Spring 2013

Exploring Supply Chains from a Technical Debt Perspective

J. Yates Monteith

John D. McGregor

Exploring Supply Chains from a Technical Debt Perspective

J. Yates Montieth and John D. McGregor
{jymonte, johnmc}@cs.clemson.edu
School of Computing, Clemson University

Abstract

Software development has evolved from software development organizations building custom solutions for every need and creating a backlog of applications needed by users to specialized organizations producing components that are supplied to other software development organizations to speed the development of their software products. Our objective is to illustrate how a manager might use supply chain information to evaluate software being considered for inclusion in a product. We investigated the Eclipse platform code to illustrate analysis methods that produce information of use to decision makers. The technical debt of the software pieces was measured using the Technical Debt plug-in to SONAR as one input into the evaluation of supply chain quality. The dependency graphs of uses relationships among files were analyzed using graph metrics such as betweenness centrality. There was a statistically significant moderate correlation between the technical debt for a file and the betweenness centrality for that file. This relationship is used as the basis for a heuristic approach to forming advice to a development manager regarding which assets to acquire.

Technical Debt – What's Not Quite Right

Metaphor to describe the coding properly and coding fast, or "not-quite-right" code. The initial "debt "represents the effort it would take to correct the code, while the "interest" can be seen as the amount of additional work that must be as a result of the initial "debt." Through using the Sonar tool, not only are we able to quantify technical debt via a discrete number of code violations, we are able to represent it in man-days and dollars.

Debt can be incurred strategically, for a number of different reasons:

- Technology is not mature enough to be integrated
- Non-critical features in the face of known bugs
- Non-code related artifacts in the face of schedule faces

Supply Chains

Traditionally, Supply Chains represent the flow of raw materials to completed products. In a software context, raw products can be thought of as developers and design techniques.

Supply chains exist within a software dependency through the flow of development assets from one organization to another. Some assets may be not be code, such as processes or architectures, while others may be code based, related through source-code dependency.

We have modeled code-based software supply chains as dependency graphs. In our graphs, each node represents a source file, while each edge connecting two nodes represents a generalized "uses" relationship.

Experiments

For our experiments, we have modeled a subcomponent of the Eclipse Platform: the Eclipse Java Development Tools (JDT). We have examined three versions, 3.4, 3.5 and 3.6 and created network representations of their internal supply chains. We have calculated technical debt using the SONAR technical debt analysis tool. We have computed additional metrics using locally developed tools and Understand, a commercial static analysis tool.

Experiment 1: Betweenness Centrality and Technical Debt

In a software supply chain those nodes that are central to the chain are the ones that aggregate the functionality to be delivered to the customer. Centrality is a measurement of a node's relative importance within a graph or network, while betweenness centrality accomplishes this through shortest-path analysis. Betweenness centrality is defined as follows:

$$BC = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Where σ_{st} is the number of shortest paths from node s to node t, and $\sigma_{st}(v)$ is the number of shortest paths from s to t that pass through v.

In our experiment, we posed the question:

- Is there a relationship between the amount of technical debt attributed to a file and the betweenness centrality of the file?

With the null hypothesis:

- There is no correlation between technical debt and betweenness centrality.

Version	Nodes / Files	Edges / Dependencies	Lines
JDT 3.4	1,429	16,361	229,110
JDT 3.5	1,437	16,553	317,487
JDT 3.6	1,564	17,222	322,642

Figure 1: Size Metrics

Version	Correlation Coefficient	One-Tailed P Value
JDT 3.4	0.43607052	< 0.0001
JDT 3.5	0.42565911	< 0.0001
JDT 3.6	0.42765676	< 0.0001

Figure 2: Betweenness Correlations

Conclusion

The calculations resulted in a significant, but moderate correlation between technical debt and betweenness centrality, sufficient to reject the null hypothesis.

Experiment 2: Longitudinal Analysis of Candidate Subgraphs

When acquiring software components from a supply chain, acceptance testing is often too costly or time consuming for most software producing organizations. Because of this, reducing the problem space for acceptance testing is integral to informed component acquisition. In order to accomplish this, we used betweenness centrality and technical debt to filter out less important nodes in our dependency graph, while using a spring-embedded edge-weighted layout to organize sub-graphs into clusters of similar dependencies.

Nodes were filtered based on technical debt. Successive refinements on increments of $1,000. After several refinements, we were left with 4 files each with over $10,000 in technical debt. By selecting the adjacent edges and neighbors connected via adjacent edges, we formed the internal supply chain for those principal four nodes.

We then utilized a spring-embedded edge-weighted layout to organize the dependency graph. Nodes are modeled as objects which repel each other, while the edges are modeled as springs, with their distance weighted by the betweenness centrality of the node. Through using this layout, we were able to identify 9 clusters which in each case at least two of our four principal nodes were dependent on.

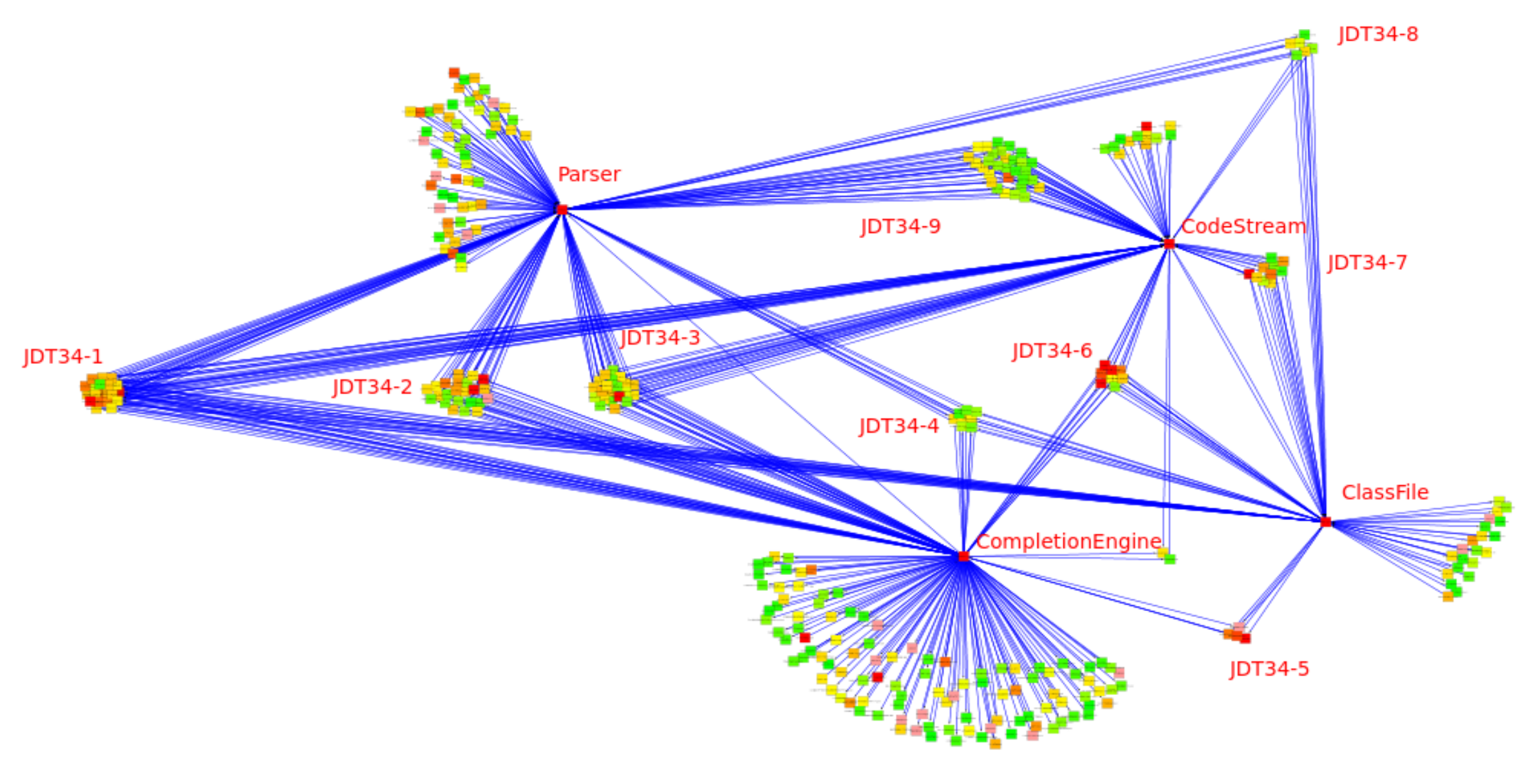

Figure 3: JDT 3.4

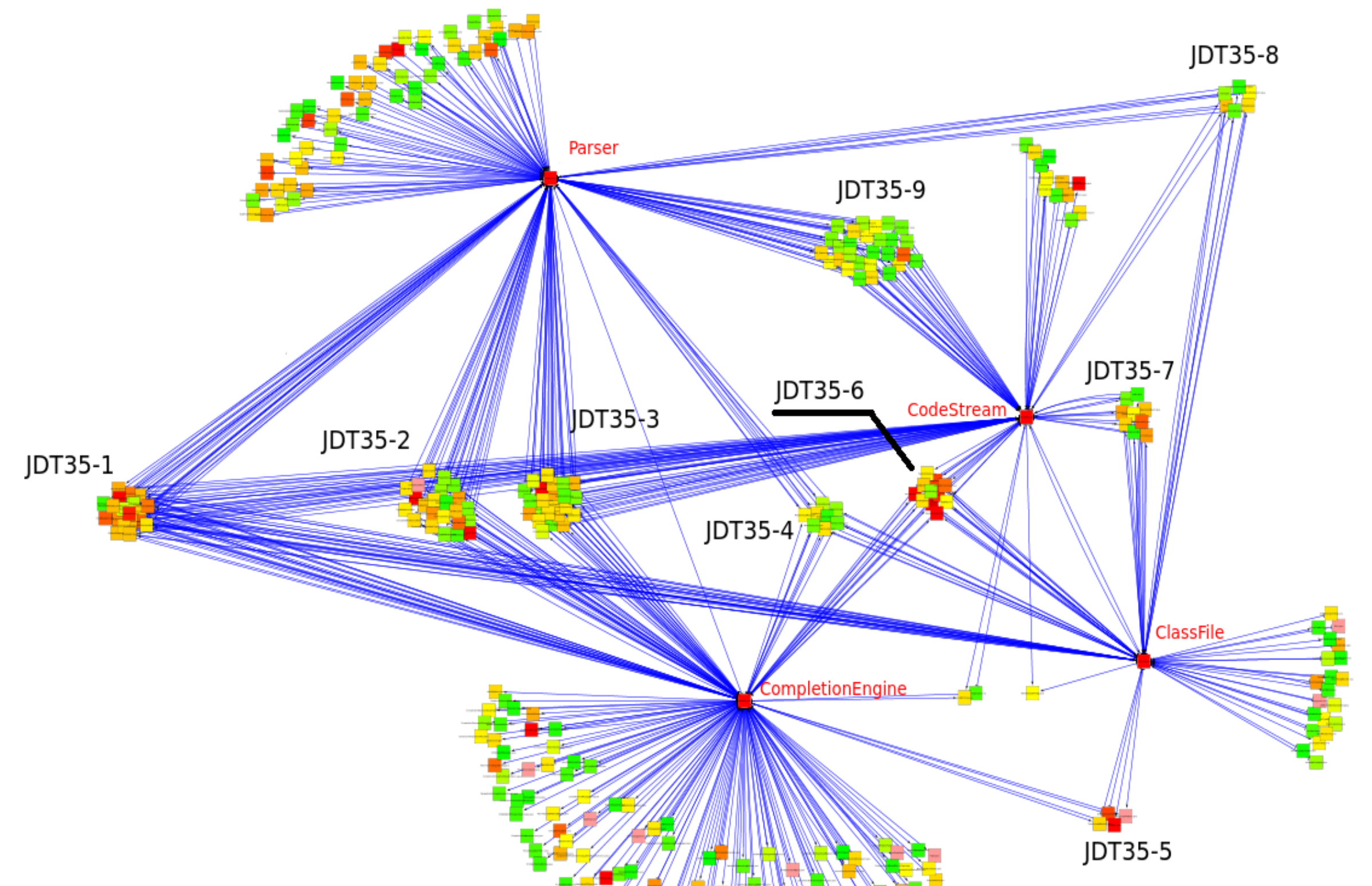

Figure 4: JDT 4.5

Experiment 2: Longitudinal Analysis of Candidate Subgraphs

By analyzing the metrics produced by both Sonar and Understand, we were able to identify a single cluster which had a significant change from one version to the next. Cluster 9 was not the biggest cluster in terms of any metrics, but saw an 80% reduction in code size, 50% reduction in code violations, and an 80% reduction in technical debt from version JDT 3.4 to JDT 3.5. What is more telling is the reduction of technical debt as measured in man-days, which indicates roughly 114 man-days of effort were put into fixing the code between these versions.

Using our method to identify and isolate candidate clusters for examination, our findings indicate that the Eclipse Developers have a technical debt, or at least quality conscience, perspective on both development and code maintenance. This is further evidenced by the relatively small amount of changes that were made from JDT version 3.5 to 3.6

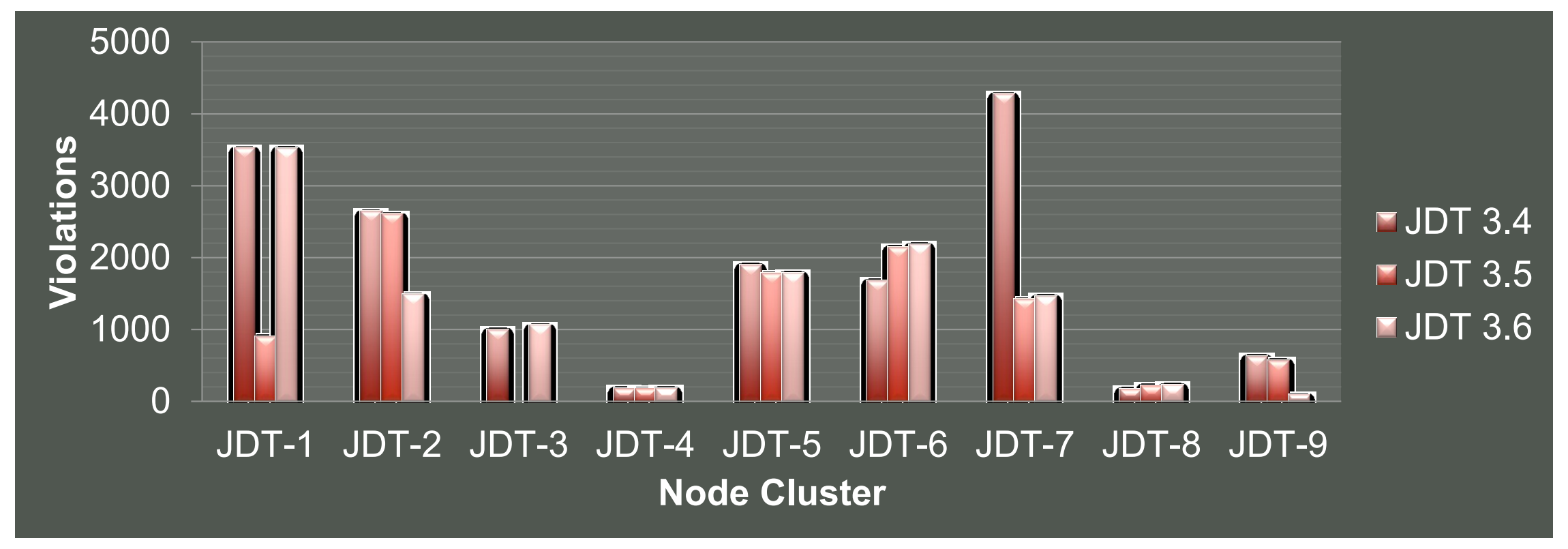

Figure 5: Technical Debt ($) of Component Clusters Across Versions

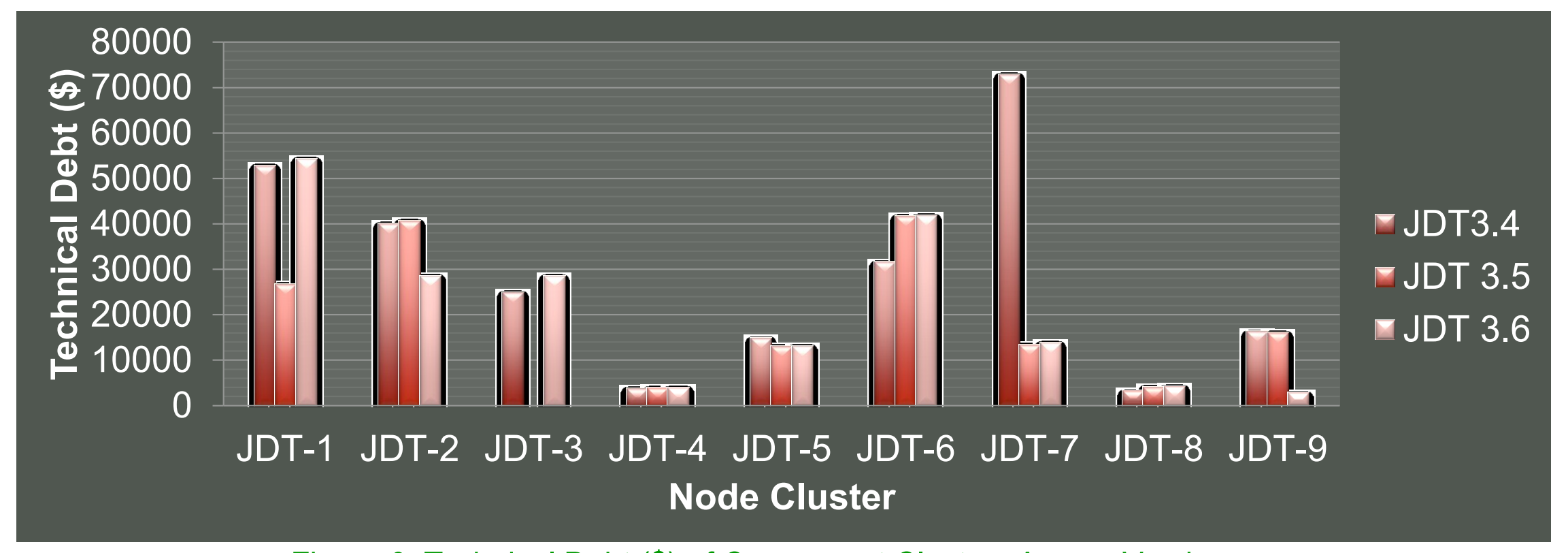

Figure 6: Technical Debt ($) of Component Clusters Across Versions

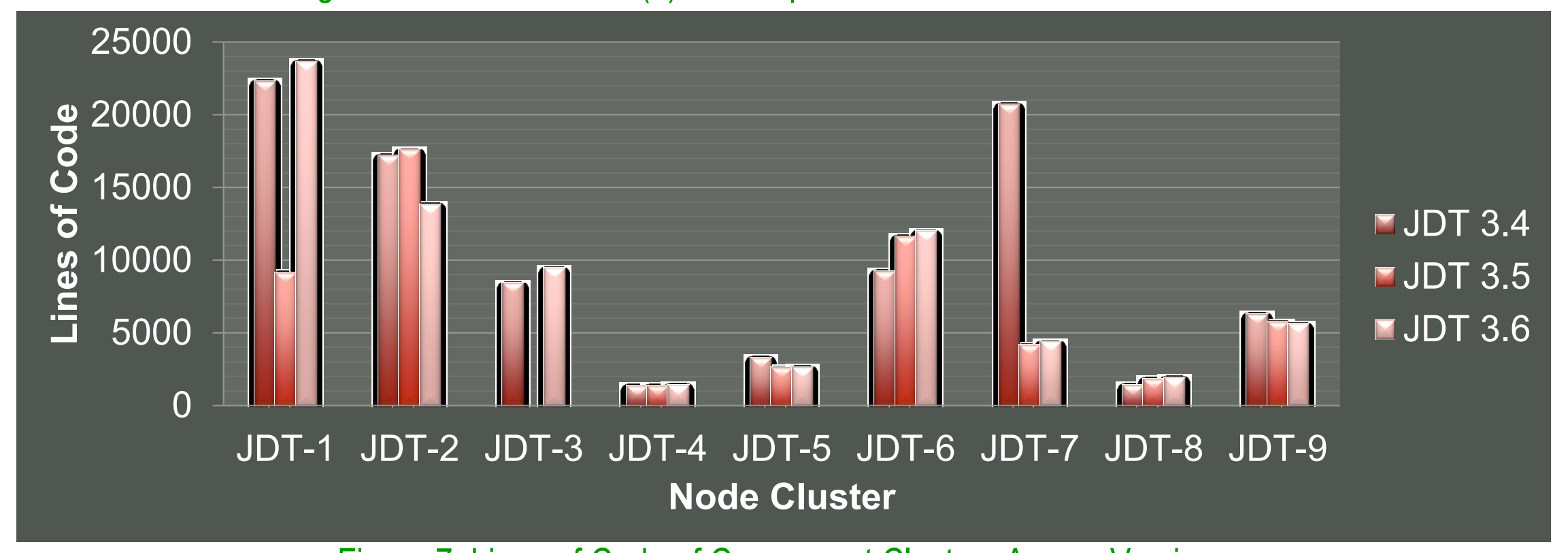

Figure 7: Lines of Code of Component Clusters Across Versions

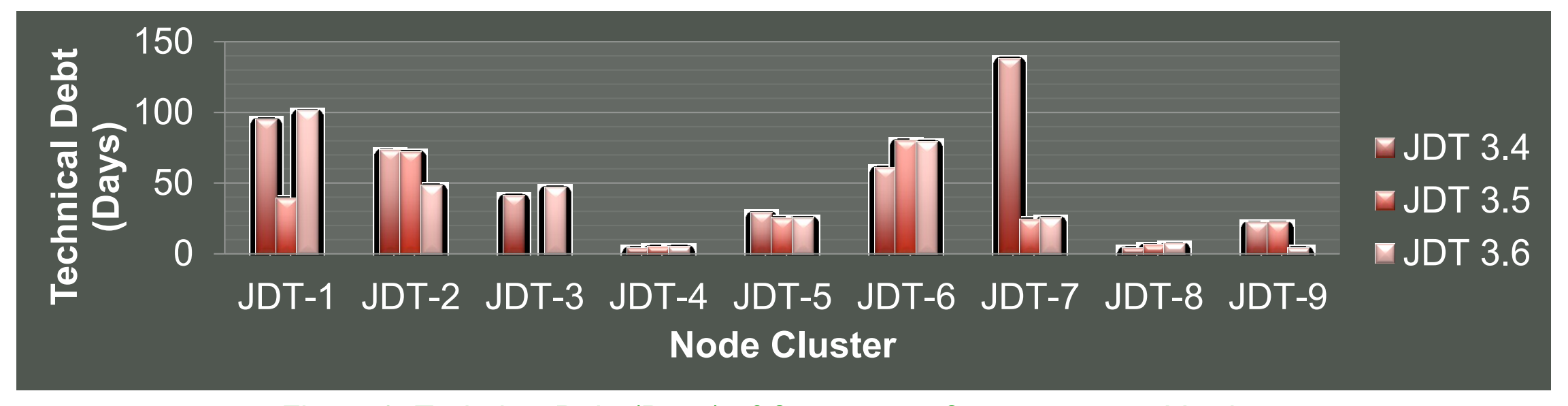

Figure 8: Technical Debt (Days) of Component Clusters across Versions

Conclusion

We have provided a heuristic approach to examining software for inclusion in an organization's supply chain. We use betweenness centrality to identify the most important files and technical debt to identify the files most in need to rework. These measures were applied to a set of versions of the Eclipse JDT package. Our heuristic pointed to an area in the internal supply chain which the experienced developers of the Eclipse platform also identified as needing rework. This distinction is the object of further study.