TAMING THE COMPLEXITY: MANAGING THE QUALITY OF ARCHITECTURE IN SOFTWARE SYSTEMS

Raghvinder S. Sangwan
Pennsylvania State University
Great Valley School of Graduate Professional Studies
Malvern, PA 19355
Email: rsangwan@psu.edu

National Quality Month Symposium 2007, October 12, 2007, Malvern, PA, USA.
The software landscape

• When systems were not large enough to merit an explicit design phase, software was considered to encompass all of the forms that were concerned with generating “executable binary code” that was intended for execution on a single machine
  – The structure of the system was largely fixed when the code was compiled and so the idea(s) of the designer were directed towards producing a single monolithic unit of binary code

• Systems are now large and typically distributed across several machines
  – The designers now have to decide how functionality and data would be partitioned or shared between machines, the form of communication mechanisms to employ, and the likely performance effects of these choices
Software design

- The process of design is divided into two distinct phases
  - *Architectural or logical design*: designer develops a highly abstract model of the system in which only externally visible properties of the model elements are included; this black-box partitioning is largely concerned with the nature and form of the problem and is less strongly influenced by the eventual form that will be adopted for its solution
  - *Detailed or physical design*: the abstract "chunks" of the problem that were identified in the first phase are mapped on to technologically-based units turning the black-box into a white-box

- Following properties of software make the design process a challenging task:
  - *Complexity*: no two parts of a system are alike and it may possess very many states during execution
  - *Conformity*: software is expected to conform to the standards imposed by other components
  - *Changeability*: software suffers constant need for change
  - *Invisibility*: this constrains our ability to conceptualize the characteristics of software and also hinders communication among those involved with its development
Measuring design quality (1)

• “When you can measure what you are speaking about, and express it in numbers, you know something about it, but when you cannot measure it, when you cannot express it numbers, your knowledge is of a meager and unsatisfactory kind.” (Lord Kelvin)
  – Ideas about measurement originally emerged largely within a community of scientists and engineers who were seeking to capture ideas about physical properties such as mass, length, velocity, etc.
  – The scales used for such properties are ratio scales; they possess well defined intervals and a zero point
• Once we move away from this context, we often are limited
  – Many of the properties of interest to us are more likely to be measured using an ordinal scale
• “Measurement is concerned with capturing information about the attributes of entities.” (Fenton and Pfleeger, 1997)
### Measuring design quality (2)

<table>
<thead>
<tr>
<th>Role</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality Concept</td>
<td>Abstract ideas about what constitutes “good” and “bad” properties of a system, and which will need to be assessed by the designers when making decisions about design choices</td>
<td>Complexity</td>
</tr>
<tr>
<td>Measurable characteristic</td>
<td>Provide a set of measurable (or at least, identifiable) characteristics of the design entities and so provide mappings between the abstract idea of a property and the countable features of the actual design (and therefore effectively correspond to the general concept of a metric)</td>
<td>McCabe’s Cyclomatic Complexity</td>
</tr>
<tr>
<td>Measure</td>
<td>Identifying lexical features of a representation that will need to be counted in order to obtain some form of values for the metrics</td>
<td>Counting nodes and edges</td>
</tr>
</tbody>
</table>

(Budgen, 2003)
Software complexity (1)

- Complexity of software is dependent on
  - the hierarchical organization of a code base, beginning at the method level and migrating up through the class, package and component levels
  - the nature of the dependencies that bind the different software elements together

- The relative impact that complexity has will vary depending upon the level of abstraction in the design hierarchy.
  - For example, cyclic dependencies between software packages and components will have greater impact than excessively complex code at the method level because a change in one package or component may adversely affect all the dependent packages or components.
Software complexity (2)

- Complex is not the same as complicated
  - “A solution should be as simple as possible but no simpler” (Albert Einstein)
    - Attempting to oversimplify since the result will be a product that will not be able to do its job
- Some measures for complexity include:
  - Complexity of control flow (McCabe, 1976)
  - Complexity of comprehension (Halstead, 1977)
  - Complexity of structure (Chidamber and Kemerer, 1994)
Software complexity and modularity

• Modular structuring makes it possible for a given problem to be considered in terms of a set of smaller components

• To make good use of a modular structure, one needs to adopt a design practice based on separation of concerns
  – A designer needs to group functions within modules in such a way that their interdependence is minimized

• Modularity provides the following benefits:
  – Modules are easy to replace
  – Each module captures one feature of a problem, so aiding comprehension (and hence maintenance), as well as providing a framework for designing as a team
  – A well-structured module can easily be reused for another problem

• Two useful measures for assessing modular structuring of software are coupling and cohesion
Modularity and coupling

- **Coupling**: measures inter-module connectivity – both form and strength

<table>
<thead>
<tr>
<th>Form</th>
<th>Feature</th>
<th>Desirability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data coupling</td>
<td>Modules A and B communicate by parameters or data items that have no control element</td>
<td>High</td>
</tr>
<tr>
<td>Control coupling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i) Activating</td>
<td>A transfers control to B in a structured manner such as by means of a procedure call</td>
<td>Necessary</td>
</tr>
<tr>
<td>(ii) Coordinating</td>
<td>A passes a parameter to B that is used in B to determine the actions of B (typically a boolean flag)</td>
<td>Undesirable</td>
</tr>
<tr>
<td>Common-environment coupling</td>
<td>A and B contain references to some shared data area that incorporate knowledge about its structure and organization. Any change to the format of the block requires that all of the modules using it must also be modified</td>
<td>Undesirable</td>
</tr>
</tbody>
</table>
Modularity and cohesion

- Cohesion: measures the extent to which the components of a module can be considered to be functionally related
  - Ideal module is one in which all the elements can be considered as being solely present for one purpose

<table>
<thead>
<tr>
<th>Form</th>
<th>Feature</th>
<th>Desirability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>All elements contribute to the execution of a single problem-related task</td>
<td>High</td>
</tr>
<tr>
<td>Sequential</td>
<td>The outputs from one element of the module form the inputs to another element</td>
<td>Quite high</td>
</tr>
<tr>
<td>Communicational</td>
<td>All elements are concerned with operations that use the same input and output data</td>
<td>Fair</td>
</tr>
<tr>
<td>Procedural</td>
<td>The elements of the module are related by the order in which the operations must occur</td>
<td>Not very</td>
</tr>
</tbody>
</table>
Comprehensive view of software complexity (1)

• When working with large, complex code bases individual metrics offer only a limited snapshot of system complexity
  – they lack the capacity to visualize the impact of dependencies on emergent design, particularly as these dependencies are rolled up through the structural hierarchy

• It is also difficult to compare the complexity of two software programs, whether they are different programs or different versions of the same program
Comprehensive view of software complexity (2)

- Structure 101 (Headway Software, 2006) is a measurement framework that provides a comprehensive view of structural complexity within a software system.
- This measurement framework is based upon two aspects of excessive structural (XS) complexity, termed ‘tangles’ and ‘fat’.
- Tangles represent cyclic dependencies between packages:
  - Martin’s (2003) ‘Acyclic Dependency Principle’ dictates, “the dependency structure between packages must be a Directed Acyclic Graph (DAG), that is, there must be no cycles in the dependency structure”.
  - Although cyclic dependencies between classes or methods within a package are often unavoidable, such dependencies among packages lead to highly coupled code that is difficult to maintain and extend since all of the packages involved in a tangle will be integrally tied to each other and affected by any changes.
Comprehensive view of software complexity (3)

- Fat measures the lack of structure, whereby packages, classes or methods, have grown excessively large and complex
  - It is determined as the number of dependencies between subcomponents
  - To ensure that fat is not hidden at the method level (complex, nested control structures), McCabe’s (1976) cyclomatic complexity metric is used to determine the amount of fat at the method level

- The degree of tangle and fat that exceeds defined threshold values at the design, leaf package, class, and method levels is quantified by the excessive (XS) complexity metric

- The XS metric is subsequently rolled up through the code-base hierarchy to calculate cumulative and average XS
  - Therefore, excessive complexity at higher levels in the design hierarchy is a much greater problem since it will impact a much larger portion of the code base

- The cumulative and average XS make it possible to quantify complexity differences between different application types and also different releases of the same application type.
Software evolution and complexity

• As a software system evolves, growing number of dependencies are introduced among its various parts potentially violating its design goals
• Lehman’s laws also predict that such emergent design would be expected to become increasingly complex over the evolution of the software unless work is done to reduce or maintain it
• This phenomenon has been observed by Schach et al (2002) in the Linux operating system and by Smith et al (2005) in 25 different open source software systems
• Therefore, it is useful to apply the new complexity monitoring technique to track and manage structural complexity in an application as it evolves through its different releases
Complexity cycle (1)

- Using the new complexity measurement methodology we tracked the structural complexity of three different open source software products as they evolved through their different releases.
- Our analysis revealed that a high proportion of structural complexity in the early releases may be found at the application code level progressively migrating to higher level design and architectural elements in subsequent releases or vice versa, a pattern that repeats itself throughout the evolution of the software product.
- We propose that such structural ‘epochs’ naturally occur during the course of software evolution whereby refactoring efforts successfully reduce complexity at the local level (e.g. within leaf packages or methods) but shifts the complexity to a higher level in the design hierarchy and design restructuring at higher levels shifts the complexity to the lower levels of the hierarchy.
Complexity cycle (2)

- If our observation holds true for most software products, then mere code refactoring may not be sufficient and software project managers may have to plan for a major restructuring of applications periodically to effectively manage structural complexity.
Case Study: JFreeChart
Early evolution epoch: complexity begins to surface

- The average XS increased sharply from 18 to 46% between the 9th and 10th release.
- During this time, the release notes indicated the addition of new functionality with respect to new plot types and changes to the combination plot framework.
Mid-evolution epoch: complexity begins to migrate

- This substantial architectural design restructuring that occurred between the release 14 and 15 of JFreeChart led to a decrease in the average XS from 54 to 30%.
Late evolution epoch: migration continues

- Although no major changes were reported in the release notes that accompanied release 31, the few changes that were made succeeded in reducing the average XS from 44% to 33% between releases 30 and 31, respectively because of package splitting.
Pattern of shifting structural complexity

- To further examine the phenomenon of shifting structural complexity, we examined two additional open source applications.
- These applications provide further evidence that excessive structural complexity shifts during software evolution but the pattern may vary from one application to the other.

### Findbugs

![Graph showing XS in 'Findbugs' Evolution]

### Hibernate

![Graph showing XS in Hibernate Evolution]

©Raghvinder S. Sangwan
Managing complexity (1)

- Visualizing the structure of a system can reveal tangles, clusters and design violations
Managing complexity (2)

- Metrics can provide supporting evidence

<table>
<thead>
<tr>
<th>Metric</th>
<th>Measure</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response factor for a class</td>
<td>Average: 13.503 Maximum: 103</td>
<td>High response factor makes classes difficult to understand, test and debug</td>
</tr>
<tr>
<td>Depth of inheritance hierarchy</td>
<td>Average: 1.567 Maximum: 19</td>
<td>Deep inheritance hierarchy implies complex design that is harder to understand and test</td>
</tr>
<tr>
<td>Data classes</td>
<td>25</td>
<td>Data classes break encapsulation</td>
</tr>
<tr>
<td>Feature envy classes</td>
<td>8</td>
<td>Feature envy classes break encapsulation</td>
</tr>
<tr>
<td>Large classes</td>
<td>Average method count: 9.737</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum method count: 75</td>
<td>Too many responsibilities packed into a class making it incohesive</td>
</tr>
</tbody>
</table>

©Raghvinder S. Sangwan
Strategic refactoring (1)

- Understand the problem domain
  - Evolution of the system should preserve the domain model
- Use patterns to achieve a flexible design that supports future requirements
Strategic refactoring (2)

- Refactor the current system using the new design and reanalyze
Strategic refactoring (3)

- Metrics can provide supporting evidence

<table>
<thead>
<tr>
<th>Metric</th>
<th>Measure</th>
<th>Comparative Analysis with the Original System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response factor for a class</td>
<td>Average: 7.31</td>
<td>Reduced by half</td>
</tr>
<tr>
<td></td>
<td>Maximum: 54</td>
<td></td>
</tr>
<tr>
<td>Depth of inheritance hierarchy</td>
<td>Average: 0.207</td>
<td>Average reduced by a factor of 7</td>
</tr>
<tr>
<td></td>
<td>Maximum: 1</td>
<td>and the maximum reduced by a factor of 19</td>
</tr>
<tr>
<td>Data classes</td>
<td>1</td>
<td>Data classes virtually eliminated</td>
</tr>
<tr>
<td>Feature envy classes</td>
<td>0</td>
<td>Feature envy classes eliminated</td>
</tr>
<tr>
<td>Large classes</td>
<td>Average method count: 5.44</td>
<td>Method count reduced by half</td>
</tr>
<tr>
<td></td>
<td>Maximum method count: 35</td>
<td></td>
</tr>
</tbody>
</table>
Conclusions (1)

- In the natural course of evolution of software, complexity shifts ascending from lower to higher structural levels (as in JFreeChart and Findbugs) or vice versa (as in Hibernate).
- This cycle of shifting complexity may lead to points of excessive complexity in a software product that we characterize as epochs.
- The discovery of these epochs suggests that if the pattern holds for most software products then mere refactoring at the code level (i.e., leaf packages and methods) may not be sufficient – every so often major restructuring at either the design or architectural level would be necessary to effectively manage structural complexity in software.
Conclusions (2)

- Measuring software complexity requires a comprehensive view
  - Techniques for software visualization, and automatic and semi-automatic approaches to assessment code quality go a long way in achieving this goal

- Introducing these techniques in a development organization has additional benefits
  - Software designers and developers become more effective in doing design and code reviews
  - Software architects can use these techniques for architecture reconstruction and for monitoring systems for architectural conformance
  - Software maintenance becomes easier as these techniques help in program comprehension
    - More than half the time during maintenance is spent understanding the system
  - These techniques stress the importance of creating systems that are easier to understand, maintain and enhance in the future
References (1)

Taming the Complexity: Managing the Quality of Architecture in Software Systems

References (2)

• R. Sangwan, P. Vercellone-Smith and P. Laplante. “Structural epochs in the complexity of software over time,” to appear in IEEE Software.