What Makes Good Software Engineering Research?

Everyday Dependability for Everyday Needs

Mary Shaw
Carnegie Mellon University
http://www.cs.cmu.edu/~shaw/

Everyday Software

- The computing game has changed
  - Internet supports mobility and a vast sea of resources
  - User expectations imply context-sensitive requirements
- Criteria for evaluating systems must change
  - Costs matter, not just capabilities
  - Specifications will inevitably be incomplete
  - “Good enough” is good enough
- The dependability game should also change
  - Reconcile conflicting objectives
  - Augment incomplete specs with user expectations
  - Use homeostasis as alternative to feedback
  - Provide compensation as alternative to repair
What Makes Good Software Engineering Research?

The Mobile Computing Challenge

- Limited hardware
  - Computer power, disk & memory capacity, battery
- Uncertain, dynamically varying services
  - Bandwidth, latency
  - Locally available information services
- Costly human attention
  - Individual, time-varying utility functions
  - Usage vs administration
  - Multi-user utility conflicts

Internet Resources as Components

Unlike conventional software components

- Autonomous
  - Independently created and managed
  - May change structure or format without notice
  - Availability, format, semantics may change
- Heterogeneous
  - Different packagings
  - Different business objectives, conditions of use
- Open affordances
  - Independent systems, not dependent components
  - Output usually for viewing, not computation
  - Incidental effects may be useful
What Makes Good Software Engineering Research?

Open Resource Coalitions

Objective: compose autonomous distributed resources
- “Coalitions” because the resources will not have a shared objective
- “Open” in contrast to control assumed for closed-shop development

This changes everything!

What’s changed?

Classical  New
Localized Distributed
Independent Interdependent
Installations Communities
Centrally-administered User-managed
Software Resource
Systems Coalitions

Institute for Software Research, International
What Makes Good Software Engineering Research?

**Everyday Software**

- The computing game has changed
  - Distributed interdependent communities of user-managed resource coalitions
- Criteria for evaluating systems must change
  - Costs matter, not just capabilities
  - “Good enough” is good enough
- The dependability game should also change
  - Reconcile conflicting objectives
  - Augment incomplete specs with user expectations
  - Use homeostasis as alternative to feedback
  - Provide compensation as alternative to repair

**Context-Sensitive Requirements**

- Different users have ...
  - ...different tolerance for system error and failure
  - ...different interests in results from a resource
  - ...different tolerance and interests at different times
- Criteria for proper operation should reflect these differences
  - Requirements can’t be tied solely to resource
  - Users need ways to express differences
- Multiple co-located users must mediate preferences
- Need user-centered requirements as part of resource composition techniques
What Makes Good Software Engineering Research?

**Sufficient Correctness**

- Traditional model of program correctness
  - Gold standard is functional correctness
  - For systems, also need extrafunctional properties

- In practice
  - Most software in everyday use has bugs …
    - … yet we get work done
  - It isn’t practical to get complete specifications
    - Too many properties people can depend on
    - Variable confidence in what we do know
    - Too expensive to collect specification information
    - Specifications should reflect users’ needs
  - We don’t really need “correctness”, but rather assurance that the software is good enough for its intended use
What Makes Good Software Engineering Research?

<table>
<thead>
<tr>
<th>Programs</th>
<th>vs</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete knowledge</td>
<td></td>
<td>Approximate knowledge</td>
</tr>
<tr>
<td>Goal: correctness</td>
<td></td>
<td>Goal: adequacy, fitness</td>
</tr>
<tr>
<td>Failure prevention</td>
<td></td>
<td>Problem remediation</td>
</tr>
<tr>
<td>Good component specs</td>
<td></td>
<td>Components poorly understood</td>
</tr>
<tr>
<td>Monolithic design</td>
<td></td>
<td>Cohesion/ coupling issues</td>
</tr>
<tr>
<td>Stable configuration</td>
<td></td>
<td>Shifting (dynamic) parts</td>
</tr>
<tr>
<td>Open loop operation</td>
<td></td>
<td>Closed loop operation</td>
</tr>
<tr>
<td>Requirements tied to</td>
<td></td>
<td>Requirements sensitive</td>
</tr>
<tr>
<td>components</td>
<td></td>
<td>to context of use</td>
</tr>
<tr>
<td>Greenfield</td>
<td></td>
<td>Brownfield</td>
</tr>
<tr>
<td>Cost not a major factor</td>
<td></td>
<td>Cost a design driver</td>
</tr>
<tr>
<td>Creating capability</td>
<td></td>
<td>Creating value</td>
</tr>
</tbody>
</table>

The Value Proposition

- Engineering seeks timely, cost-effective solutions to practical problems, preferably based on math and science
  - This entails reconciling conflicting constraints.
  - This entails making decisions with limited time, knowledge, and resources.
  - This entails understanding the contribution of design decisions to cost as well as to capability.

... and so...

- The objective of software engineering should be to create value, not simply to create capability.
What Makes Good Software Engineering Research?

Value-based Software Engineering

- Include cost-benefit tradeoffs in technical decisions
  - cost-benefit of getting information as well as of analysis
  - cost-benefit of ownership as well as of development
- Adapt techniques such as
  - machine learning, multi-attribute decision theory
  - utility theory, linear programming
  - real options, classical optimization
  - game theory, portfolio selection
  - from business, economics, social sciences
- Harnessing the knowledge of other disciplines, especially social scientist, in service of better software

Everyday Software

- The computing game has changed
  - Distributed interdependent communities of user-managed resource coalitions
- Criteria for evaluating systems must change
  - User-centered requirements
  - Sufficient correctness
  - Value creation, not just capability creation
- The dependability game should also change
  - Reconcile conflicting objectives
  - Augment incomplete specs with user expectations
  - Use homeostasis as alternative to feedback
  - Provide compensation as alternative to repair
What Makes Good Software Engineering Research?

Ways to deal with failure

- Traditional: prevent through careful development, analysis
- User centered: set criteria for proper operation to reflect user needs
- Fault tolerant: repair failures as they occur
- Compensatory: provide financial compensation

Traditional User-centered Fault-tolerant Compensatory

Validation Remediation

Global std Relative std Technical Economic

Bad thing

Prevention Repair

Detection Remediaion
What Makes Good Software Engineering Research?

Security technology portfolio selection

- Different sites have different security issues
- Elicit concerns about threats and relative priorities with multi-attribute decision techniques
  - Converts subjective comparisons to quantitative values
- Associate threat analysis with cost of successful attack and countermeasures available in the market
  - Consider cost-effectiveness and defense in depth
- Iterate, using sensitivity analysis and multiattribute techniques to refine recommendations
  - Get better understanding as well as recommendation
- Shawn Butler (finishing PhD this year)
  - Papers in ICSE 2002, CERIAS 2002

Utility-based Adaptive Configuration

- Mobile systems are resource-limited
  - Processor power, bandwidth, battery life, storage capacity, media fidelity, user distraction, ...
- Users require different capabilities at different times
  - Editing, email, viewing movies, mapping, ...
  - Dynamic preferences for quantity and quality of service
- Abstract capabilities can be provided by different combinations of services
  - Specific editors, browsers, mailers, players, ...
- Use utility theory and linear/integer programming to find best set and configuration of services
- Vahe Poladian (2nd year PhD student)
What Makes Good Software Engineering Research?

**Idea: Multidimensional cost analysis**

- **Types of cost**
  - Dollars, computer resources, user distraction, staff time, reputation, schedule, lives lost
- **Naïve view**
  - Convert all costs to a single scale, e.g., dollars
- **Problem**
  - Cost dimensions have different properties
- **Resolution**
  - Carry cost vector as far into analysis as possible
  - Convert to single scale at the latest point possible
- Butler and Poladian, independently

**Idea: Calculus of preference**

- **Needed**: a way to reconcile conflicting information
  - Multiple stakeholders
  - Multiple sources of credential information
  - Nonmonotonic information
- **Possible contributing technologies**
  - Utility theory: combining utility functions
  - Multi-attribute decision theory
  - Auctions
  - Priority scheduling
  - Engineering design judgments for reconciling conflicting constraints
Ways to deal with failure

- **Bad thing**
  - Prevention
  - Detection
  - Remediation

- **Validation**
  - Global std
  - Relative std

- **Remediation**
  - Technical
  - Economic

- **Traditional**
- **User-centered**
- **Fault-tolerant**
  - Compensatory

- Traditional: prevent through careful development, analysis
- User centered: set criteria for proper operation to reflect user needs
- Fault tolerant: repair failures as they occur
- Compensatory: provide financial compensation

Types of fault tolerance

- **Distinguish external from internal environment**
  - System has control over internal environment
  - External environment operates independently

- **Fault-tolerance through feedback**
  - Internal: detect system state, compare to criterion, repair if necessary
    - Load balancing, adaptive integration
  - External: attempt to infer external state, compare to objective, adapt if necessary
    - Control of mechanical systems

- **Homeostasis**
  - Design so normal operation maintains good conditions
    - Internet packet routing, background garbage collection
What Makes Good Software Engineering Research?

Reactive Fault Tolerance

Normal

Fault tolerant

Anomaly Detection

If you have specifications, you can detect violations
Most everyday software does not have good specs
Problem: how to discover “normal” behavior and capture this as predicates
  - Infer predicates from resource’s history
  - Set-up elicits user expectations while tuning predicates
  - Operation applies inferred predicates
Inferred predicates serve as proxies for specs
Orna Raz (PhD thesis research in progress)
  - Paper in ICSE 2002
What Makes Good Software Engineering Research?

**Reaction and Homeostasis**

- Normal
  - Normal
  - Broken
- Fault tolerant
  - Normal
  - Degraded
  - Broken
- States -> Gradients
  - Normal
  - Degraded
  - Broken

**Idea: Aggregate Reasoning**

- Recognize that software systems are too complex for exact analysis
  - We don’t understand gasses by solving the N-body problem for extremely large N. Instead, we use the aggregate gas laws $PV = nRT$.
- Seek aggregate models with system-level abstractions
  - Anomaly detection, software component insurance
  - Probabilistic certification of software components provides alternative to verification (Wallnau)
  - Exact “webs of trust” would be fragile; based on preponderance of evidence they might be robust.
  - Modeling Internet as “scale-free system” yields new results, e.g. about virus spread characteristics (Barabasi).
What Makes Good Software Engineering Research?

Ways to deal with failure

- Bad thing
  - Prevention
  - Detection
  - Validation
  - Remediation

- Traditional: prevent through careful development, analysis
- User centered: set criteria for proper operation to reflect user needs
- Fault tolerant: repair failures as they occur
- Compensatory: provide financial compensation

Compensation, not Prevention

- For everyday software, compensation may be a reasonable alternative to repair
  - Especially for time-dependent results
  - Especially if consequences of failure are large enough to matter but not large enough to be catastrophic

- Compensation techniques need
  - Actuarial model
    - Failure rate prediction based on component history
    - Definitions of share-risk pools
  - Ways to identify failure (e.g., anomaly detection)
  - Means of assessing damages

- Software component insurance
- Paul Li (2nd year PhD student)


Everyday Software

- The computing game has changed
  - Distributed interdependent communities of user-managed resource coalitions
- Criteria for evaluating systems must change
  - User-centered requirements
  - Sufficient correctness
  - Value-based software engineering
- The dependability game should also change
  - Portfolio selection
  - Anomaly detection
  - Homeostasis
  - Software component insurance

For everyday software, set criterion for dependability as “fitness for the task at hand”

Consider a wide range of approaches to achieving dependability

Achieve value through technical approaches adapted from economics and social science
What Makes Good Software Engineering Research?

Contacts

Mary Shaw
- mary.shaw@cs.cmu.edu
- http://cs.cmu.edu/~shaw/

Students
- Shawn Butler (security technology selection)
- Orna Raz (semantic anomaly detection)
- Vahe Poladian (mobile dynamic configuration)
- Paul Li (software component insurance)

Assistant
- Janet New Hilf