Course "Softwareprozesse"

Software Engineering Economics

Lutz Prechelt
Freie Universität Berlin, Institut für Informatik
http://www.inf.fu-berlin.de/inst/ag-se/

- Conventional view:
  High quality at low cost
    - Some known facts
- Economical view:
  High value at low cost

- 7 Elements of Value-Based SE:
  1. Benefits Realization Analysis
  2. Value Proposition Elicitation
  3. Business Case Analysis
  5. Concurrent System and SW Eng.
  6. Value-Based Monitoring and Control
  7. Change as Opportunity
Learning objectives

- Understand the conventional, cost-centric view of SE
- Know some key facts of conventional SE economics

- Understand the economical, value-centric view of SE
- Learn the key ingredients of value-based SE
- In particular, learn about the broad-band nature of value-centric SE projects
Conventional and economical view of software engineering

Conventional view of software engineering:
• The goal of software engineering is producing high-quality software at low cost
  • cost-efficient quality

Economical view of software engineering:
• The goal of software engineering is enabling the creation of high value (via valuable software) at low cost
  • high value-added

• Note: As a simplification, we will often talk about the value of the software, rather than the value created via using the software
Conventional view: Cost and quality

Cost of software:
- Development cost and risk
  - for requirements analysis, design, implementation, test, documentation, delivery, [...]  
  - Risk: Chance of project failure
- Maintenance cost and risk
  - for analysis, design, [...] of future changes  
  - Risk: Chance of failing to change or of degrading the SW
- Operation cost and risk
  - Cost: e.g. \(\rightarrow\) Efficiency, etc.  
  - Risk: e.g. \(\rightarrow\) Dependability, etc.
- Cost of time-to-market
  - Chances lost due to later availability of the SW

Quality of software:
- Fitness for purpose
  - Functionality
  - Compatibility
  - Dependability
    - reliability, availability, safety, security
  - Usability
    - Learnability, ease of use, tolerance for human error etc.
- Efficiency
  - Load on memory, disk, CPU, network bandwidth, user work time etc.
- Maintainability
  - Portability
  - Modifiability
  - Robustness
Observations about the conventional view

• The conventional view is highly cost-focused:
  • The cost factors anyway
  • Most quality factors as well:
    • Efficiency is focused on operation cost
    • Maintainability is focused on maintenance cost and risk
    • Much of usability is focused on operation cost
    • Dependability is focused on operation risk
    • Usability is (in parts) focused on operation risk

• Only 'Functionality' directly targets the value of the software
  • But only insofar as the requirements were 'right'
    • also, some requirements will in fact be more valuable than others
  • Correctly implementing superfluous or ill-directed requirements does not provide positive value
    • but is considered *quality* during most activities of conventional SW processes
Some known facts of conventional-view SW engineering economics


- L17: Inspections improve productivity (i.e. have high ROI), quality, and project stability
  - Hence every project should invest in inspections

- L2: The cost for removing a given defect is the larger, the later the defect is found
  - E.g. for requirements defects: often 100 times (or more) larger when found in the field as opposed to in requirements stage
  - Hence inspections of requirements and design are extremely valuable

- L15: Software reuse improves productivity (i.e. has high ROI) and software quality
  - Hence one should not develop something oneself needlessly
Some known facts of conventional-view SW engineering economics (2)

- **L24**: 80% of the defects usually come from only about 20% of the modules
  - It pays off to identify these early and then inspect them or even implement them again from scratch
- **L26**: Usability is quantifiable
  - using measures such as time spent, success rate, error rate, frequency of help requests.
  - Such quantification is useful as it guides usability improvement
- **L34**: Cost estimates tend to be too low
  - "There are always surprises and all surprises involve more work"
  - Plan for contingencies and make sure your buffer is used only for them!
- **L36**: Adding people to a late project makes it later
  - Because more people means higher coordination effort and fresh people particularly so
Economical view: Cost and value

Cost of software:
- Cost for providing value
  - Finding and agreeing on value-enabling requirements
  - Writing code and documentation
  - Fitness-improving testing
  - Delivering software and bringing it into valuable use
  - Short time-to-market
- Cost for low-value insurance
  - All other quality assurance
- Cost for cost-reduction:
  - Product-related: anything that contributes to manageability, testability, maintainability etc.
  - Process-related: Most process improvement

Value of software:
- For commercial SW products:
  - Revenue (or revenue increase) generated
- For custom software:
  - Added value and/or saved cost generated by using the software
    - This is also the basis for the revenue from commercial products if (and only if) there is no competition
- For Open Source software:
  - Its value is hard to measure

"Risk" is:
- Threats of increased cost or reduced value
Economical view: A typical cost-benefit curve
Observations about the economical view

The economical view redirects the focus of software engineering:

1. **Away from the cost of individual process steps**
   - to the cost for providing elements of the final value
   - or the cost for *preparing* to provide that value

2. **Away from the individual quality factors as such**
   - to the value they provide (fitness for purpose, efficiency)
   - or the insurance they represent (testability, maintainability, etc.)

*Note: Of course, SW engineers have always used the economical view, too.*

- But it is useful to do it more explicitly
Observations about the economical view (2)

The economical view simplifies judging the importance of SE process steps and their products:

- requirements prepare providing value, reduce risk
- design reduces costs and risk
- program code provides value
- user documentation adds value (if done well)
- defect tests add value (as long as they find value-reducing defects), reduce risk
- inspections reduce costs and risk
- process improvement reduces costs and risk
- etc.

• Note: This is very simplified. For instance process improvement wrt. requirements engineering also improves the value-providing capabilities etc.
Quality assurance → Value assurance

- **Conventional view:**
  - The goal of quality assurance activities is to build software whose quality is "as high as possible"
    - with respect to the various aspects of quality
  - It is difficult to decide on the optimal extent of these activities

- **Economical view:**
  - The goal of quality assurance activities is to **reduce the risk** to the success of the value-generating activities,
    - i.e. to ensure that potential value is actually realized ("value assurance")
  - The extent of these QA activities depends on the size of the risk and the size of the value that is to be assured
The "good enough" principle

• In the conventional view, it is difficult to decide on the level of quality to be achieved
  • e.g. 100% reliability is usually impossible. If we currently have 19 known defects (failure modes) left in the system, do we need to eliminate them all?

• In the economical view, a (seemingly) simple rule guides these decisions:
  • Is the cost of making an improvement to the product smaller than the added value generated by the improvement?
  • If yes, make the improvement, otherwise don't.
  • (Note that cost is often and value is usually hard to estimate)

• This rule leads to the "good enough" approach to SW eng.:
  • Always try to understand when the SW is "good enough"
  • and then make it at least that good
  • but probably not much better
"Good enough" example: efficiency optimization

- Assume you could reduce the processing time of a program function by a factor of 10 by spending 9 days of effort.

Should you do it?

- Depending on the importance of the function:
  - if its overall value is small, probably not. Otherwise:
- Depending on current processing time (interactive SW), e.g.
  - 3 sec: yes
  - 0.1 sec: only if the work is on a high-load server/in a game, etc.
  - 100 sec: only if the function is used daily or by many people
- Depending on the current processing time (real-time system):
  - yes if this is necessary to meet hard deadlines
  - otherwise only if it frees enough resources to make implementing other tasks much simpler (→development cost reduction)
Value-Based Software Engineering: Key elements

- Barry Boehm: "Value-Based Software Engineering", ACM Software Engineering Notes 28(2), March 2003 suggests:
  1. Benefits Realization Analysis
  2. Stakeholder Value Proposition Elicitation and Reconciliation
  3. Business Case Analysis
  4. Continuous Risk and Opportunity Management
  5. Concurrent System and Software Engineering
  6. Value-Based Monitoring and Control
  7. Change as Opportunity

- Barry Boehm, Li Guo Huang: "Value-Based Software Engineering: A Case Study", IEEE Computer, March 2003 provides an example for some of them
1. Benefits Realization Analysis (BRA): Starting point

- Fictitious company: Sierra Mountainbikes
  - Renowned for its outstanding quality bikes
  - Notorious for delivery delays, delivery mistakes, and disorganized handling of problems
- Enters a partnership with eServices Inc.
  - for joint development of better order-processing and fulfillment
- Value-realization chain (simplified):

- New order-entry system
  - Less time, fewer errors per order-entry step
  - Faster, better order-fulfillment inputs
  - Less time, fewer errors in order processing

- New order-fulfillment system
  - Fewer order-fulfillment steps, errors
  - Increased customer satisfaction, decreased operations costs

- New order-fulfillment processes, outreach, training
  - Increased sales, profitability, customer satisfaction
  - Increased profits, growth

Assumptions:
- Increasing market size
- Continuing consumer satisfaction with product
- Relatively stable e-commerce infrastructure
- Continued high staff performance
1. Benefits Realization Analysis (BRA): Consequences

- This view turns the software development project into a *business change program*
  - and identifies its stakeholders

- It involves crucial activities outside the technical domain
  - e.g. the order entry staff being willing and capable of changing the work processes

- The software people must understand and respect these aspects
  - e.g. being willing and capable to design, build, and refine the GUI and user experience in close cooperation with those staff
  - and respond to late changes during the actual process change
2. Stakeholder Value Proposition Elicitation and Reconciliation

Model-Clash Spiderweb diagram. The red lines show actual clashes in the Bank of America MasterNet system.
2. Stakeholder Value Proposition Elicitation and Reconciliation

- **Elicitation requires:**
  - Business case analysis (for basic items)
  - Groupware (for refinement)
    - brainstorming, discussion, win-win negotiation

- **Reconciliation (conflict resolution) requires in addition:**
  - Creating awareness; expectations management
    - cost models, simplifier lists, complicator lists
  - Visualization and tradeoff-analysis techniques
    - prototypes, scenarios, estimation models, etc.
  - Within-stakeholder prioritization (e.g. by pairwise comparison)
3. Business Case Analysis: General

- Analyze ROI (return on invest) of various approaches over time
  - e.g. approaches A, B, C as in the figure
- Weigh in uncertain benefits and risks
  - e.g. if early market entry is important, but competitors' speed is unknown, then pair programming may be a calendar-time risk-reduction method (C) that is preferable over compromising functionalities (B)
# 3. Business Case Analysis: Sierra Mountainbikes case study

<table>
<thead>
<tr>
<th>Projections</th>
<th>Current system</th>
<th>New system</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Date</strong></td>
<td><strong>Market size ($M)</strong></td>
<td><strong>Market share %</strong></td>
</tr>
<tr>
<td>31 Dec. 2003</td>
<td>360</td>
<td>20</td>
</tr>
<tr>
<td>31 Dec. 2004</td>
<td>400</td>
<td>20</td>
</tr>
<tr>
<td>31 Dec. 2005</td>
<td>440</td>
<td>20</td>
</tr>
<tr>
<td>31 Dec. 2006</td>
<td>480</td>
<td>20</td>
</tr>
<tr>
<td>31 Dec. 2007</td>
<td>520</td>
<td>20</td>
</tr>
<tr>
<td>31 Dec. 2008</td>
<td>560</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Expected improvements</th>
<th>Overall customer satisfaction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target date</strong></td>
<td><strong>Cost savings</strong></td>
<td><strong>Change in profits</strong></td>
</tr>
<tr>
<td>31 Dec. 2003</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>31 Dec. 2004</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>31 Dec. 2005</td>
<td>2.2</td>
<td>3.2</td>
</tr>
<tr>
<td>31 Dec. 2006</td>
<td>3.2</td>
<td>6.2</td>
</tr>
<tr>
<td>31 Dec. 2007</td>
<td>4.0</td>
<td>9.0</td>
</tr>
<tr>
<td>31 Dec. 2008</td>
<td>4.4</td>
<td>11.4</td>
</tr>
</tbody>
</table>
4. Continuous Risk and Opportunity Management

Beware of unwanted human factors!

- E.g. the programmer who is assigned to write a 4-week module finds a reuse opportunity that will reduce time to 1 week (80% chance) or (for our example's sake) may fail and then will take 6 weeks (20% chance).
  - Expected time is $0.8 \times 1 \text{ weeks} + 0.2 \times 6 \text{ weeks} = 2 \text{ weeks}$, a 50% reduction!
- A risk-averse programmer may decide *not* to use this approach
- Opportunity mgmt. should detect this case and decide whether the benefit is worth the risk
  - e.g. time-buffer available, benefits from code size reduction, etc.
5. Concurrent System and SW Engineering: Basic idea

- In VBSE, HW engineering can no longer precede SW eng.
- They need to be concurrent, in order to:
  - adapt to changes in the marketplace underway
  - let requirements emerge from prototypes
  - determine COTS constraints and manage COTS risks cost-efficiently
  - allocate tasks to HW vs. SW well
  - make good cost/value tradeoffs when designing product lines
6. Value-Based Monitoring and Control: Conventional view *Earned "Value"*

- Conventional PM uses *cost-based* earned-value tracking
  - Assumption 1: When e.g. 10% of the project work are finished, also 10% of the project's value have been earned
  - Assumption 2: 10% of the work have been finished if tasks have been finished that were *planned* to consume 10% of the total cost

This project is
- behind schedule (green line)
- but below budget (red line)
6. Value-Based Monitoring and Control: Tracking real *Earned Value*

- In contrast, PM based on the economics view would attempt to perform *value-based* earned-value tracking
  - For finished functionality as well as planned functionality

- To do this:
  1. Set up a business case to quantify the expected value (benefits)
  2. Involve more shareholders in order to perform all the additional activities that are needed to realize the benefits
     - such as changes of people behavior, changes to related processes
  3. Track actual benefit objectively (quantitatively) where possible
     - Track estimated benefit subjectively elsewhere
  4. Adjust all of these as goals, markets, constraints, and environment change or as the expected value is not realized

- **Difficult!**
6. Value-Based Monitoring and Control: Tracking *Earned Value* control loop

- For example, in our Sierra Mountainbikes case study:
## 6. Value-Based Monitoring and Control: Sierra Mountainbikes case study (1)

<table>
<thead>
<tr>
<th>Milestone</th>
<th>Schedule</th>
<th>Cost ($K)</th>
<th>Op-Cost Savings %</th>
<th>Market Share ($M)</th>
<th>Annual Sales ($M)</th>
<th>Annual Profits</th>
<th>CumΔ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Life Cycle Architecture</td>
<td>3/31/04</td>
<td>400</td>
<td></td>
<td>20</td>
<td>72</td>
<td>7.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3/31/04</td>
<td>427</td>
<td></td>
<td>20</td>
<td>72</td>
<td>7.0</td>
<td></td>
</tr>
<tr>
<td>Core Capability Demo (CCD)</td>
<td>7/31/04</td>
<td>1050</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7/20/04</td>
<td>1096</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Init. Op. Capability (IOC)</td>
<td>9/30/04</td>
<td>1400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9/30/04</td>
<td>153</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware IOC</td>
<td>9/30/04</td>
<td>3500</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10/11/04</td>
<td>3432</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deployed IOC</td>
<td>12/31/04</td>
<td>4000</td>
<td></td>
<td>20</td>
<td>80</td>
<td>8.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>12/20/04</td>
<td>4041</td>
<td></td>
<td>22</td>
<td>88</td>
<td>8.6</td>
<td>0.6</td>
</tr>
</tbody>
</table>

... Plan Act. ...
### 6. Value-Based Monitoring and Control: Sierra Mountainbikes case study (2)

<table>
<thead>
<tr>
<th>Milestone</th>
<th>Schedule</th>
<th>Late Deliv.</th>
<th>Cust. Sat.</th>
<th>ITV</th>
<th>Ease of Use</th>
<th>Risks/Opportunities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Life Cycle</td>
<td>3/31/04</td>
<td>12.4</td>
<td>1.7</td>
<td>1.0</td>
<td>1.8</td>
<td>Increased COTS ITV risk.</td>
</tr>
<tr>
<td>Architecture</td>
<td>3/31/04</td>
<td>12.4</td>
<td>1.7</td>
<td>1.0</td>
<td>1.8</td>
<td>Fallback identified.</td>
</tr>
<tr>
<td>Core Capability Demo (CCD)</td>
<td>7/31/04</td>
<td>2.4*</td>
<td>1.0*</td>
<td>2.7*</td>
<td></td>
<td>Using COTS ITV fallback.</td>
</tr>
<tr>
<td></td>
<td>7/20/04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>New HW competitor; renegotiating HW</td>
</tr>
<tr>
<td>Software</td>
<td>9/30/04</td>
<td>2.7*</td>
<td>1.4*</td>
<td>2.8*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Init. Op.</td>
<td>9/30/04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability (IOC)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td>9/30/04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IOC</td>
<td>10/11/04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deployed</td>
<td>12/31/04</td>
<td>11.4</td>
<td>3.0</td>
<td>2.5</td>
<td>3.0</td>
<td>$200K savings from renegotiated HW</td>
</tr>
<tr>
<td>IOC</td>
<td>12/20/04</td>
<td>10.8</td>
<td>2.8</td>
<td>1.6</td>
<td>3.2</td>
<td>New COTS ITV source identified, being prototyped</td>
</tr>
</tbody>
</table>

*alpha testing
Economics view of design: Modularity as buying options

- Parnas' principle of information hiding suggests to form modules such that they encapsulate a design decision:
  - Should the decision ever need to change, the required modifications will be easier.

- The economical view suggests that information hiding can be viewed as buying a *real option*:
  - Encapsulating the decision requires effort (the cost of the option)
  - but it supplies you with the option (choice) to change the decision later on at lower cost
  - the difference in change-cost is the potential value of the option

(First use of real options regarded reserving olive oil presses long before the actual olive harvest.)
Other kinds of design-related options

Other design issues can be viewed from an option perspective, too:

- **Architecture:**
  - A SW architecture does not provide value itself,
  - but it provides options to implement valuable functionality easily

- **Program generators:**
  - A code generator does not provide value itself,
  - but it provides options to implement valuable functionality easily

- **Design-related risk management:**
  - Risk assessment (e.g. by prototyping) is an investment that aims at estimating the value of certain options.
Not all decisions are encapsulated

• From these examples, it should be obvious that not all design decisions can be encapsulated – or should be
  • If you do not believe this, try encapsulating your decision for an architecture or for a programming language!

• Rather there are dependencies between design decisions
  • Decision A often depends on decision B or vice versa
    • but perhaps only for some choices of A or B
  • We call A and B "design parameters" (DP)
  • When thinking about a design, we may recognize some parameters (and many dependencies) only after a while

• We can reason economically about
  • how much certain dependencies hurt or
  • how valuable an encapsulation might be

Economics view of design: Summary

• Modularization can be understood in terms of
  • design parameters (DP, any internal design decision)
  • design rules (DPs meant to be fixed \(\rightarrow\) module interfaces)
  • environment parameters (EP, external design forces such as volatile requirements and constraints)
  • dependencies among these, which result in coordination costs for changes and possibly avalanching changes

• Modularization provides change options
  • Changes can add value to a system

• Good modularizations are good because:
  They allow changes that add more value than they cost
Further literature

- Workshop on economics-driven software engineering research
  - aka EDSER, a yearly workshop focusing on the economical view (until 2006, last instance held as “ESC” in 2007)
  - proceedings appear at ACM press
  - [www.acm.org](http://www.acm.org) (full-text access from fu-berlin.de IP numbers)
Summary

• The conventional view of SE focuses on cost and quality
• This view ignores the fact that different requirements provide different value

• An economical view of SE should take value into account
• For instance by using
  • value-based project planning
    • which automatically leads to additional business process changes and a holistic (rather than SW-focused) view of value optimization
  • value-based project progress tracking
  • a view of modularity as providing flexibility value by generating real options (namely for changes)
Thank you!