Modelling and Software Development

> *pattern n. .... **2**. a model or design or instructions according to which something is to be made ...**5.** a regular form or order in which a series of actions or qualities etc. occur....*
>
> *pattern v. **1**. to model according to a pattern*[i]

Anyone who has studied human geography has probably come across the work of Walter Christaller. In 1933 he proposed a model to describe the settlements patterns in southern Germany – see side box. While this model shed some light on where settlements had occurred nobody really expected it to actually give the location of settlements; it described settlement patterns an abstracted way.

Models, by their nature make assumptions to simplify things: when Airfix produced their model F1-11 – the company attempted to show you what a particular jet bomber looked like but they never claimed it would fly (OK, I'm sure many I was not the only boy who tried to fly the odd Airfix model out the bedroom window!)

From a programming perspective we are concerned with information models: this places us closer to the models of Christaller than Airfix.

If at this point you wonder what all this has to do with software let me spell it out: when we create computer systems we are creating models. Sometimes these are obvious: I once worked in a department modelling the electricity market, these predictions where used directly by management to decide which electricity contracts to sign; sometimes these models are less obvious: a customer relationship management (CRM) system models the expected interactions between company and customer, when the model fails we find post-it notes on peoples terminals "If Jack Smith phones transfers him to Jo."

Our models have boundaries: within these conditions and scenarios are dealt with. Some boundaries are explicit, some are tacit. Economists are familiar with these boundaries: every economics model comes with the "all other things being equal" pre-condition. Their models will attempt to describe activity provided every other boundary condition remains unchanged. In many cases these models make sweeping generalisations. The monetarist model (see side box) attempts to predict inflation in a closed economy. In itself it is useless because it is so general, but it does allow economists to reason about an economy. It also forms the starting point for sophisticated models used by make economic predictions.

What are the models we build?

In software development we build many different kinds of model, on any project the different models look at the problem domain from different perspectives.

Metaphores are small models

Drawing an analogy by metaphor is setting up a small, quick model: we rely on the fact someone already knows the metaphor, so when Kevlin compares software development to gardening, he is relying on the fact that most of use have an idea what gardening entails.

Metaphores are used in one of two modes:

- Educators use a metaphor to describe a new concept to us in terms of a known one
- We use it to reason about a system: because X is like Y in one respect, is it similar in other respects?

These are the same thing at different points. We tie our subject (e.g. software engineering) to a target (e.g. gardening) by pointing out a similarity, then we can follow this up with a less obvious similarity. In doing this we leverage knowledge in one area to increase knowledge in another. Think of it as proof by induction.

Specifications

In traditional development a business analyst writes a specification document that is then implemented. Some organisations still work this way but many companies make do with a vague statement of intent: "We intend to build an equities trading system" – this relies on the developers understanding of what features an equities trading system should contain, the developers have their own mental model of what the system needs to do. (This explains why banks prefer to hire people with experience in the financial markets and why these people can command premium wages.)

Weather a large document, in developers heads or on many individual CRC cards the specification is a model of the problem. Yet is it usually incomplete, and frequently inconsistent with itself.

Design

While I hope all systems have a design I believe most use the Topsy Design Pattern: they just grow.

A design is high level representation of the source code as such it is indisputably a model. It is a model of the solution, not the problem.

Like any model, it makes assumptions and abstractions. It is important that all working on the system understand the model: if I think we are building an Airfix Lancaster bomber and you think we building a B17 we may well get something that looks like a four engined World War II bomber but it will be neither one thing or another. Auntie Dotty may think it is a good model but her terms of reference are different to ours.

Source code

Our ultimate model of the solution: the point where we start to discover the inconstancies and holes in the specifications.

I once worked on a train time tabling system. The specification was long and inevitably contained omissions and errors. These where fixable, even when they occurred late in the development cycle we could add new rules and change existing ones. The most difficult problems occurred when rules conflicted each other, usually this wasn't obvious until the source code was examined and we found that a fix for requirement A had introduced a bug, requirement A was quite respectable, but nobody foresaw that when

implemented the result contradicted requirement B. Only when the specification was codified in the pure logic of code was this clear that neither A nor B represented the true requirement.
Our voluminous specification model was neither complete nor self-consistent and much was still locked in people heads.

Other models

Specification, design and code may be the first models that spring to mind but there are other models in software development:

- Test suites: test results part way between problem and solution; they attempt to apply the solution to the problem.
- Process models: We defined models for how we develop software, our processes. SSADM, Extreme Programming, waterfall, and such are models of process. Those who read my article on Extreme Programming[ii] will notice this as one of my criticisms: Beck sets out a model called *Extreme Programming* (XP), then, he says: you cannot modify this model, if you do so it is no longer XP. I can't accept this, XP is a process model, no team will ever have the exact conditions of the C3 team, it must be adapted for each case.
- Delivery schedules: need I say more?

What are our tools?

When we use CASE tools like Rational Rose our modelling is obvious, but even when we write C++, Java and Pascal we are codifying our logic in a language model. The languages and machines which run our models are all Turning equivalent so no computer or language is really more powerful than another, but each brings different techniques for modelling the problem, for thinking about the problem, and this is where their power lies.
Object oriented Java is not more powerful than procedural Pascal because it runs faster; it is more powerful because it allows us to think, to model, in different concepts.
Beyond language we have notations: when I draw a UML chart you know that a rectangle means one thing and circle means another. Actually, I will take exceptions with my own argument here: I think many of our notations, especially UML, rely too much on subtleties, a folded corner on a rectangle means it is different to a regular rectangle, while a dotted line is different to an solid line. I think we often try and put too much information into our notations.
I found the following story in *Software Fundamentals:* "if the presenter showed a block diagram, Dave [Parnas] would ask about the semantics – the meaning of different block shapes ... the meaning of an arrow; whether an unfilled arrow meant something different than a solid, filled arrow... Usually these frills had no meaning. They certainly didn't aid careful analysis, and they often got in the way[iii]."
In recent years the patterns community has moved to define more labels for more models. In the case of the GoF[iv] book they defined a meta-model that could be used to describe all their models. Now when I say

Singleton, *Chain-of-responsibility* or *Mediator* you know what I mean – OK, I deliberately included Mediator because it is not so well known and this is one of the problems the patterns community faces; it has been so successful in defining patterns that, outside of a core half dozen, few are widely known.

Why are our models inexact?

Like Christaller we can never expect our models to exactly describe a situation – by their nature they are abstractions. The simplification we make to create the model and generalise it return in real life. This brings us to the realm of Chaos theory, a small variation can, over time, when repeated, magnify into significant difference.

We also face the problem of Catastrophe theory - when multiple parameters are varied things start to break down. And as if this weren't bad enough we also have to face the law of diminishing returns[v].

The key with a model is identifying variability[vi] however, we frequently miss points of variability and need to adjust our models accordingly – but add too many points of variability, too many *if*'s and *but*'s and instead of a model we have just a list of special cases.

The more parameters a model has the less useful it is. There is no model of the game of soccer because there are too many parameters which can effect the game, we can make generalisations: Manchester United usually win, Everton usually lose[vii].

We should not expect our models to be exact, nor should we expect to follow them blindly. Sometimes it just doesn't make sense. Yes, we would like our singletons to be nicely destroyed at the end of the program, but what does it matter if the OS model will clean things up? Sometimes the work involved is not justified: consider the GPS system in a cruise missile, what does a memory leak matter in the final few milli-seconds before impact? Attempting to have a GPS-singleton delete itself neatly is more work for the programmers and adds more variability to a system at the exact moment when it must be totally predictable.

And not only with code and patterns: the process models of Yourdon, Jackson and Beck are really just more Christaller models. Yes, we can learn from them, we can compare ourselves to them, but to attempt to adopt them as laid out by the authors is about as sensible as ignoring the a mountain range when building your village!

Summary: Just what does a model give us?

Models provide us with many benefits:

- They give an idea or concept a label
- A label allows us to communicate more efficiently
- They allow us to compare different concepts
- They qualify the main characteristics of an idea

However, they come with drawbacks:

- No model will ever exactly describe an idea: if it does it is not a model
- Because they selectively hide elements that can be manipulated to the advantage of the modeller

- Applied incorrectly they can be wasteful on resources: you can't apply Christaller to mountainous regions; to do so would simply waste your time.

The value of a model lies in the abstractions it makes: by focusing on what are the important elements we are not distracted by the irrelevant ones. This is a classic definition of software abstraction but it also brings us back to Christaller: on the German plains Christaller accurately isolated the abstractions which describe settlement patterns. But, you would never expect to find Christaller's settlement patterns exactly because there are things like rivers, hills, particularly fertile land areas and such. Equally you should not expect your software model, your pattern, to describe exactly your software.

Conclusion

Models are an essential way of abstracting problems and patterns. Using models we can codify and communicate ideas – this in turn allows us to learn and to share ideas. However, as other disciplines know, models have limits, we should not expect our idealised models to be used straight out the box. Every model, whether it is a design pattern, a process or a programming style must be adapted to our present circumstances.

---

[i] Oxford paperback dictionary, 1983.

[ii] Overload 37, May 2000.

[iii] Introduction to *Abstract Types Defined as Class Variables*, Software Fundamentals: Collected Papers of David L. Parnas, edited by Hoffman and Weiss, Addison-Wesley 2001.

[iv] Design Patterns, Gamma et al, Addison-Wesley, 1995.

[v] See any good economics text book for a description of diminishing returns. Nor do I give references for Chaos theory, Catastrophe theory, Christaller or Monetarism – likewise these can be found in good mathematics, human-geography and economics text books. Google searches provide lots of sources on all.

[vi] See Coplien, Multi-Paradigm design in C++, Addison-Wesley, 1998 for discussions on commonality and variability analysis.

[vii] This will change next season, I'm sure.