

guage-specific conventions warrant the name patterns. For instance, the JavaBeans naming conventions—`get*` and `set*`, etc.—are just that: naming conventions. They define framework participation rules, to support meaningful introspection, for non-`BeanInfo` implementers. Therefore, although idiomatic in one sense, they are not patterns in the sense we are talking about. When the authors of the JavaBeans specification called the naming convention design patterns, they confused regular expression pattern matching with the more specific concept of design patterns.

Java idioms are being documented in a number of places, including a growing body of tentative idioms on the Wiki.[5] In some cases the patterns have a more specific context than simply Java, for instance, dealing with concurrency.[6]

Pattern Languages

Patterns often have relationships with other patterns. The related patterns may be used to resolve problems in the new context introduced by applying a particular pattern, or patterns may be used to support the development of a particular solution. For instance, Iterator is often supported by the use of a Factory Method.[1]

Patterns can be grouped together and collected in a catalog to provide a useful knowledge source; a simple software engineering handbook if you like, e.g., GoF and POSA. There may be some documented relationships between the patterns in a catalog. However, the value of patterns is more fully realized when connecting them together in the narrative framework of a pattern language[3]:

A pattern language is a collection of patterns that build on each other to generate a system. A pattern in isolation solves an isolated design problem; a pattern language builds a system. It is through pattern languages that patterns achieve their fullest power.…A pattern language should not be confused with a programming language. A pattern language is a piece of literature that describes an architecture, a design, a framework, or other structure. It has structure, but not the same level of formal structure that one finds in programming languages.

A pattern language represents a reasonable set of practices and decisions that need to be taken together to resolve a particular design challenge. The relationships help the developer determine which patterns should be applied and under what circumstances. The developer works with a connected group of patterns rather than just individual patterns.

The idea of pattern languages originated with patterns in building architecture[7], but there are now many good examples for software, for instance, in the *Pattern Language of Program Design* books.[8–10] There is even a pattern language for writing pattern languages![11]

Value-Based Programming

What kind of problem in Java needs the concerted collaboration of many patterns? There are many, but to give you a simple and complete example that we can work through in this and the next column, let us consider the issue of expressing and using values in Java. Examples of value types include strings, integers, and intervals, as well as semantically richer dates, money, and physical quantities such as length, mass, and time. We often think of objects as representing the significant chunks of a system; values, in effect, form the currency between these chunks.

Identity, State, and Behavior

An object can be characterized by identity, state, and behavior.[12] Value objects have transparent identity, significant state, and behavior directly related to the state. Knowing the identity of an object means that you can hold a reference to it. By transparent identity we mean that a value object's identity is not important to the way we use it, and one value object is substitutable for another with the same state. An example of this is a string. Our

focus is on a string's content and its manipulation, and not on the reference itself: Comparison of the content of two strings is of interest, but comparison of their identity is less useful. In other words, common usage for String is based on its overridden equals method and not on the == operator.

Service-based objects are another example where identity is incidental. However, for service objects behavior and not state is the most important feature; often service objects are stateless. Contrast this with entity objects, for which both identity and state are significant.

Values in Java

Except for the built-in types, such as int, Java currently supports only reference-based objects. Interesting proposals for language extension aside,[13] it is not currently possible for developers to create their own types to follow the same behavior as the built-ins. For instance, there is no operator overloading in Java, except for the indirect relationship between the + operator and the toString method, and passing by copy is supported only in the context of remoting, specifically java.io.Serializable types under RMI.

Nonetheless, this does not remove the need for developers to create types that act as values. The idioms for supporting fine-grained value types can be described through a pattern language.

A Pattern Language

The pattern language for value-based programming in Java that follows is a work in progress. It is drawn from common Java practices as found in published code, including the standard Java libraries. A summary and basic structure of the language, followed by a simple example that demonstrates its use, is shown this time. In the next column we will examine each of the patterns in more detail.

Overview of the Patterns

Figure 1 shows all of the patterns in the language. Lines with arrows represent successor relationships, showing how one pattern may be followed by another to support it in some way; the detail of that support is found in the text of the pattern itself.

An Immutable Value, for example, avoids the side-effect problems that arise from sharing value objects between objects, particularly across threads. However, it can be costly and awkward, in terms of object creation, to only

Table 1. **Thumbnails for value-based programming patterns.**

Name	Problem	Solution
Class Factory Method	How can you simplify, and potentially optimize, construction of Value Class objects in expressions without resorting to intrusive new expressions?	Provide static methods to be used instead of (or as well as) ordinary constructors. The methods return either newly created Value Class objects or cached objects from a table.
Cloneable Value	How can you pass a Value Class object into and out of methods without allowing callers or called methods to affect the original object?	Implement the Cloneable interface for the Value Class and use a clone of the original whenever it needs to be passed.
Enumeration Values	How can you represent a fixed set of constant values and preserve type safety?	Each constant is represented by an Immutable Value defined as a static final in the scope of the Immutable Value class, which cannot be instantiated outside the scope of that class.
Immutable Value	How can you share Value Class objects and guarantee no side-effect problems?	Set the internal state of the Value Class object at construction, and allow no subsequent modifications i.e., implement only query methods.
Mutable Companion	How can you simplify complex construction of an Immutable Value? Immutable Value objects.	Implement a companion class that supports modifier methods and acts as a factory for
Value Class	How do you define a class to represent values in your system?	Override the methods in Object whose action should be related to content and not identity (e.g., equal), and implement Serializable. The Value Class will be either an Immutable Value or a Cloneable Value. Simplify construction with a Class Factory Method.
Whole Value	How can you represent a primitive domain quantity in your system without loss of meaning?	Express the type of the quantity as a Value Class.

work with Immutable Value objects, and so it is often helpful to provide a Mutable Companion class for the Immutable Value class. In the standard Java library, java.lang.StringBuffer is an Immutable Value and java.lang.StringBuffer is a Mutable Companion.

Table 1 summarizes each of the patterns alphabetically in what is commonly known as thumbnail form: The name, essential problem, and brief solution are presented without rationale or examples.

Putting the Patterns to Work

A simple example can be used to illustrate the pattern language in action. Consider the problem of representing dates in an object system. The resulting code is shown in Listing 1 (available online in the code section of **www.javareport.com**).

The Whole Value pattern,[14] also known as the Quantity pattern,[15] and Value Class pattern offer the entry points into the pattern language. We can already guess that the best way to represent dates in our systems is directly as objects, hence the need for a class Date. The way in which Date should be implemented is as a Value Class, which describes what is involved in making its instances value-like.

A Date object is considered to be an Immutable Value[16] to avoid problems arising from sharing a single Date object among other objects. For instance, two objects sharing a Date object expect that the value it represents should remain unchanged. However, if one of the objects modifies it, the other will also experience the change—a person object holding a date of birth field may unexpectedly find its birthday moved! To simplify manipulation of dates Mutable Companion , DateManipulator, is also provided. An alternative to this approach is to make Date a Cloneable Value.

One issue that needs to be addressed is what field order should be used to initialize Date objects: YYYY/MM/DD, DD/MM/YYYY, or MM/DD/YYYY? The joy of standards is that there are so many to choose from, but if we choose one how do we enforce that choice? If int is used to represent the year, the month, and the day, there is no type checking to catch incorrect use of the other cases, e.g., given the following constructor:

```
public class Date implements Serializable
{
  public Date(int year, int month, int day) ...
  ...
}
```

All of the following will compile:

```
Date right = new Date(year, month, day);
Date wrong = new Date(day, month, year);
Date alsoWrong = new Date(month, day, year);
```

Months can be conveniently represented as Enumeration Values, also known as Typesafe Constant,[17] which deals with expressing fixed sets of constants (think enum in C and C++). A year can be conveniently wrappered as a Whole Value, making it a distinct type and therefore checked by the type system. The Year class is intended for use as part of a method's interface rather than as part of an object's representation; it is at the interface that the type safety is really needed. Given that Month and Year are now checked, it is safe to leave the day in the month as a plain int, although you may wish to make it a Whole Value for consistency.

The Class Factory Method pattern generally supports the Value Class pattern, making it easier to express new objects in expressions, as in the case of Year. The Class Factory Method pattern is a more specific variant of the Factory Method[1] pattern: Factory Method deals specifically with managing object creation in a class hierarchy, Class Factory Method focuses on providing an alternative method of object creation to calling new with a constructor.

Conclusion

Patterns are gregarious: They like company, and can work well with other patterns to assist in design. The simple issue resolved here, that of value-based programming in Java, hopefully illustrates how a pattern language combines patterns to work through a problem and support a set of principles. Next time, we will look at each of the patterns in greater detail. ■

References

1. Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison–Wesley, 1995.
2. Buschmann, F., et al., *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.
3. Coplien, J., *Software Patterns*, SIGS, 1996.
4. Alpert, S., Brown, K., Woolf, B., *The Design Patterns Smalltalk Companion*, Addison–Wesley, 1998.
5. *Java Idioms*, **http://c2.com/cgi/wiki**
6. Lea, D., *Concurrent Programming in Java: Design Principles and Patterns*, Addison–Wesley, 1999.
7. Alexander, C., et al., *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.
8. Coplien, J. and D. Schmidt, Eds., *Pattern Languages of Program Design*, Addison–Wesley, 1995.
9. Vlissides, J., J. Coplien, and N. Kerth, Eds., *Pattern Languages of Program Design 2*, Addison–Wesley, 1996.
10. Martin, R., D. Riehle, and F. Buschmann, Eds., *Pattern Languages of Program Design 3*, Addison–Wesley, 1998.
11. Meszaros, G., Doble, J., "A Pattern Language for Writing Pattern Writing," *PLoPD1998*.
12. Booch, G., *Object-Oriented Analysis and Design with Applications, 2nd edition*, Benjamin/Cummings, 1994.
13. Gosling, J., "The Evolution of Numerical Computing in Java," **http://java.sun.com/people/jag/FP.html**
14. Cunningham, W., "The CHECKS Pattern Language of Information Integrity," *PLoPD1995*.
15. Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison–Wesley, 1997.
16. Henney, K., "Java Patterns and Implementations," presented at *BCS OOPS Patterns Day*, Oct. 1997, presentation notes and whitepaper, **techland.qatraining.com/profile.htm** .
17. Warren, N., Bishop, P., *Java in Practice: Design Styles and Idioms for Effective Java*, Addison–Wesley, 1999.