Model-driven development

Perdita Stevens

School of Informatics
University of Edinburgh

Plan

- Recap of ways of using models
- Modelling slow- and fast-moving designs
- Model-driven development
- Current state of MDD

Ways of using models

At the beginning we said: UML use varies across projects and organisations, e.g.

- people scrawl UML diagrams on napkins and whiteboards – ephemeral models
- UML diagrams appear in documents (sometimes after the code has been written) – models as static documentation
- UML diagrams are developed in tools before the code, and code is generated from/in parallel with them – model-driven development (MDD)

We claimed: "Good modelling, design and testing should let you change the software quickly and without breaking it, when things change." True?

What kind of modelling?

Currently the way in which modelling can sensibly be used is strongly influenced by how fast the design is expected to change.

We'll consider separately

- modelling for slowly changing design
- modelling for fast changing design.

In future, we hope, MDD may work for both – currently works in medium-fast changing design niches!

Slowly-changing design

The waterfall idea – requirements, analysis, design, implement, test – is generally fiction (and, NB, regarded as such by W. Royce who coined the term "waterfall"!)

However, rarely, "big design up front" (BDUF) really is necessary:

- when implementation is not easy to change, e.g., involves building hardware
- when very costly verification processes need to be done – safety-critical software.

Then we typically need detailed, carefully checked, tool-supported modelling.

Fast-changing design

If change to the software is frequent, and done at code and test level, modelling has to be as quick and easy as possible.

Hence agile modelling usually done on whiteboards or scrap paper.

Do *not* want to spend hours with a UML tool or, worse, drawing tool to update a beautiful model of the current design if this will change again tomorrow anyway.

This is why organisations that mandate UML diagrams in design documents, but also need designs to change, often get models written after the code is complete!

Vision of future: MDD as easy as modelling on a whiteboard, *and* bringing extra benefits e.g. most code generated/updated automatically.

Modelling, design and testing

support each other:

- good modelling lets you pick a design that will work
- good testing helps you refactor a design when necessary
- good design lets you test effectively (via well-chosen APIs; also recall use of Factory Method to help test legacy code)

Key idea of agile development (though not unique to it): simple design is easier to change and less error prone.

As simple as possible, but no simpler... what counts as simple will depend what functionality has to be provided, which will change.

Modelling helps you find a simple design, testing helps you get to it.

Testing

Tests are important anyway, but *essential* for changing software quickly and without breaking it.

Recall refactoring approach. When you need to make a change (add functionality, fix bug):

1. Refactor the system into the state you wish you were starting from (now you are!)
2. Make the change.

1. The refactoring step

In small increments – your aim is to make the change steps just small enough that you never make a mistake, so all these tests always pass (ha!):

1. Run (at least the relevant) existing tests – presumably they pass, but this is a good sanity check
2. Do a refactoring step (recall, a small redesign step, not altering the functionality: e.g., eliminate some code duplication)
3. Rerun the tests to check they still pass.

Repeat until the codebase is how you want it to be, i.e., ideally designed to make the change you're about to make easy.

2. The change step

Now that your codebase is in good shape, with a clean simple, tested design that will support your change easily:

1. Write new tests that should eventually pass, but will currently fail (i.e. tests that demonstrate the bug you're about to fix, or that the new functionality you're about to add should pass)
2. Make your change
3. Rerun all the (relevant) tests, new and old.

It's highly likely that some test (new or old!) will fail, but because the design was so clean, it's easy to fix.

Summarising so far...

If models are not automatically kept in sync with code, the work involved in changing a model duplicates the work in changing the code.

So either:

- minimise number of changes to the model (BDUF, or write the model afterwards); or
- minimise the need to update models when things change (ephemeral modelling, or none at all)

Both have severe disadvantages.

What if we could change code and model together, for no more cost than the current cost of changing code? This, simplifying wildly, is the idea of MDD.

Model-driven development

Means different things to different people, but roughly:

- treat models as important, first-class artefacts in development
- large development may include many models, each adapted to the needs of its users (UML design model, database model, architecture model; can also regard e.g. code and documentation as models)
- use tools to avoid duplicating work, so
- decisions recorded in one model can be automatically rolled through to any other models, including code, using
- model transformations.

Model transformations

A model transformation is a program that can create or modify a model, typically using information from one or more other models. E.g.

- code generator:
  – input a UML model, output skeleton Java code.
- documentation generator, e.g. JavaDoc:
  – input a UML model, or Java code, or whatever, and generate pretty documentation.
- A more sophisticated task is roundtrip engineering:
  – input a UML model and some Java code; change them to be consistent.
  Here be dragons! Tools that do this exist, but they tend to be fragile and unpredictable. Active research area.

Current state of MDD

Some success stories (see e.g. `http://www.omg.org/mda/products_success.htm`), but not as widespread as early advocates (especially OMG) expected.

Many many reasons... some will change over time.

In your development career (if any) keep an ear to the ground and an open, if sceptical, mind.

Domain-specific modelling languages

Topic fits here because defining a modelling language specific to your domain – much simpler than UML – is one way to make model transformation easier: many MDD success stories involve DSMLs.

Has other benefits too: can define exactly the modelling concepts you need, getting simpler, clearer models.

MOF, the language in which UML is defined, is one way to define DSMLs.

Some tooling exists (see e.g. `http://www.eclipse.org/modeling/`; not very mature.