Technical Debt:

At the Intersection of Decades of Empirical Software Engineering Research

Carolyn Seaman

University of Maryland Baltimore County
Fraunhofer Center for Experimental Software Engineering
Universidade Federal de Pernambuco

ESELAW 2013
Montevideo, Uruguay
10 April 2013

Fraunhofer USA
Center for Experimental Software Engineering

information systems UMBC

Thesis

"It is only a little hyperbolic to call this a watershed moment for empirical [software engineering] study, where many areas of progress are coming to a head at the same time."

Forrest Shull, Davide Falessi, Carolyn Seaman, Madeline Diep, and Lucas Layman. "Technical Debt: Showing the Way for Better Transfer of Empirical Results." Forthcoming in "Future of Software Engineering" published in honor of the 60th birthday of Prof. Dr. H. Dieter Rombach, 2013.

Outline

- Introduction to the Technical Debt metaphor
- Contributing streams of research
  - Evolution
  - Current state
  - Role in Technical Debt research
- A solution to our problems?
  - Technology transfer
  - Evolving the discipline
- Conclusion
  - Call to action
  - A watershed, indeed?

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

Technical Debt Identification

- Different types of Technical Debt
  - Design debt
  - Testing debt
  - Defect debt
  - Others…
- Some debt is easy to find, some is not
  - Easy:
    - Test cases that weren't run
    - Defects found but not fixed
    - Classes that everyone knows are a mess
  - Hard:
    - Code that gradually decays over time
    - Breakdown of design patterns
    - Code that is so complex only one person ever works with it

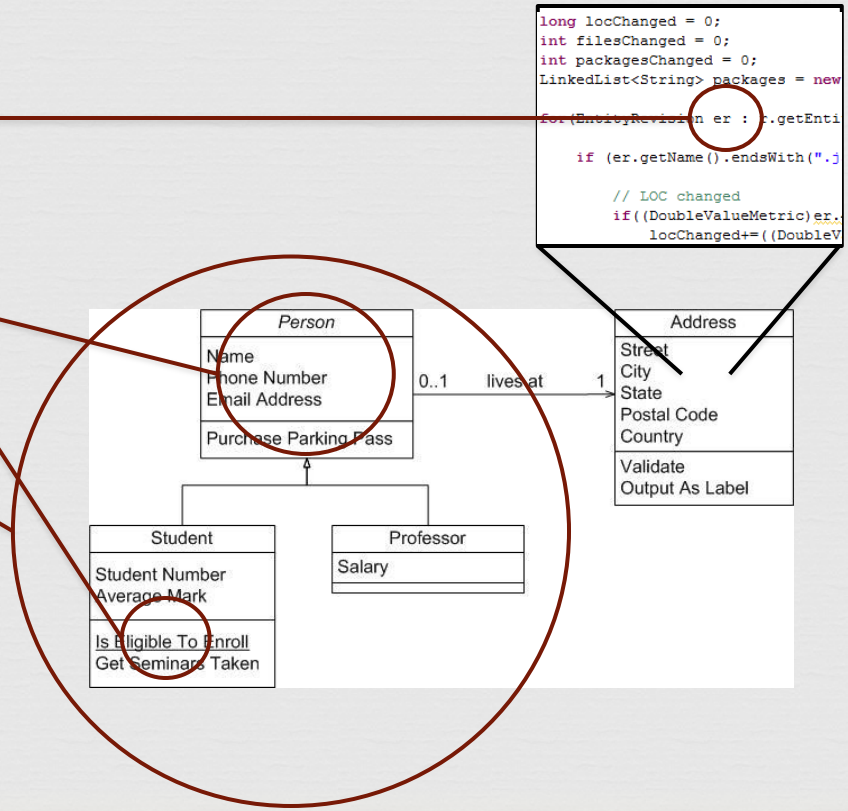
Research on Identifying Design Debt

ASA issues
(line level)

Code smells
(method and class level)

Grime
(class interaction level)

Modularity violations
(architecture level)



Research on Identifying Important Debt

- Which is better at finding the most important debt, tools or people?
- 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**

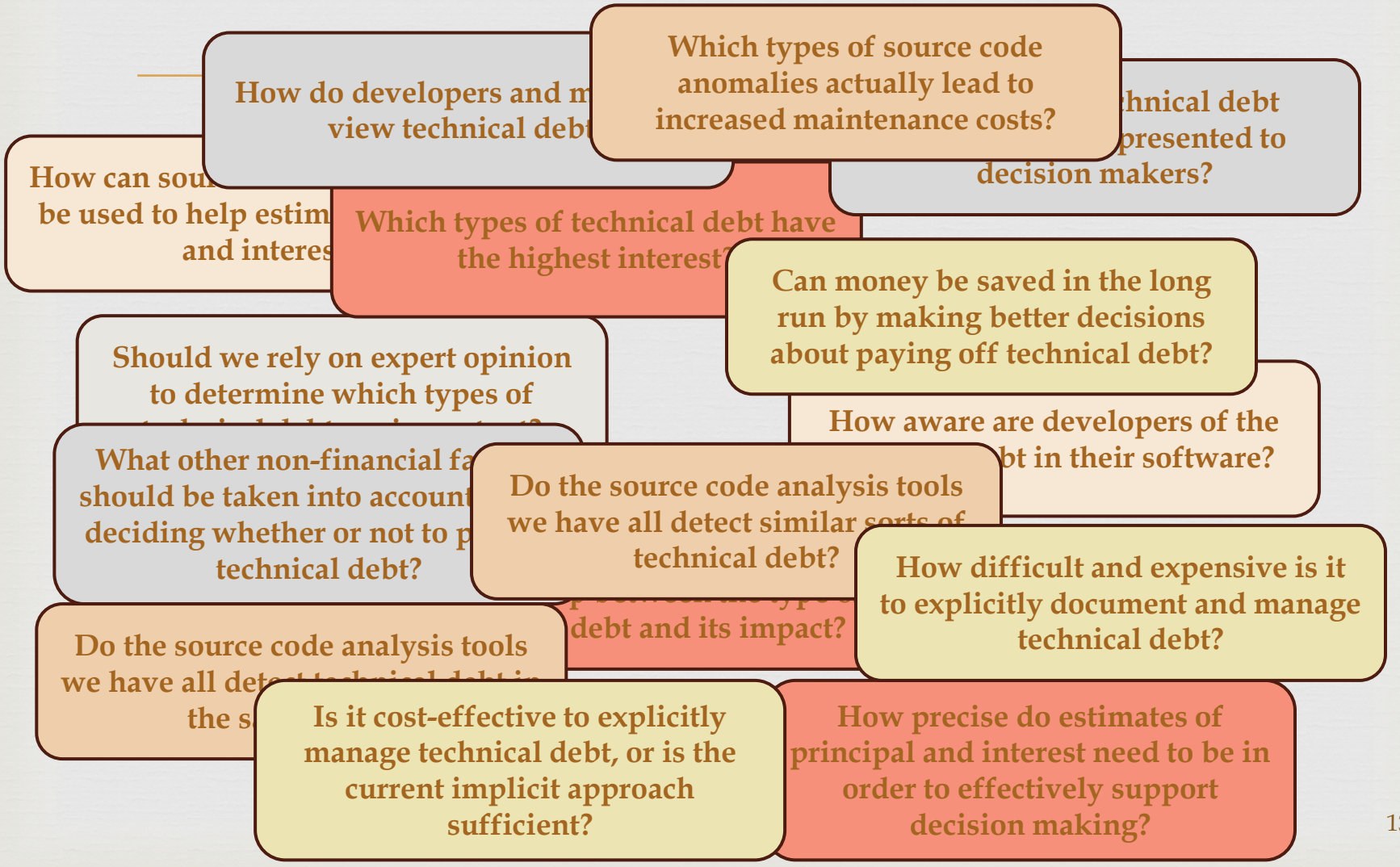
Technical Debt Management

- Managing Technical Debt, once it is identified, includes:
  - Evaluating principal and interest
  - Monitoring changes in debt (individual and collective)
  - Making decisions about debt
- Simplest possible approach: cost-benefit analysis
  - Principal = cost of paying off an instance of debt
  - Interest = benefit of paying off an instance of debt
  - Pay off the debt whose interest outweighs the principal
- Too simple
  - Lots of simplifying assumptions
  - A good place to start

Research on Technical Debt Management

- Several ongoing case studies
- Retrospective studies
  - Use historical data to simulate various decision outcomes
  - Calculate the benefits of making decisions based on information about Technical Debt
- Live studies
  - Projects try the simple approach
  - We collect data on effort and problems
  - Determine the costs of explicitly managing Technical Debt
  - Determine where the approach is too simple

Open Research Questions



Contributing Streams of Research

- Software aging and decay
- Risk management
- Qualitative methods and appreciation for context
- Software metrics
- Program analysis
- Software quality

Software Aging and Decay

- Foundational
  - Technical Debt is in some ways just a restatement of these ideas
- Another metaphor
  - Like human aging
  - Changing the software becomes harder as it evolves
- Results
  - Inability to keep up
  - Reduced performance
  - Decreased quality
- Lehman's Law of Increasing Complexity:
  - Complexity increases unless work is done to maintain or reduce it

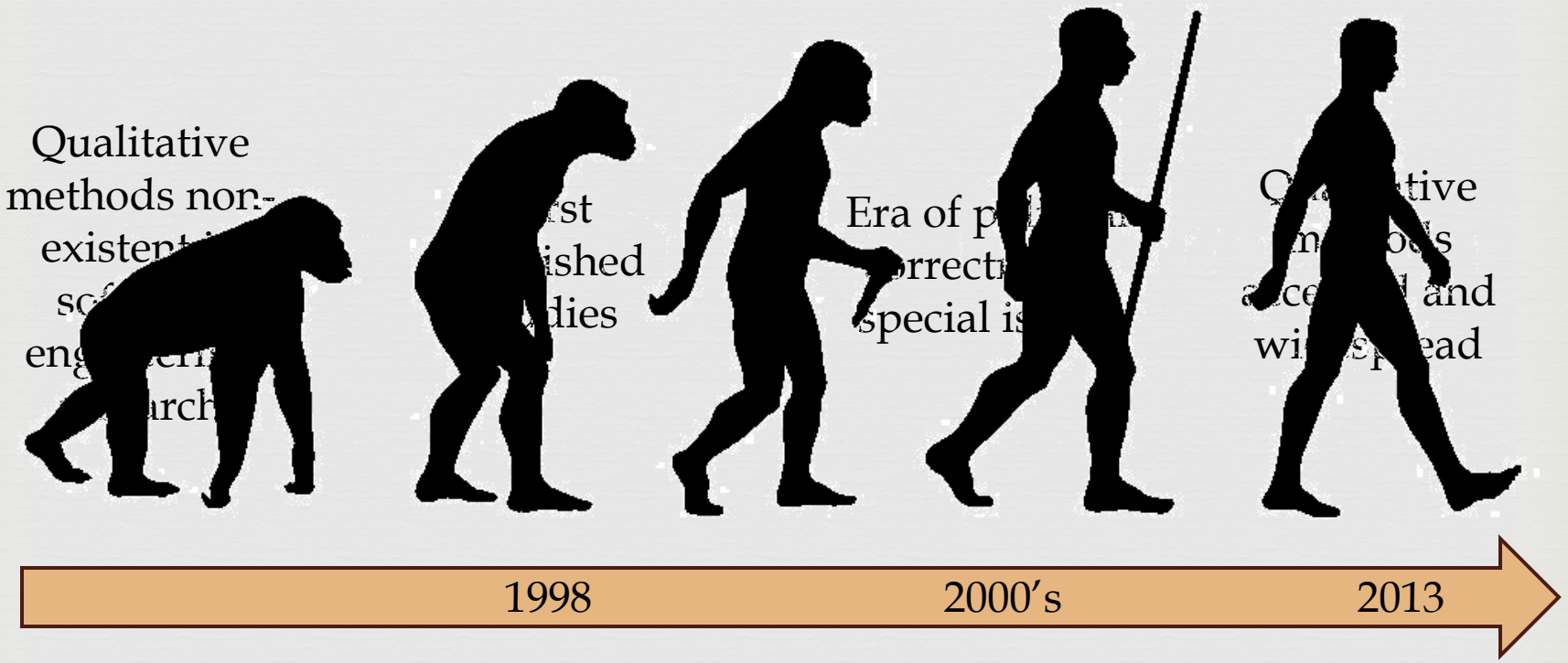
Lehman and Belady, 1985
Parnas, 1994

Risk Management

- Also foundational
  - Instances of Technical Debt constitute one type of software risk
- Risk Management cycle (identify, assess, manage) provides a template for managing Technical Debt
- Risk Assessment approaches (e.g., Risk exposure analysis) provides ways to quantifying Technical Debt
- Concept of utility loss provides a way to characterize the interest on Technical Debt

Stonebumer et al., 2002
Boehm, 1991

The Evolution of Qualitative Methods in SWE



Seaman, 1998
Dyba et al., 2011
Dittrich et al., 2007

Current State of Qualitative Methods in SWE Research

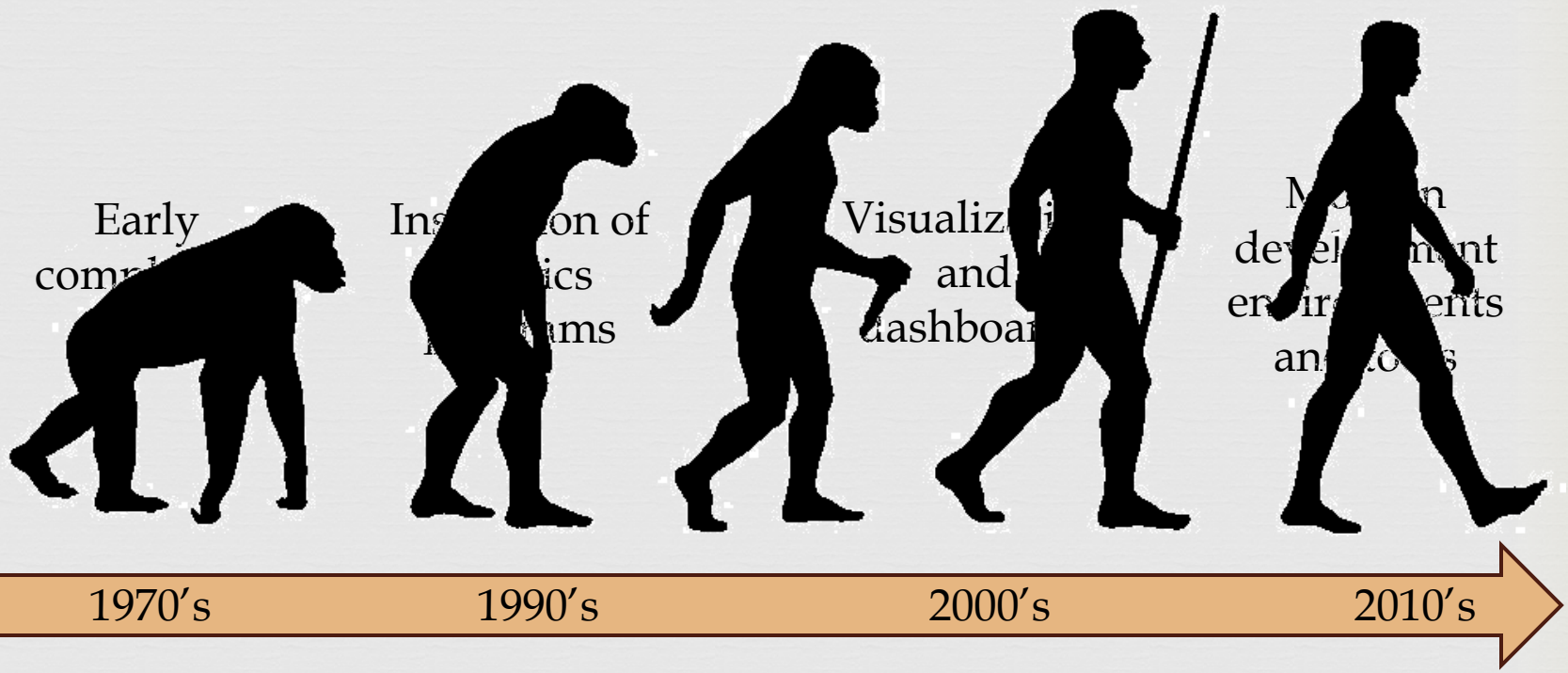
- Empirical software engineering researchers can now add a host of qualitative methods to their empirical toolkit
- Many good examples of qualitative studies are available in the literature (e.g. in special issues)
- Many experts who are highly experienced
- Starting to look at qualitative synthesis of studies (e.g. in the context of SLRs)
- **Bottom line**: We now have the tools and expertise available to fully investigate questions of human behavior and context

Qualitative Methods and Context in TD Research

- Technical Debt related concepts are context-specific
- A project's Technical Debt strategy should be based on goals and "pain points"
- Context factors can be elicited in a number of ways
- We need qualitative methods to ensure capture of all relevant factors
- Qualitative work in Technical Debt research one of the reasons for its relevance to practice
- **Bottom line**: We can't study TD properly without qualitative methods, and until recently we didn't as a community know how to use qualitative methods effectively

Lim et al., 2012

The Evolution of Software Metrics



McCabe, 1976
Halstead, 1970
Basili et al., 1994
Gaudin, 2009

Schumacher et al., 2010
Bohnet and Döllner, 2011
Snipes et al., 2011

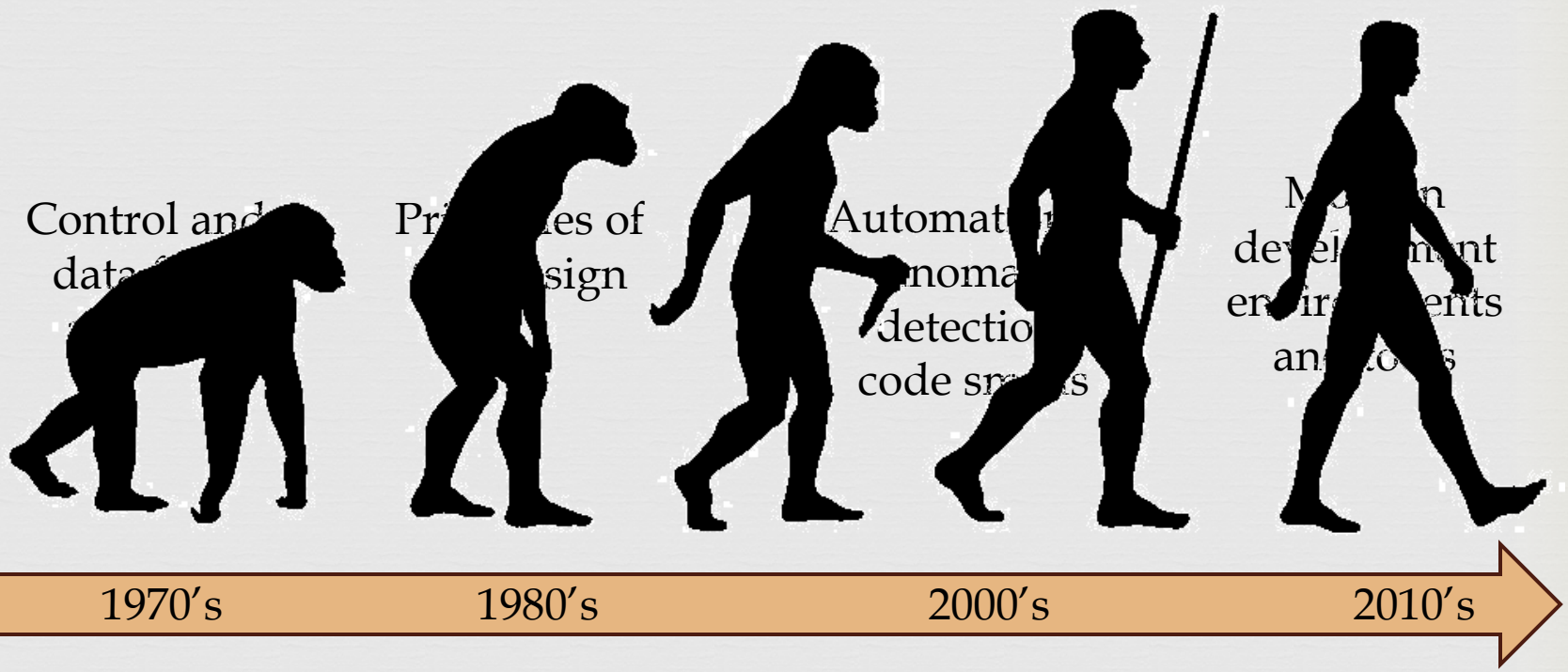
Current State of Software Metrics

- Adoption of software metrics in industry is still spotty
  - Especially in small and medium organizations
- Many large development organizations are "data-rich" environments
- Metrics no longer have to be "added on" at the end of the process – better integration is possible
- **Bottom line:** Tools are available to integrate data collection, analysis, and visualization into the software development process

Software Metrics and Technical Debt

- The relationship between software metrics and Technical Debt is complex and subject to further research
- Not evident that modules with "worse" indicators have "real" debt
- Code smell definitions try to get at the complicated relationship
- In practice, TD management often begins with monitoring metrics
- **Bottom line**: Simple views of metrics are not sufficient; we need easy ways to combine and visualize custom-fit combinations and relationships between different metrics

The Evolution of Program Analysis



Kildall, 1973
Jones, 1981
Rentsch, 1982
Booch, 1986

Ball and Rajamani, 2002
Munro, 2005
Bohnet and Döllner, 2011
Snipes et al., 2011

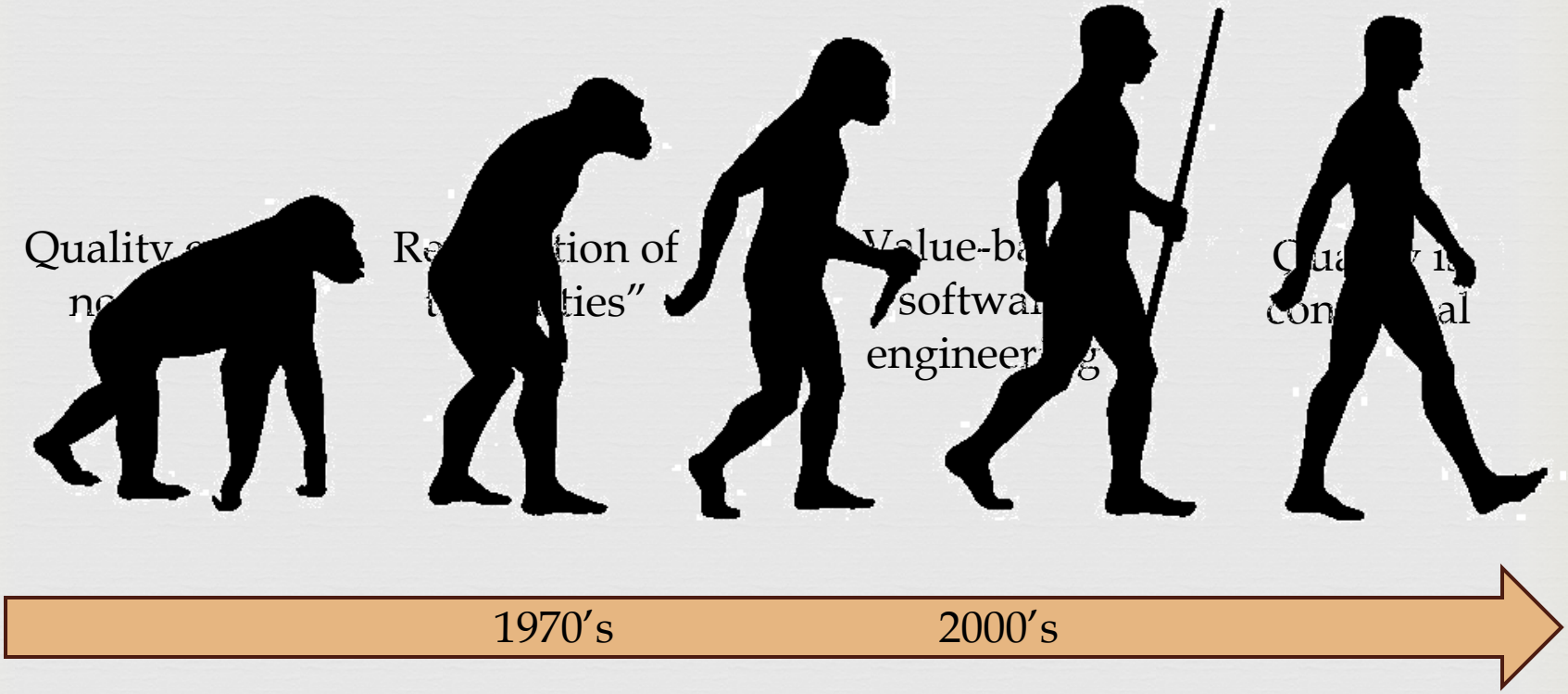
Current State of Program Analysis

- A plethora of tools available
- Easy to use
- Some cases in which program analysis is integrated into the build process
  - Even quantitative thresholds for an acceptable number of "issues"
- Generate mountains of information
- **Bottom line:** the challenge is to make sense of the analysis results – what's important?

Program Analysis and Technical Debt

- Like software metrics, the relationship between program analysis and Technical Debt is complex
- Anomaly detection through program analysis (e.g. code smells, ASA "warnings", etc.)
  - Not clear what anomalies constitute debt
  - Tools don't usually convey information about the value or importance of the anomaly
- Program analysis provides the building blocks for techniques that look at higher-level (e.g. architectural) issues
- **Bottom line**: Modern program analysis techniques provide tools only facilitate identification of debt, they need more support to identify and evaluate instances of "real" debt

The Evolution of Software Quality



Rubey &Hartwick, 1968
Boehm, 1973
Biffl et al., 2005

Current State of Software Quality

- Organizations still struggle to define quality in a meaningful way
- Maturity of understanding of quality varies
- Organizations who manage quality successfully have tied their quality indicators to business goals and desired outcomes
- **Bottom line:** Quality management is goal-driven

Software Quality and Technical Debt

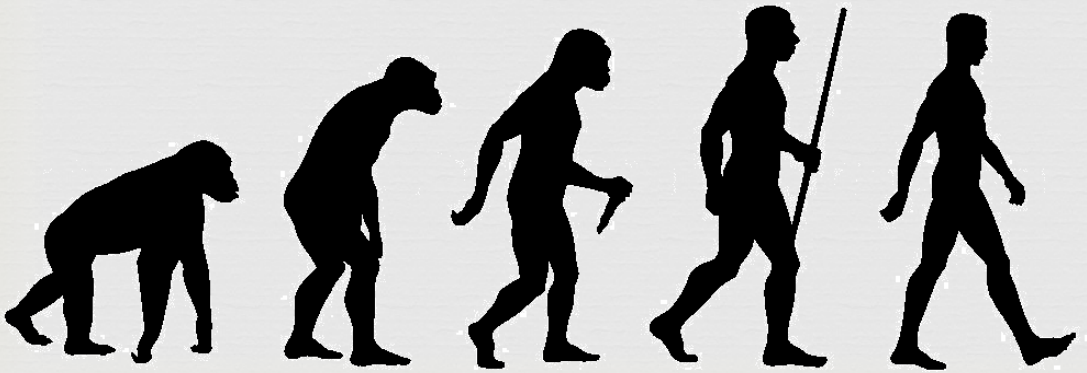
- Technical Debt is primarily concerned with the maintainability aspect of quality
- But most other "ilities" feed into maintainability
- The debt-related concepts of principal and interest are directly tied to the idea of value
  - The idea that quality contributes to value, not just function
- **Bottom line**: We now understand that quality means different things in different times and places, and it is this understanding that is crucial for the study of Technical Debt

Commercial Break

- Empirical Software Engineering and Measurement (**ESEM 2013**)
  - Baltimore, Maryland, USA
  - October 10-11, 2013
  - Short papers and posters deadline: June 11
- Co-located workshop: **Managing Technical Debt**
  - October 9, 2013
  - Working session to coordinate research in this area

Persistent Problems in Empirical Software Engineering

Technology Transfer



Evolution of the Discipline

Technology Transfer

- Too few empirical software engineering researchers get to see their ideas put into practice
- Our research too often does not start from a real problem or a real context
- Our research too often is described in terms that are not relevant for practitioners
- We're not good salespeople
- Requirements of publication and practice are not always in harmony

Technology Transfer and Technical Debt

- The Technical Debt metaphor
  - Gives us a vocabulary that both researchers and practitioners understand
  - Is a problem that practitioners care about
  - Forces researchers to view the problem from a practice point of view
- Applying Technical Debt research in practice starts with identifying the project's sources of "pain"
- Thus, research in this area by necessity is grounded in practice

Evolution of the Discipline

- Software engineering research has long suffered from an inability to build on previous results
- Too often suffers from a lack of grounding in prior literature
- Previous slides show successes in individual areas
- But we need to get better at
  - applying findings in one area to solve problems in another
  - combining diverse solutions to address a multi-faceted problem
  - see the relationships between different areas

Evolution of SWE and Technical Debt

- Technical Debt is a multi-faceted problem
- Addressing it effectively in practice relies on solutions from:
  - Software evolution
  - Risk Management
  - Qualitative assessment of context
  - Software metrics
  - Program analysis
  - Software quality
- Here's our chance to appreciate and use results from outside our own corners of the field

Recap

- Technical Debt
  - Is a metaphor that describes a real problem in software engineering practice
  - Requires solutions from a variety of different areas in empirical software engineering that have evolved over the last few decades
  - Requires solutions that are only now possible because of the level of evolution of these contributing areas
  - Provides the potential for addressing some long-term problems in the empirical software engineering research community

Call to Action

- Do more research on Technical Debt, **BUT**
  - Don't lose the industry focus
    - Keep talking to practitioners
    - Learn the vocabulary
    - Listen to where the pain is
  - Don't reinvent the wheel
    - Read the literature
    - Adapt solutions
    - Collaborate

"Watershed"?

"It is only a little hyperbolic to call this a watershed moment for empirical [software engineering] study, where many areas of progress are coming to a head at the same time."

- Are we at a historical moment in empirical software engineering research?
- Will everything be fundamentally different from now on?
- We have the right problem, we have a history of research providing at least the beginnings of the right solutions.
- It could be….

Thank you!

Questions?

References I

- Lehman, M. M. & Belady, L. A. (1985). Program Evolution: Processes of Software Change. *Academic Press Professional Inc*.
- Parnas, D. L. (1994). Software aging. *Proceedings of 16th International Conference on Software Engineering* (pp. 279–287). IEEE Comput. Soc.
- Stoneburner, G., Goguen and A., Feringa, A. (2002). Risk Management Guide for Information Technology Systems. *National Institute of Standards and Technology* (SP800-30).
- Boehm, B. W. (1991). Software Risk Management: Principles and Practices. *IEEE Software* (Issue 1, vol. 8, pp. 32-41).
- Seaman, C.B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* (vol. 25, no. 4, pp. 557-572)
- Dittrich, Y., John, M., Singer, J. and Tessem, B. (2007). For the Special issue on Qualitative Software Engineering Research. *Information and Software Technology* (vol. 49, no. 6, pp. 531–539).
- Dybå, T., Prikladnicki, R., Rönkkö, K., Seaman, C. and Sillito, J. (2011). Qualitative research in software engineering. *Empirical Software Engineering* (vol. *16, no.* 4, pp. 425–429).
- McCabe, T. J. (1976) A complexity measure. *IEEE Transaction on Software Engineering* (vol. SE-2, pp. 308 -320).
- Halstead, M.H. (1977) Elements of Software Science. Elsevier.
- Basili, V., Caldeira, G. and Rombach, H. D. (1994) The Goal Question Metric Approach. *Encyclopedia of Software Engineering*. Wiley.
- Schumacher, J., Zazworka, N., Shull, F., Seaman, C., Shaw, F. (2010). Building empirical support for automated code smell detection. *Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement* (pp. 1-10).
- Gaudin, O. (2009). Evaluate your technical debt with Sonar. *Retrieved from http://www.sonarsource.org/evaluate-your-technical-debt-with-sonar*.

References II

- Bohnet, J. and Döllner, J. (2011). Monitoring code quality and development activity by software maps. *Proceedings of the 2nd Workshop on Managing Technical Debt* (pp. 9-16).
- Snipes, W., Robinson, B. and Murphy-Hill, E. (2011). Code Hot Spot: A tool for extraction and analysis of code change history. *Proceedings of the 27th International Conference on Software Maintenance* (pp. 392-401).
- Kildall, G. A. (1973). A unified approach to global program optimization. *Proceedings of the 1st Annual Symposium on Principles of Programming languages* (pp. 194-206).
- Jones, N. D. (1981). Flow analysis of lambda expressions. *Automata, Languages and Programming* (vol. 115, pp. 114-128).
- Rentsch, T. (1982). Object oriented programming. *SIGPLAN Not* (vol. 17, no. 9, pp. 51-57).
- Booch, G. (1986). Object-Oriented Development. *IEEE Transactions on Software Engineering* (vol. SE-12, no. 2, pp. 211-221).
- Ball, T. and Rajamani, K. S. (2002). The SLAM Project: Debugging System Software via Static Analysis. *SIGPLAN Not* (vol. 37, no.1, pp. 1-3)
- Munro, M. J. (2005). Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source Code. *The 11th International Symposium Software Metrics*, 2005 (pp15).
- Rubey, R. J. and Hartwick, R. D. (1968). Quantitative measurement of program quality. *Proceedings of the 23rd ACM national conference* (pp. 671-677).
- Boehm, B. W., Brown, J. R., H. Kaspar, H., Lipow, M., MacLeod, G. J. and Merritt, M. J. (1973). Characteristics of Software Quality. *TRW Software Series* (TRW-SS-73-09).
- Stefan Biffl , Aybüke Aurum , Barry Boehm , Hakan Erdogmus , Paul Grünbacher, Value-Based Software Engineering, Springer-Verlag New York, Inc., Secaucus, NJ, 2005.