DSL Prescription

Use DSLs to Express Design Intent

We currently don't have a language for DI (Design Intent, aka Architecture, aka Business Rules).

Refactoring is a symptom of DI being embedded in code. Corollary: if you need to refactor, then it is likely that the code hasn't been split into DI and Implementation.

Excess detail is the antithesis of DI.

Most languages pride themselves on how many features they have, instead of how few features they have.

When you create DI, you don't want to care how something is implemented (e.g. Arrays vs. Lists).

Low-level efficiency is the concern of Efficiency Engineers, not Architects.

DSLs are one way to split DI from Implementation.

Automate Everything

Cheat

When possible, cheat.

Don't Do Things That the Base Language Already Does

Let the underlying base language handle the heavy lifting.

Use a DSL Only If It Saves Effort

Design

Can a design be expressed "better" - e.g. more accurately - using a DSL instead of using a detailed HLL?

Reuse of Architecture

Reuse of DI (Design Intent, aka Architecture) is more important than the reuse of code.

Code is cheap, thinking is hard.

Keep "business rules" separated from code. Use one language for DI, another for Implementation.

Coding

Does using a DSL reduce coding time?

Automation - get the DSL to write code. Write programs that write programs.

Maintenance

Does using the DSL reduce maintenance effort?

Can a Maintenance Engineer understand the DI (Design Intent) more quickly?

Does the DSL perform D.R.Y. (Don't Repeat Yourself) for you?

Can a Maintenance Engineer perform bug fixes more quickly by tweaking the DSL?

Can a Maintenance Engineer perform feature upgrades more quickly by tweaking the DSL?

Write As Little Code As Possible

Create Small DSLs

Reduce Coding Using DSLs

Rely on the Base Language to do the Heavy Lifting

Use More than Text

Management uses diagrams (e.g. on whiteboards).

Programmers should use diagrams, too.

It is OK to mix diagrams and text in the same document.

Diagrams Can Be Easy to Transpile

Diagrams can be easy to transpile.

Think *glyphs*, not pixels. Use backtracking parsers (e.g. Ohm, PROLOG, etc.).

Gedanken examples:

- How do you know if 4 lines make a box?[1]
- How do you know if one box is smaller than another box?
- How do you know if the smaller box intersects the edge of the bigger box?
- How do you know if a piece of text is completely inside a box?
- How do you know if an arrow (a glorified line) joins two boxes?
- What if the line is made up of many smaller segments?
- How do you draw a network?
- How do you draw a state machine?[2]
- What changes when you have ellipses instead of boxes?
- What changes when you have curvy lines instead of straight line segments?

When All Else Fails. Automate

Generate code, in some way, automatically. Use a pretty printer to make the code human-readable.

You can always use the generated code as if it were written manually (by someone else).

---

[1] If you know PROLOG or something like it, how would you declaratively write this relationship of 4 lines?

[2] If you don't already know, refer to Harel's StateCharts paper.

Why Management Hated DSLs

DSL (mis-)Perceptions

Management perceive DSL-writing as a sink-hole for time. This impression is based on the mistaken notion that writing DSLs is the same as writing compilers.

Management sees the up-front cost of creating a DSL. Management knows how to *measure* development time (and cost) but doesn't know how to measure maintenance (understanding) costs.

Management can't hire interchangeable units, called programmers, who already understand a given DSL. Understanding a DSL requires *thinking*, understanding a product design requires *thinking*, too.

Hiring

At the moment, we don't know how to hire *thinkers* based on only a resume.

The Profession of Engineering

The profession of Engineering encountered the problem, of hiring *thinkers*, decades ago.

The answer was to split the profession into parts.

If you attend university courses for 4 years and are rubber-stamped with an Engineering degree, then you are deemed to be an Engineer.

People who attend trades colleges for 2 years are deemed to be tradespeople.

Others are deemed to be labourers and brick-layers.

For this scheme to work, a method of communication between the strata must be used - *blueprints*.

Round Tripping

Round Tripping is Not Used in Engineering

Engineers put their seal (stamp or signature) on designs and are responsible - in Law - for their designs.

Brick-layers might detect "bugs" or "improvements" in designs, but they never make substantial changes to blueprints. The changes must be approved by the signing Engineer(s).

The practice of round-tripping is never used in labour and Engineering.

Round Tripping Is a Symptom

People use round-tripping technology when they believe that the generating technology doesn't work in all cases.

Round-tripping usually causes accidental complexity.

If you think you need round-tripping, then

a. prove that the notation doesn't work for some case
b. fix ithe notation, don't use round-tripping as a band-aid.

Blueprints

Current programming languages cannot be used like *blueprints*.

Current programming languages expose too much detail to be effectively used as communication mechanisms, such as blueprints.

In my opinion, the answer lies in *isolation*.

Drawings

Blueprints are drawings that expose little detail.

Blueprints are composed of simple elements.

Current programming languages expose too many details to be used in the way that blueprints are used in Engineering and construction.

Scalability

Further explanation:

The main problem in software design is scalability.

We want to "plug" pieces tother like LEGO blocks.

Better scalability implies fewer dependencies.

Early hardware people got this "right". They took incredibly complicated devices (semiconductors made up of various kinds of rust) and built chips / ICs (integrated circuits).

Chips were black boxes. They had a set of input/output pins. The insides of the chips were inscrutable - encased in opaque epoxy.

Nothing leaked out of or into a chip except through the pins of the chip.

Properties of a chip were described in easily-measured terms:

- voltage on a pin
- current needed by a pin
- diagram / chart of the outputs, given a set of inputs
- timing.

Then, hardware designers "discovered" that point-to-point wiring between chips led to non-scalable designs.

They built a (small) hierarchy - chips mounted on boards plugged into backplanes.

The earliest backplanes were basically point-to-point wiring harnesses. For example, an early Wang word processor I owned, had a backplane with some 400 pins, allowing a chip on one board to send signals directly to a chip on another board.

Then, came the S100 bus. It had only 100 pins. It was well defined and documented. Certain connections were not allowed, even if they could be done more efficiently as point-to-point connections.

The idea of the Bus led to Apple computers and, ultimately, the IBM desktop computer. (There was more than one Bus definition, but the market shook those out).

Can software be built like chips? I argue Yes.

We need to build software in hierarchies.

Divide and conquer.

There must be no leakage - of anything - between layers in a hierarchy. ("Anything" includes things like variables, types, control flow, dependencies of any kind, etc.).

Rigor and Trade-offs

Engineering is about making trade-offs.

Engineers don't strive to *prove* that a design works - they simply build safety margins into a design.

The current quest for provable software designs will not lead to Engineering.

What is needed is characterization of the possible trade-offs, e.g. How fast does it run? How much memory is needed? How much processing power is needed? What is the fail-safe, the "big red button"? What can be done if it crashes? Does it have a "known beginning state"? How much will it cost to design each feature? How much will it cost to test each feature? What is at stake? How thoroughly does it need to be tested? What is the worst-case throughput? What is the average throughput? What is the MTBF? Is it single-sourced or multiply-sourced and what are the implications?

Complexity

I see software as a hierarchy of black boxes. The Architect for each box *chooses* the best way to describe the design intent of a black box. The Engineer figures out how to dot the I's and cross the T's. The Production Engineer figures out how to measure and make the black box "more efficient" and the Coder lays the bricks to implement the black box.

Black Box Architecture

I see software as a hierarchy[3] of black boxes. The Architect for each box *chooses* the best way to describe the design intent of a black box. The Engineer figures out how to dot the I's and cross the T's. The Production Engineer figures out how to make the black box "more efficient" and the Coder lays the bricks to implement the black box.

Many Silver Bullets

A good Architect will have a tool-belt full of Silver Bullets. Maybe a problem is best described in Relational terms, maybe a problem is best described as a State Machine (as a diagram, yet), maybe a problem can be broken down in a synchronous manner, etc., etc.

[3] Actually, a directed, acyclic graph. Information flows upwards, control (commands) flows downwards.

PSLs

I will use the term *PSL* instead of *DSL* to emphasize problem-specific issues for every problem+solution.

PSL means problem-specific language. The older term, DSL, means domain-specific language. In my opinion, "domain" is too broad a term, we must focus down on problems and we must use specialization instead of generalization to solve specific problems.