Detecting and Tracking Enterprise Technical Debt

Felix Bachmann
Stephany Bellomo
Bottom Line Up Front

We have been working for a large organization for 2 years, conducting design reviews of a large portfolio of projects.

We also work in the area of technical debt research.

During project design reviews, we routinely surface technical debt.

We noticed that some technical debt has unique characteristics.

We call this type of technical debt *enterprise technical debt* (ETD).

In this talk, we

- share characteristics and examples of ETD
- explain how to motivate action by making ETD items, along with supporting evidence, visible to stakeholders.
Conceptual Organization Chart

IT Management

- Business Managers
  - Special Project Lead
  - Special Projects
    - Project Architects
    - Team Leads
    - Team Members

Enterprise Architecture Guides (us)

O&M Management

- Business Staff
  - Project Architects
  - Team Leads
  - Team Members

- Project Managers
  - Software Projects
    - Project Architects
    - Team Leads
    - Team Members

Interacts with
Responsible for

O&M = Operations and Maintenance

Enterprise Level

Project Level
Our Role

We work for IT management, and we serve as architecture guides for a portfolio of projects; each project has at least one architect.

Our achievements in this role include:

- establishing an incremental design review process
- establishing a dashboard for tracking design risks
- facilitating and participating in design reviews and code analyses
- educating architects and teams on software architecture practices
- mentoring architects to conduct design review and code analyses
What Is Technical Debt?

In software-intensive systems, technical debt consists of design or implementation constructs that are expedient in the short term, but set up a technical context that can make a future change more costly or impossible.

Technical debt

- exists in a **system artifact**, such as code, build scripts, automated test suites, or data
- traces to **several locations** in the system, implying ripple effects of impact of change
- has a **quantifiable** effect on system attributes of interest to developers, such as increasing number of defects, negative change in maintainability, and code quality indicators
Characteristics of Enterprise Technical Debt

Comparison to technical debt:

✓ Exists in a **system artifact**

△ Is traced to **several systems or projects**

△ Has a **quantifiable** effect at the **enterprise management level**

Often, these characteristics also exist:

✓ Requires enterprise-level involvement and investment to resolve

✓ Requires project resources to fix

✓ Conflicts with an enterprise goal or standard

✓ Impact may have ripple effects across multiple projects in the enterprise
Examples of Enterprise Technical Debt

In this section, we will walk through three examples:

1. Shared schema integration (versus service API)
2. Business logic confusion in SpringMVC framework
3. Decentralized access control
Examples of Enterprise Technical Debt

1. Shared schema integration (versus service API)
2. Business logic confusion in SpringMVC framework
3. Decentralized access control
Shared Schema Example: Background

In this example, we have a project that forces two teams from different parts of the organization to exchange data

- One team provides an external portal
- The other provides internal functionality and data used by the portal

The integration approach they used was common at the time it was built

However, now they are feeling the limitations of the approach

Neither team has control or motivation to fix the problem

This is because the enterprise has different goals from the project level; therefore, risks may impact the enterprise but not the project (and vice versa)
Example: Shared Schema Integration

**System Artifact: Shared Schema**

**Problem:** Applications A and D are integrated using a shared database schema. Changes to the schema by Application A impact D and vice versa.

- Exists in a system artifact
- Is traced to several systems or projects

Changes here directly impact Application D.
# Shared Schema Impact and Quantification

<table>
<thead>
<tr>
<th>Impact</th>
<th>Quantifiable Effect</th>
<th>Who Impacted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changes made to the shared schema by one team break the other team’s application</td>
<td>Maintenance cost, feature delays</td>
<td>IT Management, Business Managers</td>
</tr>
<tr>
<td><strong>Conflicts with organizational modernization goal to move toward service APIs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Does not scale</td>
<td>Delays making UI changes, cost of change</td>
<td>O&amp;M Management</td>
</tr>
<tr>
<td>Reusing the shared schema for more than two projects is very difficult</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functionality and data duplication are workarounds to avoid impact of changes</td>
<td>Cost of feature implementation, feature delivery delays</td>
<td>Business Management, O&amp;M Management</td>
</tr>
</tbody>
</table>

- Has a **quantifiable** effect at **enterprise level**
- Conflicts with **enterprise goal** or standard
Solution Recommendations

**Problem:** Product Owner of Application A states, “There is no money to make the life of team Application D easier.”

Therefore, the recommended solution is a two-pronged approach:

- **Enterprise Management** must allocate funds and/or reorder priorities to fix this
- **Projects J and K** must do the work to replace shared schema integration with service API

 ✓ Requires **enterprise-level involvement** and investment
 ✓ Requires **project resources** to fix
Examples of Enterprise Technical Debt

1. Shared schema integration (versus service API)
2. Business logic confusion in SpringMVC framework
3. Decentralized access control
SpringMVC Example: Background

For many years, the organization has been suffering from slow feature delivery.

A contributing factor is that business rules are PL-SQL stored procedures that have become complex and hard to change.

The organization

- set a goal to extract business logic out of the database stored procedures
- decided to use SpringMVC pattern to separate business logic from physical storage layer
- started with two projects using the SpringMVC pattern
Example: MVC Business Logic Inconsistency

**System Artifact:**
SpringMVC framework Service with focus on Controller

**Problem:**
Application B and C put business logic in SpringMVC Controller (versus Service), which creates tight coupling between UI layer and data layer

- Exists in a system artifact
- Is traced to several systems or projects
## MVC Impact and Quantification

<table>
<thead>
<tr>
<th>Impact</th>
<th>Quantifiable Effect</th>
<th>Who Impacted</th>
</tr>
</thead>
<tbody>
<tr>
<td>The organization has an enterprise goal to clean up the physical data storage tables, but because there is coupling from UI to DB layer, there may be impact if data storage level changes</td>
<td>Maintenance cost, feature delays</td>
<td>Business Management, O&amp;M Management</td>
</tr>
<tr>
<td>Changes to Controller could impact Data Layer, which would impact multiple applications</td>
<td>Delays making UI changes, cost of change</td>
<td>O&amp;M Management</td>
</tr>
<tr>
<td>Because business logic is not in consistent places, maintainers take longer to create new features</td>
<td>Cost of feature implementation, feature delivery delays</td>
<td>Business Management, O&amp;M Management</td>
</tr>
</tbody>
</table>

- ✓ Has a quantifiable effect at enterprise level
- ✓ Conflicts with enterprise goal or standard
SpringMVC Recommended Solution

Executive management must
- direct the projects to make the changes
- set aside funds for projects to do rework if needed
- reprioritize other tasks to get it done

Projects B and C must move business logic out of Controller to Service Component

- Requires **enterprise-level involvement** and investment
- Requires **project resources** to fix
Examples of Enterprise Technical Debt

1. Shared schema integration (versus service API)
2. Business logic confusion in SpringMVC framework
3. Decentralized access control  We are here
Decentralized Access Control Context

For many years, project teams have been building applications independently in silos. There is no centralized access control repository or maintenance screen. Each application writes access control roles/permission logic itself. Consequently, the teams write access control information for each application in different places in the database schemas. The project teams are fine with this, but the enterprise is suffering due to security and maintainability issues.
Example: Decentralized Access Control

System Artifact:
Access control architecture (or lack thereof)

Problem:
Each application team writes access control roles/permission logic itself; Access control information is spread all over the databases

- Exists in a system artifact
- Is traced to several systems or projects
## Decentralized Access Control Impact

<table>
<thead>
<tr>
<th>Impact</th>
<th>Quantifiable Effect</th>
<th>Who Impacted</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Security</strong> – Makes it more difficult and time consuming to delete a person from multiple applications and you run the risk of forgetting to delete the person from one or more applications.</td>
<td>Data loss cost, risk of exploitation cost</td>
<td>O&amp;M Management, Business Managers</td>
</tr>
<tr>
<td>Every application does access control independently, leading to too many different implementations, so features take longer to develop. Approach conflicts with enterprise goal to create common authentication component</td>
<td>Cost of feature implementation</td>
<td>O&amp;M Management, Business Managers</td>
</tr>
<tr>
<td>Roles are hard coded in some applications, which limits flexibility; if something changes in the organization, the code must change</td>
<td>Cost of role changes, admin change delays</td>
<td>O&amp;M Management, Business Managers</td>
</tr>
</tbody>
</table>

- Conflicts with an **enterprise goal** or standard
- Has a **quantifiable** effect at **enterprise level**
Access Control Recommended Solution

Executive management must
• direct the projects to make the changes
• set aside funds for projects to do rework if needed
• reprioritize other tasks to get it done

A Common Authentication capability should be created to store role and permission information with an administrative capability to allow project teams to make changes. Then project teams will need to adapt applications to use it.

✓ Requires enterprise-level involvement and investment
✓ Requires project resources to fix
Managing Technical Debt

Discovering enterprise technical debt is just the start; after discovery we need to do something with it

On the next few slides, we will cover these topics:

• Process overview for managing ETD
• Why and how we make ETD visible
• Implications of managing ETD as a continuous process
Process Overview for Managing ETD

To keep the diagram simple, we combine the Project Design Risk List and Project Technical Debt Item List into one list.

If a Project Design Risk has accumulation, we flag as Technical Debt Item in the list.
Making Data Visible

If not managed correctly, ETD will be forgotten until it is too late.

The key to getting action is to make data visible.

We make different kinds of data visible for different stakeholders and purposes.

Diagram:

Enterprise Level
- Discovery
- ETD List and Combined Risk Summary (Dashboard)

Project Level
- Project Design Risk List & Summary (Dashboard)
- Solution Proposal
- Solution Decision
- Executive Management Affinity Tree
Software Architecture Dashboard

Our Software Architecture Dashboard has four parts; the bottom two quadrants are project level and the top two quadrants are enterprise level.
The Project Design Risk List shows the project design risks

Each project design risk associated with an ETD increases the probability the risk will have an impact

<table>
<thead>
<tr>
<th>ID</th>
<th>Project Risk Description</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWQ-114</td>
<td>Data object definitions in Service belong in Utility</td>
<td>Project A</td>
</tr>
<tr>
<td>SWQ-108</td>
<td>Role/permission architecture is not flexible</td>
<td>Project B</td>
</tr>
<tr>
<td>SWQ-106</td>
<td>Use inheritance to get rid of unnecessary classes</td>
<td>Project B</td>
</tr>
<tr>
<td>SWQ-99</td>
<td>Design missing integration with Secure gateway</td>
<td>Project B</td>
</tr>
<tr>
<td>SWQ-77</td>
<td>Reduce branching complexity in xxDetailMgr.java</td>
<td>Project B</td>
</tr>
<tr>
<td>SWQ-71</td>
<td>Performance issue due to calls to database</td>
<td>Project B</td>
</tr>
<tr>
<td>SWQ-62</td>
<td>Duplication of data requires synchronization</td>
<td>Project D</td>
</tr>
<tr>
<td>SWQ-35</td>
<td>Add a notification component</td>
<td>Project E</td>
</tr>
<tr>
<td>SWQ-105</td>
<td>Manager and Model dependency</td>
<td>Project B</td>
</tr>
<tr>
<td>SWQ-76</td>
<td>Investigate JPA for complex SQL stmts in validation</td>
<td>Project B</td>
</tr>
<tr>
<td>SWQ-72</td>
<td>Move business rule logic to service layer</td>
<td>Project B</td>
</tr>
<tr>
<td>SWQ-36</td>
<td>Clarify data transformation responsibility</td>
<td>Project E</td>
</tr>
<tr>
<td>SWQ-27</td>
<td>Define IDs to map extranet records to intranet</td>
<td>Project E</td>
</tr>
<tr>
<td>SWQ-16</td>
<td>Use repo component via a manager</td>
<td>Project B</td>
</tr>
</tbody>
</table>
The Design/Code Review is a project-level roll-up of open issues by design or code review activity.
Enterprise Technical Debt List

This is an example ETD List derived from our current work (cleansed)

<table>
<thead>
<tr>
<th>ID</th>
<th>Enterprise Technical Debt Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWQ-131</td>
<td>ABC Portal using XYZ doc storage instead of enterprise doc storage</td>
</tr>
<tr>
<td>SWQ-128</td>
<td>The path for crossing extranet to intranet makes OSB redundant</td>
</tr>
<tr>
<td>SWQ-127</td>
<td>Externally facing data exchange Apps A and B use data syncing</td>
</tr>
<tr>
<td>SWQ-126</td>
<td>Standardized enterprise access control missing so App D team rewrites</td>
</tr>
<tr>
<td>SWQ-89</td>
<td>Microservices pattern misused, introducing overhead and latency</td>
</tr>
<tr>
<td>SWQ-88</td>
<td>Multiple sets of services for Case Management in heterogeneous ways</td>
</tr>
<tr>
<td>SWQ-85</td>
<td>There are two versions of C Application; features are duplicated in both</td>
</tr>
<tr>
<td>SWQ-74</td>
<td>Create Address Information Shared Service</td>
</tr>
<tr>
<td>SWQ-68</td>
<td>Project E moving business logic to SpringMVC PLSQL is copied &quot;as is&quot;</td>
</tr>
<tr>
<td>SWQ-66</td>
<td>Project A and C SpringMVC UI to data layer dependencies</td>
</tr>
<tr>
<td>SWQ-65</td>
<td>Enterprise authentication and access control partially finished</td>
</tr>
<tr>
<td>SWQ-64</td>
<td>A Shared Lookup Data Service partially finished</td>
</tr>
<tr>
<td>SWQ-58</td>
<td>Application A and D using a brittle shared database schema</td>
</tr>
</tbody>
</table>
Dashboard: Combined Risk Summary

The Combined Risk Summary is at the enterprise level and is a roll-up of open issues by design or code review activity.

<table>
<thead>
<tr>
<th>Projects</th>
<th>Design Risks</th>
<th>Code Risk</th>
<th>Technical Debt Item</th>
<th>TODO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project A</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Project B</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Project C</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Project D</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Project E</td>
<td>5</td>
<td>9</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Project F</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Project G</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Project H</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Project I</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Executive Management Affinity Tree

This view is useful for helping Executive Management prioritize investments

Each bullet is an item on the ETD List

For example, this chart makes clear there are a lot of issues under Data Exchanges and Service API, so this is an area for the organization to focus on
Continuous Process

Making enterprise technical debt visible is not a one-time exercise. It is part of a continuous process, and the architect is the driving force.

Diagram:
- Architecture Runway Components
- Phased Roadmap
- Ecosystem Action Decision Point
- Ecosystem TDI
- System Architecture
- Project Design Risks
- Project TDI
- Enterprise Impact
- Project Impact
- Decision Points
- Artifact
The More Evidence, the Stronger the Case

One of the architect guide’s jobs is to continuously look for more evidence in risk data. The more evidence, the stronger the case for taking action.

**ETD**
- **Status**
  - Proposed: Not sure if it is EDT (assumed evidence)
  - Defined: An open EDT item (clear evidence)
  - Resolved: Decision to pay back (it is hurting!)
  - In progress: Being resolved in one or more projects
  - Done

**Evidence**

1 ... *
Wrap-up

Summary
We discussed characteristics of ETD and provided several examples
We talked about the importance of data visibility and gave examples
We emphasized the architect guide’s role in continuously monitoring for ETD

Next Steps
We shared the ETD views with management for the first time in January
Encouraging outcome; ETD data gave us credibility when arguing for investments
Need to keep collecting data; more data will strengthen the case!
In the future, we plan to focus on tactics for proactive management of ETD
Contact Information

Felix Bachmann
fb@sei.cmu.edu

Stephany Bellomo
sbellomo@sei.cmu.edu