Course "Softwareprozesse"

Software Engineering Essentials

Lutz Prechelt
Freie Universität Berlin, Institut für Informatik

- "No Silver Bullet"?
- SEMAT Essence basic concepts
- Economical view:
  Strive for high value at low cost
  - (not for high 'quality')
- 7 Elements of Value-Based SE:

1. Benefits Realization Analysis
2. Value Proposition Elicitation
3. Business Case Analysis
4. Continuous Risk&Opportunity Mgmt.
5. Concurrent System and SW Eng.
6. Value-Based Monitoring and Control
7. Change as Opportunity

- Modularity as real options
- To-be-done refactorings as Technical Debt

Learning objectives

- Understand that most complexity in SW eng. is unavoidable
- Understand a set of 7 basic concepts for SW engineering
- Understand the conventional, cost-centric view of SE
  - Know some key facts of conventional SE economics
- Contrast it to the economical, value-centric view of SE
- Understand the key ingredients of value-based SE
- Learn to apply the economical view to software design

Part I: "No Silver Bullet": Essential vs. accidental difficulty

- Frederick Brooks: "No Silver Bullet: Essence and Accidents of Software engineering", IEEE Computer 1987
- There are two types of difficulty:

1. **Essential** difficulty comes from the problem itself
   - it can not be reduced or removed, i.e., there is no "silver bullet" to shoot this difficulty "werewolf"
     - where seemingly benign SW projects suddenly turn evil (in a manager's view)
   - *"The hardest part of the software task is arriving at a complete and consistent specification and much of the essence of building a program is in fact the debugging of the specification."*

"No Silver Bullet": Essential vs. accidental difficulty (2)

Freie Universität Berlin

2. **Accidental** complexity arises from they way we *handle* the problem
   - it can be reduced in many ways and some parts can be removed completely, by:
   - abstraction
     - ever-higher-level languages
       - OO, declarative, multi-paradigm, DSLs
     - great designers
   - automation
     - analysis tools
     - construction tools
     - test automation
   - reuse
     - libraries, frameworks, components
     - architectures, design patterns
   - good process
     - requirements-focussed
     - incremental
       - refactoring
     - with strong communication
     - robust against mistakes

No silver bullet: Experts' views 20 years later

Freie Universität Berlin

- "No Silver Bullet: Software Engineering Reloaded", IEEE Software 2008
  - after a panel discussion at OOPSLA:
- Dave Parnas:
  - a "silver bullet" would not require skill: impossible
  - education and skill are key
  - most progress is due to HW
- several:
  - OO technology is helpful
- Martin Fowler:
  - OO is rarely applied right
- Dave Thomas, M. Fowler:
  - People don't understand ideas and fundamentals
- Fred Brooks:
  - "I know of no other field where people do less study of other people's work."

Part II:
A minimum set of concepts?

Freie Universität Berlin

CMMI talks about very many things.

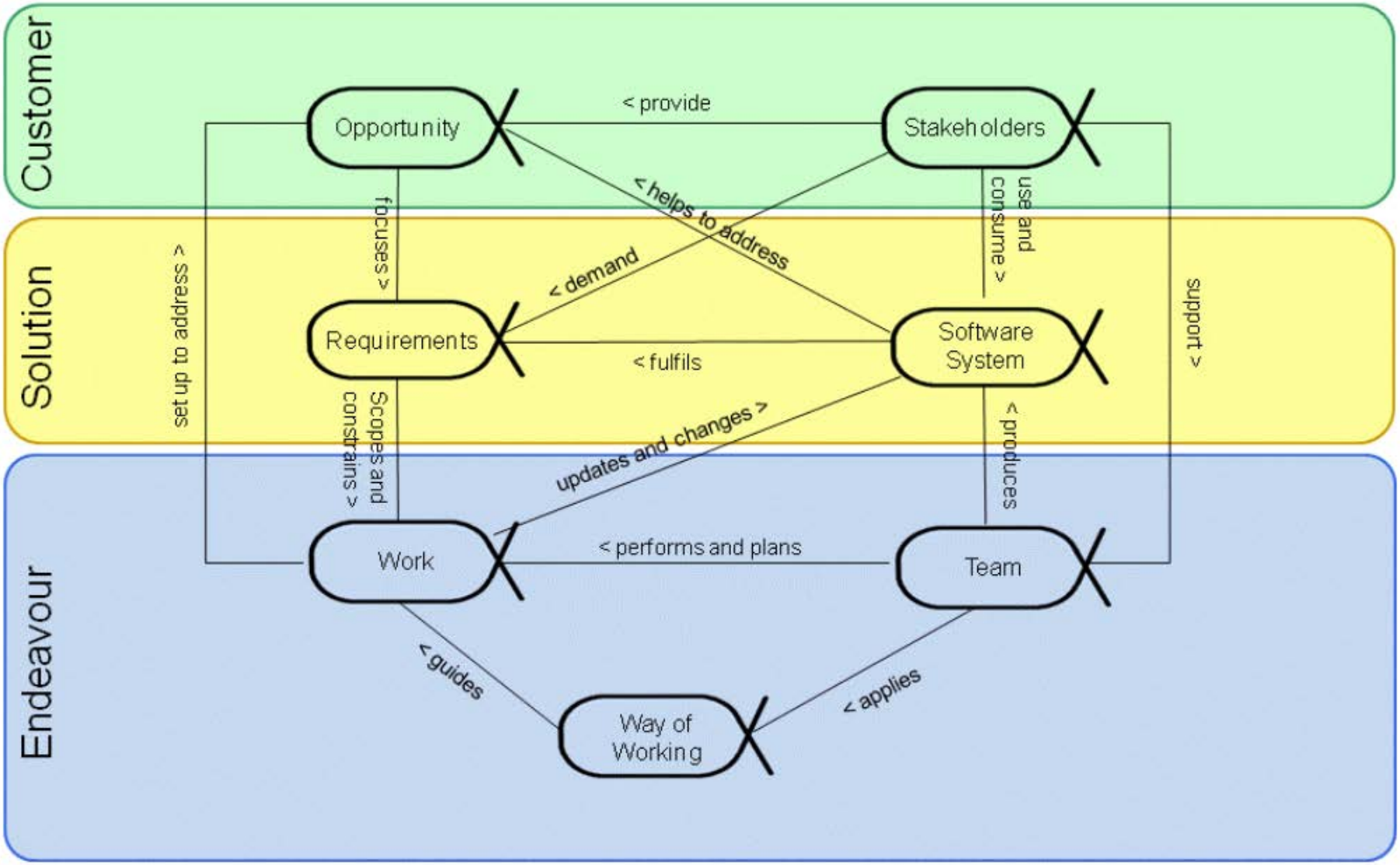
Can we reduce those to a comprehensive minimum?

The SEMAT initiative has tried this:

"*Essence* – Kernel and Language for SW Eng. Methods",
V 1.1, OMG standard, 2015 (308 pages)

- **suggests 7 basic concepts (called "alphas")**
  - in three "areas of concern": customer, solution, endeavor
- 4 + 6 + 5 "activity spaces" for the areas of concern
- 1 + 3 + 2 "competencies" for the areas of concern
  - (both not discussed here)
- Different states for each alpha to describe their lifecycles in a healthy project:
  "[alphas] have states representing progress and health, so as the endeavor moves forward the states associated with these elements progress"

The 7 *Essence* alphas and their basic relationships



Essence goals

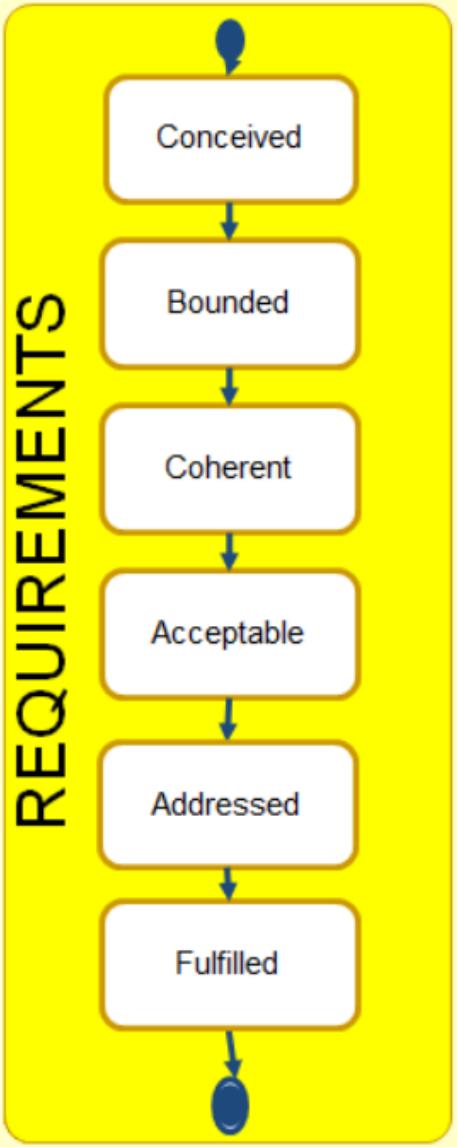
- holistic view of software development
- independent of domain, technology, development methods
- supports automation (based on an underlying formal language)

- extensible: new alphas can be added (e.g. "funding"); alphas can be broken down into elements
  - e.g. individual requirements, team members, practices, …

- actionable: alphas are "things to work with"
  - "Is every alpha in the state where it needs to be?"
  - "If not, what should we do to get there?"
    - Essence can define practices and compose them into methods

- Sensible so far. But then we look at the states, e.g.:

Essence states for the Requirements alpha



The need for a new system has been agreed.

The purpose and theme of the new system are clear.

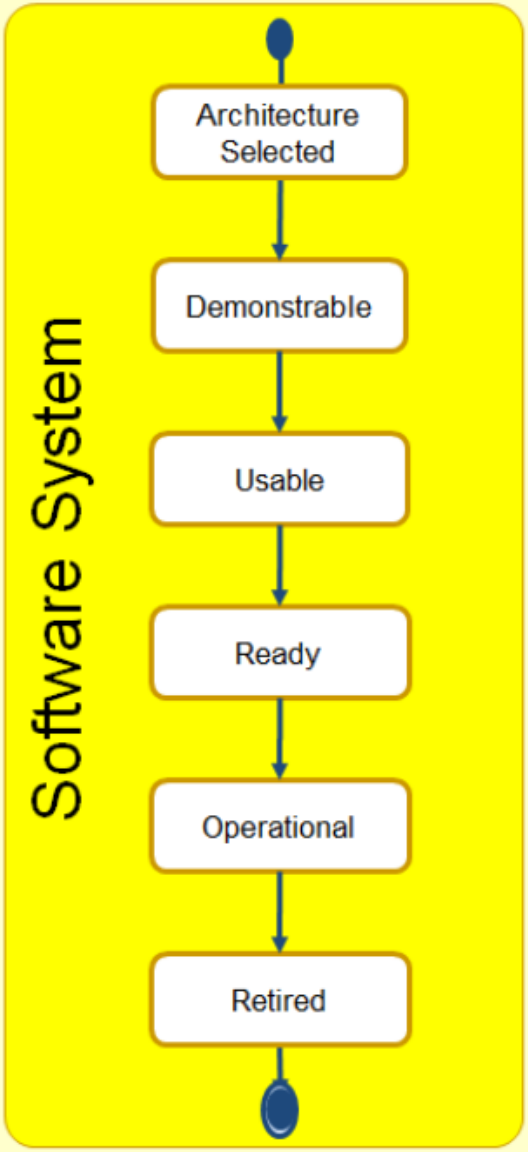
The requirements provide a consistent description of the essential characteristics of the new system.

The requirements describe a system that is acceptable to the stakeholders.

Enough of the requirements have been addressed to satisfy the need for a new system in a way that is acceptable to the stakeholders.

The requirements that have been addressed fully satisfy the need for a new system.

Essence states for the Software System alpha



An architecture has been selected that addresses the key technical risks and any applicable organizational constraints.

An executable version of the system is available that demonstrates the architecture is fit for purpose and supports testing.

The system is usable and demonstrates all of the quality characteristics required of an operational system.

The system (as a whole) has been accepted for deployment in a live environment.

The system is in use in a live environment.

The system is no longer supported.

An overly phase-oriented world-view!

Experience with Essence

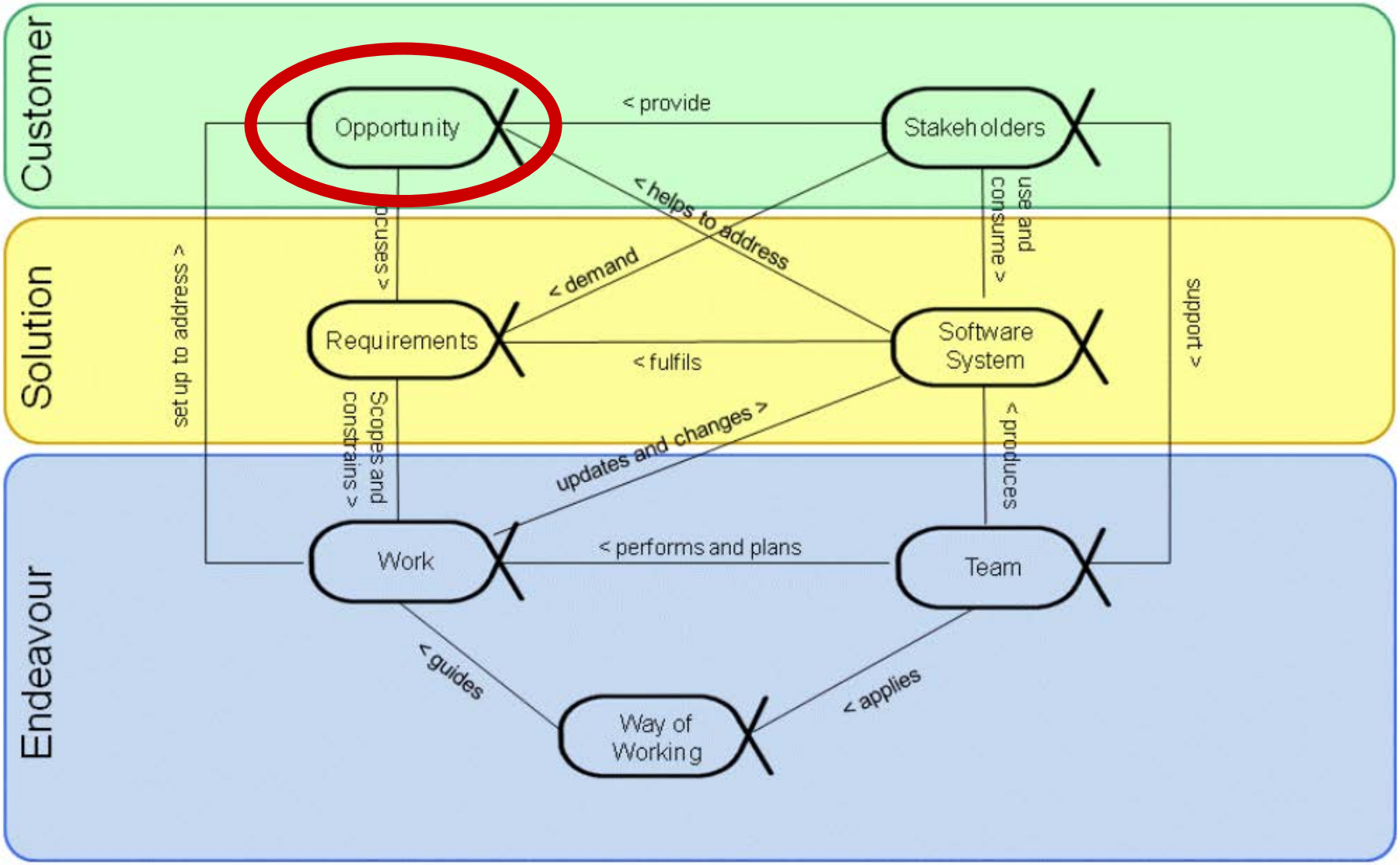
Example:

- Munich Re used it
  - "Ending Method Wars: The Successful Utilization of Essence at Munich Re", Ivar Jacobson International, 2015

- Findings:
  - Essence helps avoid talking too much about work products
  - It provides helpful joint terminology for discussions

- Outcome:
  - Munich Re added two more alphas ("funding", "acquisition")
  - Munich Re defined three development lifecycles:
    - exploratory (high-risk) projects, standard projects, small enhancement projects;
  - they differ in the states needed per alpha for each milestone.

Which alpha tends the most to get too little attention?



Part III: Economical view of software engineering

Conventional view of software engineering:

- The goal of software engineering is producing <u>high-quality</u> software at <u>low cost</u>
  - cost-efficient quality

Economical view of software engineering:

- The goal of software engineering is *enabling* the creation of <u>high value</u> (via <u>valuable</u> software) at <u>low cost</u>
  - 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

Freie Universität Berlin

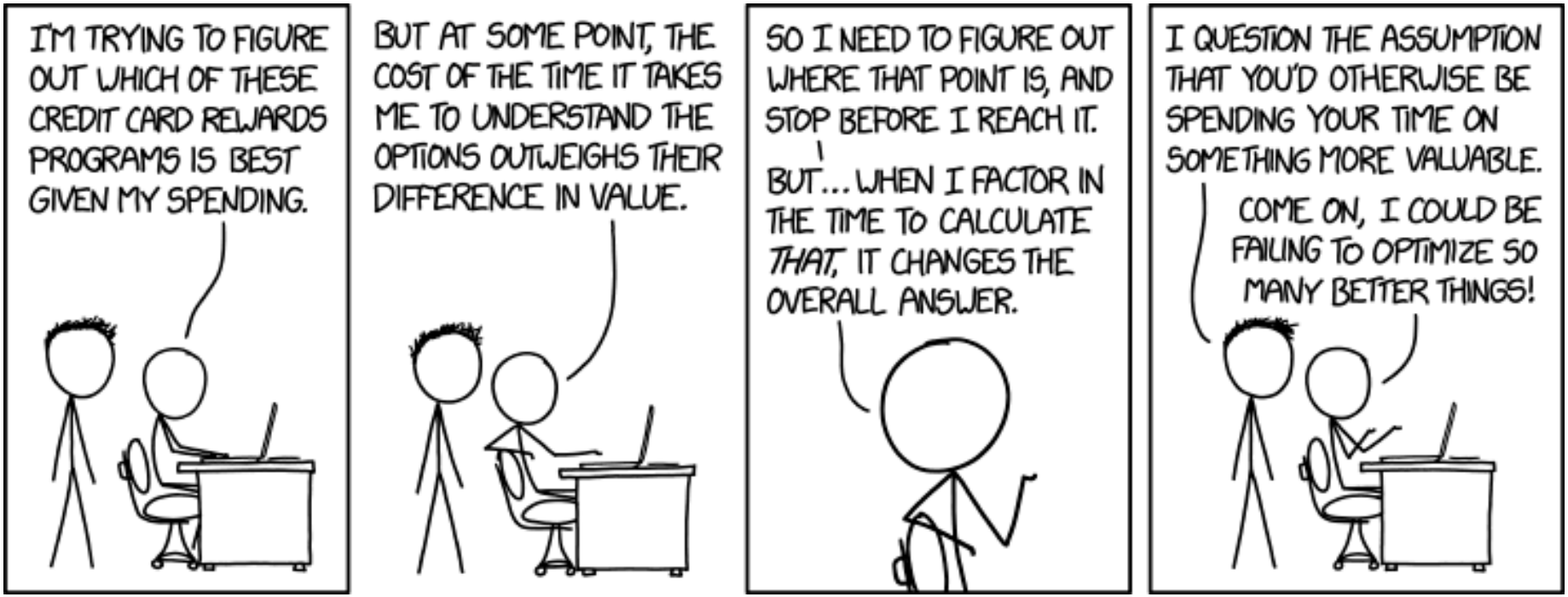
Cost of software:

- 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. →Efficiency, etc.
  - Risk: e.g. →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

We should look for "good", not for "cheapest" (or "best")



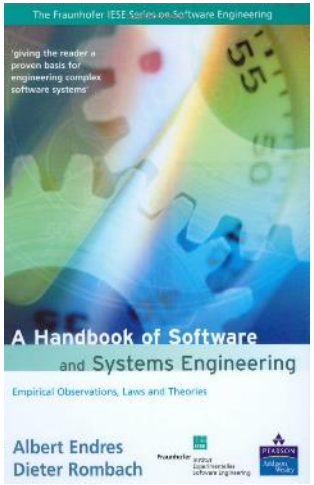
https://m.xkcd.com/1908/

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' and 'Cost of time-to-market' directly target 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



- Source: Albert Endres, Dieter Rombach: "*A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*", Pearson 2003.

- 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 (for phased processes at least)
- 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)

Freie Universität Berlin

- 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

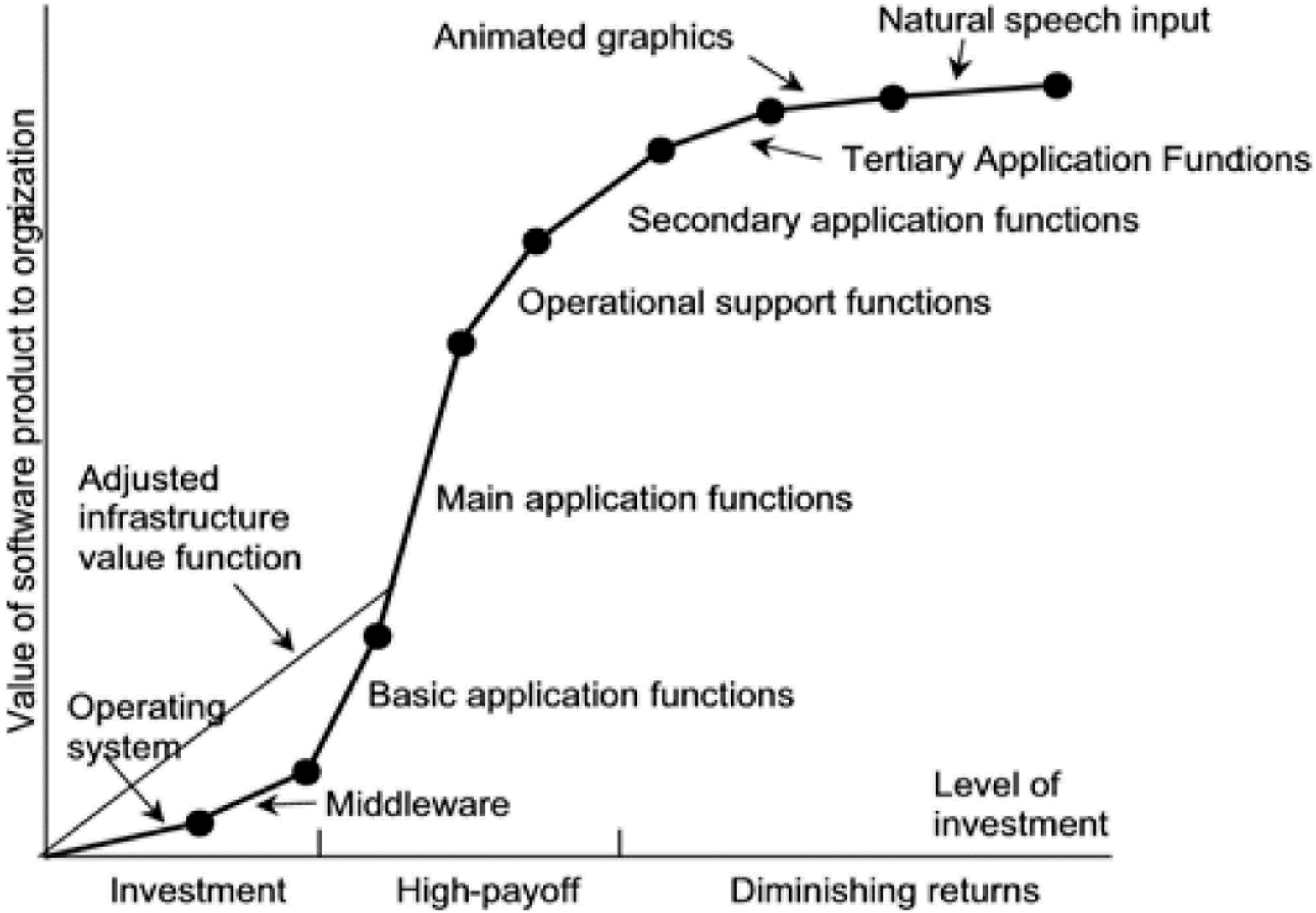
Value of software:

- 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

Freie Universität Berlin

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: Implicitly, SW engineers have always also used value considerations.
  - 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.

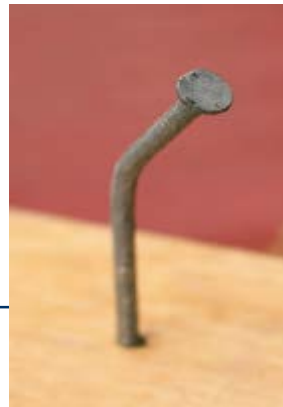
Quality assurance → Value assurance

- 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

Freie Universität Berlin

- Barry Boehm: "*Value-Based Software Engineering*", ACM Software Engineering Notes 28(2), March 2003

suggests:

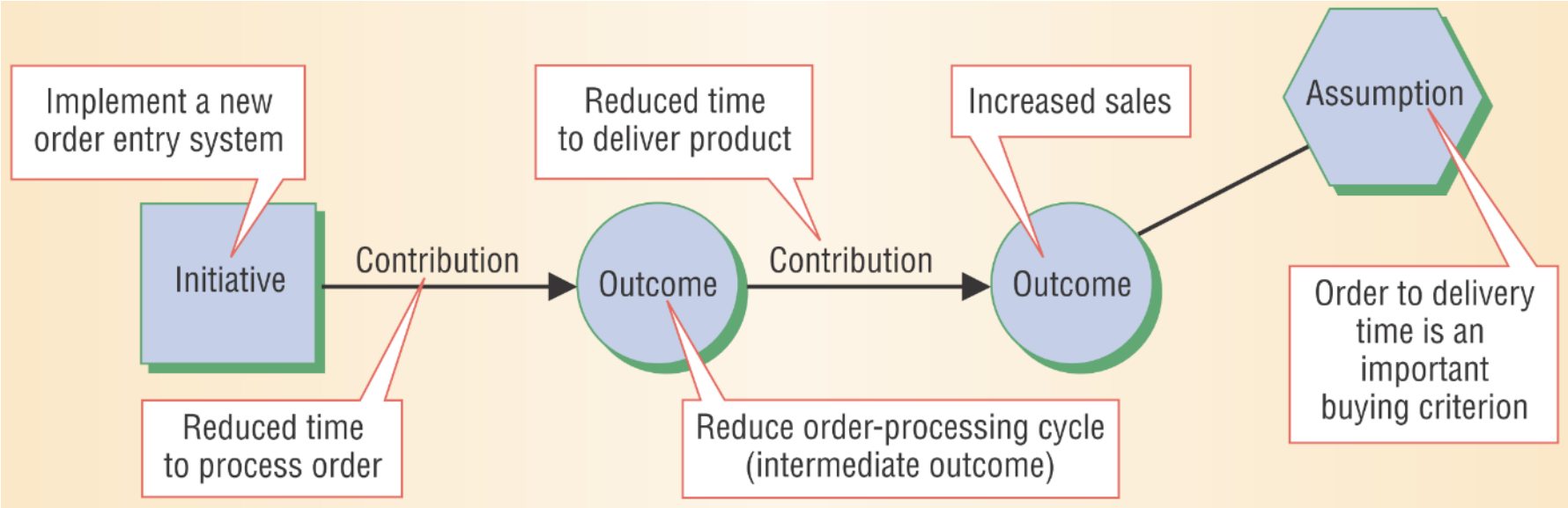
1. Benefits Realization Analysis
2. Stakeholder Value Proposition Elicitation and Reconciliation
3. Business Case Analysis
4. Continuous Risk and Opportunity Management
5. Concurrent System and Software Engineering
6. Value-Based Monitoring and Control
7. Change as Opportunity

- Barry Boehm, Li Guo Huang: "*Value-Based Software Engineering: A Case Study*", IEEE Computer, March 2003

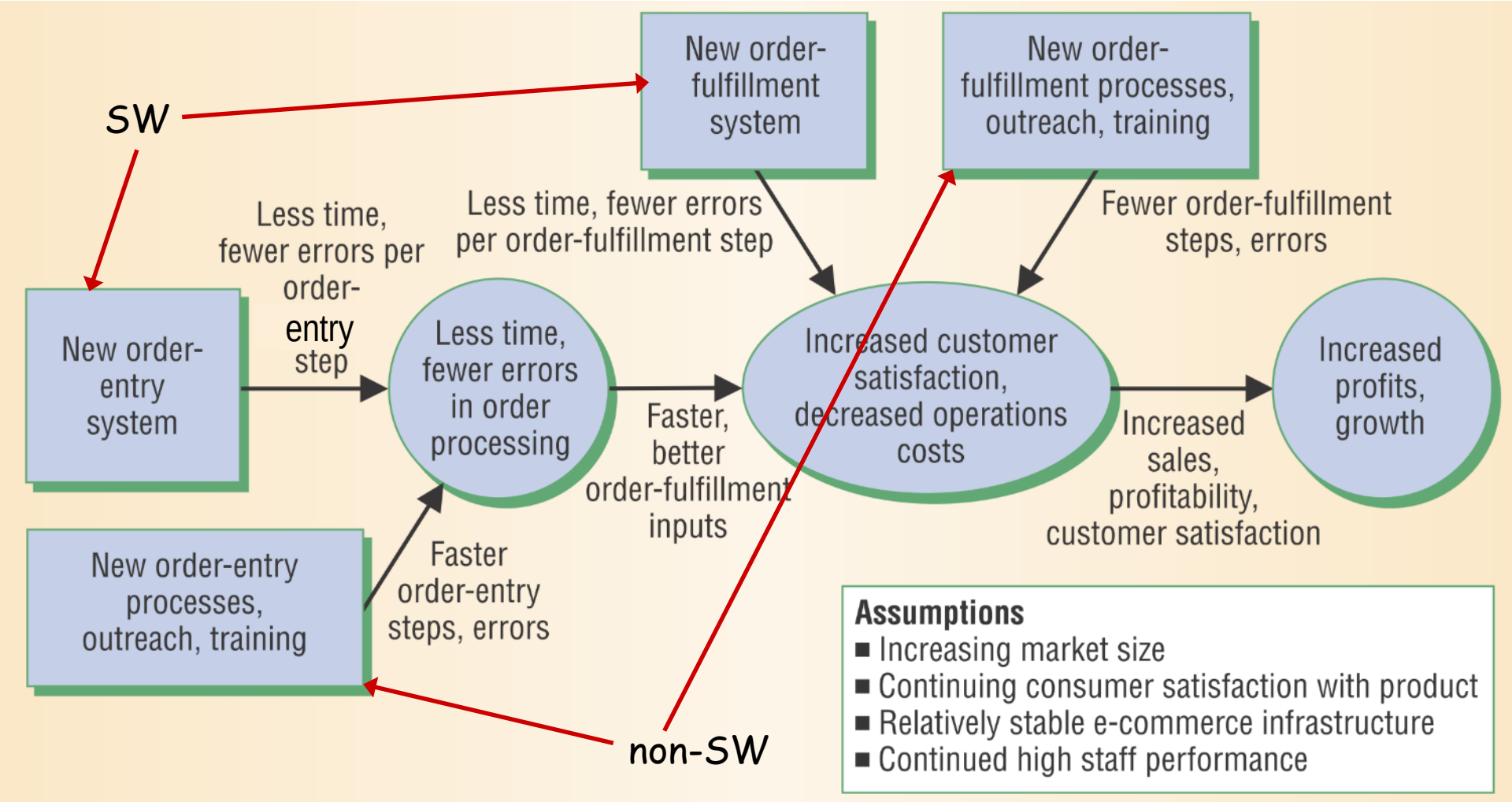
provides an example for some of them

1. Benefits Realization Analysis (BRA): Starting point

- Fictitious company: Sierra Mountainbikes
  - 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):



1. Benefits Realization Analysis (BRA): More detailed realization chain



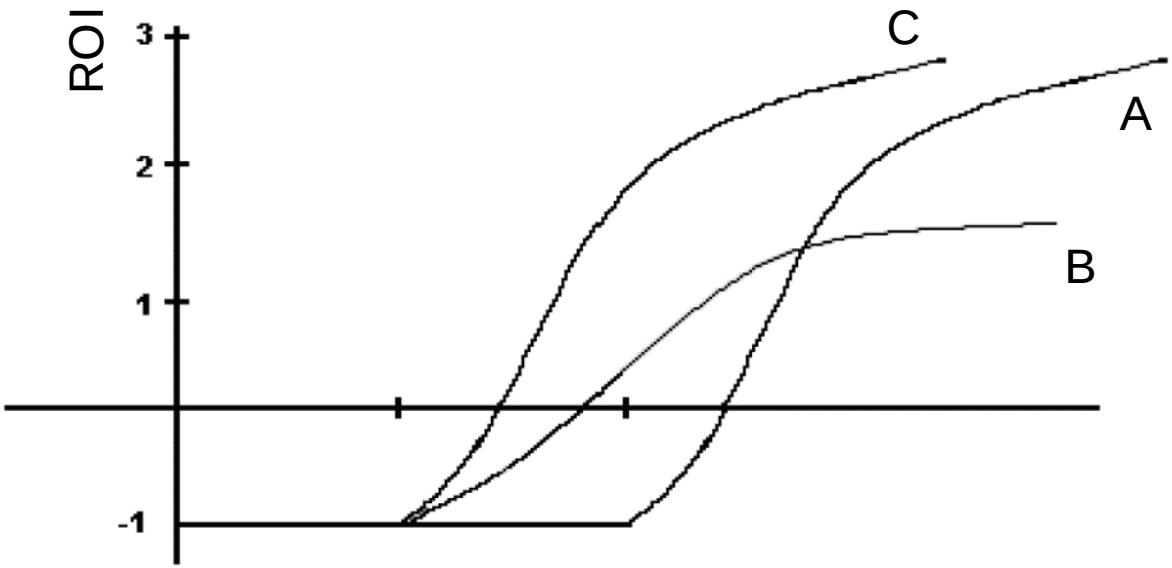
1. Benefits Realization Analysis (BRA): Consequences

- This view turns the software development project into a *business change* <u>*program*</u>
  - 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

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

Freie Universität Berlin

Projections		Current system			New system		
Date	Market size ($M)	Market share %	Sales	Profits	Market share %	Sales	Profits
31 Dec. 2003	360	20	72	7	20	72	7
31 Dec. 2004	400	20	80	8	20	80	8
31 Dec. 2005	440	20	88	9	22	97	10
31 Dec. 2006	480	20	96	10	25	120	13
31 Dec. 2007	520	20	104	11	28	146	16
31 Dec. 2008	560	20	112	12	30	168	19

Time	Expected improvements						Overall customer satisfaction		
Target date	Cost savings	Change in profits	Cumulative change in profits	Cumulative cost	Return on investment	Late delivery (percent)	Customer satisfaction	In-transit visibility	Ease of use
31 Dec. 2003	0	0	0	0	0	12.4	1.7	1.0	1.8
31 Dec. 2004	0	0	0	4.0	–1	11.4	3.0	2.5	3.0
31 Dec. 2005	2.2	3.2	3.2	6.0	–.47	7.0	4.0	3.5	4.0
31 Dec. 2006	3.2	6.2	9.4	6.5	.45	4.0	4.3	4.0	4.3
31 Dec. 2007	4.0	9.0	18.4	7.0	1.63	3.0	4.5	4.3	4.5
31 Dec. 2008	4.4	11.4	29.8	7.5	2.97	2.5	4.6	4.6	4.6

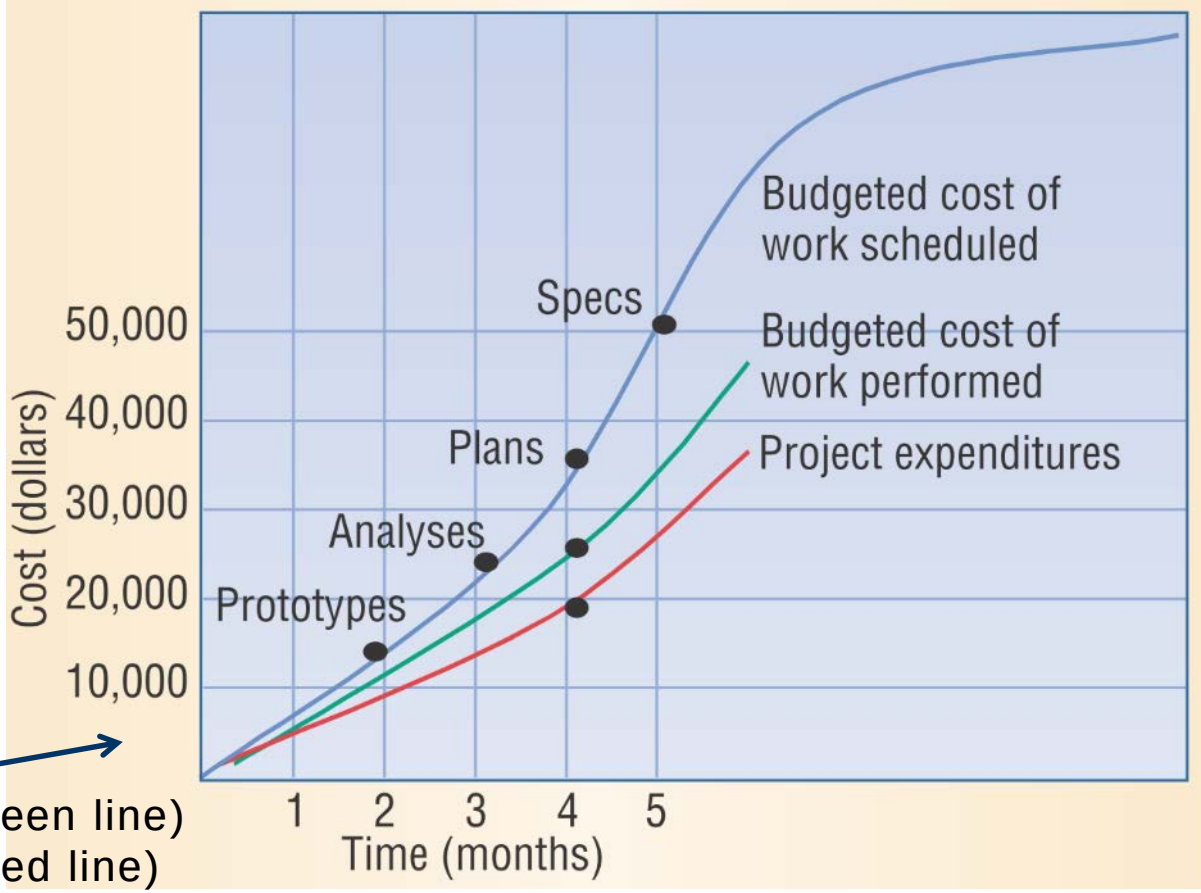
5=perfect

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.

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



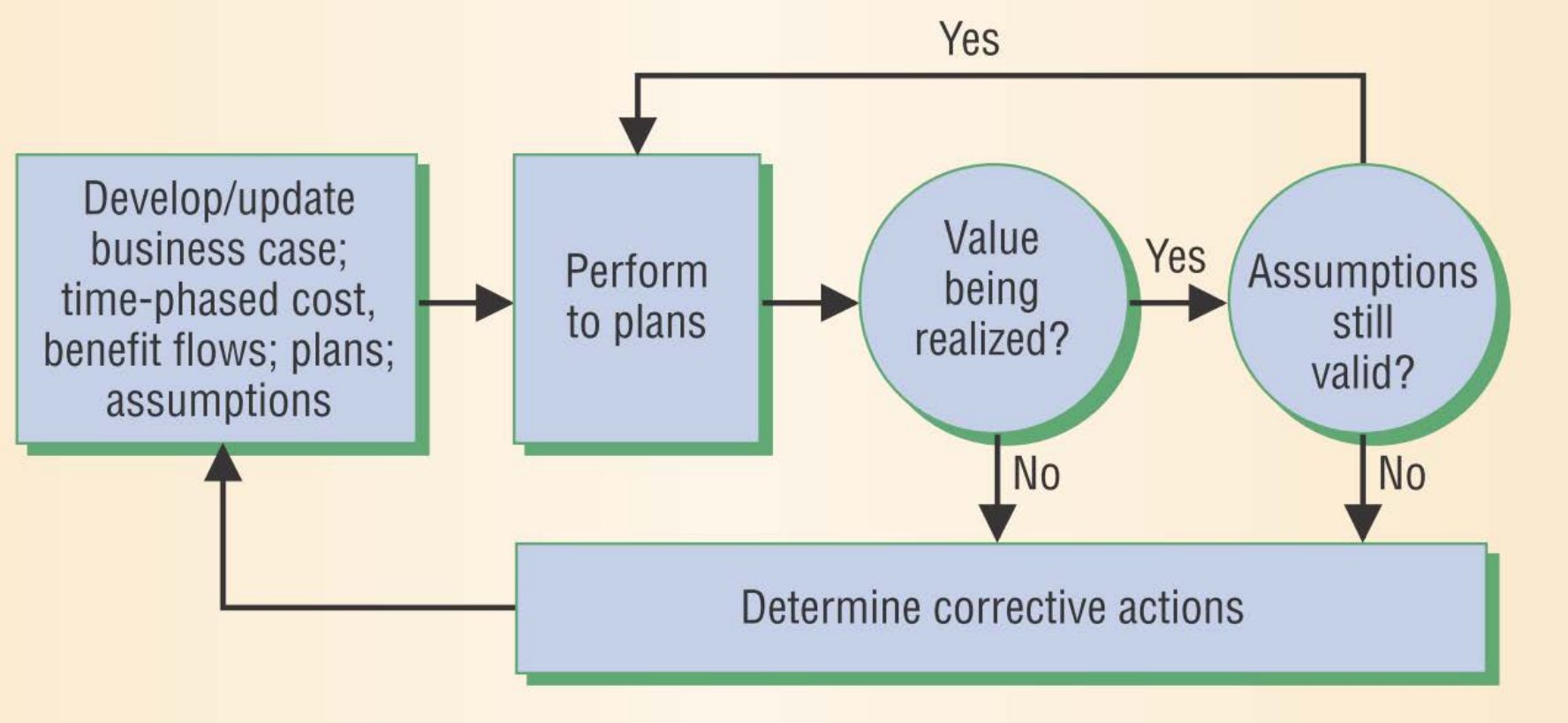
This project is
- behind schedule (green line)
- but below budget (red line)

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 need 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)

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

more columns follow →

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

Freie Universität Berlin

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

Part IV: 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.)

Thales of Miletus
ca. 600 B.C.

Olive oil press

Other kinds of design-related options

Freie Universität Berlin

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

Sullivan, Griswold, Cai, Hallen: "*The Structure and Value of Modularity in Software Design*", ESEC/FSE 2001

Related concept: Technical debt

Freie Universität Berlin

- When our solution knowingly has a non-ideal structure
  - due to then-non-careful work or now-improved understanding
  - we can often save some time implementing it
  - but have to pay this back in change resistance and defects later.

- This situation is called *Technical Debt*
  - Ward Cunningham: "The WyCash Portfolio Management System", OOPSLA 1992
  - http://wiki.c2.com/?WardExplainsDebtMetaphor
    - "I am in favor of writing code to reflect your current understanding of a problem even if that understanding is partial. [...]
      [Y]ou are wise to make that software reflect your understanding as best as you can, so that when it does come time to refactor, it's clear what you were thinking when you wrote it, making it easier to refactor it into what your current thinking is now. "
  - see also https://www.martinfowler.com/bliki/TechnicalDebt.html
    - "The tricky thing about technical debt, of course, is that unlike money it's impossible to measure effectively."

see also http://drops.dagstuhl.de/opus/volltexte/2016/6693/

Summary

- Focussing on very few basic concepts may be helpful
  - e.g. opportunity, stakeholders, requirements, SW system, work, team, way of working
- The conventional view of SE focuses on cost (missing above) and quality (→ SW system)
  - ignoring that different requirements provide different value
- An economical view of SE considers value (→ opportunity)
  - for instance by using value-based project planning & tracking
    - which automatically leads to additional business process changes and a holistic (rather than SW-focused) view of value optimization

- Software design considerations can use the ideas of
  - modularity as providing flexibility value by generating real options (namely for changes);
  - designs that trail behind the current best understanding as Technical Debt

Thank you!