Designing & documenting software interfaces

Andy Buckley
University of Glasgow

MCnet Computing School, Mariaspring, 5 September 2018

Intro

- Design is the best bit! Chance to pour all your creativity and experience into making the interface you want
- It's satisfying: craftsmanship, a bit of artistry, applied psychology, ...

Intro

- Design is the best bit! Chance to pour all your creativity and experience into making the interface you want
- It's satisfying: craftsmanship, a bit of artistry, applied psychology, ...

- *But it's hard:* need to think 3 steps ahead, and keep many things in mind

Intro

- Design is the best bit! Chance to pour all your creativity and experience into making the interface you want
- It's satisfying: craftsmanship, a bit of artistry, applied psychology, ...

- *But it's hard:* need to think 3 steps ahead, and keep many things in mind

- And leave yourself room to manoeuvre: you *will* get some things wrong, so how to avoid getting locked into bad ideas?

  $\Rightarrow$ **it's easier to add things than to take them away**

Think of the user first

- My golden rule: think as a user
  - You will need to agonise, sweat, and write clever (or repetitive) code to make it *seem* simple
  - Ideally they will have no idea how much is going on below the surface!

Think of the user first

- My golden rule: think as a user
  - You will need to agonise, sweat, and write clever (or repetitive) code to make it *seem* simple
  - Ideally they will have no idea how much is going on below the surface!
- "Make simple things simple; and difficult things possible"

Think of the user first

- My golden rule: think as a user
  - You will need to agonise, sweat, and write clever (or repetitive) code to make it *seem* simple
  - Ideally they will have no idea how much is going on below the surface!
- "Make simple things simple; and difficult things possible"
- **Metaphors are key:** a good metaphor means that an interface "flows"…and less docs needed!

Think of the user first

- My golden rule: think as a user
  - You will need to agonise, sweat, and write clever (or repetitive) code to make it *seem* simple
  - Ideally they will have no idea how much is going on below the surface!
- "Make simple things simple; and difficult things possible"
- **Metaphors are key:** a good metaphor means that an interface "flows"…and less docs needed!
- **Consistency**
  ⇒ "Principle of least surprise"

A few good acronyms

- **KISS**

  Keep it simple, stupid

A few good acronyms

- **KISS**

  Keep it simple, stupid

- **DRY**

  Don’t repeat yourself

A few good acronyms

- **KISS**

  Keep it simple, stupid

- **DRY**

  Don't repeat yourself

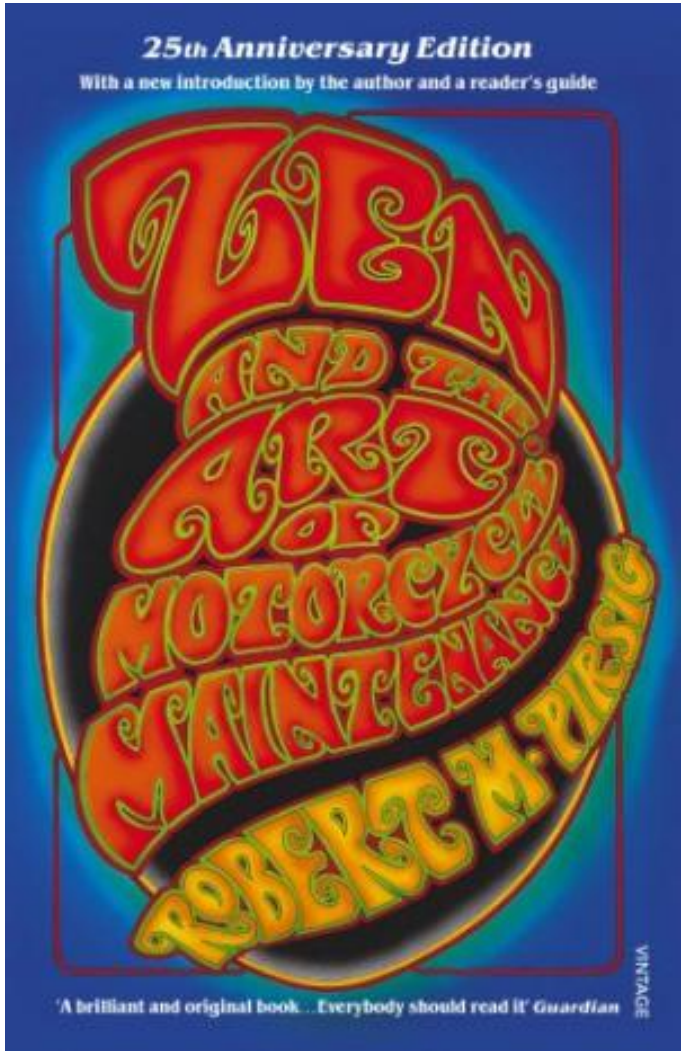
- **YAGNI**

  You ain't gonna need it

Quality

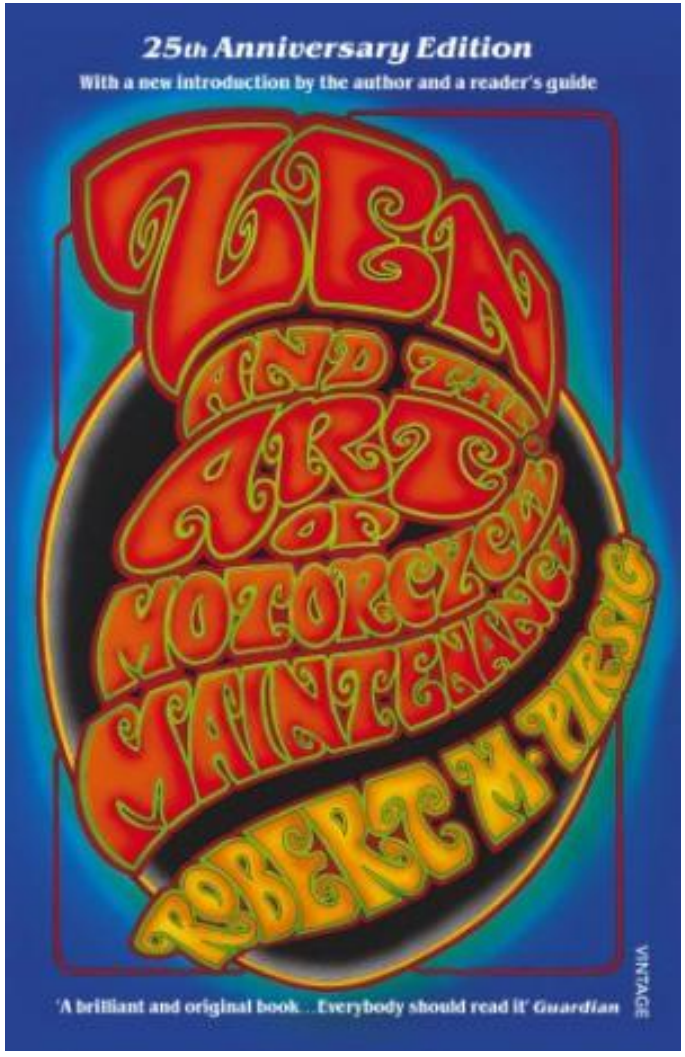
Emphasise *quality*:
take pride in getting it right

- Implies iteration: "Build one to throw away – you will anyway"
- Plan for wrong turns ⇒ pro-active versioning and deprecation strategies
- Defensive design ⇒ caution, think ahead, and be prepared for extra up-front work



Quality

- **Avoid short-cuts…**
  except when you don't!
  - Don't forget to be proportionate: if you're really hacking a one-off, do what gets a good enough result quickly
  - No point in heavily engineering a throw-away
  - But as you get better, robust designs will be familiar and not take significantly longer



Caring

What qualities do really bad programmers share? (Quora)

They don't care.

- They don't care about the reason why they're coding – the thing they're supposed to make work
- They don't care about how they go about coding – best practices, methodology, modularization, architecture...
- They don't care about understanding what needs to be done before they start coding – they jump right in
- They don't care about the language they're going to use – anything goes, the looser the better
- They don't care about putting in the work that's needed to build things the right way – they'll take shortcuts just because
- They don't care about tracking their progress and making sure others can understand what they're doing – after all, bug trackers are only needed when your code has bugs, right?
- They don't care about writing tests and documentation – if you want to understand what the code does, why don't you just look at it?
- They don't care about your advice nor your opinion, because, quite frankly, they already know better

Design methodologies

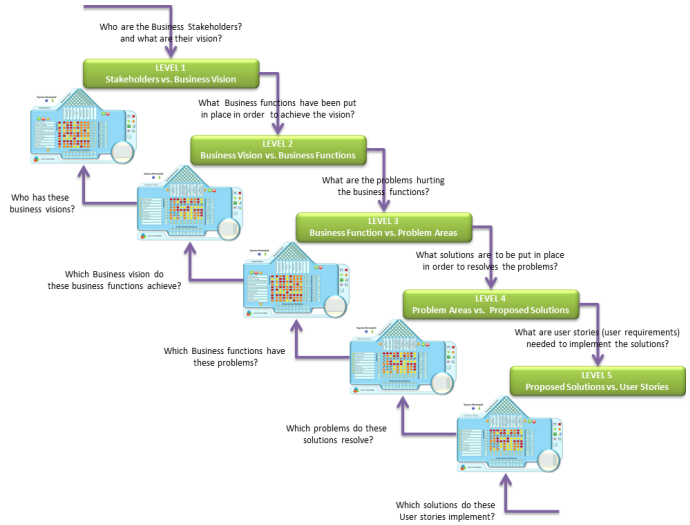
First, collect requirements. Think before you act. Then…

- **"Lone genius":** single-minded focus *can* drive design coherence
  - Everyone has met some unsatisfying system "designed by committee"
  - But you *will* need to work with others
- **Discussion / prototyping:**
  - Get feedback early
  - Linus' Laws: "Release early, release often"; "With many eyeballs, all bugs are shallow"
  - Maybe try card-passing exercises for OO design: you are the objects

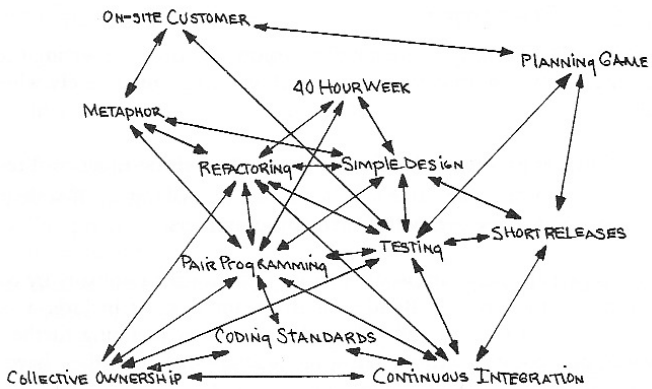
Design methodologies

First, collect requirements. Think before you act. Then…

- **The Spec**
  - Design everything to the finest detail on paper in advance
  - Does anyone still believe this is possible?!



- **"Agile" / "Extreme" / "SCRUM"**
  - Lightweight, minimal spec'ing, pair programming, fast evolution, unit testing & code review
  - Requires really good people (?)
  - + YAGNI: a good motto when deciding whether to add obscure features "for completeness / for symmetry / because I'm smart"



APIs

Application programming interfaces – i.e. code interfaces

- **A library or an executable? *Why not both?***
  - Is multi-language a possibility? Use each's strengths.
- **Many complex problems are made worse by misalignment of problem and code design**
  - Prototype (maybe in Python, even for C++ projects), and keep going until you feel it click
  - Try writing your fantasy ideal user script, then make the API that enables it to work
- **Identify what is the "atomic" *key object* in the problem**
  - This may be a single thing or a collection of them.
  - **Divide and conquer** along natural relationships. . . which may not be immediately obvious. **One job, done well**
  - Look for orthogonality and composability
  - You're designing, not abstractly classifying: cf. the object-oriented toaster . **Remember the use-case!**
- **Some reading: Qt little manual , Pixar C++**

Coding convention

- **Choosing good names is *really* important. Way more than you think**
- Use convention to minimise user/maintainer need to check definitions
- Micro-consistency is worthwhile: e.g.
  - "`mk`" or "`make`"?
  - `camelCase` or `under_scores`?
  - `CONSTANTS`, `Types`, `functions`. (+ `variables`, internally)
  - Clearly mark private class members (I like Python leading underscores)
  - Pass common arguments in the "same order" everywhere (or $\Rightarrow$ single object)
  - Attention to detail pays off in *lack of surprise*

Coding convention

- Make user-life pleasant! Save keystrokes while maintaining readability
- Distinction between this and *style*
  - Whitespace is not an API thing – but choose a decent, widespread convention and *be consistent*
  - And respect other people's conventions when working on their code
  - Great opportunities for bikeshedding – **STOP IT!**

Exercise: designing APIs

Get in groups of 4. We're going to try a design exercise!
Collect requirements and sketch main features of one of:

- **Design a 4-vector class for use in an MC event record**
  - How much of the abstract maths needs to be displayed?
  - What about numerical stability?
    Coordinates (hidden/explicit?) and $m = \sqrt{E^2 - p^2}$. Units?
  - Interoperation with established tools?
  - What little things will make users' lives better?
- **Design a set of histogram classes**
  - Separate classes for separate dimensionalities? Or one unified design?
  - What about overflows?
  - Are the user requirements the same for statistics and for plotting?
  - What to interoperate with, and how best to do it?

Documentation

"I didn't want to write this, but I wanted to read it and this seemed the only way" — Xkb HOWTO

- Everyone hates writing documentation (and tests) – but if your interface makes intuitive sense, you need less of it
- "Out-of-date documentation is worse than no documentation". Unless you make it *really* easy, it'll go out of date
- Reduce barriers ⇒ Doxygen, Sphinx, doctest direct from structured comments
- Can integrate "traditional documentation" with building the project website
- Self-documenting code structure, cf. queriable Rivet analysis metadata

Doc generators

Great idea cf. *literate programming* — make the documentation part of the code

Doxygen: really just for C++. Mark up your code with special comments, and lots of directives:

```
/// This class is a bit of a waste of time
///
/// @deprecated I used to think this was a good idea, but now
/// realise the error of my ways.
class Useless : Pointless {
  ...
};
```

Easy to generate and update config files with **`doxygen -g`** and **`doxygen -U`**

Doc generators

Python docstrings: similar, but docstrings are part of the lang

```
def mult(a,b):
    "Multiply two numbers, inconveniently"
    import math
    return math.exp(math.log(a) + math.log(b))
```

- Try direct in the interpreter
- Pydoc: **`pydoc -w mydemo`** . . . mmmmm!
- Sphinx: now main Python doc tool

  `mkdir doc && cd doc`

  `sphinx-quickstart` . . . *conf hack*

  `sphinx-apidoc -o source ..`

  `make html`

  https://gisellezeno.com/tutorials/sphinx-for-python-documentation.html
- readthedocs.org – very convenient online hosting of technical docs

Command-line interfaces

The design rules are exactly the same!

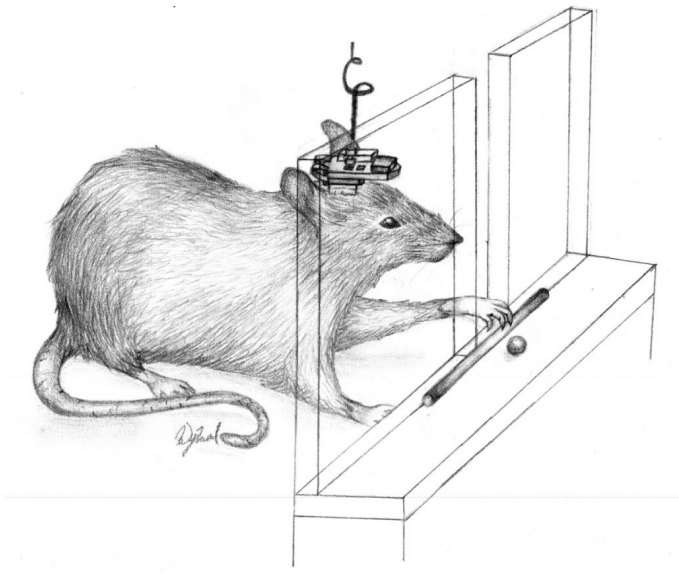
- Unix philosophy of "small tools, each doing one job well" is excellent advice. You can't anticipate all use-cases, so composability & orthogonality maximise possibilities
- Split applications into separated components. Communicate via established OS features
- Be consistent: e.g. the same option flags for the same tasks. Minimise surprise, encourage successful guesses
- Micro-advice:
  - Names: avoid underscores, avoid file extensions (?), use a common prefix for "tab searching"
  - Build in looping potential via "last-arg = open-ended list" $\Rightarrow$ saves users need to write shell loops
  - Use standard option parsing libraries e.g. Python ~~optparse~~ argparse

Data format design

- I'm a big fan of plain text for non-huge data
  - Robustness, human readability/debuggability
  - Space efficiency can be traded
  - Parsing efficiency can be improved with some tricks
- Base on standard meta-formats as much as posssible: YAML, JSON, (XML), HDF5, (ROOT), ...
- Think again about how you'll handle changing requirements: include versioning from the beginning, to handle "schema evolution"
- Steering/config file formats and user interfaces suit restricted subsets of usage possibility: don't be tempted to make a Turing-complete CLI or config syntax!
- Weird uses can use your beautiful, flexible API!

Summary

- Good interfaces are 50%+ of good design. Spend time on them
- The exact methodology is less important than care, creativity, and being prepared to throw away & improve
- Every rule is to be broken. . . but don't do it lightly: *agonise* about it!
- When it's good, you'll *know it*!
- Interfaces are everywhere: APIs, CLIs, data. They all deserve proper design



Extras

Object orientation

- Object orientation is about polymorphism and encapsulation first: "need to know"
- In many situations it is a perfect match to the problem: interacting self-contained objects are the natural picture
- Avoids the need for developer omniscience ⇒ maximum-entropy code problem
- But **not *everything* needs to be an object!**
  - This was the error of Java, "the OO language"
  - Parables: object-oriented toaster (not every logical base class is necessary), kingdom of nouns (not everything is an object)
  - Keep it flexible; mix and match styles; Zen/kung-fu pastiche *x* yet *y*

Yeeesh...

Object orientation

- You do not need to use all the language features! Think of the user…
- Are objects compatible with scalability, and parallel/vectorised performance requirements?
  - Encapsulation is conceptually great for the programmer, potentially fatal for CPU caches
  - "Views" on to contiguous data arrays may work (are there OO langs that help?!)
  - **It depends** on how your users will use your tools

Yeeesh…

C(++) specifics

- **C++ was not designed to minimise coupling, provide clean interfaces, etc.**
- What to expose? Not everything should have a public header
- *"Does this code unit need a header at all?"*
- Use namespaces, forward declarations, and avoid `using` in headers – unless namespaced
- C++ include system is broken: exposes implementation details via memory layout of internal data, forces wide rebuilds
  - Strategies, e.g. PIMPL method to decouple dependencies
- Pointers as a hint of memory management; references otherwise.
  - I like to use "mk" names when memory is allocated, as a signal for the user to watch out