How to Analyze and Visualize a Large, Interconnected Software System: A Study of Fedora 20 with Lessons for All

January, 26th, 2015

Dan Sturtevant
CEO of Silverthread Inc.
Researcher at Harvard Business School
dan@silverthreadinc.com
o: 617-992-9290, m: 781-223-8200

Dave Allan
Director of Software Engineering
Silverthread Inc.
dave@silverthreadinc.com

Webinar video and slides available at http://sdm.mit.edu
Engineering organizations and the technical systems they create are so complicated that no understands them anymore.

Problem: Complexity

“the fundamental nature of software… involves basic and poorly understood problem-solving processes combined with unprecedented and multifaceted complexity”

- Shapiro

Threat: Technical debt, Technical ‘Bankruptcy’

Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation.

- Cunningham
Result: software engineering in a perpetual state of crisis

6.4% of large projects successful

41.4% failures
• Abandoned
• Started again from scratch

52% challenged
• Budget overrun
• Schedule overrun
• Bad functionality

The Standish Group,
with a database of 50,000 development projects
http://www.computerworld.com/s/article/9243396/
Healthcare.gov_website_didnt_have_a_chance_in_hell_
Topics for today

① Complexity, architecture, and technical debt
② Risk, operational performance, and financial performance in large software development enterprises
③ Building insight and improving outcomes using analysis and visualization
④ A look at Fedora Linux with lessons for all
Why we fail: Modern systems exceed bounds of human understandability

“[o]n a large software project one is lucky if one person in 50 has anything resembling an overall understanding of the conceptual structure of the project, and divinely blessed if that person has the ability to explain it in lay terms.”

- British Royal Academy of Engineering

As complexity grows it becomes hard to understand how

- Individual components work
- Different efforts interact
- The system as a whole functions

Agile competitors and mounting problems threaten the viability of the firm.

Confusion

Risk
- Project non-delivery
- Reliability & safety problems
- Inability to compete

Operational
- Teams unproductive
- Staff turnover and knowledge loss
- Project delays, cost overruns

Financial
- Loss of revenue
- Loss of customers
- Excess costs
- Inability to maintain market position

What it Costs
So why do complex systems ever work? Architecture.
Some architecture patterns that constrain complexity

Architecture = hierarchy and modularity
 (= “near-decomposability”) of a complex system

Herbert Simon, Architecture of Complexity, 1962

Hierarchical and modular systems can evolve easily
Architecture dramatically influences change cost
Some architectures are easier to change than others

Carliss Baldwin, Author of Design Rules: The Power of Modularity
Architectural properties help people manage complexity in technical systems and development enterprises

**Modularity**
- Prevent changes in one area from impacting others
- Bound cognitive burden placed on individuals
- Bound communication requirements

**Commonality**
- Long term waste reduction
- Build on and help harden mature capabilities
- Many eyes detect problems faster
- New efforts move more quickly

**Layering**
- Create powerful concepts and abstractions
- Make overall system more flexible
- Insulate people in different domains
- Hide information

**Hierarchies**
- Facilitates bottom-up, top-down design
- Complex problems broken up into small simple ones
- Scalable – can grow without increasing local complexity
- Prevent feedback

**Part Quality**
- Easier to understand
- Better performance
- Easier to interact with
- Easier to enhance
- Easier to debug
- Easier to test
- Better test coverage possible

---

**Technical Debt**: When these properties break down, problems are more likely
Network methods for visualizing software and team interactions

Direct Dependencies

Indirect Dependencies

Traditional Network View

Design Structure Matrix
The impact of modularity on organizational performance

Team A

Team B

Integral System

Modular System
Architectural complexity in a large commercial codebase

Hidden core spanning >2500 source files caused major financial & organizational problems at one firm

Modularity has broken down
Complexity has measurable risk and financial consequences

<table>
<thead>
<tr>
<th>Firm with ~130 products &amp; $1.7B revenue</th>
<th>Fortune 100 engineering conglomerate</th>
<th>Fortune 50 consumer software firm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cost &amp; schedule overruns</strong></td>
<td><strong>Defects &amp; risk of failure</strong></td>
<td><strong>Long-term maintenance cost</strong></td>
</tr>
<tr>
<td>Technical debt in a large codebase</td>
<td>Technical debt increased defects</td>
<td>Technical debt responsible for</td>
</tr>
<tr>
<td>impacts productivity</td>
<td>found after a safety-critical system went live</td>
<td>cost variance between products</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Developers in low-complexity code
  - 2.5X more productive
- At least 30 FTE / 12% worth of effort wasted per year
- Over $4 million annual waste
- 3X defects in complex code

<table>
<thead>
<tr>
<th>% of fielded system with critical bugs</th>
<th>Dollars spent fixing critical defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well structured code</td>
<td>Code with technical debt</td>
</tr>
<tr>
<td>00.629%</td>
<td>13.004%</td>
</tr>
<tr>
<td>00.629%</td>
<td>12 cents per LOC</td>
</tr>
<tr>
<td>13.004%</td>
<td>81 cents per LOC</td>
</tr>
</tbody>
</table>

- In code with a measurably better architecture:
  - 10% higher developer productivity during maintenance
  - 14% less time to release security patches
  - 25% fewer incomplete or incorrect fixes
Financial and operational performance Pre- and Post-refactoring at a Fortune 500 Software Firm

Refactoring underway

Refactoring complete

Rate of new product development accelerates dramatically
Fedora Community: a large scale development organization

- Fedora is a Linux distribution produced by the Fedora Project and sponsored by Red Hat\(^1\)
- Started in 2003
- As of Fedora 20, contains 2574 source packages producing 4071 binary packages
- Hundreds of millions of lines of code and billions of dollars in development effort\(^2\)
- Creates package dependency metadata so that package management tools can automatically install dependencies
- The package metadata make it possible to create network diagrams of the relationships between the individual open source communities that make up the distribution.

1. Fedora and the Infinity design logo are trademarks of Red Hat, Inc.
Fedora product architecture: Technology dependencies

- 4000 installable software packages in Fedora 20
- Some are low level, such as the Linux OS kernel
- End-user applications, such as browser & spreadsheet programs sit at a higher level
Fedora organizational structure: 703 core development teams cyclically interdependent on each other.
What does it mean to be interdependent?

All of these development teams / projects depend on each other **directly or indirectly**

This fact is **NOT OBVIOUS** to the development teams involved
Cyclic interdependencies amplify change propagation, cause side-effects, can cause organizational deadlock, etc.

Kernel development can clearly influence gstreamer plugin development

But the kernel development organization can also be impacted by the gstreamer plugin team via non-obvious & circuitous routes

NOTE: The path shown here has the shortest path length.
Cyclic interdependencies amplify change propagation, cause side-effects, can cause organizational deadlock, etc.

Alice introduces an incompatibility

Which makes Bob work at night, so he introduces a bug

Which makes Catherine miss a release date

Which means Doug is missing a needed feature

Which means Elle’s project is scrapped and she quits

So an emergency patch has to go out on Tuesday

Which means Perl has a bug in the regular expression engine

But the kernel development organization can also be impacted by the gstreamer plugin team via non-obvious & circuitous routes

NOTE: The path shown here has the shortest path length.
Cyclic interdependencies amplify change propagation, cause side-effects, can cause organizational deadlock, etc.

This may seem like a contrived example, but with millions of interdependencies between 703 development organizations, low-probability events will happen regularly.

The more teams are interlocked, the more conservative they have to be. This impacts culture, agility, and productivity.

Core teams behave much differently those without as much impact on others.
Imagine that you lead an organization with 2500 development teams working on interrelated products. What might be a good strategy for deciding where to focus?

1. Determine which products are most important from a revenue & competitive perspective. Place focus on them.
2. Determine which products they depend on, both directly and indirectly, focusing on the enterprise architecture. Figure out which products need attention even if they did not seem important at first.
3. Use technical debt measures to allocate effort and scrutiny.
4. Where appropriate, break unnecessary links to eliminate hidden structure that impacts teams. Doing so will allow them to operate independently at different clock-speeds.
Exploring complexity at multiple levels

**Dot size: Product size**

- **System level Complexity**: Core
- **Files with High internal complexity**: 9,557
- **Num files in package**: 96,483
- **Architectural complexity internal to package**: 0.0686
- **Unmaintainable files**: 2,345
- **Direct coupling**: All

**Dot size: Within product code complexity measure**

- **System level Complexity**: Core
- **Files with High internal complexity**: 9,557
- **Num files in package**: 96,483
- **Architectural complexity internal to package**: 0.0686
- **Unmaintainable files**: 2,345
- **Direct coupling**: All
Exploring complexity at multiple levels

Dot size: Measure of Product’s direct coupling to other products

- Package Name: kernel
- System level Complexity: Core
- Files with High internal complexity: 8,557
- Num files in package: 69,483
- Architectural complexity internal to package: 0.0666
- Unmaintainable lines: 2,345
- Direct coupling: All

Dot size: Within product architectural complexity

- Package Name: kernel
- System level Complexity: Core
- Files with High internal complexity: 8,557
- Num files in package: 69,483
- Architectural complexity internal to package: 0.0666
- Unmaintainable lines: 2,345
- Direct coupling: All
Now, imagine that you want to focus in on an individual software product. Repeat process one level down.

① Determine which software capabilities or components or files are most important and Place focus on them. Use network metrics to find those things they depend on.

② Determine which software elements they depend on, both directly and indirectly, focusing on the software architecture.

③ Figure out which functions, classes, etc. need attention even if they did not seem important at first. Use technical debt measures to think about risk and allocate effort and scrutiny in the codebase.

④ Refactor to break links and eliminate hidden structure interlocking teams. Doing so will free them to operate more independently.
The Linux Kernel is very well structured (low architectural complexity at the product level overall relative to benchmarks.)

Within the Kernel, there are regions that are more complex than others. This complexity is neither “good” nor “bad.” Some functionality is vital, obviously central to the function of the OS, and warranted. (i.e. memory manager, scheduler.)

Regions with higher architectural complexity may be more risky, costly, and harder to work in.

It may make sense to review the codebase to see if refactoring could reduce risk & cost by reducing reliance on components that are not obviously central.
The Silverthread team

Management team

Dan Sturtevant
- Chief Executive Officer

Dave Allan
- Director of Development

Sunny Ahn
- Director of Bus Development

Board members

Carliss Baldwin
- Founder & Board Member
- Professor of Finance at Harvard Business School

Michael Davies
- Founder & Board Chairman
- Senior Partner, Endeavour Partners
- Faculty member MIT ESD
- Boston Consulting Group

Alan MacCormack
- Founder & Board Member
- Professor of Technology Operations & Management, Harvard Business School
Conclusions

- Leaders of large development enterprises currently have limited visibility, making it hard for them to know where to focus.

- In this presentation we discussed ways to manage large scale development, focus on areas of highest risk, and act in ways that make the overall organization more effective.

- By exploring the dependency structure in Fedora 20, we found that at least 703 open source development teams depend on each others’ code to get their own jobs done, and were surprised to find a high degree of interdependence among teams. Some of those interdependencies might be unintended, and may be a drag on the overall effectiveness of the community.

If you can’t measure it you can’t manage it.
To learn more

Let us analyze your code: http://silverthreadinc.com

Community and academically oriented site: http://technical-debt.net

Or reach me directly:
Dan Sturtevant  dan@silverthreadinc.com  m: 781-223-8200