The Value Proposition for Everyday Dependability

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

Dependability in the real world

- Dependability needs arise from user expectations
  - Different users need different properties, quality levels
  - “Good enough” is often good enough
- Costs matter, not just capabilities
  - Costs are of many types: money, delay, disruption, …
  - Few can afford highest dependability regardless of cost
- Uncertainty is inevitable
  - Specifications will not be complete
  - Actual operating environment is not known
- Integration is a bigger challenge than components
  - Interoperability and shared assumptions are hard
  - Many stakeholders mean many objectives
The Value Proposition for Everyday Software

Themes

Thinking about dependability

Dependability and value for specific users

Uncertainty in ultra-large-scale systems

Predicting value from design

A note about choosing research problems

An important part of research is picking good questions – good problems – to work on

An important part of software engineering research is picking research questions whose answers will improve our ability to design, develop, and maintain software

Therefore I will discuss good research opportunities as well as established results
The Value Proposition for Everyday Software

Themes

Thinking about dependability
Dependability and value for specific users
Uncertainty in ultra-large-scale systems
Predicting value from design

Ways to achieve dependability

Potential problem (bad thing)

Validation

Prevention

Remediation

Reaction

- Global std
- Relative std
- Policy std
- Technical reactive
- Technical adaptive
- Economic

Traditional
User-centered
Ultra-large scale
Fault-tolerant
Self healing
Compensatory

- Traditional: prevent problems through careful development, analysis
- User centered: set dependability criteria to reflect user needs
- Ultra-large-scale: negotiate dependability criteria among stakeholders
- Fault tolerant: detect and repair problems when they occur
- Self healing: operate adaptively to avoid problems
- Compensatory: provide financial compensation
Specifications: Conventional Doctrine

Component specification is

- **sufficient and complete** -- says everything you need to know or may rely on
- **static** -- written once and frozen
- **homogeneous** -- written in single notation

“Three prerequisites must be met for a component to be used in more than one system:
- complete, opaque enclosure;
- complete specification of its external interface;
- and design consistency across all sites of reuse.

Without these, reuse will remain an empty promise.”

Real-World (Incomplete) Specifications

- **Heterogeneous**
  - Many kinds of information: functional, structural, extra-functional, family properties
  - Many types of values: integer, formula, narrative
- **Intrinsically incomplete**
  - Open-ended needs: cannot anticipate all properties of interest
  - Cost of information: impractical, even for common properties
- **Evolving**
  - Partial information: understanding commensurate with amount of information
  - New properties: additional properties added as discovered

This is the case for many components and most systems.
When is “Good” Enough?

- Traditional model of program correctness
  - Gold standard is functional correctness
  - Also need extrafunctional properties such as performance, reliability, security, accuracy, usability, ...

- 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

How much must you trust your software?

- Degree of Oversight
  - Full oversight, manual operation
  - Near real-time weather
  - Automatic sports statistics
  - Stock market alerts
  - Ambulance scheduling
  - Web search for disease information
  - Appointment scheduling
  - Drug interactions
  - Patient monitoring
  - Missile guidance
  - Nuclear safety devices

- Consequences of Failure
  - Catastrophe
  - Inconvenience

- Full oversight, manual operation
- None: full automation, unattended operation
Mobile Computing

- 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

- Information: unstructured text, formatted text, databases, live data feeds, images, maps, current status (e.g., inventory)
- Calculation: reusable software components, applications that can be invoked remotely
- Communication: messages, streaming media, synchronous communication, agent systems, alert/notification services
- Control: coordination for use of resources, registration and subscription services
- Services: secondary (processed) information, simulation, editorial selection, evaluation, responsive experts
Internet Resources

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 afforances
  - Independent systems, not dependent components
  - Output usually for viewing, not computation
  - Incidental effects may be useful

Open Resource Coalitions

Objective: compose autonomous distributed resources

- “Coalitions” because the resources will not share objectives
- “Open” in contrast to control assumed for closed-shop development

This changes everything!
Technical Issues for Resource Coalitions

**What does a resource do?**
- Specifications are worse than component specifications
  - Must handle partial and uncertain information

**How do we create coalitions?**
- More than screen-scraping, glue, XML, std types
  - Use loose coupling, substitutability, new architectures

**How well must a coalition work?**
- Connectivity, format, semantics can all change
  - Determine when results are good enough
  - Create adaptive architectures

---

What’s changing?

<table>
<thead>
<tr>
<th>Classical</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>Localized</td>
<td>Distributed</td>
</tr>
<tr>
<td>Independent</td>
<td>Interdependent</td>
</tr>
<tr>
<td>Insular</td>
<td>Vulnerable</td>
</tr>
<tr>
<td>Installations</td>
<td>Communities</td>
</tr>
<tr>
<td>Centrally-administered</td>
<td>User-managed</td>
</tr>
<tr>
<td>Software</td>
<td>Resource</td>
</tr>
<tr>
<td>Systems</td>
<td>Coalitions</td>
</tr>
</tbody>
</table>
Everyday Software

The computing game has changed
- User expectations lead to context-sensitive requirements
- Internet supports mobility and a vast sea of resources

Criteria for evaluating systems must change
- Value is in the mind of the beholder
- “Good enough” is good enough
- Costs matter, not just capabilities

The dependability game should also change
- Augment incomplete specs with user expectations
- Reconcile conflicting objectives
- Use background adaptation as alternative to reaction
- Provide compensation as alternative to repair

Themes

Thinking about dependability

Dependability and value for specific users

Uncertainty in ultra-large-scale systems

Predicting value from design
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.

Value can not be evaluated in isolation

- It only makes sense as value to some stakeholder.
- Value is benefit net of cost to a particular stakeholder.

Ways to achieve dependability

Potential problem (bad thing)

Validation

- Global std
- Relative std
- Policy std

- Traditional
- User-centered
- Ultra-large scale

Remediation

- Technical reactive
- Technical adaptive
- Economic

- Fault-tolerant
- Self-healing
- Compensatory

- Prevention
- Reaction

Traditional: prevent problems through careful development, analysis
User centered: set dependability criteria to reflect user needs
Ultra-large-scale: negotiate dependability criteria among stakeholders
Fault tolerant: detect and repair problems when they occur
Self healing: operate adaptively to avoid problems
Compensatory: provide financial compensation
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
- Evaluation criteria should reflect these differences
  - Requirements can’t be tied solely to resource
  - Users need ways to express individual preferences
- Multiple co-located users must mediate preferences
- Need *user-centered requirements* as part of resource composition techniques

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 (CMU ‘03)
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 series of configurations of services

Vahe Poladian (5th year PhD student)

Ways to achieve dependability

Potential problem (bad thing)

Validation
  - Global std
  - Relative std
  - Policy std

Remediation
  - Technical reactive
  - Technical adaptive
  - Economic

Validation
  - Traditional
  - User-centered
  - Ultra-large scale

Fault-tolerant
  - Self healing

Compensatory
  - provide financial compensation

Traditional: prevent problems through careful development, analysis
User centered: set dependability criteria to reflect user needs
Ultra-large-scale: negotiate dependability criteria among stakeholders
Fault tolerant: detect and repair problems when they occur
Self healing: operate adaptively to avoid problems
Compensatory: provide financial compensation
Reactive Fault Tolerance

**Normal**

```
Normal -> Pos -> Broken
```

**Fault tolerant**

```
Normal -> Pos -> P^BN -> Broken
P^BN -> NoPos -> P^BD
P^BD -> NoPos -> P^BP
P^BP -> NoPos -> Degraded
Degraded -> NoPos -> P^BN
```

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
    - Multiple statistical, data mining techniques
    - Set-up: elicit user expectations while tuning predicates
      - Using templates that show what techniques can express
    - Operation: apply inferred predicates to detect anomalies
- Inferred predicates serve as proxies for specs
- “Anomaly” is in the eye of the specific user
- Orna Raz (CMU ‘04)
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
    - Feedback control of embedded systems
- Homeostasis
  - Design so normal operation maintains good conditions
    - Internet packet routing, background garbage collection

Ways to achieve dependability

- Traditional: prevent problems through careful development, analysis
- User centered: set dependability criteria to reflect user needs
- Ultra-large-scale: negotiate dependability criteria among stakeholders
- Fault tolerant: detect and repair problems when they occur
- Self healing: operate adaptively to avoid problems
- Compensatory: provide financial compensation
Reaction and *Homeostasis*

**Normal**
- Normal
- Broken

**Fault tolerant**
- Normal
- Fault tolerant
- Degraded

**States \(\rightarrow\) Gradients**
- Normal
- Degraded
- Broken

**Idea: Homeostasis**

- Instead of distinguishing “good” and “bad” states, react to all change in a way that preserves goodness.
- Analogy: biological and ecological systems.
- Examples:
  - Background maintenance (garbage collection)
  - Dynamic resource selection (Internet packet routing)
  - Slack, excess capacity (service capacity)
- Advantages:
  - Mechanisms operate over a wide range of performance.
  - Performance improves independent of health status.
  - Does not require precise distinctions between internal states, freeing that effort for other aspects of design.

- Paper at WOSS02

Institute for Software Research
Ways to achieve dependability

Potential problem (bad thing)

Validation
- Prevention
  - Global std
  - Relative std
  - Traditional
Remediation
- Reaction
  - Policy std
  - User-centered
  - Ultra-large scale
  - Fault-tolerant
Technical
- Technical reactive
  - Self healing
  - Economic
Compensatory

- Traditional: prevent problems through careful development, analysis
- User centered: set dependability criteria to reflect user needs
- Ultra-large-scale: negotiate dependability criteria among stakeholders
- Fault tolerant: detect and repair problems when they occur
- Self healing: operate adaptively to avoid problems
- 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 moderate

- 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 (5th year PhD student) working on defect rates
The Value Proposition for Everyday Software

Themes

Thinking about dependability

Dependability and value for specific users

Uncertainty in ultra-large-scale systems

Predicting value from design

Ways to achieve dependability

Potential problem (bad thing)

Validation

Remediation

- Traditional: prevent problems through careful development, analysis
- User centered: set dependability criteria to reflect user needs
- Ultra-large-scale: negotiate dependability criteria among stakeholders
- Fault tolerant: detect and repair problems when they occur
- Self healing: operate adaptively to avoid problems
- Compensatory: provide financial compensation
Ultra-Large-Scale Systems

- Large size on many dimensions
  - Lines of code, users, users’ objectives, amount of data, dependencies among components, etc
- Scale changes everything

Characteristics
- Decentralized operation and control
- Conflicting, unknowable, diverse requirements
- Continuous evolution and deployment
- Heterogeneous, inconsistent, changing elements
- Indistinct people/system boundary
- Normal failures
- New forms of acquisition and policy

Analogy: Cities and city planning

- Cities are complex systems
  - Built of individual components chosen by individuals
  - Constantly evolve
  - Withstand failures and attacks
- Cities are not centrally controlled
  - Standards for infrastructures
    - Building codes, highway standards
  - Policies that allow individual action within constraints
    - Zoning laws
  - Regulations that govern individual action
    - Enforcement after the fact, rather than prior constraint
**Decentralized operation and control**

- ULS system scale offers only limited possibilities for central or hierarchical control
  - Long life, multiple users and objectives, span of physical jurisdictions are the norm at ULS scale
  - Many versions of subsystems must work together
  - Modifications are developed and installed by independent groups
  - Spontaneous, unanticipated new uses arise

Undermines common assumption:
- All conflicts must be resolved and must be resolved uniformly

**Conflicting, unknowable, diverse requirements**

- ULS systems serve wide range of competing needs
  - Competing users contend for requirements
  - Understanding of problem evolves
  - Dependability is “better/worse”, not “right/wrong”
  - Wicked problems

Undermines common assumptions:
- Requirements are known in advance, evolve slowly
- Tradeoff decisions will be stable
Wicked Problems

Wicked problems violate the assumptions that must be made to use the problem solving methods of our familiar tame problems.

Wicked problems:

- cannot be easily defined so that all stakeholders agree on the problem to solve;
- have better or worse solutions, not right and wrong ones;
- have no objective measure of success;
- have no given alternative solutions-these must be discovered;
- often have strong moral, political or professional dimensions.


Characteristics of Wicked problems

1. There is no definitive formulation of a wicked problem.
2. Wicked problems have no stopping rule.
3. Solutions to wicked problems are not true-or-false, but good-or-bad.
4. There is no immediate and no ultimate test of a solution to a wicked problem.
5. Every implemented solution to a wicked problem has consequences.
6. Wicked problems do not have a well-described set of potential solutions.
7. Every wicked problem is essentially unique.
8. Every wicked problem can be considered to be a symptom of another problem.
9. The causes of a wicked problem can be explained in numerous ways.
10. The planner (designer) has no right to be wrong.

Institute for Software Research
Continuous evolution and deployment

- ULS systems have long lives and multiple independent developers
  - Different groups may install capability for their own needs; this may conflict with other groups
  - Evolution can’t be controlled centrally, must be shaped by rules and policies that protect critical services and allow diversity at the edges
Undermines common assumption:
- System improvements introduced in “releases”
- Users know about releases and can choose to accept them or not

Heterogeneous, inconsistent, changing elements

- ULS systems will be composed from independently-created components
  - Heterogeneous: many sources, no single interface standard, often incorporating legacy systems
  - Inconsistent: evolution spontaneous, not planned; different objectives may cause inconsistent versions
  - Changing: hardware, software, operating environment change based on local decisions
Undermines common assumptions:
- Effect of change can be predicted adequately
- Configuration information is accurate and controlled
- Components and users are fairly homogeneous
Indistinct people/system boundary

- ULS systems’ service to a user depends on actions of other users; user/developer distinction soft
  - User actions may affect overall system health
  - System must adapt to changing usage patterns
  - Aggregate analysis may be better than exact analysis

Undermines common assumption:
- Users’ behavior doesn’t affect overall system
- Collective behavior of people is not relevant
- Social interactions are not relevant

Normal failures

- ULS system scale implies inevitable failures, so systems must do protection/recovery/enforcement
  - Hardware failures inevitable because of scale
  - Legitimate use of software and services outside planned capability will cause degradation/failure
  - Malicious use will cause problems

Undermines common assumptions:
- Failures will be infrequent and exceptional
- Defects can be removed
New forms of acquisition and policy

ULS systems will evolve, but there must be governance to prevent anarchy

- Success of system depends on organic evolution
- Individual developers won’t fully understand core infrastructure
- Need effective guidance on allowed/unallowed change

Undermines common assumption:
- There is a single agent responsible for system development, operation, and evolution

ULS Research Opportunities

- Human interaction
- Computational emergence
- Design
- Computational engineering
- Adaptive system infrastructure
- Adaptable and predictable system quality
- Policy, acquisition, and management
The Value Proposition for Everyday Software

Themes

Thinking about dependability

Dependability and value for specific users

Uncertainty in ultra-large-scale systems

Predicting value from design

We need better ways to analyze a software design and predict the value its implementation will offer to a customer or to its producer
The Value Proposition for Everyday Software

Engineering design

Engineers . . .
- iterate through design alternatives
- reconcile client’s constraints
- consider cost & utility as well as capability
- recognize that early decisions affect later costs

. . . but . . .

Software engineers . . .
- lack adequate techniques for early analysis of design
- design for component spec rather than client expectation
- rarely include cost as 1st-class design consideration
Why does early design evaluation matter?

Cost of repair

- Fixing problems after delivery often costs 100x more than fixing them in requirements and design
- Up to half of effort goes to avoidable rework
  - “avoidable rework” is effort spent fixing problems that could have been avoided or fixed earlier with less effort
- Early reviews can catch most of the errors

Cost of delaying risk management

-- Barry Boehm

Institute for Software Research
Why does early design evaluation matter?

Cost of repair
- Fixing problems after delivery often costs 100x more than fixing them in requirements and design
- Up to half of effort goes to avoidable rework
  - "avoidable rework" is effort spent fixing problems that could have been avoided or fixed earlier with less effort
- Early reviews can catch most of the errors

... but ...

Confidence in estimates is lowest early in a project

Confidence in estimates

Software costing and sizing accuracy vs phase
Why does early design evaluation matter?

- **Cost of repair**
  - Fixing problems after delivery often costs 100x more than fixing them in requirements and design
  - Up to half of effort goes to avoidable rework
  - “avoidable rework” is effort spent fixing problems that could have been avoided or fixed earlier with less effort
  - Early reviews can catch most of the errors

  ... but ...

- Confidence in estimates is lowest early in a project
- Early decisions commit most of the resources

 Costs, commitment, and uncertainty

- Engineering involves deciding how to make irreversible commitments in the face of uncertainty

![Graph showing cost commitment and uncertainty](image)

- Usual view: cumulative costs incurred to date
- Risk-aware view: costs committed to date

— Art Westerburg, personal communication
The Value Proposition for Everyday Software

Current software design evaluation

- Relatively little attention to early design evaluation
  - design
- Software-centric evaluations
  -
- Minor role for costs other than development
  -
- Sparse, scattered, inconsistent evaluation methods
  -

Leverage lower cost of change during design

Software-centric evaluations

- Consider user-specific preferences, or perceived value

Minor role for costs other than development

- Expand role for larger-scale economic issues

Sparse, scattered, inconsistent evaluation methods

- Find ways to use models together
**Value–based approach to dependability**

- Value is benefit net of cost
  - Value to a given client is benefit net of cost to *that* client
- Dependability involves uncertainty
  - ... about properties of software components
  - ... about interactions of components or systems
  - ... about operating environment
  - ... about consequences of failure
- Value of dependability involves prediction and risk management
- Benefits and costs are largely set early in design
  - But at that time benefits and costs can only be predicted

**Economists’ view of value**

- A firm’s goal is typically to maximize total revenue minus cost of the inputs, represented by
  \[
  \max \left( B(z) - C(y) \right) \quad \text{such that} \quad F(y, z) \leq 0
  \]
- Here
  - In vector \( z \), \( z_j \) represents quantity of product \( j \) sold
  - \( B(z) \) is the total revenue from selling those products
  - In vector \( y \), \( y_i \) represents quantity of input \( i \) consumed
  - \( C(y) \) is the total cost of those inputs
  - \( F(y, z) \) is a vector, as well, so \( F(y, z) \leq 0 \) represents a list of equations representing constraints on the problem
The Value Proposition for Everyday Software

Early, code-free, design evaluation

- Target of evaluation
  - very high level design, before “software design” methods start elaborating the box and line diagrams
  - evaluation that weighs costs as well as capabilities
  - evaluation that recognizes user needs and preferences
  - evaluation that does not depend on access to code

- Long-term objective: framework to unify models
  - general, to handle models for various specific attributes
  - open-ended, esp. with respect to the aspects considered
  - flexible, handling various levels of detail and precision

Model for predictive analysis of design

\[ U(d, \theta) = B(x, \theta) - C(d, x, m) \] for \( \{ x : F(d, x, m) \} \), where \( x = P(d, m) \)

Value \( U \) of design \( d \) to a client with preferences \( \theta \) is benefit \( B \) net of cost \( C \) provided the desired result \( x \) is achievable and attributes \( x \) of implementation are predicted by \( P \)

Let
- \( d \) be a design in some appropriate notation
- \( x \) be in \( A^n \) an open-ended vector of capabilities
- \( \nu \) be in \( V^n \) a multidimensional value space
- \( m \) be in some notation a development method
- \( \theta \) express user pref a multidimensional utility space
- \( B \) express benefits predicted value \( \nu \) of \( x \) to user with pref \( \theta \)
- \( C \) express costs cost \( \nu \) of getting \( x \) from \( d \) with method \( m \)
- \( F \) checks feasibility whether \( d \) with \( x \) can be achieved with \( m \)
- \( P \) predicts capabilities attributes \( x \) that \( m \) will deliver for \( d \)
Basic value proposition

\[ U(d, \theta) = B(x, \theta) - C(d, x, m) \text{ for } \{ x : F(d, x, m) \}, \text{ where } x = P(d, m) \]

Following economics, value is benefit net of cost

Adopting a software tool will cost \( \$X \), and it will save you \( \$Y \), right away, on your current project.

\[ U = \$Y - \$X \]

Value based on product attributes

\[ U(d, \theta) = B(x, \theta) - C(d, x, m) \text{ for } \{ x : F(d, x, m) \}, \text{ where } x = P(d, m) \]

The value of a design is the benefit, net of cost, of the implementation as represented by its capabilities.

Let \( d \) be a design in some appropriate notation
\( x \) be in \( \mathbb{R}^n \) an open-ended vector of capabilities
\( v \) be in \( \mathbb{R} \) value in dollars

\( B \) express benefits predicted value \( v \) of \( x \) to user
\( C \) express costs cost \( v \) of getting or using \( x \)
Ex 2: Choosing a representation

- You store maps to view and edit in drawing package
- Only 1 of every 50 reads leads to a write
- Cost: $10K per sec read/write, $0.1/KB storage
- You get data for your typical data sets:

<table>
<thead>
<tr>
<th>File type</th>
<th>Seconds to open (read)</th>
<th>Seconds to write (save or export)</th>
<th>File size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>6</td>
<td>93</td>
<td>6243</td>
</tr>
<tr>
<td>EMF</td>
<td>9</td>
<td>88</td>
<td>17908</td>
</tr>
<tr>
<td>EPS</td>
<td>5</td>
<td>17</td>
<td>20909</td>
</tr>
<tr>
<td>PDF</td>
<td>7</td>
<td>95</td>
<td>6243</td>
</tr>
<tr>
<td>WMF</td>
<td>5</td>
<td>86</td>
<td>11038</td>
</tr>
</tbody>
</table>

Best representation *for this application*

<table>
<thead>
<tr>
<th>Design d &lt;format&gt;</th>
<th>Attributes x &lt;time,size&gt;</th>
<th>Cost v &lt;total $&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>&lt;393, 6243&gt;</td>
<td>$4554</td>
</tr>
<tr>
<td>EMF</td>
<td>&lt;538, 17908&gt;</td>
<td>17908</td>
</tr>
<tr>
<td>EPS</td>
<td>&lt;267, 20909&gt;</td>
<td>4761</td>
</tr>
<tr>
<td>PDF</td>
<td>&lt;445, 6243&gt;</td>
<td>5424</td>
</tr>
<tr>
<td>WMF</td>
<td>&lt;336, 11038&gt;</td>
<td>4464</td>
</tr>
</tbody>
</table>
Ex 3: Determining value of features

For spreadsheets,
- Adherence to dominant standard $\rightarrow$ 46% higher price
- 1% increase in installed base $\rightarrow$ 0.75% increase in price
- Quality-adjusted prices over 5 years declined 16%/year

Hedonic model a good predictor
- Hedonic model estimates value of product aspects to consumer’s utility or pleasure; it assumes price is a function of product features

Econometric analysis of spreadsheet market, 1987-92

Predicting attributes from design

\[ U(d, \theta) = B(x,\theta) - C(d,x,m) \text{ for } \{ x : F(d,x,m) \} \text{ where } x = P(d,m) \]

We often need to predict the implementation properties \( x \) before the code is written

\( d \) be a design in some appropriate notation
\( x \) be in \( \mathbb{R}^n \) an open-ended vector of capabilities
\( v \) be in \( \mathbb{R} \) value in dollars
\( m \) be in some notation a development method

\( B \) express benefits predicted value \( v \) of \( x \) to user
\( C \) express costs cost \( v \) of getting \( x \) from \( d \)

\( P \) predicts capability capabilities \( x \) of implementation of
Ex 4: Predicting size from function points

COCOMO Early Design

- Examine design to count function points

<table>
<thead>
<tr>
<th>Type</th>
<th>Complexity Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
</tr>
<tr>
<td>Internal logical files</td>
<td>7</td>
</tr>
<tr>
<td>External interface files</td>
<td>5</td>
</tr>
<tr>
<td>... etc ...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Choose programming language
- Use pre-calibrated table to estimate code size

<table>
<thead>
<tr>
<th>Language</th>
<th>LOC per Fcn Pt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada 95</td>
<td>49</td>
</tr>
<tr>
<td>C++</td>
<td>55</td>
</tr>
<tr>
<td>Java</td>
<td>53</td>
</tr>
<tr>
<td>PERL</td>
<td>27</td>
</tr>
<tr>
<td>VB 5.0</td>
<td>34</td>
</tr>
</tbody>
</table>

-- Boehm, COCOMO II, 2000

Ex 5: Predicting mobile performance

Given a configuration of applications to support a user task, what will its resource requirements be?

Design $d$ is “configuration” expressed as

$\{<\text{application}, (QoS settings)>\}$

$\{<\text{Windows Media Player},
(24 \text{ fps}, 300x200, \text{high quality audio})>
<\text{MS Word},
()>,
<\text{Firefox},
(5 s, \text{text})>\}$
Ex 5: Predicting mobile performance

Empirical profiling yields resource usage
Implementation attributes maintain distinctions among resource consumers:

\{ <application, (QoS settings), resource usage> \}
\{ <Windows Media Player,
  (24 fps, 300x200, high quality audio),
  (25\%, 256 Kpbs, 30 MB)>,
  <MS Word,
  ( ),
  (2\%, 0 Kpbs, 28 MB)>,
  <Firefox,
  (5 s, text),
  (8\%, 56 Kpbs, 10 MB)> \}
**Time ≠ Money**

\[
U(d, \theta) = B(x, \theta) - C(d, x, m) \text{ for } \{ x: F(d, x, m) \}, \text{ where } x = P(d, m)
\]

Capabilities \(x\) and values \(v\) are multidimensional; they may be measured on different scales.

Let
- \(d\) be a design in some appropriate notation
- \(x\) be in \(A^n\) open-ended vector of arbitrary attributes
- \(v\) be in \(V^n\) open-ended vector of arbitrary attributes
- \(m\) be in some notation a development method

\(B\) express benefits predicted value \(v\) of \(x\) to user
\(C\) express costs cost \(v\) of getting \(x\) from \(d\) with method \(m\)

\(P\) predicts capability capabilities \(x\) that \(m\) will deliver for

**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
Multidimensional Cost Analysis

» Different factors in a problem are appropriately measured in different ways
  ✤ Dollars, computer resources, user distraction, staff time, reputation, schedule, lives lost

» It’s tempting to convert everything to dollars, but this can lead to …
  ✤ Loss of information related to different properties
  ✤ Errors by converting nominal, ordinal, or interval scales to a ratio scale
  ✤ Loss of flexibility by early choice of conversion
  ✤ Confusion of precision with accuracy

» Many analysis techniques require a single cost unit, but you should delay conversion as long as possible

Measurement Scales and Scale Types

<table>
<thead>
<tr>
<th>Scale</th>
<th>Intuition</th>
<th>Preserves</th>
<th>Example</th>
<th>Legitimate transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>Simple classification, no order</td>
<td>Differences</td>
<td>Horse, dog, cat</td>
<td>Any one-to-one remapping</td>
</tr>
<tr>
<td>Ordinal</td>
<td>Ranking according to criterion</td>
<td>Order</td>
<td>Tiny, small, medium, large</td>
<td>Any monotonic increasing remapping</td>
</tr>
<tr>
<td>Interval</td>
<td>Differences are meaningful</td>
<td>Size of difference</td>
<td>Temperature in Celsius or Fahrenheit</td>
<td>Linear remappings with offset (ax+b)</td>
</tr>
<tr>
<td>Ratio</td>
<td>Has a zero point</td>
<td>Ratios of values are meaningful</td>
<td>Absolute temperature (Kelvin), values in currency units</td>
<td>Linear remappings without offset (ax)</td>
</tr>
</tbody>
</table>
Properties of Resources

- **Perishable: lost if not used**
  - Perishable bandwidth
  - Nonperishable disk space

- **Fungible: convertible to other resources**
  - Complete common currency
  - Partial bandwidth vs CPU (compression)
  - None calendar time vs staff months

- **Rival: use by one person precludes use by another**
  - Rival money, labor, bandwidth
  - Nonrival information goods

- **Measurement scale: appropriate scale & operations**
  - Nominal, ordinal, interval, ratio

Ex 6: Algorithmic Complexity

- Analysis of algorithms tells you how running time will scale with problem size
  - A sort algorithm might be $O(n \log n)$
  - Scalability is not a scalar attribute!!

- In this case
  - $d$, the design, is the pseudo-code of the sort algorithm
  - $x$, the capabilities, is $O(n \log n)$ scalability
  - $v$, the value space, includes a scalability dimension
  - $m$, the development method, is a programming technique
  - $P$ predicts competent implementation $\Rightarrow$ expected runtime
  - $C$ is the cost (e.g., performance) of $O(n \log n)$ execution time
Considering development method

\[ U(d, \theta) = B(x, \theta) - C(d, x, m) \text{ for } \{ x : F(d, x, m) \} \]

where \( x = P(d, m) \)

We don’t have the code during early design, so we have to predict the implementation properties \( x \) assuming \( d \) is implemented by method \( m \)

Let
- \( d \) be a design in some appropriate notation
- \( x \) be in \( \mathbb{R}^n \) an open-ended vector of capabilities
- \( v \) be in \( \mathbb{V}^n \) a multidimensional value space
- \( m \) be in some notation a development method

\( B \) express benefits predicted value \( v \) of \( x \) to user
\( C \) express costs cost \( v \) of getting \( x \) from \( d \) with method \( m \)

\( P \) predicts capability capabilities \( x \) that \( m \) will deliver for

Ex 6a: Algorithmic Complexity, again

Analysis of algorithms tells you how running time will scale with problem size
- A sort algorithm might be \( O(n \log n) \)

In this case
- \( d \), the design, is the pseudo-code of the sort algorithm
- \( x \), the capabilities, is \( O(n \log n) \) scalability
- \( v \), the value space, includes a scalability dimension
- \( m \), the development method, is a programming technique

\( P \) predicts competent implementation \( \Rightarrow \) expected runtime
\( C \) is the cost (e.g., performance) of \( O(n \log n) \) execution time

Implementation must be competent, not just correct
- I once saw an \( O(n^3) \) implementation in a class assignment!
Ex 7: COCOMO II Early Design Model

COCOMO predicts effort (PM) & schedule (TDEV)
PM = A (Size)^E \prod_i EM_i where E = B + 0.01 \sum_j SF_j
- A, B are calibrated to 161 projects in the database
- EM_i and SF_j characterize project and developers
- TDEV is similar

But it depends on Size, and LOC aren’t known early
- Count unadjusted function points (UFP) in requirements
- Use COCOMO II’s conversion table (previous example!!)
Size = KSLOC(programming language, UFP)

Ex 7: Predicting development effort

C(d,x,m) = C(\text{Size}, x, <A, B, EM_{i}, SF_{j}> ) = <PM>
= < A x Size^E \Pi_i EM_i,> where E = B + 0.01 \sum_j SF_j
= < A x KSLOC(pl, UFP(d))^E \Pi_i EM_i,>

With nominal values for A, B, SF_j, EM_j
= < 2.94 x KSLOC(pl, UFP(d))^{1.0997}>

For 100KSLOC system,
= < 465.3153 person-months >
Client-focused Value

\[ U(d, \theta) = B(x, \theta) - C(d, x, m) \text{ for } \{ x : F(d, x, m) \} , \text{ where } x = P(d, m) \]

Most significantly, value can only be reckoned relative to the needs and preferences (utilities) of a stakeholder – in this case, the client or user.

Let
- \( d \) be a design in some appropriate notation
- \( x \) be in \( \mathbb{R}^n \) an open-ended vector of capabilities
- \( v \) be in \( \mathbb{V}^n \) a multidimensional value space
- \( m \) be in some notation a development method

\[ \theta \text{ express user pref } \quad \text{ a multidimensional utility space} \]

\[ B \text{ express benefits } \quad \text{ predicted value } v \text{ of } x \text{ to user with pref } \theta \]

\[ C \text{ express costs } \quad \text{ cost } v \text{ of getting } x \text{ from } d \text{ with method } m \]

\[ P \text{ predicts capability } \quad \text{ capabilities } x \text{ that } m \text{ will deliver for } \]

---

Ex 8: Mobile configuration utility

\[ U(d, \theta) = B(x, \theta) - C(d, x, m) \text{ for } \{ x : F(d, x, m) \} , \text{ where } x = P(d, m) \]

We previously saw prediction of \( x \) from \( d \)

\( x \) is qualities of delivered service (e.g. video fidelity)

\( d \) is application configuration (player + editor)

\( v \) is <user utility, seq of configurations, resource use>

Objective is a sequence of configurations \( d \) with the that best satisfies each user’s personal preferences \( \theta \)

<table>
<thead>
<tr>
<th>Video player</th>
<th>Windows media</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RealPlayer</td>
<td>0.8</td>
</tr>
<tr>
<td>Frame rate</td>
<td>10 fps</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>18 fps</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>24 fps</td>
<td>1.0</td>
</tr>
</tbody>
</table>

... etc...
Ex 8: Mobile configuration utility

For the configuration design point

\[
\{ \langle \text{Windows Media Player,} \\
(24 \text{ fps}, 300x200, \text{high quality audio}), \\
(25\%, 256 \text{ Kpbs, } 30 \text{ MB}) \rangle, \\
\ldots \text{ etc } \ldots \}
\]

The utility is weighted by attribute

\[
\langle \text{player, frame rate, frame size, audio} \rangle \sim \langle .5, 1.0, .5, 1.0 \rangle
\]

Then the player component of the utility is

\[
.5 \times \theta(\text{Media Player}) + 1.0 \times \theta(24 \text{ fps}) + .5 \times \theta(300x200) + \\
1.0 \times \theta(\text{high})
\]

\[
= .5 + 1.0 + .5 + 1.0
\]

\[
= 3.0
\]
Uncertainty in values

\[ U(d, \theta) = B(x, \theta) - C(d, x, m) \]

for \( \{ x : F(d, x, m) \} \), where \( x = P(d, m) \)

Capabilities \( x \) and values of \( B, C \) may be contingent and uncertain, so the value space may express uncertainty such as ranges, probabilities, future values.

Let \( d \) be a design in some appropriate notation
\( x \) be in \( \mathbb{R}^n \) an open-ended vector of capabilities
\( v \) be in \( \mathbb{V}^n \) a multidimensional value space
\( m \) be in some notation a development method
\( \theta \) express user pref a multidimensional utility space
\( B \) express benefits predicted value \( v \) of \( x \) to user with pref \( \theta \)
\( C \) express costs cost \( v \) of getting \( x \) from \( d \) with method \( m \)

\[ P \text{ predicts capability } x \text{ that } m \text{ will deliver for} \]

Ex 9: Present Value Analysis

Purchase or license a component?
- Benefit $60K/year, realized at end of year
- License cost $50K/year, due at beginning of year
- Purchase cost $120K, at beginning
- Interest rate 5%/year

<table>
<thead>
<tr>
<th>End m</th>
<th>Purchase</th>
<th>License</th>
<th>Benefit</th>
<th>( (1+I)^{-n} )</th>
<th>Purchase</th>
<th>Benefit</th>
<th>License</th>
<th>Benefit</th>
<th>( \text{Val} = \text{Ben} - \text{Cost} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>120</td>
<td>50</td>
<td>0.00</td>
<td>1.00</td>
<td>120.00</td>
<td>50.00</td>
<td>60.00</td>
<td>50.00</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td>60</td>
<td>0.95</td>
<td>47.62</td>
<td>72.62</td>
<td>57.14</td>
<td>47.62</td>
<td>57.14</td>
<td>62.86</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>60</td>
<td>0.91</td>
<td>45.35</td>
<td>84.95</td>
<td>52.44</td>
<td>45.35</td>
<td>52.44</td>
<td>62.86</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>60</td>
<td>0.86</td>
<td>43.19</td>
<td>92.94</td>
<td>51.83</td>
<td>43.19</td>
<td>51.83</td>
<td>62.86</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>60</td>
<td>0.82</td>
<td>41.08</td>
<td>99.96</td>
<td>51.20</td>
<td>41.08</td>
<td>51.20</td>
<td>62.86</td>
</tr>
<tr>
<td>sum</td>
<td>200</td>
<td>240</td>
<td>1.00</td>
<td>120.00</td>
<td>186.16</td>
<td>186.16</td>
<td>120.00</td>
<td>186.16</td>
<td>212.76</td>
</tr>
</tbody>
</table>
Economic Value in a SW Project

<table>
<thead>
<tr>
<th>Development</th>
<th>T</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
<td>C-M</td>
</tr>
</tbody>
</table>

Note the times at which variables are evaluated
- Development cost (I) is PV at time 0 of development cost
- Asset value (C) and Operation cost (M) are PV at time T
- Risk (d) is used as discount rate to move C&M to 0
- Flexibility value (Ω) measures value of strategic flexibility

\[
NPV = \frac{(C-M)}{(1+d)^T} - I + \Omega
\]

--Erdogmus, Comparative evaluation of development strategies with NPV, EDSER-L, 1999

Usage scenarios

- Evaluating a given design, comparing products
  - Most of the previous examples explore this scenario
- Composing evaluation functions
  - COCOMO Early Design composes code size estimate with the effort and schedule estimators
- Optimizing among design alternatives
  - We show dynamic reconfiguration for mobile devices
- Deciding what design information to capture
  - Look at the design representations used the the predictors that may be appropriate
- Exploring tradeoff spaces
  - We now show how to use COCOMO in this way
Ex 10: Tradeoffs in development costs

Most of EM_i and SF_j describe development method, but four describe characteristics of the product:
- SCHED (required development schedule constraint)
- RCPX (required reliability and complexity)
- RUSE (required reusability)
- PDIF (platform difficulty)

We can restate the Early Design estimators to retain these as parameters:
- For simplicity, use only RCPX, SCHED

COCOMO II, Product Factors Isolated

\[ U(d, \theta) = B(x, \theta) - C(d, x, m) \] for \{ x : F(d, x, m) \}, where \( x = P(d, m) \)

\[ x = \langle RCPX, SCHED \rangle, \ x_i \ in \ \{XL, VL, L, N, H, VH, XH\} \]

d is Size = KSLOC(prog lang, UFP(rqts))

\( v \) is value space \( <PM, TDEV, RCPX, SCHED> \)

\( m \) is encoded in the adaptive factors

\[ <A, B, EM_{i,not RCPX, SCHED}, SF_k> \]

COCOMO (P) then predicts the cost element of \( v \):

\[ PM = A \ (Size)^E \ \Pi_{i,not RCPX, SCHED} EM_i \times EM_{RCPX} \times EM_{SCHED} \]

where \( E = B + 0.01 \Sigma_j SF_j \)
The Value Proposition for Everyday Software

Cost of Achieving Given RCPX, SCHED

\[ C(d,x,m) = C(d, <RCPX, SCHED>, <A, B, Em, SFk>) \]
\[ = <PM, TDEV, RCPX, SCHED> \]
\[ = <A \times Size^E \Pi_{i \neq RCPX, SCHED} EM_i x EM_{RCPX} x EM_{SCHED} , TDEV, RCPX, SCHED> \]

where \( E = B + 0.01 \Sigma_j SF_j \)
\[ = <A \times KSLOC(pl, UFP(d))^E \Pi_{i \neq RCPX, SCHED} EM_i x EM_{RCPX} x EM_{SCHED} , TDEV, RCPX, SCHED> \]

With nominal values for \( A, B, SF_j \), all \( EM_i \) but RCPX, SCHED
\[ = <2.94 \times KSLOC(pl, UFP(d))^{1.0997} x EM_{RCPX} x EM_{SCHED} , TDEV, RCPX, SCHED> \]

For 100KSLOC system,
\[ = <465.3153 x EM_{RCPX} x EM_{SCHED} , TDEV, RCPX, SCHED> \]

Effort to Achieve Given RCPX, SCHED
Review: Examples

- **Toy examples**
  1. Value as simple benefit minus cost
  2. Selection of representation for a task
  3. Feature value from econometric analysis of spreadsheets
  4. KSLOC prediction from requirements via function points
  5. Real models
  6. Performance prediction based on algorithmic complexity
  7. Schedule and effort from COCOMO II
  8. Present value analysis for buy vs license decision

- **Real models**
  9. Performance prediction based on algorithmic complexity
  10. Schedule and effort from COCOMO II

- **Current and recent research**
  - Multidimensional costs
  - User-oriented configuration of mobile devices

Other examples

- **Security Attribute Evaluation Method (SAEM, Butler)**
  1. Elicit client’s threat, asset protection priorities ($\theta$)
  2. Evaluate per-threat countermeasure effectiveness
     
     ( $x = P(d,m)$ )
     
     of candidate security technology to add ($d$)
  3. Weight countermeasures by priorities ($B(x,\theta)$)

- **Cognitive modeling for UIs (Keystroke, GOMS)**
  1. Design UI and select common tasks
  2. Use cognitive model to predict task times ($x = P(d,m)$)

- **Real options to evaluate delayed decision**
  1. Additional cost now to preserve flexibility
  2. Cost to exercise flexibility later
     
     ( $C(d,x,m)$ expresses implementation and design cost now
     
     $B(x,\theta)$ expresses option value for exercising flexibility later)
FAQ

Is it sound? No, it’s light!
Is the model correct? Maybe not, it’s a first cut
Is it complete? No, it’s opportunistic
Is it universal? No, it takes user view of value
Does it work? Maybe. We’ll see
So, is it useful? We already think so
What does it not do? Things that need code

What needs to be done?

» Make early predictive design evaluation viable
  » Identify existing techniques that apply early
  » Explain them in a consistent way
  » Determine how to compose them
  » Develop new techniques

» Provide a unifying model
  » Be explicit about interfaces
  » Be clear about method and confidence

» Support it with tools
We need better ways to analyze a software design and predict the value its implementation will offer to a customer or to its producer.

Many techniques provide early, but selective, evaluation.
- They are not organized to use systematically.
- Economic view offers promise for unification.

Themes

Thinking about dependability

Dependability and value for specific users

Uncertainty in ultra-large-scale systems

Predicting value from design