Technical Debt

Webinar
November 10, 2011

Carolyn Seaman
University of Maryland Baltimore County

Nico Zazworka
Fraunhofer Center for
Experimental Software Engineering

http://www.technicaldebt.umbc.edu/
Outline

• The TD Metaphor – and why TD needs to be managed and communicated
• Types of Debt
  • Design debt: Code smells, Grime, ASA issues, Modularity violations
  • Defect debt
  • Testing debt
  • Documentation debt
• Managing TD
  • The TD list
  • Metrics – principal and interest
  • Communicating debt
• Decision making
  • Basic decision scenario
  • Other decision strategies: Portfolio model, Options, AHP
Software Maintenance

• Large inventory of operational systems that need to be maintained
  • Fixed
  • Enhanced
  • Adapted
• Such systems need constant modification in order to remain useful
• Most such systems are too expensive to replace, so considerable resources go into their maintenance
• However, maintenance, even more than development, is characterized by tight budget and time constraints
Technical Debt

- Technical Debt is the gap between:
  - Making a change **perfectly**
    - Preserving architectural design
    - Employing good programming practices and standards
    - Updating the documentation
    - Testing thoroughly
  - And making the change **work**
    - As quickly as possible
    - With as few resources as possible
Everyday Indicators of Technical Debt

“Don’t worry about the documentation for now.”

“The only one who can change this code is Carl”

“It’s ok for now but we’ll refactor it later!”

“ToDo/FixMe: this should be fixed before release”

“Let’s just copy and paste this part.”

“Does anybody know where we store the database access password?”

“I know if I touch that code everything else breaks!”

“Let’s finish the testing in the next release.”

“The release is coming up, so just get it done!”
Technical Debt Metaphor

• A **metaphor**, NOT a theory or a scientific concept

**Definition**

• Incomplete, immature, or inadequate **artifact** in the software development lifecycle (Cunningham, 1992)

• Aspects of the software we **know are wrong**, but don’t have time to fix now

• Tasks that were left **undone**, but that run a **risk** of causing future problems if not completed

**Benefits**

• Higher software **productivity** in the current release

• Lower **cost** of current release

**Costs**

• “**Interest**” – increased maintenance costs

• **Risk** that the debt gets out of control

• Little scientific research, but

  • Discussions in blogs, forums, etc.

  • Strongly related to Risk Management
How Technical Debt Happens

The focus of our research
Wow, this module is really bad. It’s going to be very hard to make any changes to it.

Hey, Miriam, I think we should take some time to refactor this module in the next release.

Why would we do that? That would take a lot of time and effort.

But if we don’t refactor it soon, I have a gut feeling it’s going to cause major problems later.

OK, I’ll put in the plan for the next release.

David is pretty smart, and he’s usually right about these kinds of things.

David
Developer

Miriam
Manager
Wow, this module is really bad. It’s going to be very hard to make any changes to it.

Hey, Miriam, I think we should take some time to refactor this module in the next release.

Those developers always try to make their code perfect. I need some evidence that this is worth it.

What is the ROI of this refactoring?

RO...WHAT?!?

Let’s stick with implementing important features.
Potential Payoffs of Explicitly Managing TD

- Lowered maintenance costs
  - Avoiding “interest payments”
  - Avoiding unnecessary “perfecting” work
- Increased maintenance productivity
  - Better prioritization of tasks in each release
  - Maintenance always performed on code that is easier to work with
- Avoiding surprises
  - Fewer components that fail without warning
  - Fewer unexpectedly large over-budget maintenance tasks
  - Better estimation of the costs and risks of postponing maintenance tasks
Technical Debt in the Industry

- To our knowledge, no one is formally monitoring and managing TD explicitly.
- However, most practitioners we talk to understand the concept and have informal mechanisms for keeping an eye on debt:
  - A personal “TD list”
  - Backlogs
  - To-do lists
- TD-related issues go by many names:
  - risk management
  - code quality
  - conformance to standards
- Some TD may be unintentional, and may go undetected for some time, most is intentional – practitioners know when it is being incurred and why.
- The TD metaphor provides a way to unify all these related concepts in a tailored way.
Identifying Technical Debt
Identifying Design Debt

- Design debt — any kind of anomaly or imperfection that can be identified by examining source code and/or related documentation, that leads to decreased maintainability if not remedied

  - ASA issues (line level)
  - Code smells (method and class level)
  - Grime (class interaction level)
  - Modularity violations (architecture level)
ASA Issues

• ASA: Automatic Static (Code) Analysis
  • Tools that identify problems (e.g. potential bugs) on line level

1 Person person = aMap.get("bob");
2 if (person != null) {
3   person.updateAccessTime();
4 }
5 String name = person.getName();

• Well established stable tools for issue detection: FindBugs (Java), Resharper (C#)
• Studies show that these approaches produce a large number of false positives: often more than 1000 warnings that have to be inspected.
• Approach gaining significant traction in practice: used during Google Fixit events

Links: http://findbugs.sourceforge.net/
Code Smells

- Methods and classes that violate the principles of good object oriented design:
  - Clearly defined single responsibility
  - Encapsulation
  - Information hiding
  - Few and clear interfaces
  - Proper use of inheritance

- Code Smells point to potential problems: require investigation and final judgment by developer

- Set of 20 more or less formally defined Code Smells available
Code Smells Applied

- Automatic approaches have been proposed and implemented to automatically detect Code Smells in object oriented code
  - Based on Radu Marinescu’s Detection Strategies
  - For Java: CodeVizard and Marple
  - For C#: ReSharper, CodeRush, Gendarme, FxCop

- Example: **God Class**
  - Class with too many responsibilities and is more interested in data of other classes than its own.

Further reading: [http://www.codevizard.com](http://www.codevizard.com)
Code Smell Research Findings

• Studies indicate that automatic detection works well:
  • Low number of false positives and false negatives
• Research focus: God Classes (concept is easy to understand)
  • Require higher maintenance attention (TD interest): God Classes are 5-7 times more change prone
  • Require essential rework (TD interest): God Classes are 4-17 times more defect prone
• Baseline from our experience: most systems have 2%-8% God Classes
• Recent studies show that Dispersed Coupling code smell points to defect and maintenance prone classes
  • Dispersedly coupled classes access methods from many other classes
• Take away message: from the larger set of code smells Dispersed Coupling and God Classes seem to have the greatest negative maintenance impact (highest TD interest)
Design Patterns and Grime

- Design patterns promise code to be more maintainable and less defect prone
  - Describe how multiple classes work together
- Design patterns can decay over time as systems evolve
- **Grime**: accumulation of non-pattern code in classes following a design pattern
  - **Rot**: changes that break the integrity of a design pattern
- Research results indicate that often grime goes along with increased coupling
  - Grime buildup decreases the testability of code
Modularity Violations

• Organization of software software systems: **inter-dependent modules**
  • Proper architecture leading to a clear structure of relationships promotes **reuse** of modules and **smaller ripple effects**.

• Dependencies indicate how modules should change together:
  • Example: If the Model is changed Controller A and Controller B might require changes.

• Modularity Violations: recurring changes on classes within modules that are **not depending on each other**:
  • Example: Classes in View 1 and View 3 changing together
Modularity Violations Research

- Studies have shown that modularity violations are an excellent indicator of defect prone classes.
- Tools: CLIO (Drexel University)

<table>
<thead>
<tr>
<th></th>
<th>Multithread Correctness</th>
<th>Dispersed Coupling</th>
<th>God Class</th>
<th>Modularity Violations</th>
<th>All 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Avg. prec.</strong></td>
<td>0.78</td>
<td>0.81</td>
<td>0.96</td>
<td>0.85</td>
<td>0.77</td>
</tr>
<tr>
<td><strong>Avg. recall</strong></td>
<td>0.13</td>
<td>0.12</td>
<td>0.11</td>
<td>0.21</td>
<td>0.33</td>
</tr>
</tbody>
</table>

Precision and Recall for predicting defect prone classes in Hadoop:
85% of classes having modularity violations are among defect prone classes.
21% of all defect prone classes have a modularity violation.

Other types of debt
Testing Debt

• Tests that were planned but:
  • not implemented
  • not executed
  • or they got lost

“There’s no tests for buttons other than `<input type="submit">` yet. I’m pretty sure also `<input type="button">` works if other `<input>`s work, but `<button disabled="disabled">Text</button>` should be tested separately.”

http://code.google.com/p/robotframework-seleniumlibrary/issues/detail?id=163

“While updating the package of html5lib to 0.90 in Debian I realized that the unit tests are gone. To ensure the keep the package in a good working shape while it transitions through new Python versions and new versions of the modules it depends on, it would be *very* appreciated if the unit tests would be shipped in the zipfile again.”

Documentation Debt

- Documentation that is not kept up-to-date, e.g.
  - Installations and run instructions
  - Architecture documentation
  - Requirements and use case documentation
  - API documentation

“Except there is no such class or field in the SDK. It is outdated documentation that definitely needs to be updated.”
http://code.google.com/p/android/issues/detail?id=8483

“There is not much documentation available regarding the format of .xclangspec files. As a starting point, see for instance the outdated documentation at:
http://maxao.free.fr/xcode-plugin-interface/specifications.html”
http://code.google.com/p/go/source/browse/misc/xcode/go.xclangspec?r=30b0c392132645259e053a2ba8904383a55bab03

“This was apparently the old behavior and it's changed now, but the documentation doesn't so say.”
http://code.google.com/p/redis/issues/detail?id=514
Defect Debt

- Known defects that are not yet fixed
  - Low priority defects
  - Low severity defects
  - Manifest rarely
  - Workarounds present

“There are a couple of latent bugs in the linux TLS implementation. I'm filing a single issue because they are so small and easy to fix.”

http://code.google.com/p/dynamorio/issues/detail?id=358
Managing Technical Debt
Managing Technical Debt

• An Initial Technical Debt Management Framework

TD List

- TD Identification
- TD Estimation
- Decision Making
Technical Debt List

- A list of **TD Items**
  - Tasks that were left undone, but that run a risk of causing future problems if not completed.
  - Examples: Components/modules/classes that need refactoring, testing that needs to be done, etc.

- **Content of TD Item**
  - **Description** – what, where, why?
  - **Principal** – how much will it cost to do the work?
  - **Interest** – what happens if we don’t do this work?
    - **Amount** – amount of extra work if this causes problems later
    - **Probability** – probability that this will cause future problems

- **TD List Update Policy**
  - The TD list should be reviewed **after each release**, when items should be added as well as removed.
## TD Item Example

<table>
<thead>
<tr>
<th>ID</th>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>3/31/2008 (Release 3.2)</td>
</tr>
<tr>
<td>Responsible</td>
<td>Joe Developer</td>
</tr>
<tr>
<td>Type</td>
<td>Design</td>
</tr>
<tr>
<td>Location</td>
<td>Method m in Module X</td>
</tr>
<tr>
<td>Description</td>
<td>In the last release, method m was added quickly and is thread-unsafe.</td>
</tr>
<tr>
<td>Estimated principal</td>
<td>Medium (medium level of effort to modify m)</td>
</tr>
<tr>
<td>Estimated interest amount:</td>
<td>High (if we wait to modify m, there might be more dependent modules that need to be modified)</td>
</tr>
<tr>
<td>Estimated interest probability</td>
<td>Low (not likely to be adding simultaneous calls to m)</td>
</tr>
</tbody>
</table>
TD Attributes

- Three attributes of a TD item
  - Principal
  - Interest probability
  - Interest amount

- Start with a rough estimate of the attribute values
  - High, Medium, Low

- Defer more precise estimation until data is available
  - Fault detection ability and defect density => testing debt
  - Cost of fixing a defect pre-release & post-release => defect debt
  - Time and effort for updating documentation => documentation debt
TD Attribute Estimation

• **Principal** => historical effort data
• **Interest Probability** => historical usage, change, and defect data
  • Example questions
    • How likely is it that a defect will occur in the untested part?
    • How likely is it that code containing a known error will be exercised?
  • A time element
• **Interest amount**
  • Assume that the item has an effect on future work
  • Example questions
    • How much more it will cost to deal with defects later in the system’s lifetime than in testing?
• These are all hard to estimate with any certainty
• Historical data will help
• Any estimation is better than the current method – “gut feeling”
Decision Making Scenario

• Question
  • When and which technical debt items should be paid?

• Example
  • Significant work is planned for component X in the next release, should we pay down some debt on component X at the same time?

• Assumptions
  • There is an up-to-date TD list that is sorted by component and has high, medium, and low values for principal and interest estimates for each item.

• Process

  principal for all remaining TD items related to component X

  [Flowchart: Select → Re-evaluate → Estimate → Compare → Add up]
Other Decision Models

- The proposed TD management strategy is based on a simple cost/benefit analysis
- But TD occurs in complicated development and business scenarios
  - TD items are inter-related
  - Business factors are important, too
  - Prediction is hard
- Other strategies for making decisions might be appropriate
  - Portfolio model
  - Options
  - AHP
Portfolio Approach

- Portfolio
  - Combination of different types of assets
  - Risk reduction strategy
  - Decision making process
    - Determining the types and amounts of assets
- Modern Portfolio Theory
  - Mathematical model of the diversification problem
- Application to TD Management
  - TD item -> Asset
  - Principal – interest (net benefit) -> Asset return
  - Interest standard deviation -> risk of asset return
- Process
Options

- Paying off technical debt can be seen as an investment decision
  - relatively certain short-term costs—
  - more uncertain longer-term benefits
  - so timing matters
- Black-Scholes valuation formula
  - has been used to evaluate architectural investments (e.g. refactoring)
  - difficult to estimate key parameters
- Baldwin and Clark’s Net Option technique
  - similar but uses statistical methods to calculate parameters
  - key parameter is called the technical potential of a module, which measures how likely the module will be changed
Analytic Hierarchical Process

• Pairwise comparison of alternatives (i.e. TD items)
  • By experts
  • Based on defined, but subjective criteria
• Output is a ranking of items based on an aggregation of the pairwise comparisons
• In the TD management scenario, criteria are mixed
  • Comparisons based on quantitative criteria (e.g. cost, benefit) can be done automatically
  • Comparisons based on qualitative and/or subjective criteria (e.g. business value) can be done with human intervention.
Technical Debt Management - Recap

- **Technical Debt List**
  - Central artifact
  - List of TD items, each of which captures a task left undone

- **Technical Debt Identification**
  - Design, Defect, Documentation, Testing Debt
  - Source code analysis tools

- **Estimation and measurement**
  - Principle, interest amount, interest probability

- **Decision making**
  - Using the TD list during release planning
So where are we?
Summary

• Technical Debt is a metaphor that describes a very real phenomenon
• Provides a way to talk and reason about the difficulties of software maintenance
• Technical Debt comes in a variety of forms, all of which can be detected in different ways
• Technical Debt can be documented and tracked in a TD list
• TD Attributes like principal and interest can be estimated at various levels of precision as needed
  • Expert estimation
  • Historical data
• Ultimately, the value of explicitly managing technical debt is in better decision making
What’s next...

• **For us:**
  • Future research
    • Refining TD management mechanisms and measurement
    • Exploring source code analysis strategies
    • Extending the model to decisions about incurring TD
  • Collaboration
    • Industrial collaboration is key to research in this area
    • Data, sites for experiments, feedback

• **For you:**
  • Start using tools for source code analysis – some commercial tools available, others in research environments and can be requested
  • Start a TD list – can use our template - probably some developers already have one
  • Explore some types of TD retrospectively – what’s really a problem?
  • Refine release planning process to incorporate TD
Contact us...

Carolyn Seaman
cseaman@umbc.edu

Nico Zazworka
nzazworka@fc-md.umd.edu

or, find us on the web:
http://www.technicaldebt.umbc.edu/
Acknowledgments

• UMBC
  • Yuepu Guo, Ph.D. candidate
• Fraunhofer Center
  • Forrest Shull, Division Director
  • Antonio Vetró, Ph.D. student
  • Many past, current, and future interns
• Other academic collaborators
  • Yuanfang Cai, Drexel University
  • Clemente Izurieta, Montana State University
  • Fabio da Silva, Universidade Federal de Pernambuco, Brazil
  • Sunny Wong, now at Siemens Healthcare
• Industrial collaborators
Tools for TD Detection

• ASA Issues:
  • FindBugs: http://findbugs.sourceforge.net/
  • Resharper: http://www.jetbrains.com/resharper/

• Code Smells
  • CodeVizard: http://www.codevizard.net
  • Marple: http://essere.disco.unimib.it/reverse/CodeSmellDetector.html
  • CodeRush: http://devexpress.com/Products/Visual_Studio_Add-in/Coding_Assistance/
References


References


