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

• Tracking earned value
  • conventional
  • value-centric

• Design as real options
  • valuating modularity
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 about the broad-band nature of value-centric SE projects
• Learn about the view of modularity as real options
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 managability, 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 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)
Project management: Tracking earned value

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

• For an example, see Barry Boehm, Li Guo Huang: "Value based software engineering: A case study", IEEE Computer, March 2003 (see next few slides)
Value-tracking case study (1): Starting point

- **Fictitious company: Sierra Mountainbikes**
  - Renown 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):**

![Value-realization chain diagram]
Value-tracking case study (2): Proposed initiatives

- Full value-realization chain:

![Diagram showing the full value-realization chain with initiatives and outcomes.

Assumptions:
- Increasing market size
- Continuing consumer satisfaction with product
- Relatively stable e-commerce infrastructure
- Continued high staff performance]
Value-tracking case study (3): Business case of initiatives

<table>
<thead>
<tr>
<th>Projections</th>
<th>Current system</th>
<th>New system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Market size ($M)</td>
<td>Market share %</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>Expected improvements</th>
<th>Overall customer satisfaction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
</tr>
<tr>
<td>date</td>
<td>savings</td>
</tr>
<tr>
<td>2003</td>
<td>0</td>
</tr>
<tr>
<td>2004</td>
<td>0</td>
</tr>
<tr>
<td>2005</td>
<td>2.2</td>
</tr>
<tr>
<td>2006</td>
<td>3.2</td>
</tr>
<tr>
<td>2007</td>
<td>4.0</td>
</tr>
<tr>
<td>2008</td>
<td>4.4</td>
</tr>
</tbody>
</table>
### Value-tracking case study (4):
#### Value tracking year 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>Core Capability Demo (CCD)</td>
<td>7/31/04</td>
<td>1050</td>
<td></td>
<td>20</td>
<td>72</td>
<td>7.0</td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>9/30/04</td>
<td>1400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Init. Op. Capability (IOC)</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>Deployed OC</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>IOC</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>

...
Value-tracking case study (5):
Value tracking and countermeasures

<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>
</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>
</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>
</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>
</tr>
<tr>
<td>Software Init. Op. Capability (IOC)</td>
<td>7/20/04</td>
<td>2.4*</td>
<td>1.0*</td>
<td>2.7*</td>
<td></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>
</tr>
<tr>
<td>Init. Op. Capability (IOC)</td>
<td>9/30/04</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware IOC</td>
<td>9/30/04</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deployed IOC</td>
<td>10/11/04</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>
</tr>
<tr>
<td>Deployed IOC</td>
<td>12/20/04</td>
<td>10.8</td>
<td>2.8</td>
<td>1.6</td>
<td>3.2</td>
</tr>
</tbody>
</table>

Risks/Opportunities:
- Increased COTS ITV risk.
- Fallback identified.
- Using COTS ITV fallback.
- New HW competitor; renegotiating HW
- $200K savings from renegotiated HW
- New COTS ITV source identified, being prototyped

*alpha testing
Design process: 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

Design structure matrix (DSM)

- From an options perspective, a design is described by its parameters and their dependencies
  - Parameter: Choice of algorithm, data structure, interface, technology, etc.
  - Dependency: Choice of one parameter influences the decision space of the other
    - e.g. choice of component technology influences design of components
- Design structure matrix captures parameters and dependencies:

  "(col) has influence on (row)"

<table>
<thead>
<tr>
<th>&quot;(row) depends on (col)&quot;</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.</td>
<td>.</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>X</td>
<td>.</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td>X</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
Dependencies are costly

- Whenever A changes, B may have to change, too
  - Thus, the person working on B has to be notified
- If now B changes, C may have to change, too
  - Thus, the person working on C has to be notified
- If now C changes, B may have to change again etc.

"has influence on" (used by)

<table>
<thead>
<tr>
<th>&quot;depends on&quot; (uses)</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>X</td>
<td>.</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>X</td>
<td>.</td>
</tr>
</tbody>
</table>
How to break dependencies

- Dependencies that result in such ripple effects can often be avoided by voluntarily introducing an additional design parameter and keeping it fixed:
  - I (an interface to A) is introduced and defined as unchanging
  - Now B no longer depends on A and A can change freely
  - We call this operation "splitting" and call I a "design rule" (DR)
- If I is fixed, the resulting example design is now modular:
  - A forms one module, B+C form another, I forms a design rule
The value of modularity

- By exercising our option to define I as it is
  - and then keeping it fixed
- we gain the option to modify A and B+C independently
  - Beforehands, we could only modify both of them together
  - This difference is crucial: It allows for incremental improvement
    - Individual changes do no longer require changes elsewhere
    - Individual changes do no longer require coordination

- We call I a *visible module*
- We call A and B+C *hidden modules*
  - We can change the value (implementation) of the parameters, within a hidden module but this fact is not declared anywhere
- Other design parameters that depend on anything else than a design rule would be *non-modular*
Modularization case study: KWIC: Keyword-in-context

- Parnas' classical modularization example: Keyword-in-context (KWIC)

- Input: a list of book titles. For example
  - UML reference
  - UML user guide
  - Eclipse user manual

- Output: "Keyword-in-context" index
  - Eclipse user manual
  - guide UML user
  - manual Eclipse user
  - reference UML
  - UML reference
  - UML user guide
  - user guide UML
  - user manual Eclipse

David Parnas: "On the criteria to be used in decomposing systems into modules", Communications of the ACM 15(12):1053-1058, December 1972
Modularization case study: "Strawman" design of KWIC

- Design parameters (DP) are: the data types ("Data"), procedure interfaces ("Type"), and implementations ("Alg")

---

**Input**
- read list of titles

**Circ**
- tokenize titles and perform circular shift

**Alph**
- sort alphabetically

**Out**
- write KWIC index to output

**Master**
- control execution
Modularization case study: DSM for "strawman" design

'type' = call interface

Design rules
Implementation constraints
Modules
Very little freedom!

A - Input Type
D - Circ Type
G - Alph Type
J - Out Type
B - In Data
E - Circ Data
H - Alph Data
K - Out Data
C - Input Alg
F - Circ Alg
I - Alph Alg
L - Out Alg
M - Master
Modularization case study: DSM for information hiding design

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>A</th>
<th>D</th>
<th>G</th>
<th>J</th>
<th>O</th>
<th>P</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>F</th>
<th>H</th>
<th>I</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>N - Line Type</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>A - In Type</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>D - Circ Type</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>G - Alph Type</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>J - Out Type</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>B - In Data</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C - In Alg</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>E - Circ Data</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>F - Circ Alg</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>H - Alph Data</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>I - Alph Alg</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>K - Out Data</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>L - Out Alg</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>M - Master</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Introduces three new parameters N, O, P (Line Store module).

Design rules

Implementation constraints

Modules: more hidden information gives more room for improving the design

Everybody calls the line store, thus decoupling the other parts
Modularization case study: Modeling external forces: EP, EDSM

- A design is good if none of the likely changes needed for the system require modifying a design rule (DR)

- To model this, we extend the DSM into an **EDSM** by introducing **environment parameters** (EP)
  - Other than the DP, the EP are not controlled by the designer
  - So we now have EP and DP (the latter as normal DP and as DR)
  - KWIC EPs: X. Computer has more or less memory, Y. input corpus (title list) may be long or short, Z. user req's may change (input format, shifting, sorting).
  - Note that introducing EP is useful: discussing EPs is more focused than discussing design decisions

- In a good design, no DR depends on an EP
  - When an EP changes, all influence should be on normal DP only
  - i.e., we must adapt only internals of modules, but not interfaces

First the "strawman" design again:
Modularization case study: EDSM for "strawman" design

<table>
<thead>
<tr>
<th>Environment parameters (EP)</th>
<th>Design rules (DR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Influence of EP on DR: <strong>that's bad!</strong></td>
<td>Influence of EP on implementation DP: acceptable</td>
</tr>
</tbody>
</table>
Modularization case study: EDSM for information hiding design

<table>
<thead>
<tr>
<th>X - Computer</th>
<th>Y - Corpus</th>
<th>Z - User</th>
<th>N - Line Type</th>
<th>A - In Type</th>
<th>D - Circ Type</th>
<th>G - Alph Type</th>
<th>J - Out Type</th>
<th>O - Line Data</th>
<th>P - Line Alg</th>
<th>B - Input Data</th>
<th>C - Input Alg</th>
<th>E - Circ Data</th>
<th>F - Circ Alg</th>
<th>H - Alph Data</th>
<th>I - Alph Alg</th>
<th>K - Out Data</th>
<th>L - Out Alg</th>
<th>M - Master</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>.</td>
</tr>
<tr>
<td>X</td>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>.</td>
</tr>
<tr>
<td>X</td>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Environment parameters (EP)

Design rules (DR)

**No more influence of EP on DR!**

Influence of EP on implementation DP: acceptable
Modularization case study: Model for valuating modularization

- Any module M provides the option to replace it
- An arbitrary replacement $M_R$ may be better (i.e., add value)
  - We model the improvement $I = V(M_R) - V(M)$ as a random variable with normal distribution with mean zero
  - the variance is proportional to the module's technical potential
- Building $M_R$ costs something (which reduces value)
  - proportional to the complexity of M
- To use $M_R$ costs something more
  - proportional to the number of modules that depend on M

- Depending on these costs, it may or may not be helpful to make several experiments building a new $M_R$ each to find the best replacement
  - The expected added value for the optimal number of experiments represents the option value of M
Modularization case study: Option values of the KWIC modules

information hiding design

(some parameter details are missing)

strawman design

complexity and coordination cost are assumed as 1 per parameter
Modularization case study:
Conclusion

• Modularization can be understood in terms of
  • design parameters (DP, in particular fixed ones: design rules DR)
  • environment parameters (EP)
  • 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 focussing on the economical view
  - proceedings appear at ACM press
  - [www.acm.org](http://www.acm.org) (Volltexte verfügbar aus dem Uninetz heraus)
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!