silverthread

Primer on design quality, the (more important) half of steering software projects

*Durable business agility = product agility * process agility. You need both.*

Process measures are not enough

Most software leadership steers and evaluates projects by measuring the development process. A very common failure pattern is to create micromanaged process controls that burden development teams with unnecessary overhead and frustrating busywork. Using techniques like Gantt charts, critical path analysis, CMMi, or earned value management, leadership gains *process* control and insight, but only limited *product* control and insight. The foundations of modern agile methods, DevOps principles, and lean systems engineering revolve around quick software delivery, faster feedback cycles, and analytics that quantify evolving product releases.

The secret of software measurement

Building on lean measurement techniques, we need to quantify *work* in progress, not *activities* in process. This means measuring code in the product pipeline and relying less on measures of the supplemental artifacts of the process pipeline. The direct measures of the code base are more factual and trustworthy mechanisms for steering software delivery projects.

Many enterprises measure *code quality* of the code subjected to unit testing. This is not enough. Decades of Harvard/MIT research and field experience provide strong evidence that the *design quality* of the code base subjected to integration testing correlates better with economic outcomes. Figure 1 illustrates a typical set of metrics from a large vendor of commercial software products.

Figure 1: The typical impact of design quality

How is design quality different from code quality?

Code quality and design quality are complementary product measures, and both can be extracted objectively from an evolving code base. Code quality assessments analyze the parts; design quality assessments analyze the whole. Developers can identify and fix code quality issues without having much impact on design quality. Code quality tools identify issues within a part by scanning and analyzing specific lines of code. Many code quality tools measure McCabe complexity. SonarQube identifies "bad smells" such as "if" "elsif" constructs with no final "else"; Coverity identifies likely buffer overflows; and HP Fortify identifies likely security vulnerabilities. But these tools do not quantify design quality. That is Silverthread's forte.

Silverthread's design quality assessments provide insight into the architectural properties of code bases that make them more manageable and understandable. When you quantify the relationships between the parts and the larger scale structures that they form, you can understand the macro-level health of the forest along with the micro-level health of the trees. Design quality tools identify complexity issues by scanning and analyzing dependencies among all the parts. Some insights include:

- Visual summaries of modularity, cohesion, and coupling
- Quantification of hierarchy and cyclical dependency
- Quantification and identification of commonality and reuse

By scanning a code base, you can extract the structure of a system as it really is, which is frequently much different than what it was intended to be. Design documents or design models capture intentions, but because they are supplementary artifacts that rely on manual change propagation, they are frequently wrong or out of synch with the evolving coded product.

Why should we invest in understanding design quality?

The most important characteristic of software is that it is "soft." The faster and easier software is to change, the faster and easier it is to achieve other required characteristics. Higher design quality leads to more predictable delivery, more understandable systems, and easier adaptation. Design quality is a relatively intuitive measure that gives both technical and nontechnical managers improved insight and control when leading a software development effort. Better design quality of the architecture subjected to integrated testing is the dominant factor impacting long-term durability, quality, business agility, and market agility.

You probably already use traditional project management measures like those shown in Figure 2. These process measures provide only half of the insight you need. Design quality and code quality are the other half, the more important product measures that teams need to steer software outcomes more predictably.

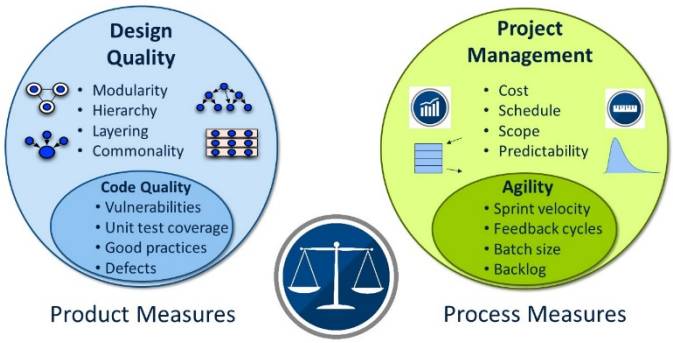

Figure 2: Product measures complement process measures

Principles of good software design

Software-based systems are so large and complex that no single person can understand how everything works. There are no laws of physics to constrain solutions. Software systems are complex networks of unbounded abstractions and an unbounded number of connections. System designers use well-accepted architectural principles to gain control over this complexity. These principles allow a complex system to be decomposed into more understandable chunks so that teams of cognitively bounded humans can work on different parts as independently as practical.

How can we measure design quality?

Objectively understanding design quality is one of the software industry's holy grails. Our research and field applications demonstrate valuable insights that can be realized through code scans and visualized through design structure matrices (DSMs). A DSM is a visual representation of the network of entities and relationships that make up a software system. We will elaborate these later, after we establish a foundation of principles.

Here are a few axioms of measuring design quality:

1. Design quality and structural complexity are context-dependent. Higher quality is in the eyes of the beholder.
2. Design quality is the dominant factor in long-term software economic outcomes.
3. Better design quality is analogous to a lower interest rate on technical debt.
4. Design quality drives the breadth and depth of interpersonal communications across the enterprise.
5. Design quality should be understood and quantified before investing heavily in code quality.
6. Design quality involves economic tradeoffs between efficiency (resources consumed) and effectiveness (user satisfaction with the value delivered).
7. Design quality involves architectural tradeoffs between static compile-time structures (the code base) and dynamic run-time interactions (system behaviors in integrated testing).
8. Design quality is best measured by quantifying structural analytics (efficiency) and integrated test analytics (effectiveness), as illustrated in Figure 3.

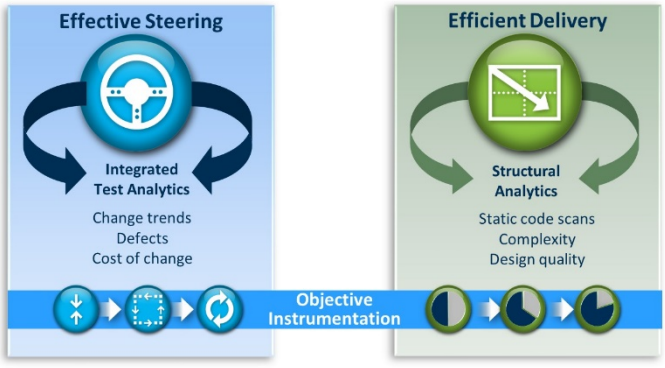

Figure 3: Balancing efficiency and effectiveness

The last axiom asserts two specific classes of measurement. *Structural analytics* provide quantifiable measures of design quality that can be extracted from a code base. More structural complexity leads directly to more overhead and inefficiencies in managing communications among people and activities across teams. *Integrated test analytics* provide measures of defects and change trends from the configuration control and issue-tracking tools. Quantifying the resources consumed to change a system allows teams to understand the consequences of structural complexity and design quality in terms of economic efficiency. Figure 4 illustrates basic design quality measures through a few quantifiable aspects of structural complexity.

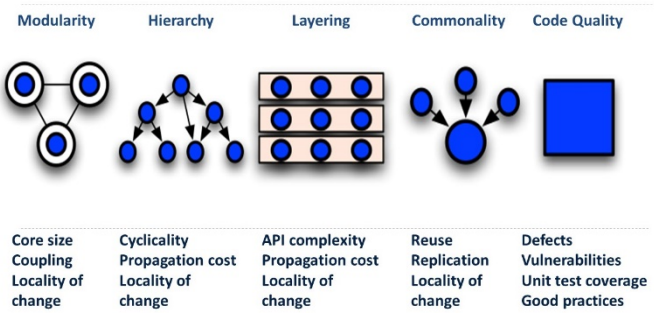

Figure 4: First-order measures of design quality

Design Principle 1: Modularity

Modular systems have separate parts that are cohesive (atomic units that perform a well-defined function) and loosely coupled (operate independently) with the other parts of the system. Modularity allows a complex abstraction or component to be organized into separate parts. Benefits of modularity include:

- Locality of change impact
- Atomic functions for more understandable parts
- Optimal assignment across teams of diverse skill
- Diversification of risk (in the same way that diversifying financial investments minimizes risk of financial loss)

Design Principle 2: Hierarchy and noncircular dependencies

Dependencies among modules should flow in one direction – from higher levels to lower levels in a hierarchy. Circularity or bi-directional dependency among modules should be minimized to avoid confusion and unintended consequences. When dependencies and control flow one way, from top to bottom without circularity, the system is called hierarchical. Control elements usually exist at the top of the hierarchy, while shared utilities reside at the bottom. Benefits include the flexibility to:

- Design bottom-up, top-down, or middle-out
- Scale up without increasing complexity
- Divide and conquer applications into more manageable pieces

Design Principle 3: Layering

Complicated abstractions can be simplified through layering multiple levels of classes and services so that consistent capabilities can be provided across varying platforms (of software, middleware, hardware, or third-party applications). The TCP/IP communications protocol hides an enormous amount of complexity (such as hardware transport, addressing, routing, packetization, and data quality checking) and provides a simple interface for using the internet. Layering benefits include:

- Simplified abstractions and complexity hiding
- Commonality, reuse, and transportability of code
- Simple, understandable APIs

Design Principle 4: Commonality and reuse

Similar functions should be performed by single pieces of code. When code is cut and pasted, then modified into multiple pieces of similar function but different coded implementation, change management becomes more complicated. Such duplication adds cost, complexity, and bloat without adding value. Benefits of commonality include:

- Reduction of rework and change management overhead
- More resilient and reliable common elements
- Fewer lines of code and simpler designs

Design Principle 5: Balanced complexity and code quality

The complexity contained inside a module or layer should maximize understandability. Code is written once but is read many times, frequently by multiple people. Cohesive, self-documenting, and understandable coding practices are important. If a module becomes too big for a single person to understand, it should be split. If the numbers of modules within a component, subsystem, or layer become too interdependent or complex for an architect to understand, they should be refactored. Benefits of balanced complexity include:

- Understandability of a system's subsystems and modules
- Simplified testing, diagnosis, and modification
- Simplified allocation across teams of people

Visualizing design quality

Every code base, no matter the language, size, age, framework, or programming paradigm, can be thought of as a collection of *entities* and *relationships* between entities. Entities include source code files, functions, classes, methods, data types, modules, components, and layers. Relationships include call, encapsulate, subclass from, and depends on. DSMs capture entities and relationships as networks of dependencies. They organize the entities into modules and logical components that illustrate the cohesion and coupling extracted directly from the source code. Modules and components are arranged so that those lower in the component hierarchy are further to the left, and those higher in the hierarchy are further to the right. After 15 years of Harvard/MIT research capturing a diverse spectrum of systems, we found recurring patterns of good and bad quality. Figure 5 illustrates two DSMs that are typical for the opposite ends of the quality spectrum.

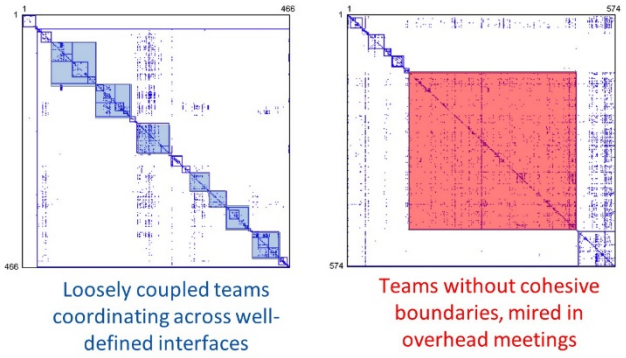

Figure 5: DSMs help us visualize design quality principles

The modular DSM on the left exhibits locally tight coupling within components but relatively loose coupling among components. The alarming structure on the right exhibits tight coupling across a large, dominant core component (red), with smaller peripheral components. The "core" is the component that is most complex and least hierarchical.

An ideal DSM looks like the illustration in Figure 6. Better quality systems exhibit no overly dominant core component that has substantially higher complexity than other components. They tend to have a horizontal layer of hierarchical control dependencies from a well-structured control component, with very few other component-to-component dependencies. They also tend to have a vertical column of dependencies for shared utilities and reusable services, APIs, classes, and data elements.

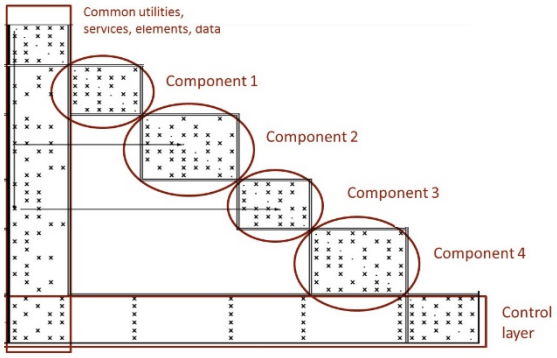

Figure 6: An idealized design structure matrix

How to read a design structure matrix

Assessing good design quality indicators and red flags from DSMs starts with some simple first-order observations. Human judgment is usually required to validate assessments in context, and many second-order analyses are usually needed to draw meaningful conclusions with confidence.

Modules along the diagonal. Files in the same module are placed side by side so that dependencies between modules combine to form logical components along the diagonal.

Cohesion and coupling. Many dependencies inside modules indicate high degrees of cohesion. Relatively fewer dependencies outside module boundaries indicate lower external coupling.

Commonality and reuse. Vertical bands on the left side of the matrix indicate reused modules at the bottom of the hierarchy.

Control elements. Horizontal bands on the bottom of the matrix represent control elements at the top of the hierarchy.

Hierarchical/noncyclical module dependencies. Intermodule links positioned below the diagonal flow in a single direction, from the top of the component hierarchy to the bottom. Links above the diagonal flow in both directions and indicate cyclicality.

Cyclicality contained. Circular dependencies should be mostly isolated within components. They represent significant complexity and should be isolated to a single developer or team.

Small modules. Modules should be limited in size, including lines of code, dependencies, data, entities, and relationships. As modules grow, becoming overly complex and unmaintainable, they should be refactored and split into smaller parts.

Ideal-looking DSMs, like the one in Figure 6, rarely show up in practice. In Silverthread's database of empirical results, spanning thousands of projects, more than 85% of our scanned code bases look more like Figure 7, with obvious challenges in understandability,

complexity, and malignant change propagation. One root cause is a lack of insight into how complexity grows as code bases are evolved over long periods of release and maintenance.

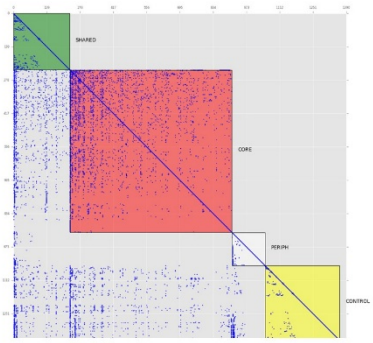

Figure 7: A typical design structure matrix with issues

Excess coupling. With numerous and expanding dependencies outside component boundaries, teams experience increased difficulty in understanding design intentions and anticipating the side effects of changing modules.

Cyclicality in a complex core. Although developers believe the system in Figure 7 has balanced components, a large overly complex component dominates the defect reports and volume of changes.

Team coordination problems. Development teams cannot operate independently because of coupling between the components for which they are responsible. The result is lots of meetings and excessive communications overhead.

What a large-scale DSM can look like

Understanding a quantified and visual perspective of design quality, especially in large-scale, long-lived evolving software systems, allows a team to manage technical debt, understand the relationships among teams, and improve the predictability of changes. Figure 8 shows a component view extracted from a larger scale aerospace application using CodeMRI® tools. In this software code base, about 4,500 files are connected by 500,000 interfile dependencies. When printed as a poster and appropriately labeled, this view was an eye-opener for the development team. They were surprised by many of the dependencies and insights that were exposed by the DSM.

Many aspects of good design quality are shown in Figure 8. Vertical bands indicating reuse are evident. Higher within-module coupling and lower external cohesion are also apparent. Circularity tends to be contained within modules. Some modules are arguably too large. Of the more than 500,000 dependencies, only about 10,000 illegal intermodule dependencies exist above the diagonal. While this was a relatively good result, it still identified many opportunities for the team to improve design quality and reduce the malignant changes they were occasionally experiencing.

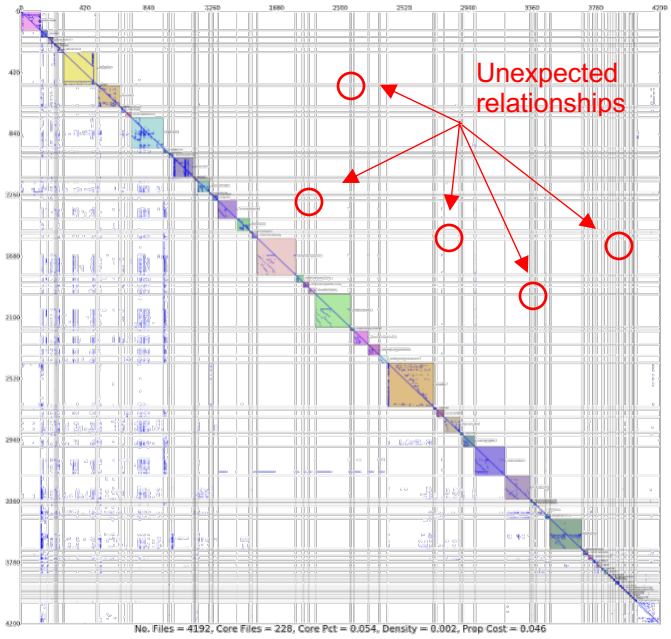

Figure 8: A large-scale DSM

Silverthread's CodeMRI® capability

Complex software systems should be structured as hierarchies of modules with reuse. Module dependencies should be explicit and noncircular. Modules should be small enough to be managed by a team. For simplicity, individual modules should have low McCabe Cyclomatic complexity scores. Reports produced by Silverthread's CodeMRI® tools highlight these module properties as well as architectural complexity.

Understand the structural complexity of your architecture

In a software system of reasonable size, a development team should be able to identify these sources of structural complexity:

- Distinct software components and modules
- The source code files that belong to each
- Dependencies among components and modules
- Core components, the highly complex regions in a code base where quality and productivity problems accumulate
- Files with high McCabe Cyclomatic scores, the complex elements in a code base where quality suffers

CodeMRI® views, metrics, and benchmarks can help to diagnose your current design health and identify opportunities for improvement.

Contact Us

Silverthread's mission is to advance the state of software measurement practice by quantifying complexity and design quality. Our measurement know-how can establish a more trustworthy foundation for improving software economics.

http://silverthreadinc.com