Measuring and Monitoring Technical Debt

USP, 27 March 2013

Carolyn Seaman
cseaman@umbc.edu
University of Maryland Baltimore County
Fraunhofer Center for Experimental Software Engineering
Visiting Researcher – UFPE Center for Informatics

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

Fraunhofer USA
Center for Experimental Software Engineering

information systems UMBC
AN HONORS UNIVERSITY IN MARYLAND

Outline

- **Introduction** to the Technical Debt metaphor
  - Definitions in the literature and beyond
  - Examples of everyday debt
- Technical Debt **Framework**
  - Technical Debt properties: principal vs. interest
  - Recording and communicating Technical Debt
- **Identifying** important Technical Debt
  - Design debt: Code smells, Grime, ASA issues, Modularity violations
  - Other types of Technical Debt
- **Management strategies** to pay down Technical Debt
- Open **research questions**
- Research and practice outlook

What is Technical Debt?

- Context: 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 maintenance 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

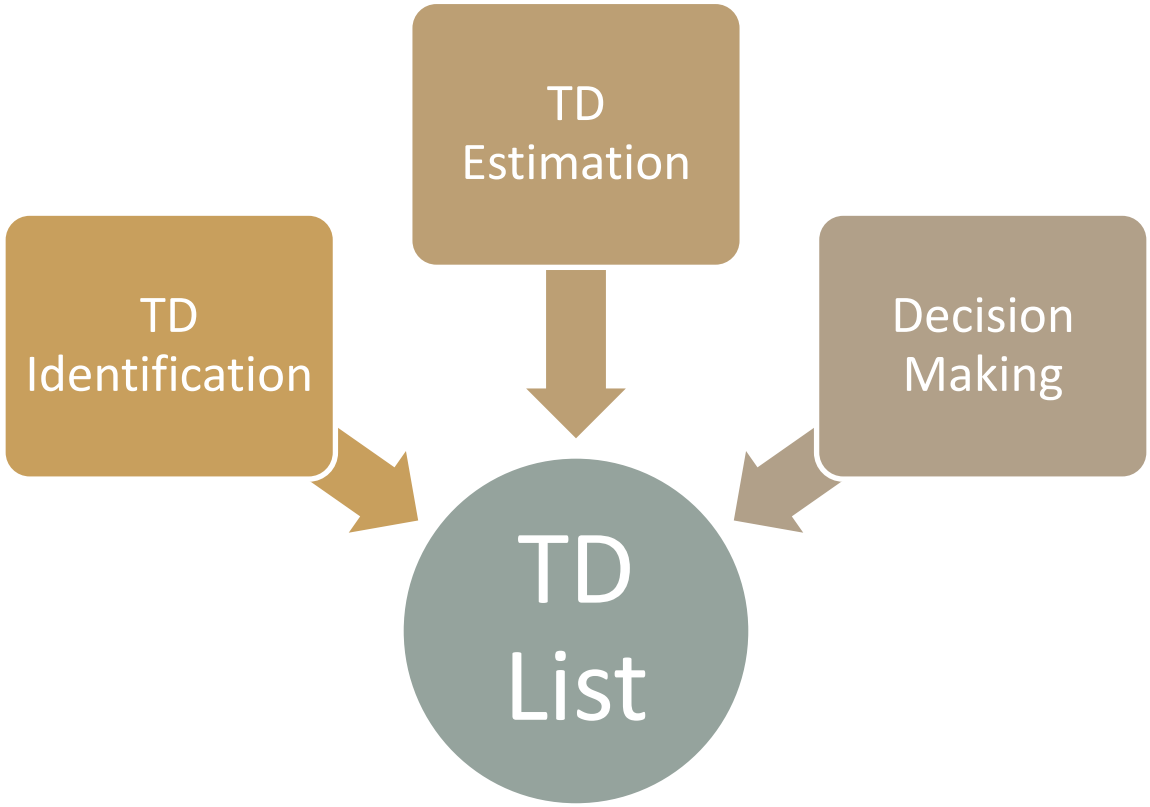
Potential Payoffs of Explicitly Managing TD

- 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

Nothing new…

- Technical debt is **not** a new phenomenon
  - Related to
    - Software decay, software aging - Belady and Lehman
    - Risk management – Boehm, etc.
    - Software quality research
- The metaphor, however, **provides**:
  - A way to apply years of research in
    - Architecture
    - Software metrics
    - Software quality
    - Software risk management
  - A new way to **talk** about maintenance issues
    - Intuitively appealing to practitioners
  - **Inspiration** for a host of potential new solutions
    - Finance is a mature domain with lots of tools to try

An Initial Technical Debt Management Framework



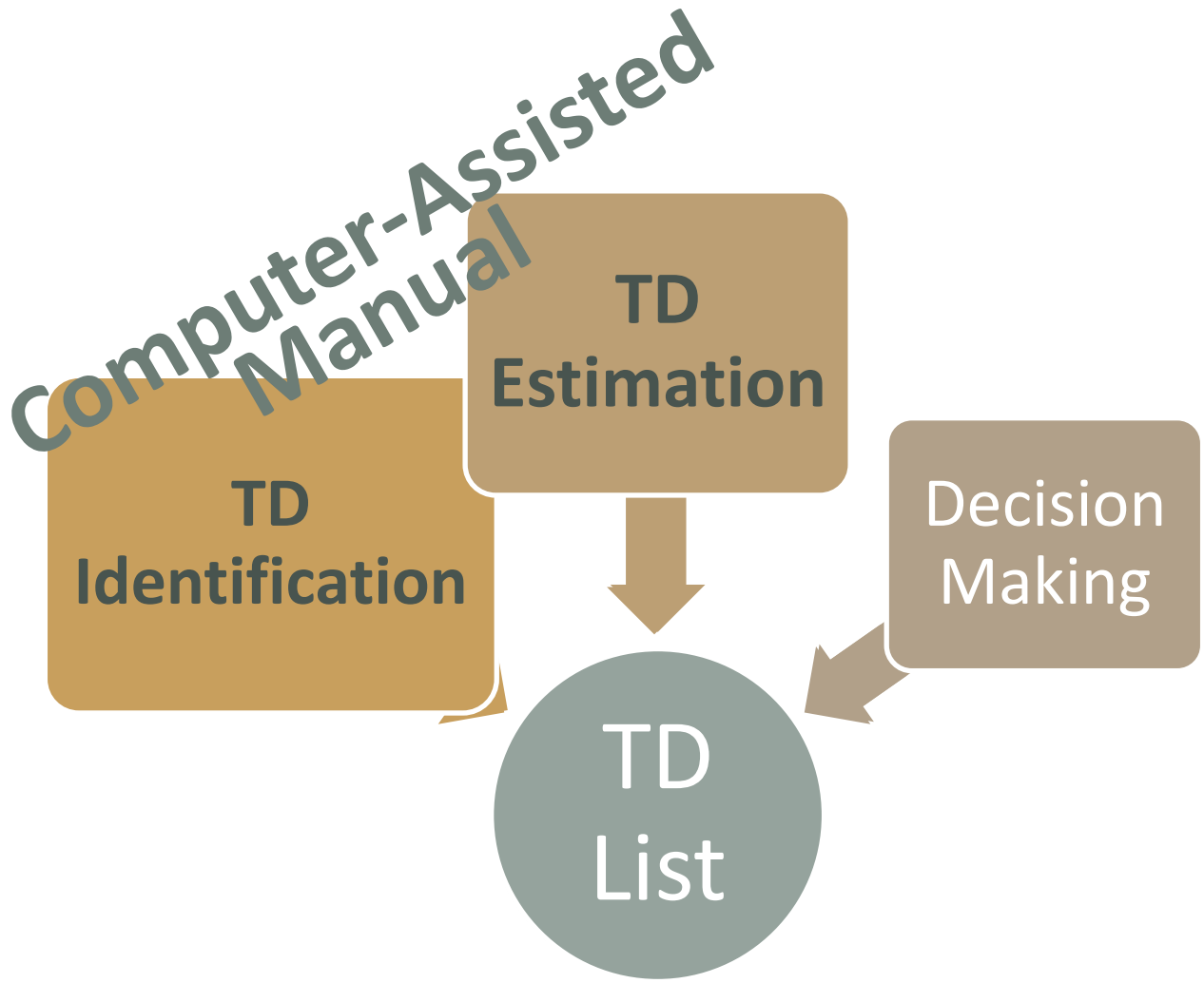
Technical Debt List

- 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

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

Identifying Technical Debt

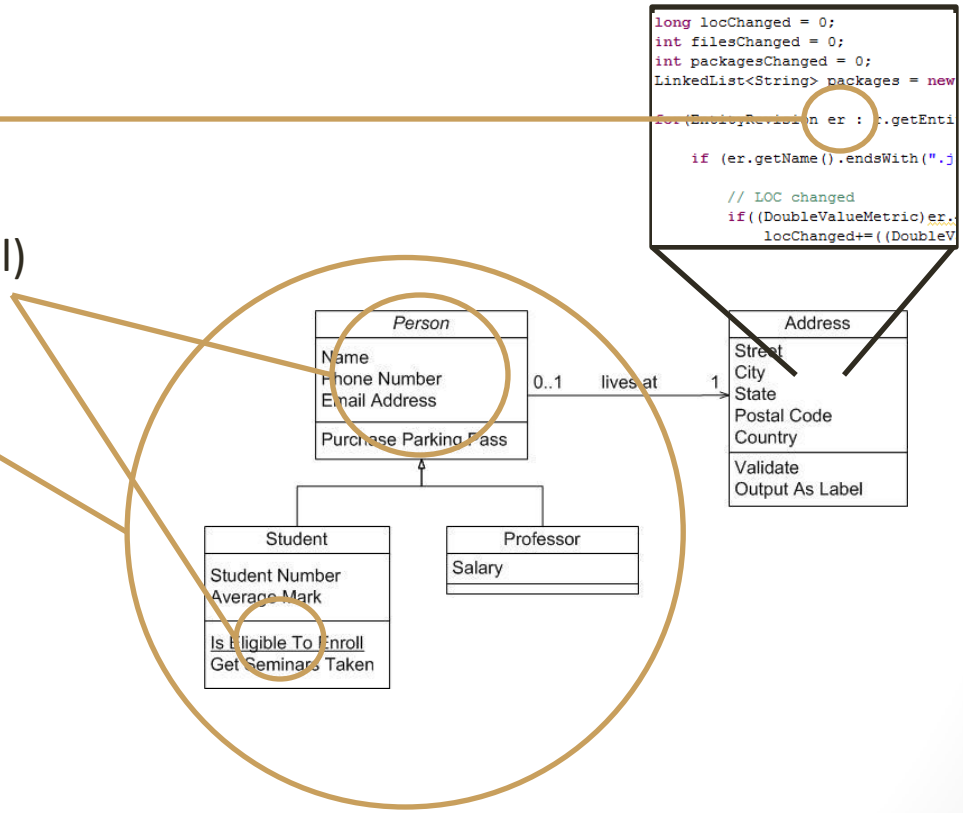


Types of Technical Debt

- **Design/code debt** – can be identified by examining source code and/or related documentation
- **Testing debt** – planned tests that were not run, or known deficiencies in the test suite (e.g. low code coverage)
- **Documentation debt** – missing or inadequate documentation of any type
- **Defect debt** – known defects that are not fixed
- **Infrastructure debt** - delayed upgrade decisions

Identifying Design Debt

- 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
  - Identify problems on line level:


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

Potential Null Pointer Exception

- Inexpensive
- Point to the problem, suggest solution
- Downside: Many (thousands) issues identified
  - Many are false positives, but interest is negligible
- Gaining significant traction in practice:
  - Used by Google to identify problems
  - Google Fixit Event
- Recent research results:
  - **Multithread correctness** and **Correctness** issues are located in classes with higher **defect-proneness**

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

Code Smells

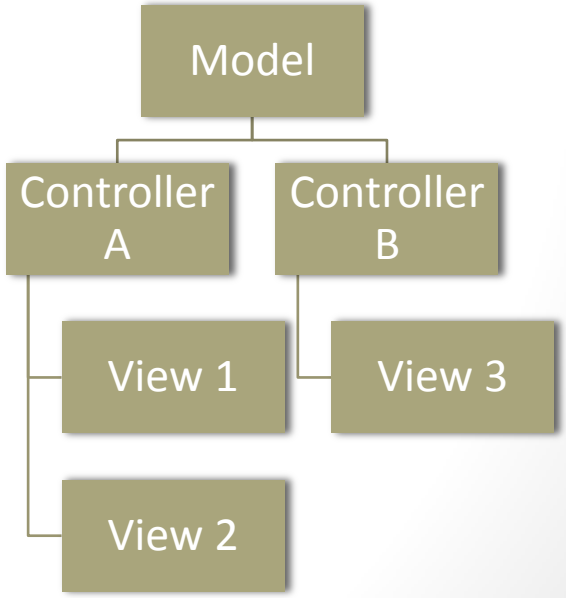
- Methods and classes that violate the principles of good object oriented design, e.g.:
  - Clearly defined single responsibility
  - Encapsulation
  - Proper use of inheritance
- Like ASA warnings, Code Smells point to *potential* problems
- Set of 20 more or less formally defined Code Smells
  - Tools and detection strategies available
- Research focus: **God Classes** (concept is easy to understand)
  - God Classes are **5-7 times more change prone**
  - God Classes are **4-17 times more defect prone**
  - Baseline from our experience: most systems have **2%-8%** God Classes
- **Dispersed Coupling** code smell points to defect and maintenance prone classes

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
- Early results show that grime has a noticeable effect on testability
  - As grime builds up, more test cases break
  - In turn affects productivity during the testing phase
  - Leads to testing debt
- Grime is often related to increased coupling

Modularity Violations

- Organization of 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 and change prone classes.
- Tool: CLIO (Drexel University)
- When applied, with other TD detection approaches, to an open source system, the results for predicting defects were:

	Multithread Correctness	Dispersed Coupling	God Class	Modularity Violations	All 4
Avg. prec.	0.78	0.81	0.96	0.85	0.77
Avg. recall	0.13	0.12	0.11	0.21	0.33

- Also, modularity violations were highly correlated with modules that developers later chose to refactor

Further reading: http://www.slideshare.net/miryung/icse-2011-research-paper-on-modularity-violations

Manual TD Detection

- Asked developers to **manually** report TD items
  - "If you had a week to do nothing but improve the maintainability of the software product, what would you work on?"
- Ran ASA, code smell detection, and metrics **tools**
- Are developers concerned about the same sorts of technical debt that is found and reported by tools?
  - Answer: **Yes and no**
- Details
  - Analysis tools found most of the modules that had developer-identified **defect debt** and about half of the modules that had developer-identified **design debt**.
  - But the tools also found **lots** of problems in modules that the developers did not care about
  - Not surprisingly, the tools could not find **testing or documentation** debt, although developers found these types of debt **important**

Testing Debt

- Tests that were planned but:
  - not implemented
  - not executed
  - or they got lost
- Inadequate tests
  - Test cases not updated for new/changed functionality
  - Low coverage
- Can be detected by:
  - Comparing test results with test plans
  - Code coverage tools
  - Comparing requirements changes with test suite changes

"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 trough 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."

http://code.google.com/p/html5lib/issues/detail?id=134&colspec=ID%20Type%20Status%20Priority%20Milestone%20Owner%20Summary%20Port

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
- Can be detected by:
  - Comparing code changes with documentation changes
  - Comment density metrics

"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
- Can be detected by:
  - Examining defect repositories
  - Test results

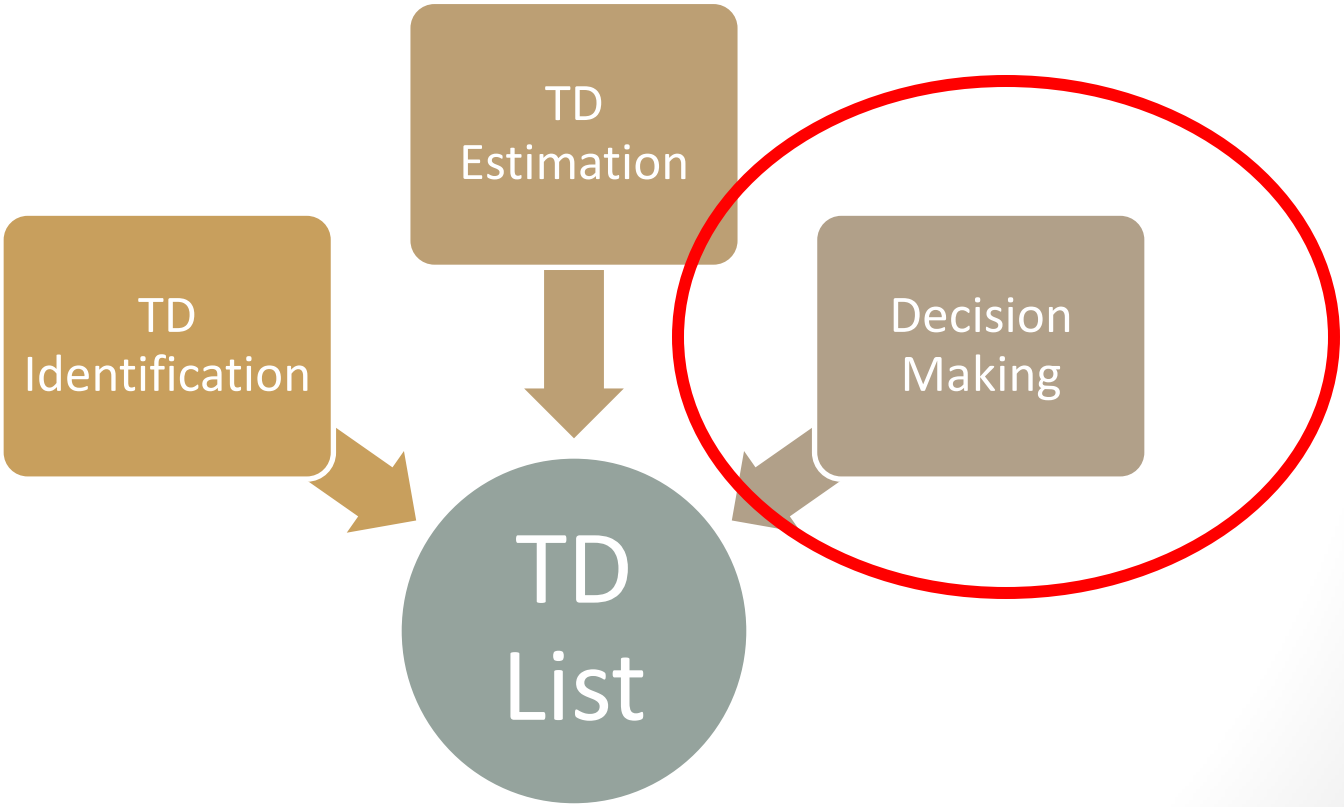
"There are **a couple of latent bugs** in the linux TLS implementation. I'm filing a single issue because they are so small and easy to fix."

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

Bottom line for detecting TD

- **Different** techniques detect **different** instances and types of technical debt
  - No one approach is sufficient by itself
  - No one approach is the right one for everyone
- The solution is a **strategy** based on
  - Business and development goals
  - Most painful types of debt
  - A **combination** of approaches that focus on the most pain
- Don't try to automate it all
  - Some kinds of technical debt can only be **detected by humans**
  - Most kinds of technical debt can only be **interpreted by humans**
  - No substitute for **talking** about it

Technical Debt in Decision Making



Simple Cost/Benefit Approach

- 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 absolutely necessary – use historical data when possible:
  - 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
- These are all hard to estimate with any certainty
- Historical data will help
- Any estimation is better than the current method – "gut feeling"

Example Scenario

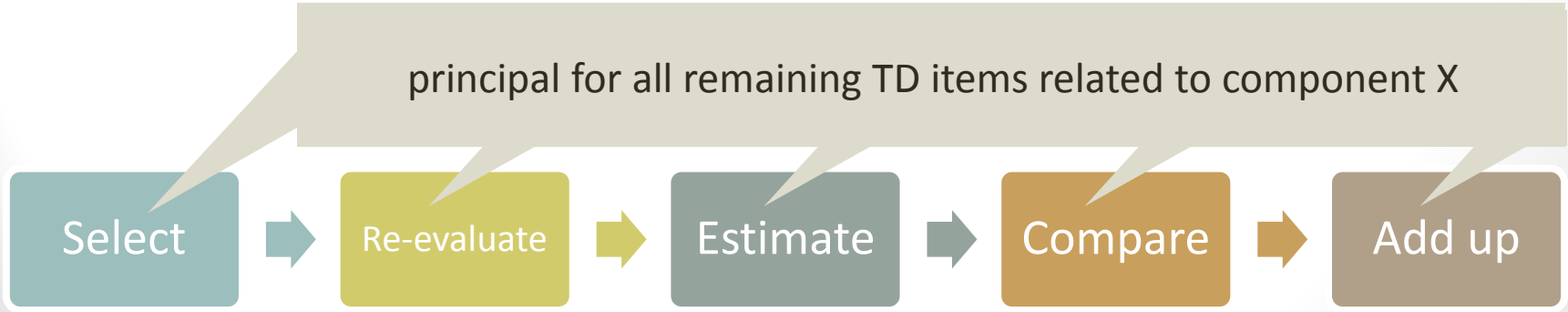
- Technical Debt item: One of your code modules is in need of refactoring
  - TD Principal: Refactoring the entire module will cost $10,000
- From historical data you have established that:
  - This module is modified in 75% of all releases
  - The cost of changing this module has gone up 10% each time it's been changed over its last 5 changes:
    - 5 changes ago cost $10,000
    - Last change cost almost $15,000

Example Scenario (cont.)

- Technical Debt Computation
- For the next release
  - Principal for paying off debt: refactoring the module costs **$10,000**
  - Interest:
    - Cost of the next change to the module
      - If refactored first: $10,000
      - If not refactored first: $16,000
      - Extra cost if not refactored: about $6000
    - Interest avoided = interest amount * interest probability
      - $6,000 * .75 = $4500

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



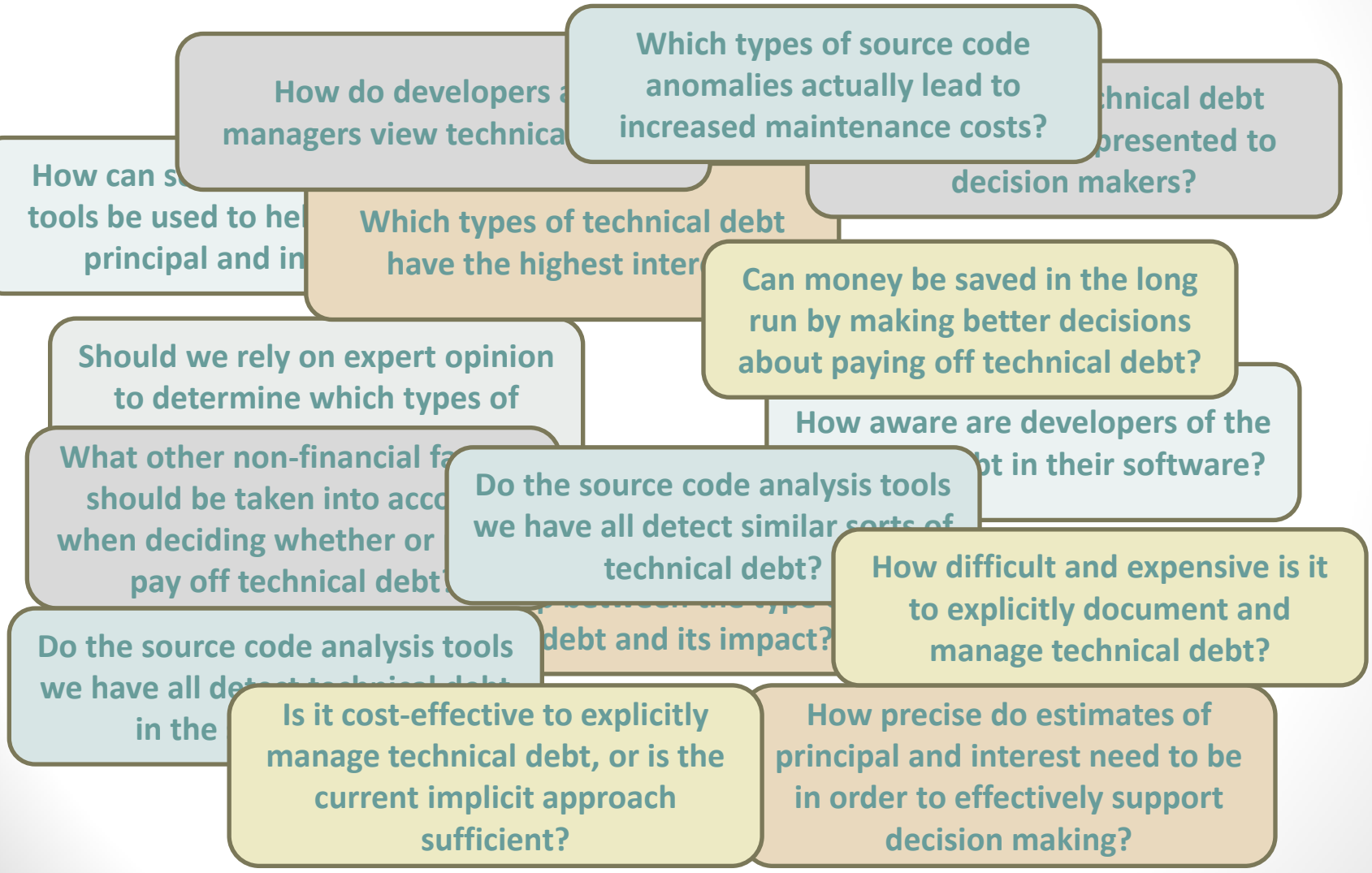
Limitations of the simple approach

- **VERY** simplistic
- Does not consider
  - **Non-financial** considerations
  - **Relationships** between TD items
- Relies on ability to **estimate** based on **historical** data
  - Better estimates come from better data
  - BUT even **good guesses** will work
- Good place to **start**
  - Serves as a **baseline** from which more sophisticated solutions can be derived

Proposed approaches to TD management and decision making

- Our proposed approach
  - Simplistic cost-benefit analysis
- Models from finance and other domains
  - Portfolio management
  - Options
  - AHP
- SQUALE
- Nugroho et al.

Open Research Questions



Proposed study designs (1)

- **Do different source code analysis techniques reveal the same or different instances of technical debt?**
- **Procedure**
  - Apply two or more analysis techniques to the same code base
  - Compare the outputs to see if the same modules are indicated as anomalous by multiple techniques
  - Optional: get feedback from the developers as to which tools did a better job of detecting “important” anomalies
- **Requires**
  - An industrial **project** willing to run tools on their code base and share the output (and, optionally, access to developers) or **OSS**
  - A **student** who can install, configure, and run the tools and run statistical analysis on the outcomes

Proposed study designs (2)

- **How useful are source code analysis techniques in helping to identify technical debt items and quantify their properties?**
- **Procedure**
  - Apply an analysis technique to a code base
  - Focus group with developers:
    - How would you use the output of the analysis to manage technical debt? including
      - How could you quantify the debt that is indicated by the analysis?
      - How would you decide when (or if) to pay off the debt?
      - What are the consequences of the debt likely to be?
- **Requires**
  - An industrial **project** willing to run tools on their code base and share the output, as well as access to developers for a focus group
  - A **student** who can install, configure, and run the tools and run a focus group

Proposed study designs (3)

- **What are the concrete negative effects of technical debt on software quality?**
- **Procedure**
  - Produce two versions of a software module, a “clean” version and a version that contains some type of technical debt
  - Two groups of subjects, one using the “clean” version and the other using the debt-ridden version, perform the same maintenance task
  - Data is collected on maintenance effort, difficulty, predictability, and resulting quality and compared between the two groups
- **Requires**
  - A programming **course instructor** willing to use his/her class for this study
  - A **student** who can run the study and analyze the data

Proposed study designs (4)

- **Does explicitly considering technical debt in decision making have an impact on maintenance cost?**
- **Procedure**
  - Collect detailed historical effort, change, defect, test, and release planning data on several past releases of a software product
  - Mine the data for instances of rework that could indicate technical debt being paid off (e.g. refactoring effort)
  - Simulate the effect of NOT carrying out that rework on future maintenance
  - Demonstrate the impact (or lack thereof) of paying off the debt.
  - Interview developers to verify the constructed history of the technical debt
- **Requires**
  - An industrial **project** with extensive historical data they are willing to share, and access to some developers
  - A **student** who can collect and mine the historical data

Proposed study designs (5)

- **What is the best way for a development team to integrate technical debt management into their process?**
- **Procedure**
  - Orient and train an existing development team that is maintaining a product already in use in technical debt concepts, and in one of the proposed technical debt management approaches (e.g. the simplistic approach presented earlier)
  - Monitor the team as they explicitly track and manage technical debt over 3-4 sprints or releases
  - Collect data on how long technical debt management activities take, what decisions are made, and any problems encountered
  - Interview decision makers after several releases to gather issues and improvement suggestions and to ask if they want to continue explicitly managing technical debt
- **Requires**
  - An industrial **project** willing to try something new, and access to some developers
  - A **student** who can carry out all aspects of the case study

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
- The ultimate aim of **managing** technical debt is to be able to improve **decision making**
- The types of Technical Debt that are relevant for a particular situation depends on past experience, organizational culture, and business environment, i.e. an organization's **pain points**.
- While the research is still early, it does provide some guidance as to Technical Debt identification **strategy**.
- There are many **open research questions**
- There are lots of **studies** waiting to be done

First steps towards tracking TD…

- Identify your pain points
- Decide what types of Technical Debt are relevant for you
- Choose a small set of tools and indicators
- Start a TD list – can use our template - probably some developers already have one
- Track the history of the TD items revealed by the tools to see if they are detecting “real” debt
- Refine release planning process to incorporate TD
- Track your success in reducing the “pain”
- Add new detection strategies to fill the gaps
- Call us if you need help
- Tell us how it’s going!

Thank you!!

- To Graziela and Alfredo
  - For inviting and hosting me
- To USP
  - For sharing your work and your space with me
- To CNPq and the Brazilian taxpayers
  - For funding my sabbatical
- To you
  - For being a great audience!

Questions?

- Carolyn Seaman
- cseaman@umbc.edu
- carolyn.seaman