Forgetting UML

A Useful Guide to Formal Modeling

“The great thing about standards is that there are so many to choose from.”
—Andrew S. Tanenbaum
In The Beginning…

There were cocktail napkins.

\[ \nabla \cdot D = \rho \]
\[ \nabla \cdot B = 0 \]
\[ \nabla \times E = -\frac{\partial B}{\partial t} \]
\[ \nabla \times H = J + \frac{\partial D}{\partial t} \]

(*call sheng after work*).
And…

- Informalities are good, for
  - Ideas
  - Art

- Formalities are good, for
  - Software
  - Laws
  - Introductions

- Informalities are bad, for
  - Software
  - Bank account mgmt

- Formalities are bad, for
  - Jazz music
  - Ad lib
But…

- For the longest time, practiced software was in the informal category when it came to implementation
  - Terse/fast code was an art, not a science
  - Large codebases came together in an ad hoc manner
- How can you find out what the software does?
  - Look at the documentation (?)
  - Look at the code (?)
  - Look at what the code does (?)
And then there were 3…Amigos

- Ivar Jacobson
  - Developer and proponent of OOSE (Object-Oriented Software Engineering)
- James Rumbaugh
  - The Object Modeling Technique (OMT)
- Grady Booch
  - Color commentator, all-around funnyman, and founder of aptly named “Booch Method”

All for one!

And one for all!

And 3 for $1
Creating formalisms

- The 3 Amigos agreed that some uniform method should exist for creating software
- **Goal:**
  - a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems
- **Problem:**
  - should we use fluffy clouds to represent objects, or boxes??

Boxes!

Boxes with curved corners!!

Clouds!!!
Long story short:

- Compromises were made, committees made animals, hands were shaken, and UML was born (c. 1997)

- The Unified Modeling Language
  - (not to be confused with Uniform, Universal, or Unicorn modeling languages)

- And then, there was the bandwagon
  - Dozens of mature software tools,
  - Hundreds of companies,
  - Thousands of consultants
  - Helping businesses do UML
Solution through diagrams

- In 1997, UML solves some, but not all problems
  - Use case diagram
  - Class diagram
  - Behavior diagrams:
    - Statechart diagram
    - Activity diagram
    - Interaction diagrams:
      - Sequence diagram
      - Collaboration diagram
  - Implementation diagrams:
    - Component diagram
    - Deployment diagram

Interfaces and ‘so what’ definition(s)
Structural definition(s)
Behavioral definition(s)
High-level layouts and interaction definition(s)
What they really mean to you

- **Use case diagram**
- **Class diagram**
- **Behavior diagrams:**
  - Statechart diagram
  - Activity diagram
  - Interaction diagrams:
    - Sequence diagram
    - Collaboration diagram
- **Implementation diagrams:**
  - Component diagram
  - Deployment diagram

Pre-design...

Static architecture design

"Mostly" documenting

Show it to your boss...
Formalities

The myth that some people come away believing, when exposed to the notion of a formal language, is that a “formal language” is a formal-looking language; that any language that contains lots of Greek letters and mathematical symbols is formal.

— David Harel, Bernard Rumpe, “Syntax, Semantics, and all that Stuff”
Question:

- Is UML a formal language?
Answer (1997):

- Which part of UML?
  - Class diagrams
  - Sequence diagrams
  - Use-case diagrams

“Turns out your friend here is just mostly dead.”
— The Princess Bride
Real problem (for UML):

- Almost everyone who was around in the late 1990s learned something about UML, and what they learned was one of the following things:
  - On a whole, UML is semi-formal (at best)
  - Piece by piece, the strongest part of UML was class diagrams, which tells you nothing about behavior or constraints
  - None of the UML tools did a really good job at helping you solve the real problems of software engineering
    - Design changes
    - Constraint guarantees
    - Cross-cutting concerns (e.g., security or speed)
So, people lost the faith

• “Better IDEs are what we really need…”

• “Support for software aspects would solve the problem…”

• “A new language with ‘$’ in it would get people motivated to standardize… no wait, make that a ‘#’…”

• “There will never ever be a single standard modeling/programming language…”
What truths are there?

- “Better IDEs are what we really need…”
  - What do you do with the IDE once you have it?
- “Support for software aspects would solve the problem…”
  - How do you identify the aspects for a project or system?
- “A new language with ‘$’ in it would get people motivated to standardize… no wait, make that a ‘#’…”
  - What if this new language makes existing programs impossible to program, or infeasible to implement?
- “There will never ever be a single standard modeling/programming language…”
  - Now you may be on to something there…
Standard of non-standards

• **One language will never be expressive enough**
  
  – So, many languages will be required

• **Ideas are easy to express informally**
  
  – Is there a way to formalize these ideas, without making them seem formal?

• **The software is not the problem, it’s the systems that the software describes**
  
  – Can we define the systems, without defining the software?
Standard of non-standards

- Software is getting bigger and bigger, so we need to have IDEs that can scale up
  - Can we make the same intuitive leap as from assembly to high-level, to go from high-level to system software?
- Systems are too complex to build without examining cross-cutting concerns
  - Can we build the systems abstractly, and extrude the software from the system models?
- The intersection of the set of programmers and set of domain experts is ridiculously small
  - Can we give domain experts programming tools that are disguised as cocktail napkins?
What did UML teach us?

- “If you standardize it, they will come”
  - Find a niche, define it, and create a standard for it
- Problem:
  - Every niche is different 😊
- Solution:
  - Lots of standards
- Problem:
  - It is a whole lot of work to go through a standardization process. 😊
What did we learn:

• If you spend all of your time making standards, you’ll never have time to write programs

• How can you do it better?
  – Don’t model code, model systems
  – Model a system by modeling its family of systems, and specializing that model
Framework:
Modelling domain vs. modeling code

- **Domain Idea**
  - Solve problem in domain terms
  - Map to UML
  - Domain Model
  - No map!

- **Domain Model**
  - Map to UML
  - UML Model
  - Generate, Add bodies
  - Code
  - Map to code, implement

- **Code**
  - Map to code, implement
  - Assembler

- **Finished Product**
  - Map to lower level

Slide by Jeff Gray, Juha-Pekka Tolvanen and Matti Rossi

2004/09/14
Why Model Domains?

The myth that some people come away believing, when exposed to the notion of a formal language, is that a “formal language” is a formal-looking language; that any language that contains lots of Greek letters and mathematical symbols is formal.

— David Harel, Bernard Rumpe, “Syntax, Semantics, and all that Stuff”
Why Model Domains?

- Domain modeling can be
  - Formal
  - Intuitive
  - Useful

Do you know what the funny thing is about domain modeling? It’s the little differences.
Example(s)

• **PowerPoint**
  - Domain: Visual Presentation

• **Excel**
  - Domain: Accounting/number crunching

• **MATLAB**
  - Domain: Discrete systems

• **LaTeX**
  - Domain: Typesetting (sub: Academic papers, books, posters…)

**Problems:**

- How long does it take to create one of these environments???
- What happens if you try to use one of these environments for something it was not intended???
- What about creating domains for non-traditional systems???
Creating Domain-Specific Modeling Environments (DSMEs)

- A working application for system design
- A customized modeling environment which is a restricted input layer that enforces some meaning
- An implementation reflecting a domain’s familiar and consistent
  - methodologies
  - notation
  - semantics
- An efficient user interface
Working

- Remember these requirements types:
  - Functional - detail how the system should behave
  - Performance - concerned with the speed of operation when performing functional requirements
  - Reliability - deal with the overall robustness of the system, and frequently with safety or dependability issues

- All must be taken into account to capture a working system

- In order to work, some output artifact should be produced
  - Don’t forget this during DSME design
Customizing

• Restricting the input set of a language to be composed solely of domain-specific types
  - The metamodel of the system should reflect all types of inputs possibly made to any system (i.e., abstract syntax)

• The language should be versatile enough to be used on more than one system, but not so low-level that it is counter-productive to learn the syntax
  - UML behavior diagrams, ‘nuff said
Reflection

• The current state of the art should be reflected in the DSME
  - What methodologies are used in the creation of systems should be reflected in the layout of the tool
  - Whatever notation is useful for explanation should be useful for specification
  - The semantics of the objects used in the language should be clear to a domain expert, i.e., they should be able to make intelligent guesses about meaning without a manual
Efficiency

• Don’t make it “visual” programming, make it domain-specific programming
  – Common semantics should be hidden from modeler
  – Domain concepts should be mapped so that they efficiently (and “prettily”) represent the notation

• Use the modeling concept that best suits the domain concept
How to build these?

Metaprogramming Interface

- Formal Specifications

Meta-Level Translation

Environment Evolution

Application Evolution

MIPS Environment

- Model Builder
- Models
- Model Interpreters

Application Domain

- App. 1
- App. 2
- App. 3

Model Interpretation

Application Evolution

Evolution

Evolution
How to build these?

- Class diagrams can be used to generate the abstract syntax of a domain-specific language
- Constraints can be added to assert certain “correctness” rules on models that are built
- Generic editors can be used to create models, while restricting inputs to the domain
- This research is heavily done at Vanderbilt University (not surprisingly, involved in Chess...)

Jonathan Sprinkle, UC Berkeley
UML (2004)

- Finally realized that one-size-does-not-fit-all
  - The new UML has what is called the Model-Driven Architecture
    - Create abstract models, and refine them according to their domains
    - Generate code through transformations from model types to model types
- Have re-cast Case/Class/Etc. diagrams in the correct light as pieces which are used in their own way, but not to be misused
How does this apply to Chess?

- Embedded systems are hard to build
  - Cross-cutting concerns
- Hybrid systems are hard to define in different idioms
  - Generate implementations from central representation
- Software is hard for domain experts to create
  - Abstract software into system building blocks, take advantage of polymorphous behavior of blocks in different contexts
- Seemingly small changes in architectures can result in rippling effects
  - Anticipate effects through automation, and simulation, by having a model of the system in existence
Conclusions

• Formal modeling is hard, but not impossible, to do
  - Don’t forget that some graphical models are formal too

• UML has given all kinds of models a bad rap
  - By thinking outside the box/cloud, it is possible to take UML’s lead by rapidly creating domain-specific environments, but without worrying about the standardization process

• Future Chess seminars will begin with actual systems built and languages used with this mindset
Questions?

“Well HAL, I’m damned if I can find anything wrong with it.”

“Yes. It’s puzzling. I don’t think I’ve ever seen anything quite like this before.”

-- 2001: A Space Odyssey