Software Engineering Research
@ The School of Graduate & Professional Studies

Jim Nemes, Division Head, Professor of Mechanical Engineering
Colin Neill, Associate Professor of Software Engineering
Raghu Sangwan, Assistant Professor of Software Engineering
Phil Laplante, Professor of Software Engineering

Software Engineering Group www.personal.psu.edu/cjn6/ASERG
PENN STATE GREAT VALLEY

- Penn State Great Valley School of Graduate Professional Studies, part of PSU's Graduate School
- Special Mission Campus
- Created to accommodate the busy schedules of full-time professionals
- Enrollment of 1900, 44 full-time and 69 part-time faculty members, student/professor ratio of 18:1
- Divisions: Engineering, Management, and Education
PSGV ENGINEERING
STUDENT
DEMOGRAPHICS

• Status
  – 95% Part-time

• Program Distribution
  – 60% Information Science
  – 25% Software Engineering
  – 15% Systems Engineering

• Common Employers
  – Lockheed Martin
  – Verizon
  – NAVSEA
  – PJM
  – Vanguard
  – L-3

• Age and Gender
  – Average is 33 yrs old
  – 75% Male, 25% Female

• Alma Maters
  – Penn State University
  – Lehigh University
  – Drexel University
  – Temple University
  – University of Michigan
  – University of Delaware
Presence in Philadelphia – Philadelphia Navy Yard

- Four million occupied square feet.
- 68 private companies.
- 6,500 Jobs.
- The Navy Yard Master Plan proposes a mixed-use development with capacity to support over 30,000 workers.

Building 100 – Innovation Center
The Navy Yard Keystone Innovation Zone (KIZ)

Technology commercialization initiative that promotes collaborative innovation between academic institutions, government research, and private industry.

The Navy Yard KIZ supports the growth of research and business activity focused on the following technology sectors:

- Power and Energy
- Nanotechnology
- Advanced Manufacturing
- Communications and IT
- Homeland Security
- Life Sciences
What PSGV Brings to the Center

- Solid Research Core in Software Engineering to complement existing expertise.
  - Software issues are key part of network security
- Liaisons with companies in the Philadelphia region
- Opportunities for large number of graduate students to become involved in applied research.
- Outreach opportunities in Malvern and PNY
- Students provide rapid knowledge transfer back to companies
Threats to Network Security

• Many threats can be tied to software vulnerabilities.
• But, security concerns are usually “tested-in” to the system rather than designed in from the beginning. This is typical of systemic properties, and these are the focus of our research.
• Using architecture-centric design we can develop systems that better meet their non-functional requirements by treating them as the primary drivers of architectural elaboration.
The Software Engineering Research Group

• Key members:
  – Dr. Colin J. Neill, Dr. Phillip A. Laplante & Dr. Raghu Sangwan

• Mission
  – Despite claims to the contrary, software systems decay over time. This is the driving force behind our research.
  – We seek methodologies to develop systems that are immune to this decay, develop techniques and tools to identify decay creeping into the system, and investigate approaches to repairing the decay without rebuilding.

• Projects
  – Architecture-centric design
    • Design approach that focuses on a systems “ilities” first, and its functionality second
  – Strategic refactoring and design repair
    • Repairing legacy systems where decay has made the system fragile and brittle.
  – Software complexity and stability
    • Tools and techniques to measure and monitor architectural coherence
  – Distributed and agile development
    • Methodological guidance in the technical and socio-cultural problems in complex system development.
Architecture-Centric Design

- Analysis and design techniques that focus on the design qualities of the system rather than the functional needs
- Benefits:
  - A system’s fitness to purpose is more closely related to design qualities than functionality
  - Design qualities are systemic properties so without specific attention, they will soon be obfuscated – particularly in distributed development.
  - Furthermore, these qualities are driven directly from business requirements so the system better reflects the needs of the business.

<table>
<thead>
<tr>
<th>Business Goal</th>
<th>Goal Refinement</th>
<th>Quality Attribute</th>
<th>Quality Attribute Scenario</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expand by entering new and emerging geographic markets</td>
<td>Support several international languages</td>
<td>Modifiability</td>
<td>A developer is able to package a version of the system with new language support in 80 person hours.</td>
<td>(H, M)</td>
</tr>
<tr>
<td>Support regulations that require life critical systems, such as fire alarms, to operate within specific latency constraints</td>
<td>Performance</td>
<td>Performance</td>
<td>A life critical alarm should be reported to the concerned users within three seconds of the occurrence of the event that generated the alarm.</td>
<td>(H, H)</td>
</tr>
</tbody>
</table>
• We have developed a design methodology that integrates architecture-centric design into standard object-oriented analysis and design.
• Approach was tested on a 2 year globally-distributed development project involving 8 universities in 5 countries.
• Reported in:
  – *Journal of Systems and Software*, May 2008 and
Starting from a monolithic system (left), we iteratively apply a series of architectural tactics corresponding to each quality attribute. The result is the coarse-grained architecture shown below.

From here we can design the fine-grained elements using standard OOAD techniques and patterns.
Strategic Refactoring

- Legacy systems exhibit significant architectural erosion, termed decay. At these times, extending and repairing the system can be overwhelming, and we are left with ‘wrapping’ it, or starting over.
- We’ve seen that tactical refactoring only shifts complexity around the system with no significant improvement.
- Instead, we’ve developed an alternative approach termed strategic refactoring:
  1. Determine target architecture
  2. Define target macro-architecture
  3. Identify micro-architectural patterns to implement changes
  4. Tease away

- Demonstrated it on several large scale applications:
  - Web-based equity management system that was deemed impossible to reuse. (IEEE IT Pro, 2003)
  - Imaging toolkit written in Java that was impossible to extend (Journal of Imaging Science and Technology, 2005)
## Starting point

<table>
<thead>
<tr>
<th>Domain Model</th>
<th>Design Model</th>
</tr>
</thead>
</table>

### Metric

<table>
<thead>
<tr>
<th>Metric</th>
<th>Measure</th>
<th>Compared to 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 hierarchy</td>
<td>Average: 0.207</td>
<td>Average reduced by a factor of 7 and the maximum reduced by a factor of 19</td>
</tr>
<tr>
<td></td>
<td>Maximum: 1</td>
<td></td>
</tr>
<tr>
<td>Data classes</td>
<td>1 (down from 25)</td>
<td>Data classes virtually eliminated</td>
</tr>
<tr>
<td>Feature envy classes</td>
<td>0 (down from 8)</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>
Architectural complexity and stability

- Lehman’s laws state that:
  - To remain useful a system must evolve
  - The structure of an evolving system degrades

- Consequently, regardless of the approach we take “emergent design” would be expected to become increasingly complex over the lifetime of the software unless work is done to reduce or maintain it

- Therefore, it is critical to monitor complexity in an application as it evolves through its different releases
Monitoring complexity

- We have developed a methodology for the comprehensive analysis of complexity in systems.
- We have employed this methodology on over 8 million SLOC in 6 medium and large scale open source systems.
- Our analysis revealed that:
  - A high proportion of structural complexity in the early releases is found at the application code level progressively migrating to higher level design and architectural elements in subsequent releases.
  - As evolution progresses low-level changes soon overwhelm the architecture once more, and the complexity reappears at the code level, and the pattern repeats.
- This suggests that mere code refactoring is not sufficient and project managers should plan for a major restructuring of systems periodically to effectively manage structural complexity.
  - We envision an “architecture dashboard” to provide automatic monitoring to support efforts.
The graph shows the degree and source of excessive complexity in each version of the system. V1.0.1 reveals a dramatic shift in the source of complexity between Tangles and Design Fat. These indicate that the package structure was radically collapsed.

In this chart we plot the packages based upon the abstractness and stability, color code them for their conformance with design heuristics, and expand them relative to their complexity. This reveals that 90% of excessively complex packages violate one (yellow) or two (red) core heuristics, and many deviate dangerously from the ‘main sequence’.