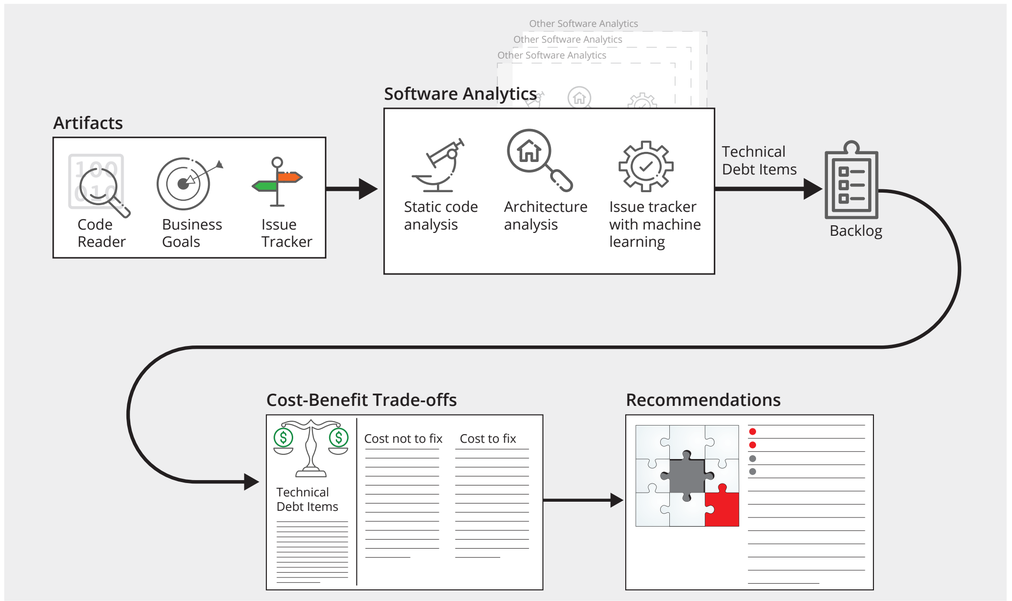
Technical Debt Analysis through Software Analytics

Technical debt conceptualizes the tradeoff between the short-term benefits of rapid delivery and the long-term value of developing a software system that is easy to evolve, modify, repair, and sustain. In this work, we are developing tools that integrate data from multiple, commonly available sources to pinpoint problematic design decisions and quantify their consequences in a repeatable and reliable way for uncovering technical debt.

Our research approach includes:

- Automatically extract potential TD issues from issue trackers using data mining techniques
- Enrich these clusters by incorporating information from code analyses and software repositories.
- Rank each TD cluster in terms of accumulating rework to date (e.g., total change and bug churn, number of related issues).

Technical Debt Analysis Workflow



Categorization and classification:

- Created a classifier that utilizes n-gram feature engineering and gradient boosting
- Our data shows that developers do identify long-lasting design issues as technical debt
- We are in the process of improving the classifier to help identify technical debt.

Model	Precision	Recall	F-Measure	Test Count	Training Count
1	0.60	0.42	0.50	510	157
2	0.71	0.32	0.44	411	256
3	0.77	0.42	0.54	311	356
4	0.74	0.70	0.72	161	506

Results: Our models are improving, especially in recall (more true debt is identified).

TD clusters

Design problems, frequently the result of optimizing for delivery speed, are a critical part of long-term software costs. Detecting such design issues with tool support is a high priority for technical debt analysis. We developed an approach where existing static analysis rules are mapped to design issues. The goal of the analysis is to help teams focus on rework causing issues earlier in the lifecycle.

		P1	P2	P5	P6	P7	All
	Rule	#	#	#	#	#	#
Logging	Standard outputs should not be used directly to log anything	334			7		343
Data consistency	@Override annotation should be used on any method overriding (since Java 5) or implementing (since Java 6) another one	1	172	1285	340	11	1799
Security & performance	Exception handlers should preserve the original exceptions	3	7	235	161	922	451
Maintainability	Source files should not have any duplicated blocks	133	36	16	23		260
	Methods should not be empty		65	463	74	67	615
	Control flow statements “if”, for, while, switch and try should not be nested too deeply	794	53	8	10	2	911
	Unused local variables should be removed	1	91	189	48	106	401
	Dead stores should be removed	5	491	512	372	291	1437
	Methods should not be too complex	79	208	157	1		498
	Sections of code should not be “commented out”	91	2	167			453
	Unused “private” fields should be removed	68	42	27	12		154
	Collapsible “if” statements should be merged	159	11	1	3		178

Results: Analysis across 8 projects reveals that our approach highlights hidden issues.

Government acquisition managers need capabilities to assess what kind of technical debt is created throughout the software lifecycle. The SEI team has been a pioneer in advancing the research agenda in this regard. Our ongoing work is focused on creating analytical tools towards an integrated software analytics approach.

Ipek Ozkaya | ozkaya@sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM17-0745
Technical Debt Analysis through Software Analytics