Tallying and Taming Technical Debt

SPA 2019 Workshop
Andy Longshaw, Nick Rozanski and Eoin Woods
24th June 2019

Introduction

"Tallying and Taming Technical Debt" was a workshop run at the SPA 2019 conference in June 2019 in London. The aim of the workshop was to get attendees thinking about the types of technical debt that they had encountered in their career and how the different types can be identified, measured and remediated.

The workshop contained two exercises, one to identify candidate types of technical debt, and classify them into groups, and then a second exercise to take one of the types and consider how to identify, measure and remediate it.

This document captures the outputs of the two exercises. Thanks to all those who came to the session and contributed so much.

Exercise 1: Classes of Technical Debt

We started with the three technical debt classes of *Architecture*, *Code* and *Production*, from the book "*Managing Technical Debt*" by Philippe Kruchten, Rod Nord and Ipek Ozkaya. During the workshop, we realised that there seemed to be a missing class, which was for technical debt related to mismatches between the different models in a software application (the real world, the domain model in the code, the stored information model and so on). We called this the "*Model and Design*" class as a working title.

The types of technical debt that the participants identified are captured in the table below.

Architecture	Model & Design	Code	Production
Undocumented short term decisions Limited cross-functional capability (e.g. using in-memory storage while volumes are low) Lack of contact testing Needing other system or service to be continually available Security vulnerability scans specific to single build Out dated software platform (e.g. unsupported database)	Mismatch between system model and real domain Business domain model changes not reflected in system domain model (e.g. deferring schema changes) Domain model properties used to manage technical infrastructure Test data not kept up to date with business domain changes (e.g. not extending the text fixtures when the domain changes)	Known ignored or skipped tests Ignored code checking or compiler warnings Dead code TODOs (particularly those with outdated content!) Commented out code Flaky tests Fragile tests Overspecified tests Lack of tests Duplicated functionality	Known ignored alerts Lack of monitoring Known unmonitored system limitations (e.g. in-memory data store) Cluttered environments Premature scaling and optimisation Disaster recovery deferred Mismatch between pre-production and production

Hard tie in to framework version Missing core security functions Rapid delivery by using technology without good skills availability	Features implemented in such a way that they do not fit the system domain model Poor or inappropriate UX for need ~~Lack of early customer engagement~~ [not technical debt]	"Raw" error messages Code hard to understand Writing code that should be reused (e.g. sorting) Known inconsistent behaviour (e.g. when switching on debugging) Deferred refactoring Lack of automation Missing or ineffective error handling Inefficient code	Assumptions about hardware and network infrastructure Pointless alerts Log severity inflation Duplicated log messages (catch / log / rethrow / catch / log / rethrow / …) Complexity in single step build Employee departure without knowledge handover Loss of knowledge that has been built from the ground up Using patches to ship into production as a shortcut (=> need to train people to remediate, increases complexity in production, limits architectural change, difficult to refactor)

Exercise 2: Finding and Mitigating Technical Debt

Name	Environment Dissonance
Summary	Production environment not like other environments (preprod, staging, …)
Measurement Approach	1. Define what metrics matter (hardware, software and versions) 2. Describe each environment using the metrics 3. Compare metrics to identify differences
Consequences	Testing may be invalidated Production failures: may be difficult to remediate; developers may cause incidents due to invalid assumptions about the production environment.
Avoidance Approach	• Software based infrastructure (containers, SDN, OpenStack, …) • Define and document metrics of interest • Automate creation, provisioning and cross-environment comparison (e.g. OpenStack) • Automate desktop provisioning • Make sure to include software

Remediation Approach	• Set up the above • At the same time: do (1) get devs aligned with (1), do (2) and (3), upgrade pre-prod or downgrade prod • Who: developers if possible, otherwise infrastructure administrators

Name	Deliberate Limit to Scalability
Category	Architecture
Measurement Approach	• Time spent operation vs building the system • Resource usage • Response times • Throughput • Error rates (failed requests)
Consequences	• Outages • No time for new feature work • Slow response times for customer • Poor relationship with stakeholders • Cascading failures in upstream systems • Low morale • Delayed response to issues
Avoidance Approach	• Clear service level obligations • Be able to profile the application (in production) • Scaling plan (know how long it will take, set up thresholds to give you the time to address it) • Record decisions made to limit scalability with planned remediation, monitoring approach etc • Don't run at 100% utilisation
Remediation Approach	• Ability to run in degraded service state (e.g. load shedding policy) • Have two scaling plans (strategic or planned, emergency) • Invest in autoscaling • Throw money at the problem (e.g. via AWS) • Test scaling

Name	Known Ignored Alerts
Category	Production
Measurement Approach	• Number of unassigned alerts • Alert severity post-incident • Number of known ignored alerts • Alerts with no human action required
Consequences	• People treat real alerts with less importance • Annoying for on call => people don't want to do it

	• Increased alert noise hides real issues
Avoidance Approach	• Regular review of ignored alerts (e.g. weekly) • Have criteria and heuristics for creating new alerts
Remediation Approach	• Apply the new alert criteria and heuristics to existing ignored alerts (who: the team responsible for alerts; when: weekly • Fix valuable alerts, remove those not valuable