The Death of Patterns

Noel Welsh

June 20, 2003

1 What Are Patterns?

There are many definitions of what a pattern is. A nice general definition from [1] is:

> A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns.

For over a decade the patterns movement in software engineering has been seeking to discover and document these insights so that others can benefit from them. To this aim patterns are often presented in a formal style called a pattern language:

> A pattern language guides a designer by providing workable solutions to all of the problems known to arise in the course of design. It is a sequence of bits of knowledge written in a style and arranged in an order which leads a designer to ask (and answer) the right questions at the right time. [2]

Finally we should note that patterns are eminently practical. In [5], widely considered the first great book of the patterns movement, we find:

> [A pattern] also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.

There are two points we can extract from the above:

- Patterns are written documents. Code is the implementation or expression of patterns, but patterns are not code.
- Patterns aren't formal in the mathematical sense. They are artifacts of complex ill-defined design decisions and so are necessarily informal.

1.1 The Importance of Patterns

Many people on first encountering patterns look at them and think: "What's the big deal? I've been doing that for ages." There is a temptation to dismiss patterns as hype, just a way of selling knowledge already known to most developers. This is not the case. Naming a concept is incredibly powerful. It enables

[chunking]

The nature of software is to tackle more and more complex tasks. The nature of software engineering is to find better ways to handle that complexity. Patterns are a way of handling complexity and a very powerful way. They enable the software engineer to reason about their code at the higher-level of patterns rather than th

Abstractions = efficiency

1.2 Example Patterns

Our first pattern example is a well established pattern that originally appeared in [5] though our version is somewhat condensed and comes from [4]:

1.2.1 Pattern: Command

Intent Encapsulate a request as a parameterized object; allow different requests, queue or log requests and support undoable operations

Solution Client creates commands as needed, specifying the Receiver object and parameters the command will use to execute later.

Each SpecificCommand is inherited from a common abstract class which defines the basic execute interface Invoker uses to trigger commands when needed.

Each SpecificCommand knows the Receiver it works with and the parameters it needs to perform its encapsulated action by calling methods in its associated Receiver.

Consequences Decouples an object from the operations performed on it.

Related Patterns Commands may be assembled using Composite or Chain of Responsibility.

There are many uses for the Command pattern. For example, adding actions to be called from menu items in a GUI is a good place to use it. The implementation of this pattern in Java would typically require one interface or abstract class for the `Command` and implementing or subclassing classes for each `SpecificCommand`.

Our next example is due to Guy Steele, writing on the `ll1` mailing list [6]:

1.2.2 Pattern: If

Intent You want to do one of two things S and T depending on the value of some boolean predicate P.

Solution Code pattern:

```
   evaluate P
   branch if not(P) to L
   execute S
   branch unconditionally to E
L: execute T
E:
```

Consequences [fill me in]

What's going on here? This is clearly a joke: every programming language provides an `if` statement. In pattern terminology `if` is an idiom as the language supports it directly.

2 The Software Crisis

According to [7] productivity (in lines of code) in the software industry is increasing at about 9% a year or about 175% over a 20 year period. From [3] we see an order of magnitude increase in the lines of assembler to lines of source code every 20 years. So with a high level language and modern development practices we can produce about 20 times more assembler for the same effort than we could 20 years ago. In the same time we've seen hardware increase speed by about six orders of magnitude. There are two important points here:

- Programmer productivity isn't even close to keeping up with the hardware. Although there are some tasks that tax the modern computer most of the time they are idle. The figures suggest this need not be the case, but it appears we lack the ability to write code that can use the power now available in a desktop PC.

- Most of the increase in productivity comes from adopting more advanced languages. The increase in the number of lines of code due to better development practice is overwhelmed by the increase due to high level languages.

3 The Death of Design Patterns

Let's return to the `if` pattern above. We dimissed it as a joke, but its exactly the kind of productivity improvement we found to be so important. Nobody writes

```
   evaluate P
   branch if not(P) to L
   execute S
   branch unconditionally to E
L: execute T
E:
```

anymore. They just write whatever `if` statement their language provides and the compiler generates assembler very much like that above.

Languages grow by acquiring patterns: the patterns of structural programming have become our `if`, `while` and `for` loops. More recently object-oriented patterns have been absorbed in languages. Unfortunately language evolution is slow - it took 20 years for just some of Smalltalk's innovations to become mainstream. So we have the situation today when the cognitive tools we use to design our programs are two decades ahead of the tools we express that program in. Programmers are effectively compiling patterns into a form our programming languages can deal with.

This is a very strange situation to be in. We noted earlier the benefits to be gained by naming a unit of thought. Design patterns allow us to do this in our design but not in our programs, so we have to resort to comments or other conventions to retain this information. This is incredibly wasteful. In addition to the chance of manual error by having this discontinuity between thought and action we greatly harm our productivity. Using the rough heuristic that our languages are about 20 years behind our patterns we're missing out on around another order of magnitude improvement in productivity. Indeed figures bear this out: a study of a large project using the Erlang language [9] showed both productivity and quality increasing by a factor of between 4 and 10 compared to existing programs written in C++.

Alternatively we can see how advanced languages compare against mainstream languages in their ability to directly express patterns. One such comparison is [8], which compares the implementation of the 24 design patterns of [5] in a language with reflection, first-class functions, multiple dispatch and macros to their C++ equivalent. It shows that most of the 24 patterns are either subsumed by language features or made substantially simpler in a more advanced language. For example, in a language where functions are first-class, that is, where functions can passed as function parameters, assigned to variables and so forth, the Command pattern can simply be replaced by a function.

At the beginning of this document we made two claims about patterns:

- Patterns are written documents. We have seen this is not the case. Patterns are language features, or idioms, that a particular language happens not to support.

- Patterns aren't formal. This again is not the case. To return to the Command pattern example, the properties of first class functions have been extensively studied in the lambda calculus. One can find rigourous formal treatment for every feature: reflection, first-class functions, multiple dispatch and macros in the advanced language discussed above.

So patterns have failed us. The death of patterns is the birth of idioms.

4 A Reconciliation

We can't change languages every few years - the cost is too high. And do we expect to throw out over 10 years worth of patterns knowledge and put ourselves in the hands of language designers? Neither is a good option. Better is to have a malleable language - one that we can change to incorporate the patterns we find in our programs. Such a language is a language with macros.

4.1 Not Your Father's Macro System

Most programmers are familiar with C preprocessor macros. Thankfully modern macro systems are both more powerful and easier to use.

Reports of the death of patterns have been greatly exaggerated. Patterns still have to be identified before they can become language features and we can't go changing our language every year - there is simply too much investment in libraries for this to feasible. What we need is a malleable language.

Computational core.

Patterns et mort. Vive le patterns!

References

[1] Brad Appleton. Patterns and software: Essential concepts and terminology. http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html, Feb 2000.

[2] Kent Beck and Ward Cunningham. Using pattern languages for object-orientated programs. Technical report, Tektronix Inc., 1987.

[3] Barry Boehm. Software productivity perspectives, paradoxes, pitfalls and perspectiv. http://sunset.usc.edu/classes/cs577b_2000/EC/22/EC-22.pdf, 1999.

[4] David Van Camp. The object-orientated pattern digest. http://patterndigest.com/, 2002.

[5] Rich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Orientated Software.* Addison Wesley, 1994.

[6] Guy Steele Jr. if pattern. http://www.ai.mit.edu/ gregs/ll1-discuss-archive-html/msg01629.html, 2002.

[7] Doug Putman. You need the right map to know where you're going with process improvement. http://www.qsm.com/process_02.html.

[8] Gregory T. Sullivan. Advanced programming language features for executable design patterns "better patterns through reflection". Technical report, Massachusetts Institute of Technology, 2002. http://www.ai.mit.edu/ gregs/.

[9] Ulf Wiger. Four-fold increase in productivity and quality. In *Workshop on Formal Design of Safety Critical Embedded Systems.*, 2001. http://www.erlang.se/publications/Ulf_Wiger.pdf.