Changing the engines without landing the plane

Robert Bentall, January 2022

About me

- I’ve worked as a software engineer across a range of domains, organisations, technology stacks
- I am currently a Principal Software Engineer at Martin-Baker, where my team works on software used in ejection seats
- I’m fascinated by the process of software engineering
- Especially how to produce higher quality software within tighter schedules and cost constraints

In this talk…

- What is technical debt?
- Case studies
- Framework
- Antipatterns
- Antidotes
- The future

None of the case studies or data in this presentation relates to Martin-Baker

What is technical debt?

- “The implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer” (Wikipedia)
- Stuff in the future takes longer because of short-cuts taken now
- Not just code quality, but also:
  - Design debt (internal quality)
  - Documentation
  - Configuration management
  - Verification

Case studies

Software that should never have been written

Updating libraries in a large shrink-wrapped product

Onboarding a (small) legacy desktop application

Software that should never have been written

- Accounting application for a small organisation
- Written by a company with no experience in the domain
- Written for a company that thought they needed a custom application
- C. 25000 LOC SQL / VB, desktop application with SQL Server backend
- No installation scripts for the database
- No test scripts (manual or automated)
- No CI server to build the application/installer
- Bug list was a spreadsheet
- No documented release process or change control process
- Customer thought it was “nearly done”

Remediation

- Build awareness and trust with customer
- Agree a process for identifying, prioritising, fixing defects => put customer in control
- Needed to do enough for the system to be valuable to the business
- Produced manual test scripts to provide documentation + repeatable V&V
- Repeated antipatterns in the code made debugging quite straightforward
- I could predict defect fix cost for a tranche of work to with c. 10%

Reflections

- Three problems:
  - A customer that thought they needed a custom system
  - A supplier that thought they knew how to build it, but only got partway through
  - The customer thought it was nearly finished
- Lesson?
  - Don’t do custom software unless you (a) need to, (b) know what you are doing

Library updates on a shrink-wrapped product

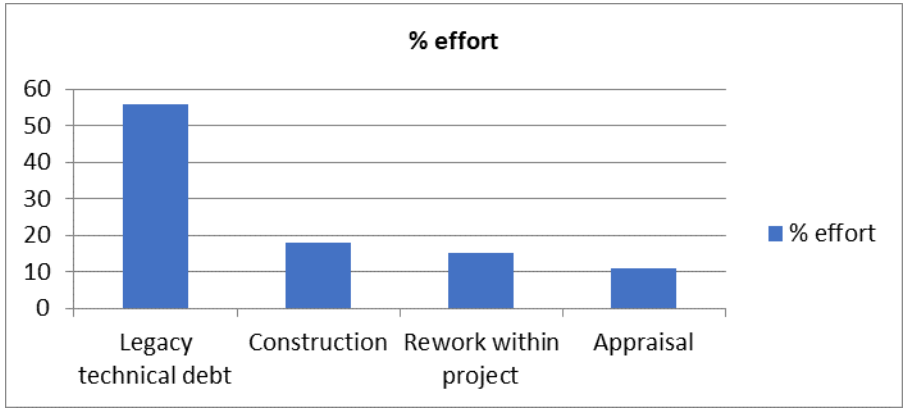
- Compiler upgrade on large shrink-wrapped product
- C. 5-7 million lines C++/C#, c. 90 projects in Visual Studio solution
- Our team was responsible for upgrading 2nd/3rd party libraries
- We agreed how we were going to approach the work:
  - Workflow for upgrades
  - Definition of done
  - Daily stand-ups + burn-down charts
- Technical debt became obvious once work started:
  - Previous maintainers had not documented configuration options
- Expectation to complete was 5-6 weeks, reality was c. 4 months

Remediation

- The to-do list got (a lot) bigger
- Took a while for the whole team to accept we had a problem
- Some engineers just wanted to get it done as quickly as possible – so we ended up re-doing some of our own work
- Senior management was supportive
- Our approach changed:
  - much more up-front investigation to identify what the library needed to do,
  - how it needed to be configured
  - Emphasis on automation + good enough documentation

Some metrics

COQ Group	COQ Subgroup	Task Hours	%
Costs of construction	N/A	26	18
Costs of control	Appraisal	16	11
Costs of failure of control	Rework from project	23	15
	Rework from previous projects	83	56
	Totals	148	100



- I analysed my task-hours from the work
- More than half of the time was spent on rework from previous projects – technical debt doubled the cost of the work
- Senior management response was positive:
  - Recognised the value of the work we had done
  - Recognised the approach needed to be different for future projects

Reflections

- Library management during early phases of project
  - Libraries are helpful if correctly used
  - But they impose an ongoing cost
  - Up-front work needs to be done to manage and control this
  - It’s not sexy or fun
  - If you take shortcuts, chances are someone else will have to deal with the fallout
  - Shortcuts had been taken in the past
- Elaboration of work prior to commencing the upgrade
  - Not enough elaboration done up-front
  - Work had been poorly planned

Onboarding a small desktop application

- In-house desktop utility, widely used
- Written as a personal development project by someone who wasn’t a software engineer
- Release process was to:
  - build on developer machine
  - copy the installer to a network folder
  - ask a colleague to install/test it for you

Remediation

- Planned rewrite of application
- Re-architect the internals to be buildable/testable
- Ensure CI/test support from the word go

Reflection

- There’s a difference in skill/experience between:
  - someone who can cut code & build an application
  - someone who can manage all aspects of the lifecycle
- The “boring stuff” is arguably what pays the biggest dividends
  - High-maturity processes (PSP/TSP)
  - Ubiquitous automation (build/test/deploy)
  - This doesn’t have to equate to lots of paper

What impact will technical debt have on your project?

Questions

Context

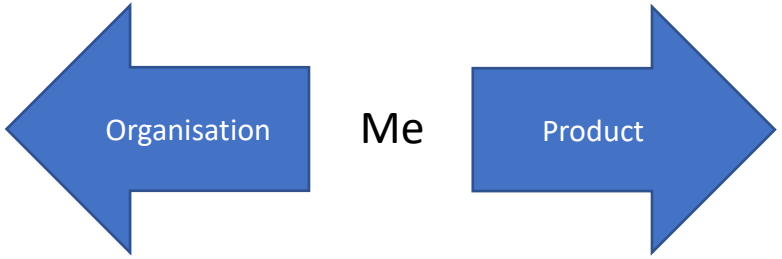
Consequences

Action

Questions I ask (a starting point)

Identifying the software - talking to people

- What is it?
- Where is it?
- Is it in source control?
- How do I know what's in the latest release?
- What happens if it goes wrong?
- How do I know what it is supposed to do?



The build process – getting to grips with it

- Can I build it on my PC?
- Can I debug and test it on my PC?
- Can I build it on a CI server?
- Can I test it on a CI server (as part of a build process)?
- Can I produce a release quickly/cheaply/safely?

Increased toolchain maturity

The context you are working in

- Organisational context – the values and dynamics of the organisations you are working in/with
- People – the skills & experience of you, your colleagues, your customers
- Processes – “how stuff happens” – an organisation’s encoded learning
- Technology – “the plumbing” – IT systems, toolchain, build and test infrastructure

These will define the boundaries of what is acceptable/desirable within your organisation => therefore impacts what you can do to solve the problem

Consequences

- Understanding where you are is critical:
  - What do you need to do to get the technology into an acceptable state?
  - What does the organisation want you to do?
  - How do you reconcile these?
- You need to be steely and diplomatic…
- If your customer trusts you, and they understand you are working in there best interests, you have a fighting chance

Action

- You need buy-in from the customer to be able to fix technical debt in a sustainable way
- They need to be able to control the work – they are paying for it
- A well-maintained engineering backlog, together with clear communication about the business benefits of a change will pay dividends
- Sometimes incremental improvement is OK
- Sometimes you'll need a bigger planned refactor
- You will both see the benefits

Antipatterns

Skill

Process management

Automation

Skill

- Unconscious incompetence
  - Domain knowledge
  - Technical capability
- Little (or poor) design
- Little (or poor) planning

Fred Brookes (Mythical Man Month, 1975):

“More software projects have gone awry for lack of calendar time than for all other causes combined”

Process management

- Lack of process
- Manually-driven processes
- Siloed systems
- Little or no measurement
- Metrics you can’t trust

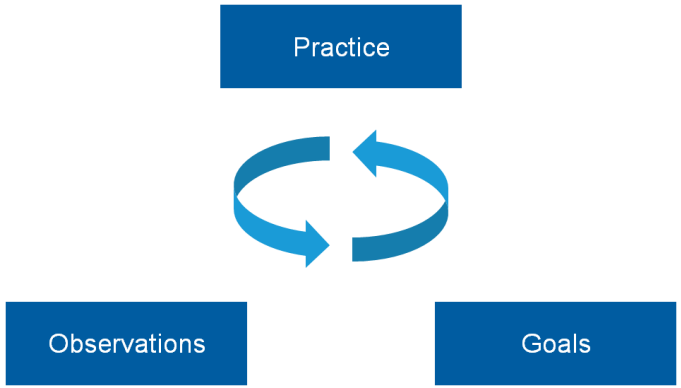
Automation

- Lack of automation
- Automation that “sort of works”
- Semi-automation

So what?

- Any one of these will slow you down AND reduce your ability to learn
- If you have all of them, you are in a lot of trouble
- If you want to improve your organisation's performance, you need to address all three – they feed off each other
  - Skills – start and execute projects with a realistic assessment of your risks
  - Process – stuff is done in a repeatable, measurable way
  - Automation – builds and tests can be trusted, drudge work is eliminated

You need feedback to improve….



Antidotes

You

Your team

The environment

Investing in the future

You

- Know thyself
- Set your own direction
- Know what to ignore

Your team

- Build enthusiasm (or at least tolerance)
- Grow together
- Plan for average execution

The environment

- Socialise the process, learning, benefits
- Find ways to help others be more productive

Invest in the future

- Training
- Infrastructure

What might the future look like?

- There is no silver bullet…
- The problem is likely to get bigger in the future:
  - More people can code
  - Barriers to entry are lower – app marketplaces make it easy to sell your work, cloud providers reduce the overhead to managing your infrastructure
  - More people are likely to be producing apps that need maintaining in the future
  - A lot of these will be learning in isolation
- But ubiquitous automation goes a long way to helping you (Gitlab)
- Tools like SQALE provide mechanisms for automatically quantifying technical debt (but only based on code…)
- You'll never have one big system to cover all your needs… and infrastructure components will need to be swapped in and out over the years as products and services evolve
- Toolchains will likely still require components from multiple vendors, so plug & play integration is going to become more important

Thank you

- Any questions?