Avoiding the Big Ball of Mud

Mario Kröplin

Big Ball of Mud

Brian Foote and Joseph W. Yoder
Fourth Conference on Patterns Languages of Programs
(PLoP '97/EuroPLoP '97)

http://www.laputan.org/mud/mud.html

“the de-facto standard software architecture”

Big Ball of Mud

A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle. We’ve all seen them. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.

BIG BALL OF MUD

alias SHANTYTOWN, SPAGHETTI CODE

You need to deliver quality software on time, and under budget.

Therefore, focus first on features and functionality,
then focus on architecture and performance.

Inscrutable code might, in fact, have a survival advantage over good code,
by virtue of being difficult to comprehend and change.

THROWAWAY CODE

alias QUICK HACK, KLEENEX CODE, DISPOSABLE CODE, SCRIPTING, KILLER DEMO, PERMANENT PROTOTYPE, BOOMTOWN

You need an immediate fix for a small problem,
or a quick prototype or proof of concept.

Therefore, produce, by any means available, simple, expedient,
disposable code that adequately addresses just the problem at-hand.

The real problem with THROWAWAY CODE comes when it isn't thrown away.

PIECEMEAL GROWTH

alias URBAN SPRAWL, ITERATIVE-INCREMENTAL DEVELOPMENT

Master plans are often rigid, misguided and out of date.
Users' needs change with time.

Therefore, incrementally address forces that encourage change and growth.
Allow opportunities for growth to be exploited locally, as they occur.
Refactor unrelentingly.

How Buildings Learn: "Maintenance is learning"

KEEP IT WORKING

alias VITALITY, BABY STEPS, DAILY BUILD, “FIRST, DO NO HARM”

Maintenance needs have accumulated, but an overhaul is unwise, since you might break the system.

Therefore, do what it takes to maintain the software and keep it going. Keep it working.

Aggressive unit and integration testing can help to guarantee that this goal is met.

SHEARING LAYERS

alias ?

Different artifacts change at different rates.

Therefore, factor your system so that artifacts that change at similar rates are together.

“Our basic argument is that there isn't any such thing as a building.
A building properly conceived is several layers of longevity of built components.”

SWEEPING IT UNDER THE RUG

alias POTEMKIN VILLAGE, HOUSECLEANING, PRETTY FACE, QUARANTINE, HIDING IT UNDER THE BED, REHABILITATION

Overgrown, tangled, haphazard spaghetti code is hard to comprehend, repair, or extend, and tends to grow even worse if it is not somehow brought under control.

Therefore, if you can’t easily make a mess go away, at least cordon it off.
This restricts the disorder to a fixed area, keeps it out of sight,
and can set the stage for additional refactoring.

RECONSTRUCTION

alias TOTAL REWRITE, DEMOLITION, THROWAWAY THE FIRST ONE, START OVER

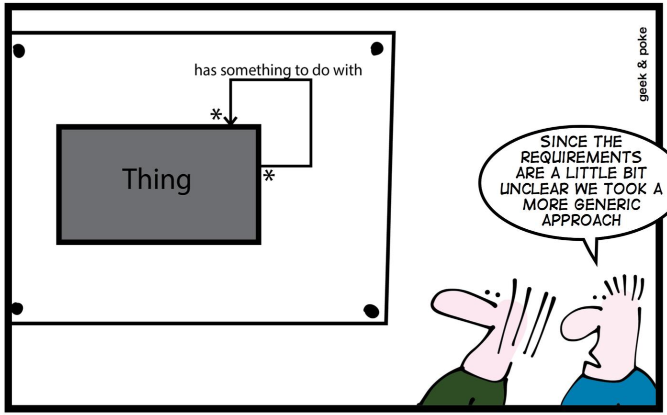
Your code has declined to the point where it is beyond repair, or even comprehension.

Therefore, throw it away and start over.

Software is often treated as an asset by accountants, and can be an expensive asset at that. Rewriting a system, of course, does not discard its conceptual design, or its staff’s experience.

Forces

- **Time** - architecture is a long-term concern
- **Cost** - good architecture is expensive
- **Experience** - bringing fresh blood aboard
- **Skill** (expertise, predisposition and temperament)
  “on average, average organizations will have average people”
- **Visibility** - who cares what it looks like on the inside?
- **Complexity** - the software is ugly because the problem is ugly
- **Change** - real world requirements are inevitably moving targets
- **Scale** - “good ideas don't always scale”


has something to do with
*
*
Thing
SINCE THE REQUIREMENTS ARE A LITTLE BIT UNCLEAR WE TOOK A MORE GENERIC APPROACH
geek & poke

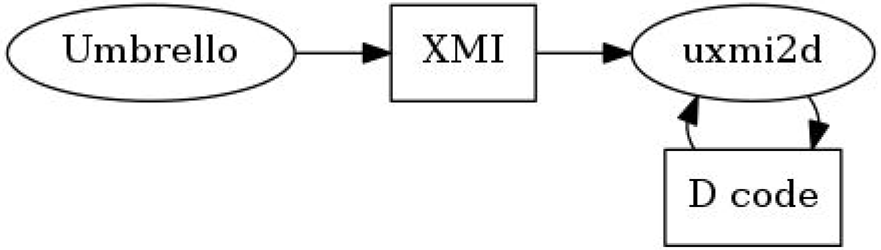
HOW TO CREATE A STABLE DATA MODEL

UML to D

- living documentation
- skeleton code generation
- generation of accessors
- code conventions
    - one class per file
    - first fields, then member functions
    - order of attributes
- documentation comments for contracts

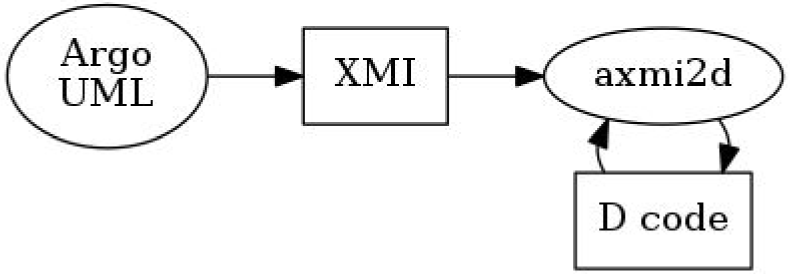
Umbrello XMI to D

https://umbrello.kde.org/



ArgoUML XMI to D

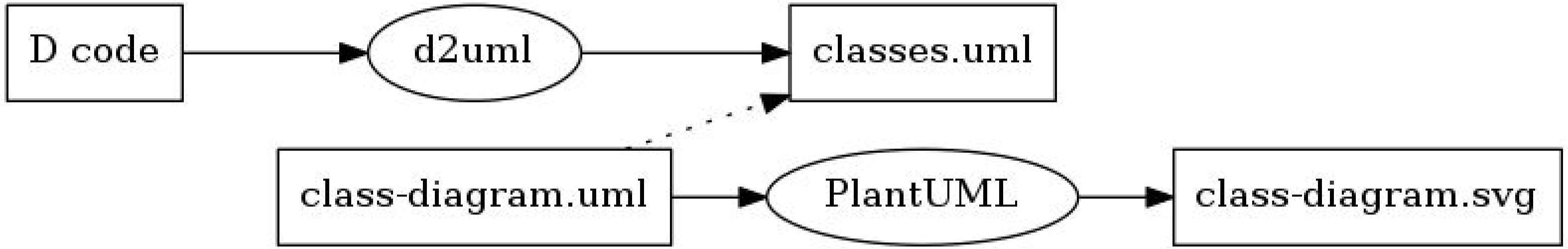
http://argouml.tigris.org/



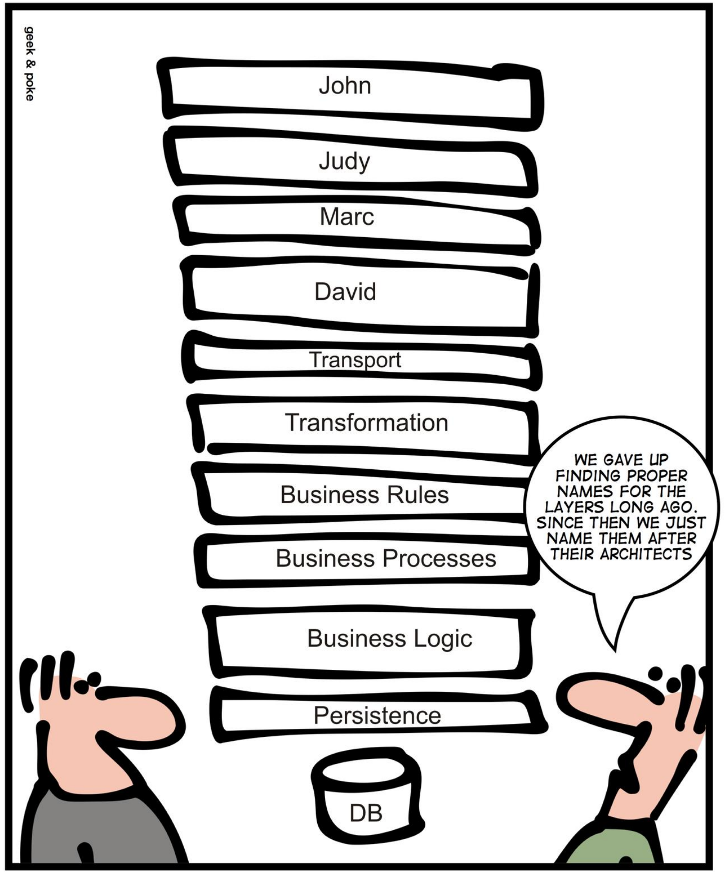
D to UML

Reverse engineering of D source code into PlantUML class outlines

https://github.com/funkwerk/d2uml
http://plantuml.com/



Demo



A GOOD ARCHITECT LEAVES A FOOTPRINT

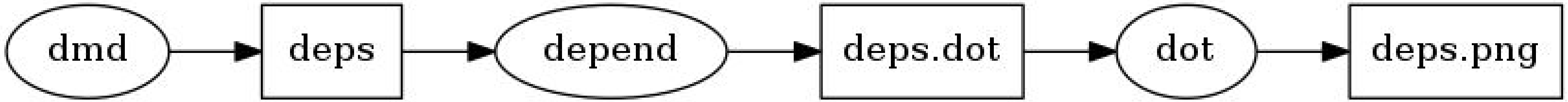
Architecture Management Tools

“A superb weapon in our continuing battle against architecture entropy.”

http://structure101.com/

Dependency Tool for D

https://github.com/funkwerk/depend



Example: vibe.d

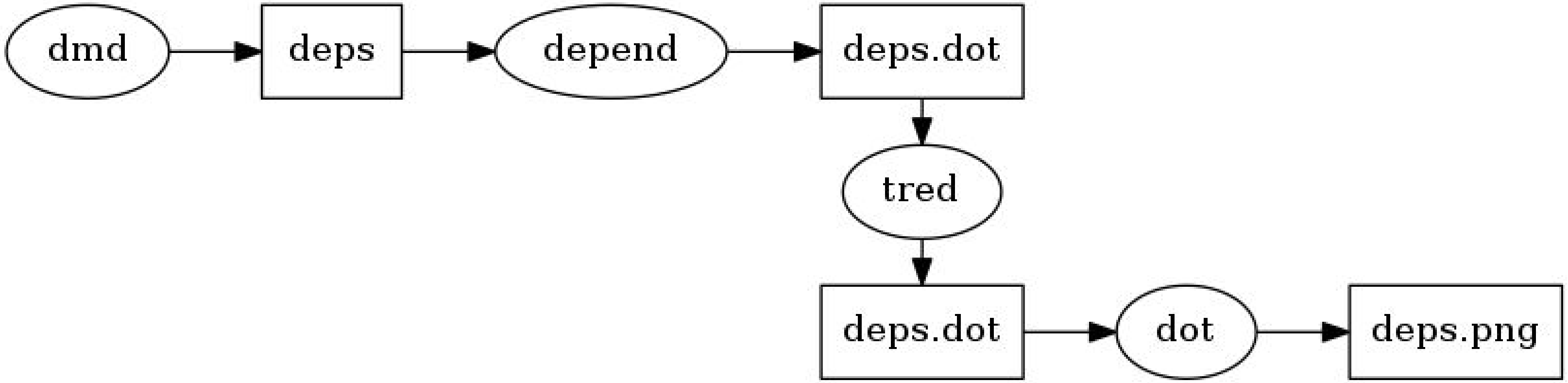
High-performance asynchronous I/O, concurrency and web application toolkit

http://vibed.org/
https://github.com/rejectedsoftware/vibe.d

Demo

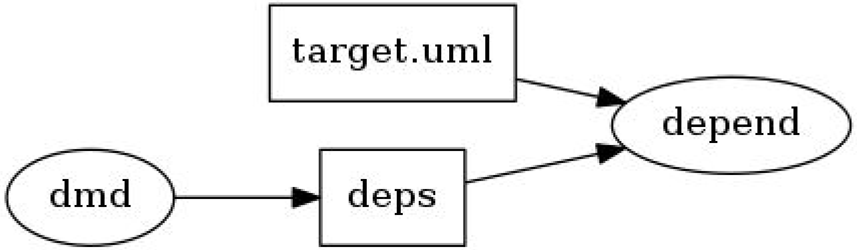
Dependency Tool for D

1. visualize module or package dependencies



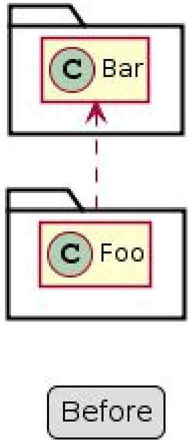
Dependency Tool for D

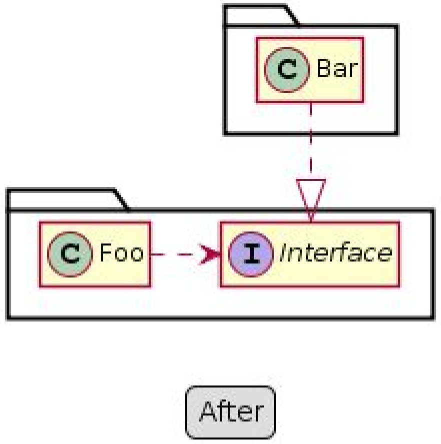
2. check actual dependencies against target dependencies



Dependency inversion principle

https://en.wikipedia.org/wiki/Dependency_inversion_principle



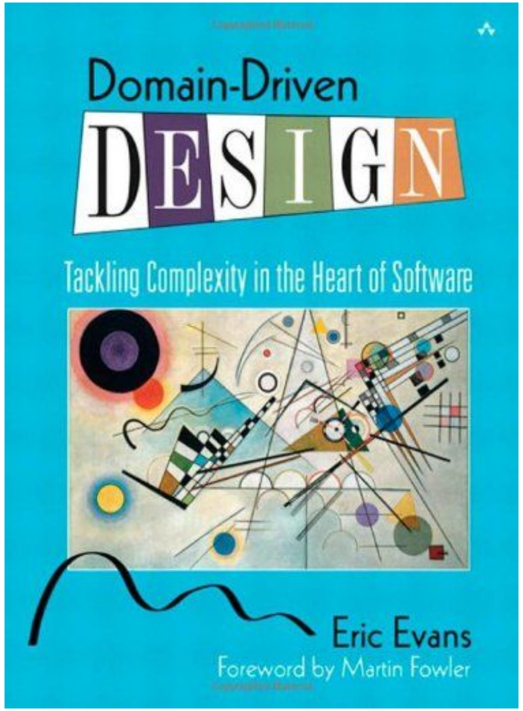


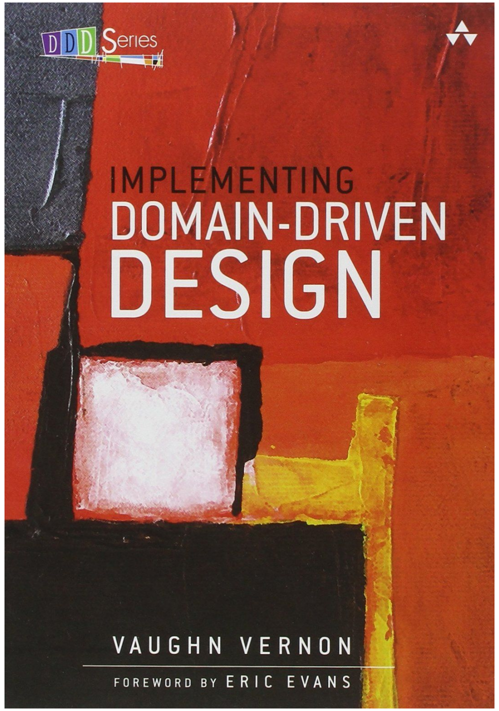
One more thing...

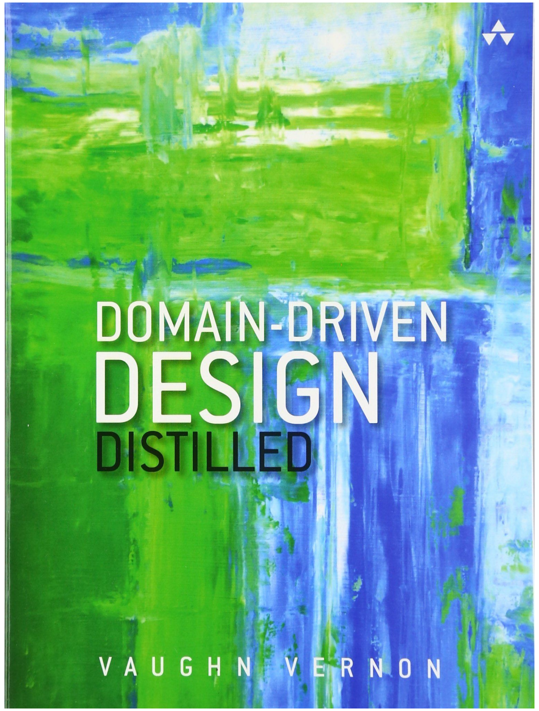
accessors

Generate D getters and setters automatically

https://github.com/funkwerk/accessors


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler


DDD Series
IMPLEMENTING
DOMAIN-DRIVEN
DESIGN
VAUGHN VERNON
FOREWORD BY ERIC EVANS


DOMAIN-DRIVEN
DESIGN
DISTILLED
VAUGHN VERNON