Software Sustainability

Alexander v. Zitzewitz
hello2morrow, Inc.

Code Quality? Yes please, if it is free…

- Do you have binding rules for code quality?
- Do you measure quality rule violations on a daily base?
- Is your architecture defined in a formal way?
- Do you measure architecture violations on a daily base?
- Does quality management happen at the end of development?
- Do you think, that more needs to be done in that area and that this would be beneficial for the team and the company?

Part I: Symptoms of Structural Erosion

Erosion of Architecture – Symptoms (Robert C. Martin)

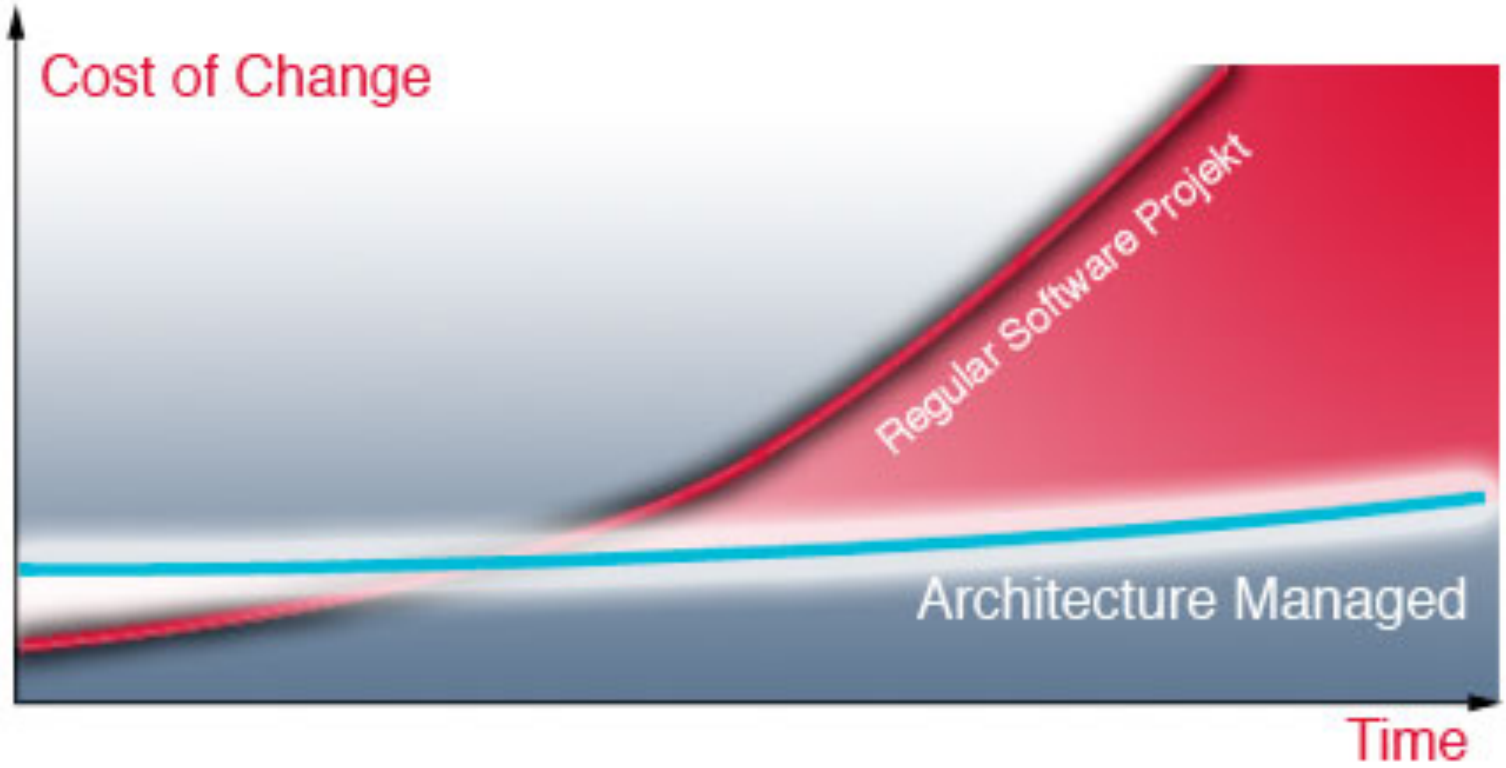
- Rigidity – The system is hard to change because every change forces many other changes.
- Fragility – Changes cause the system to break in conceptually unrelated places.
- Immobility – It's hard to disentangle the system into reusable components.
- Viscosity – Doing things right is harder than doing things wrong.
- Opacity – It is hard to read and understand. It does not express its intent well.

Overall: "*The software starts to rot like a bad piece of meat*"

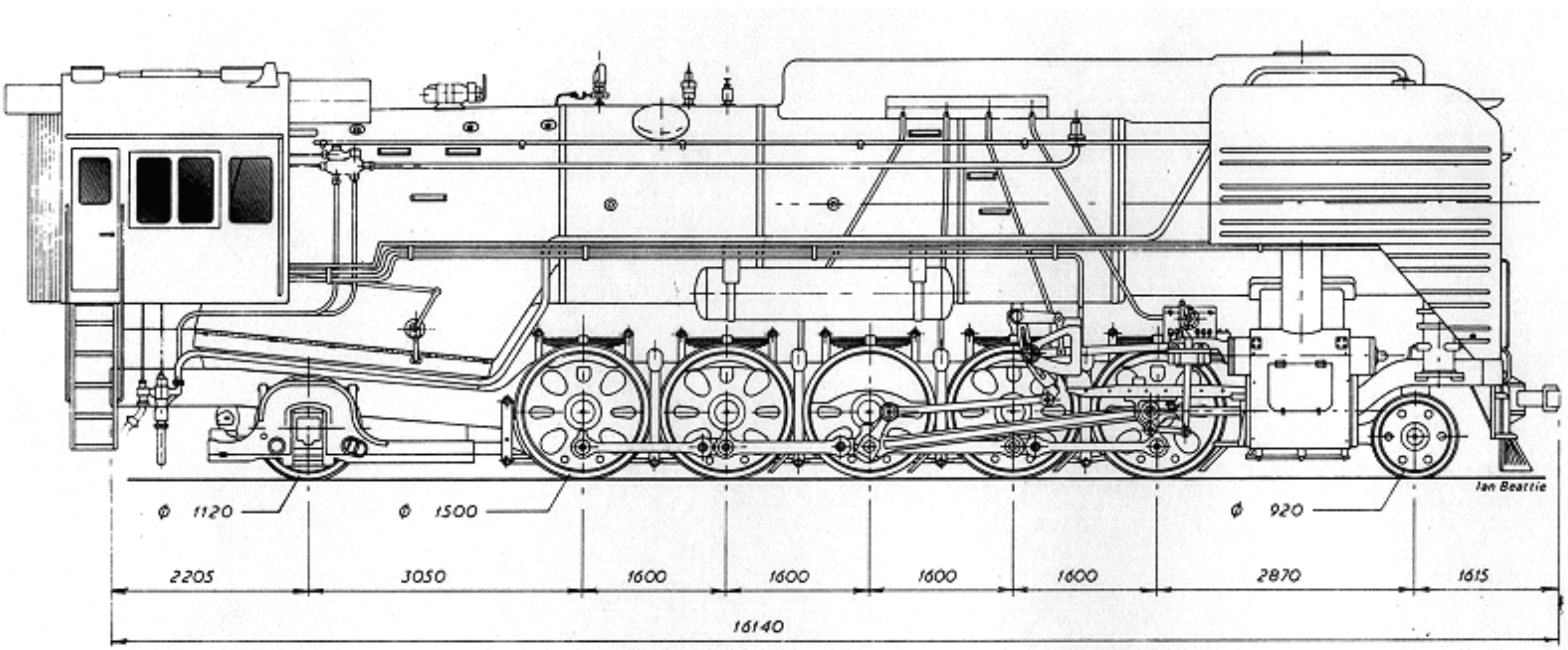
Erosion of Architecture – Reasons

- System knowledge and skills are not evenly distributed
- Complexity grows faster than system size
- Unwanted dependencies are created without being noticed
- Coupling and complexity are growing quickly. When you realize it, it is often too late
- Most projects don't measure quality on a regular base
- Management considers software as a black box
- Quality measurement is done at the end of development
- Time pressure is always a good excuse to sacrifice structure
- The Law of Entropy?

Cost of Structural Erosion / Technical Debt



Part II: Technical Quality and Sustainability



How to Define Technical Quality?

“Technical quality of software can be defined as the level of conformance of a software system to a set a set of rules and guidelines derived from common sense and best practices. Those rules should cover software architecture, programming in general, testing and coding style.”

- Technical quality cannot be achieved by testing only
- Technical quality manifests itself in very line of code
- Four aspects of technical quality:
  - Architecture / Dependency-Structure
  - Software metrics
  - Programming rules
  - Testability and test coverage
- Which of those aspects has the biggest cost impact?
- Measuring of technical quality requires static analysis

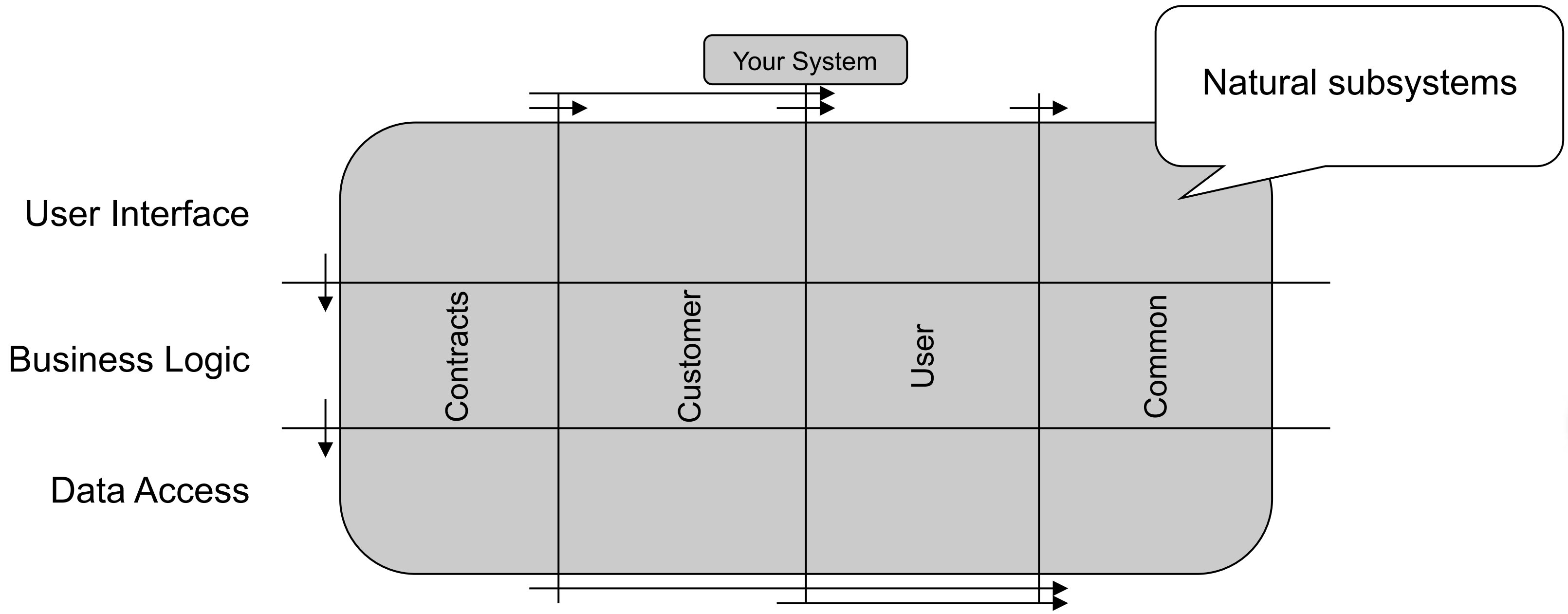
How to Achieve Software Sustainability?

- Sustainability cannot be achieved without the implementation of rules and guidelines
- Achieving sustainability requires effort and this effort needs to be considered in iteration planning.
- By investing a relatively small additional effort today a huge future effort can be avoided.
- On the short term, building sustainable software always costs more. On the long term it can reduce the overall cost of a project by more than 50%.
- Many projects suffer from being short-sighted. Mostly there is no long term planning or strategy in place to achieve a sustainable code base.
- Typically it is sufficient to spend about 20% of the time available in each iteration on sustainability.

Sustainability and Technical Quality

- Sustainability and technical quality are two sides of the same coin.
- Technical quality is a precondition for changeability, maintainability, testability and extensibility.
- Investments in technical quality only pay off in the medium and long term, but the return on investment is close to astronomical.

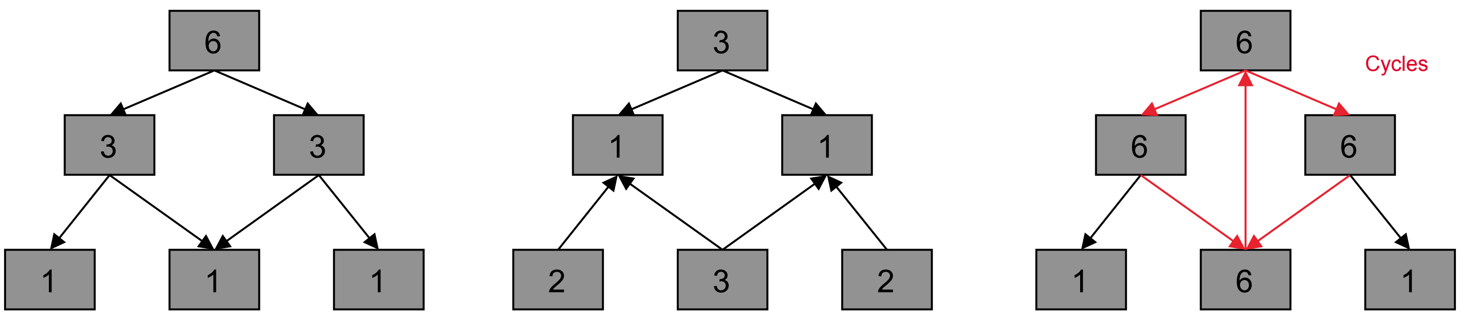
How to model Architecture



- Step 1: Cut horizontally into Layers
- Step 2: Cut vertically into vertical slices by functional aspects
- Step 3: Defines the rules of engagement

How to measure coupling

- ACD = Average Component Dependency
- Average number of direct and indirect dependencies
- rACD = ACD / number of elements
- NCCD: normalized cumulated component dependency



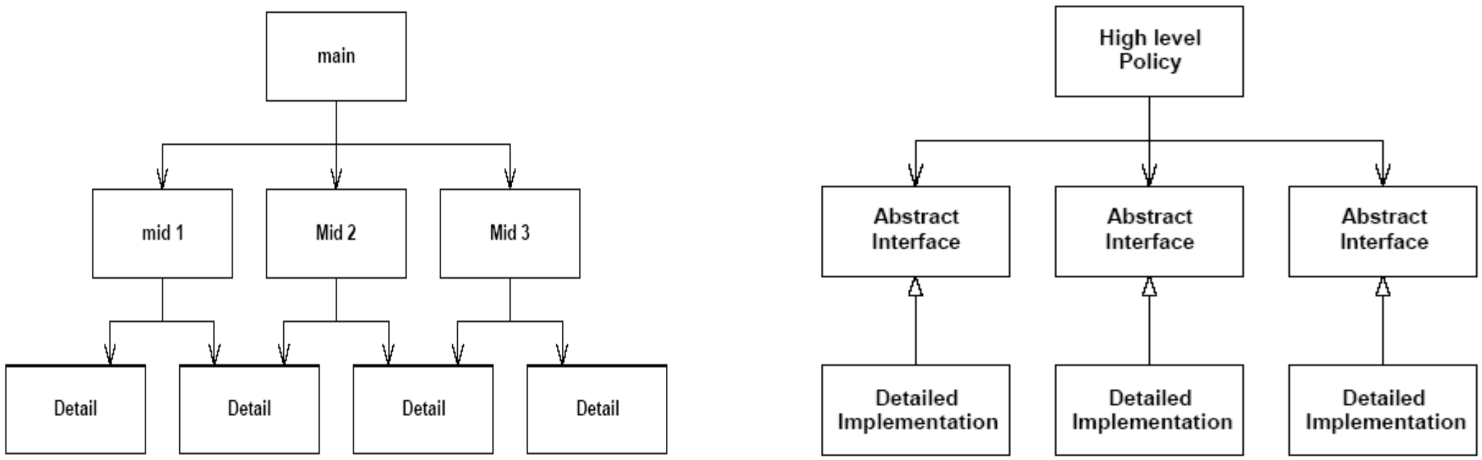
CCD = 15
ACD = 15/6 = 2,5

Dependency Inversion
ACD = 12/6 = 2

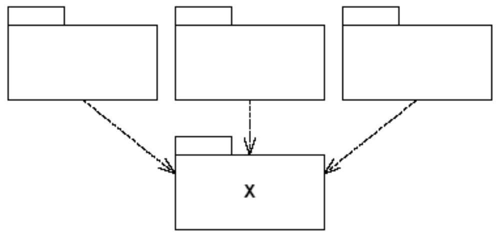
ACD = 26/6 = 4,33

How to keep the coupling low?

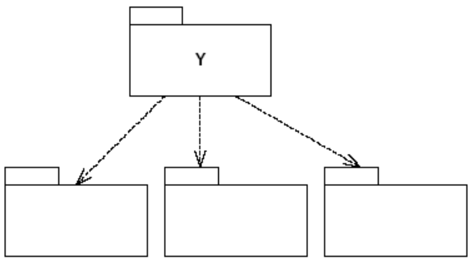
- Dependency Inversion Principle (Robert C. Martin)
  - Build on abstractions, not on implementations
  - Best pattern for a flexible architecture with low coupling
  - Have a look at dependency injection frameworks (e.g. Spring)



Architecture metrics of Robert C. Martin



X is „stable“



Y is „instable“

D_i = Number of incoming dependencies

D_o = Number of outgoing dependencies

Instability $I = D_o / (D_i+D_o)$

Build on abstractions, not on implementations

Abstractness (Robert C. Martin)

N_c = Total number of types in a type container

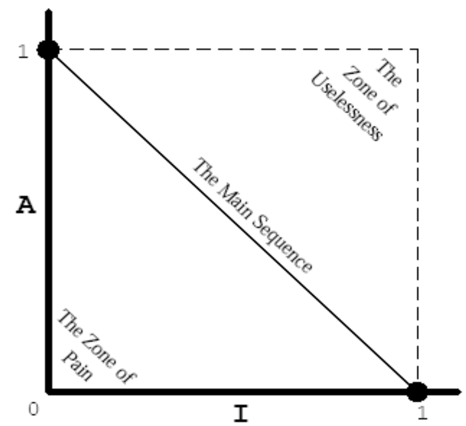
N_a = Number of abstract classes and interfaces in a type container

Abstractness $A = N_a/N_c$

Metric „distance“ (Robert C. Martin)

$D = A + I - 1$

Value range [-1 .. +1]



- Negative values are in the „Zone of pain“
- Positive values belong to the „Zone of uselessness“
- Good values are close to zero (e.g. -0,25 to +0,25)
- „Distance“ is quite context sensitive

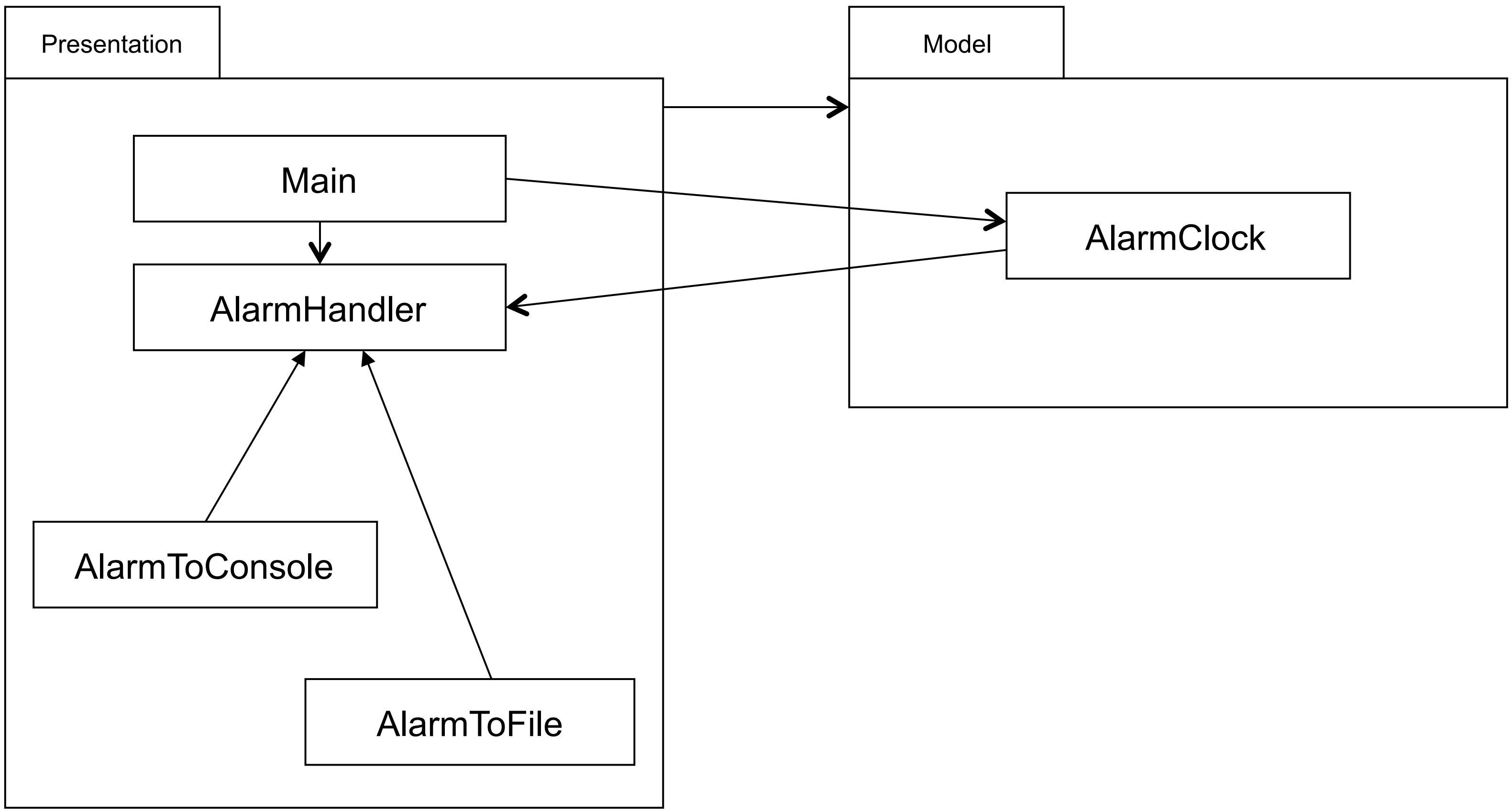
Cyclical Dependencies are Harmful

- "Guideline: No Cycles between Packages. If a group of packages have cyclic dependency then they may need to be treated as one larger package in terms of a release unit. This is undesirable because releasing larger packages (or package aggregates) increases the likelihood of affecting something." [AUP]

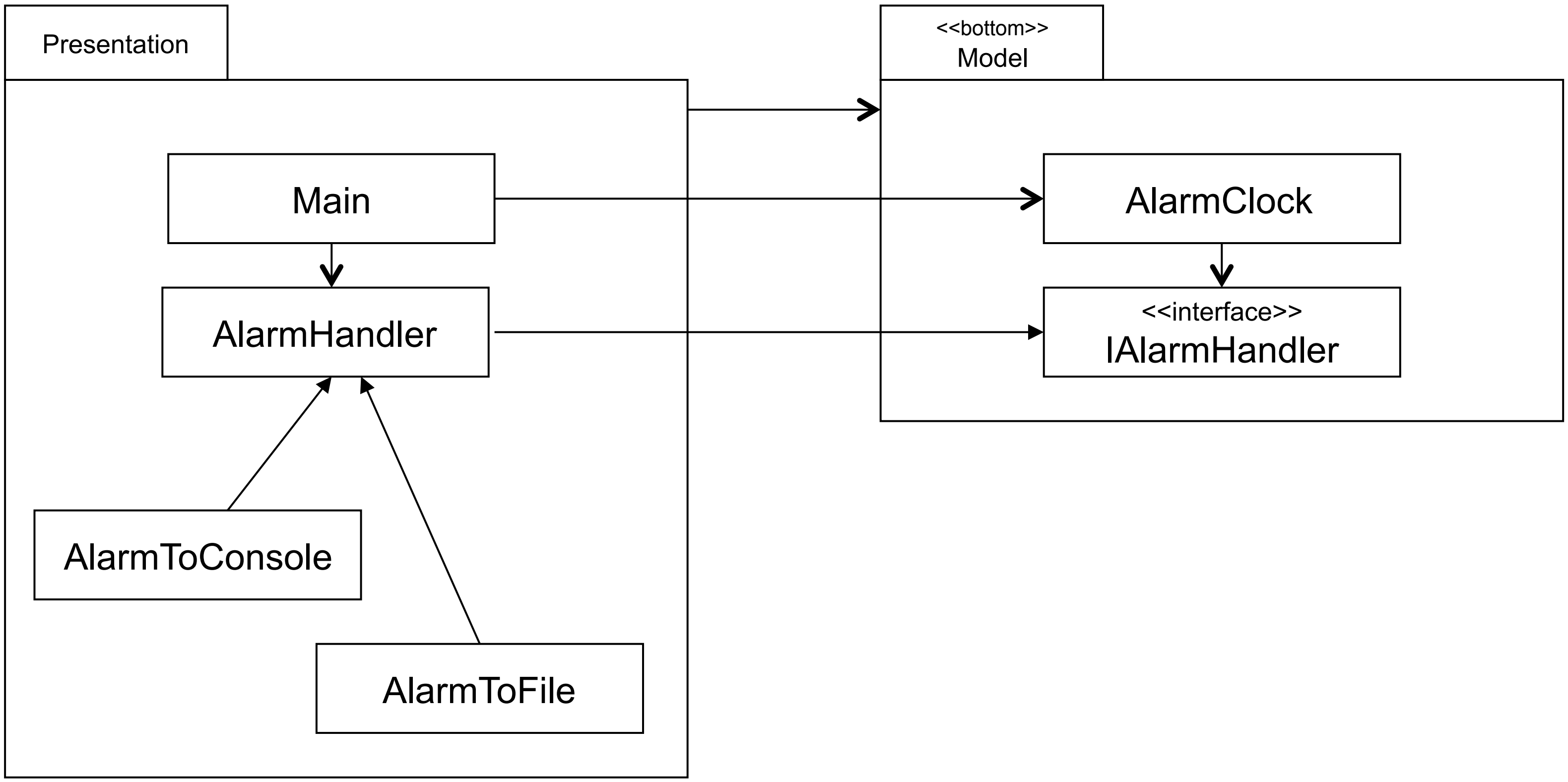
- "The dependencies between packages must not form cycles." [ASD]

- "Cyclic physical dependencies among components inhibit understanding, testing and reuse. Every directed a-cyclic graph can be assigned unique level numbers; a graph with cycles cannot. A physical dependency graph that can be assigned unique level numbers is said to be levelizable. In most real-world situations, large designs must be levelizable if they are to be tested effectively. Independent testing reduces part of the risk associated with software integration " [LSD]

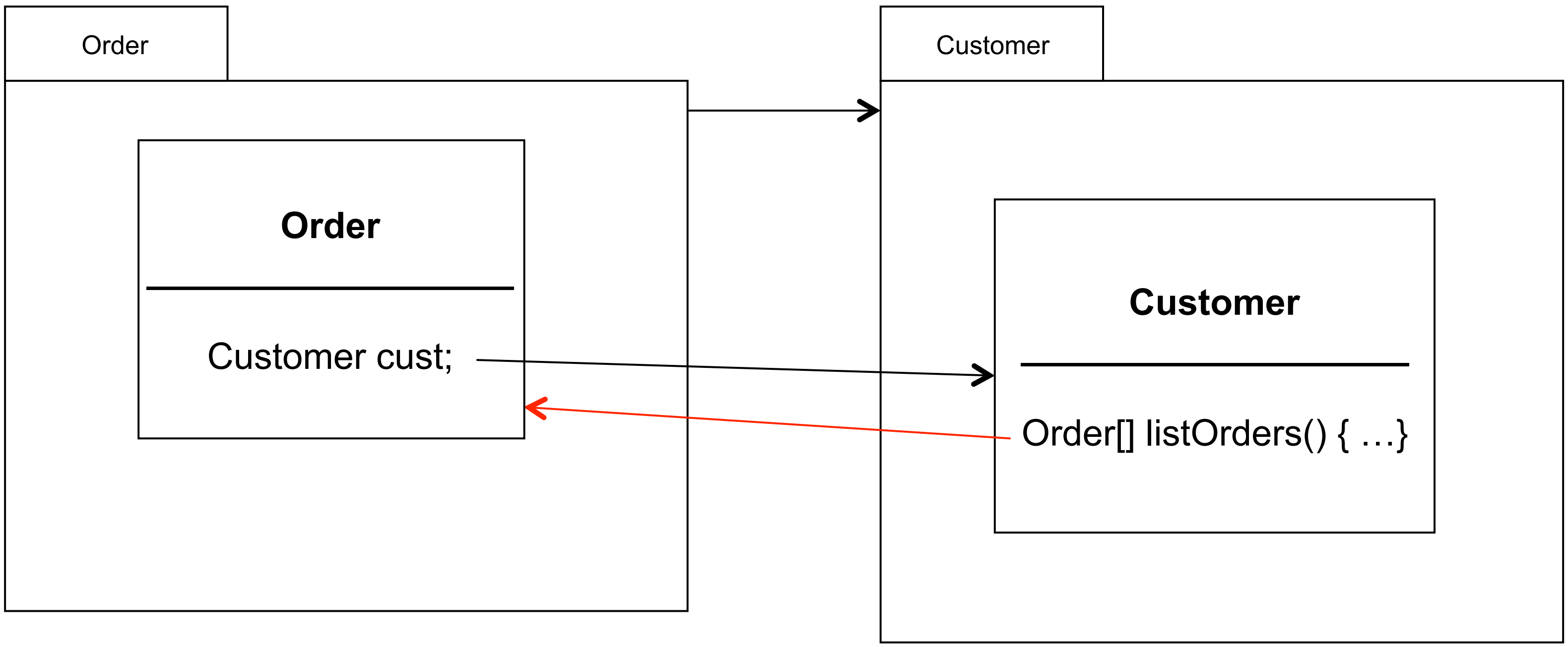
Example: Cyclical Dependency



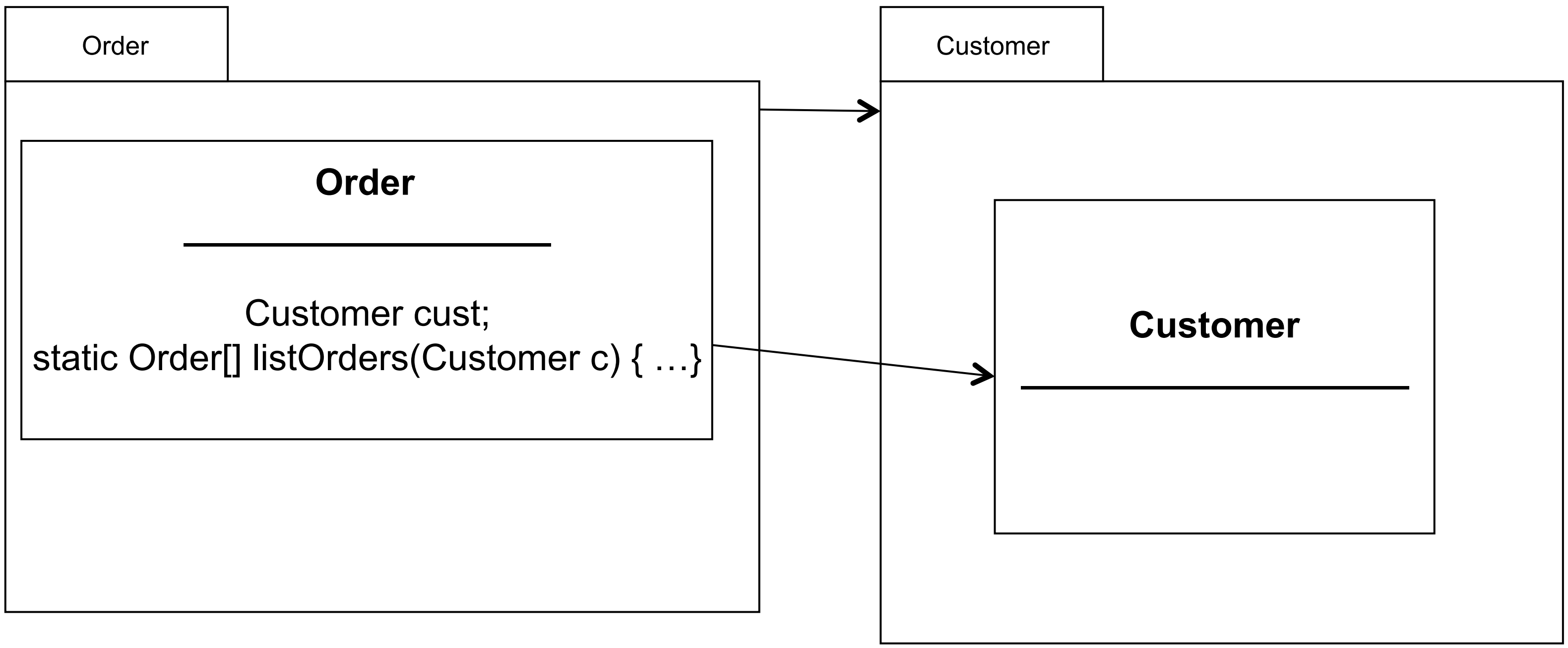
Breaking the Cycle



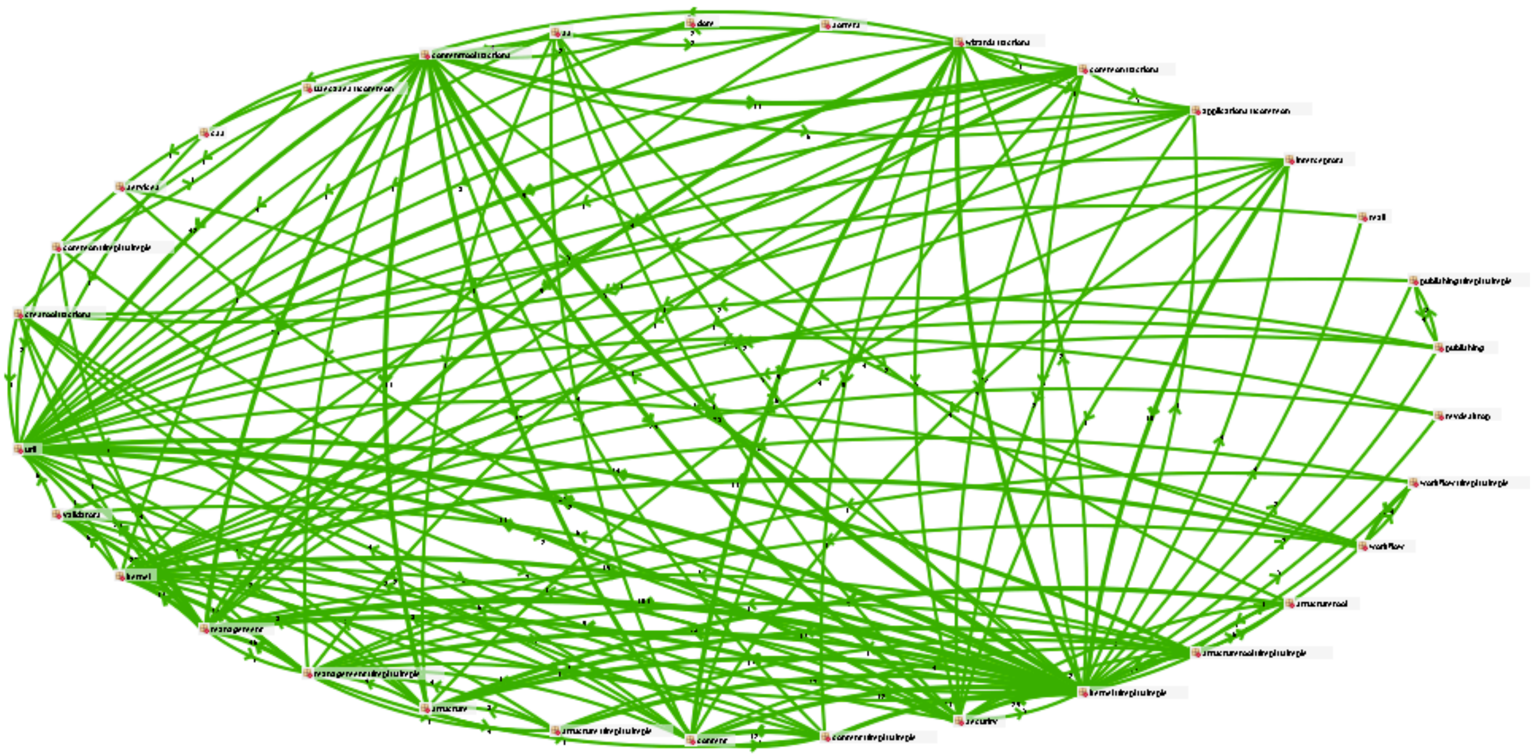
Another Cyclical Dependency



Cycle broken...



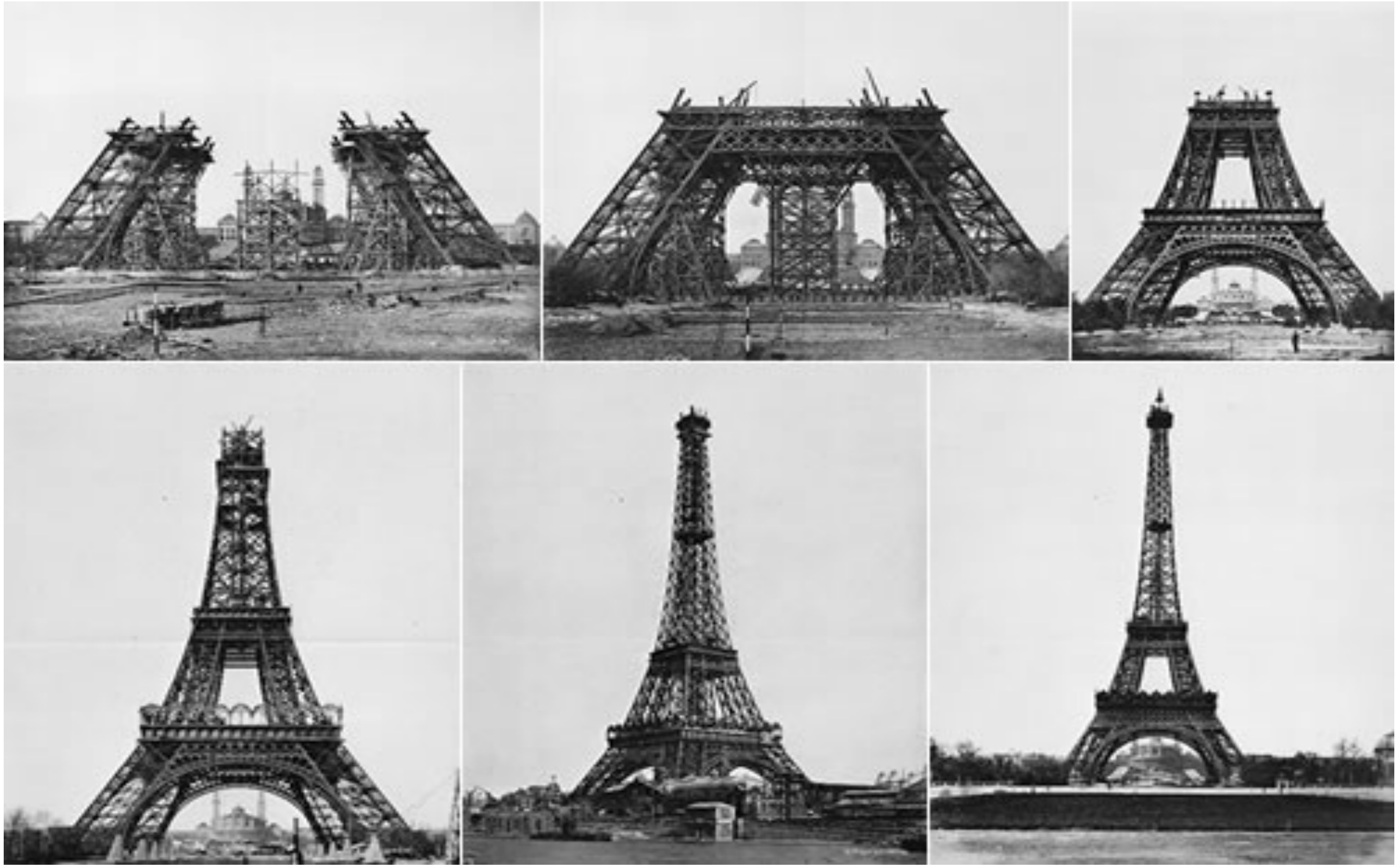
Consequences of Structural Erosion



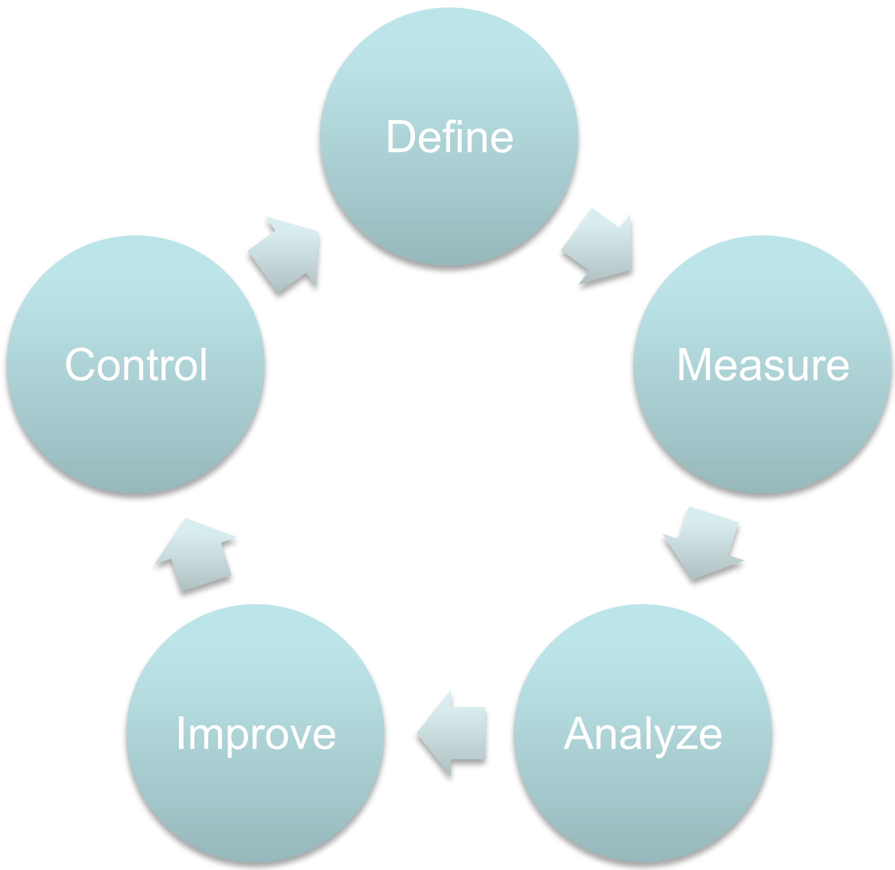
Metric “Structural Debt Index”

- Packages that are part of package cycle groups are sorted by calculating the difference between outgoing and incoming dependencies. Special rules for draws.
- Packages with more outgoing dependencies are above packages with more incoming dependencies
- All upward going dependencies are considered bad
- SDI = 10 * (type dependencies to cut) + (code refs of dependencies to cut)
- Metric should give an idea how difficult it is to clean up a tangled mess

Part III: How to Implement Sustainability



Improvements Require Transparency
Six Sigma for Software



Preconditions for Sustainability

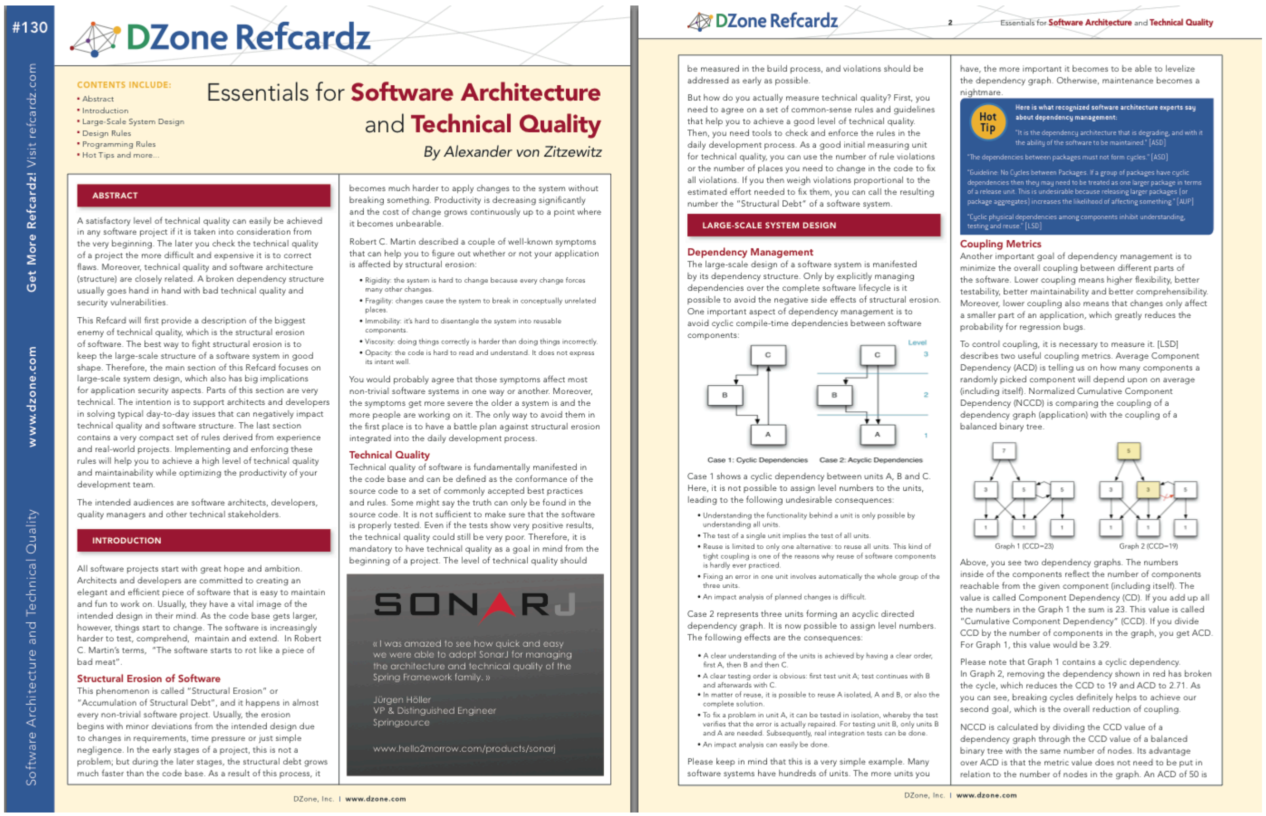
- Nothing can be delivered that does not meet the standards defined fro technical quality.
- Rules and guidelines are documented and checked in an automated way.
- Each project needs to defined an architectural model.
- Cyclical dependencies have to be avoided.
- Quality metrics and checking for rule violations are part of the daily/nightly build.
- Quality criteria are a core component of development guidelines.
- Sustainability as a goal must be supported by all management levels.

Some Simple Rules for Sustainable Projects

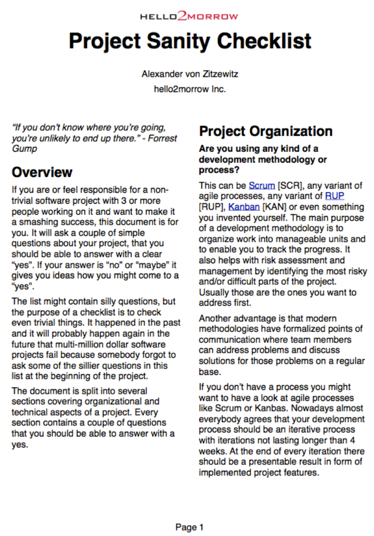
- Rule 1:
  Define a cycle free logical architecture down to the level of subsystems and a strict and consistent package naming convention
- Rule 2:
  Do not allow cyclic dependencies between different packages
- Rule 3:
  Keep the relative ACD low (< 7% for 500 compilation units, NCCD < 6)
- Rule 4:
  Limit the size of Java files (700 LOC is a reasonable value)
- Rule 5:
  Limit the cyclomatic complexity of methods (e.g. 15)
- Rule 6:
  Limit the size of a Java package (e.g. less than 50 types)

DZone's "Designing Quality Software" Refcard

Refcard #130: http://refcardz.dzone.com/



Relevant White-Papers:



HELLO2MORROW

Project Sanity Checklist

Alexander von Zitzewitz

hello2morrow Inc.

"If you don't know where you're going, you're unlikely to end up there." - Forrest Gump

Overview

If you are or feel responsible for a non-trivial software project with 3 or more people working on it and want to make it a smashing success, this document is for you. It will ask a couple of simple questions about your project, that you should be able to answer with a clear "yes". If your answer is "no" or "maybe" it gives you ideas how you might come to a "yes".

The list might contain silly questions, but the purpose of a checklist is to check even trivial things. It happened in the past and it will probably happen again in the future that multi-million dollar software projects fail because somebody forgot to ask some of the sillier questions in this list at the beginning of the project.

The document is split into several sections covering organizational and technical aspects of a project. Every section contains a couple of questions that you should be able to answer with a yes.

Project Organization

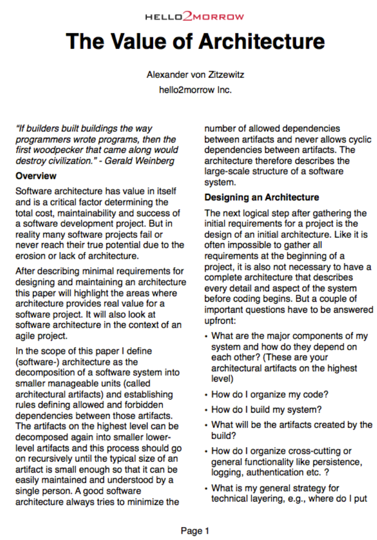
Are you using any kind of a development methodology or process?

This can be Scrum [SCR], any variant of agile processes, any variant of RUP [RUP], Kanban [KAN] or even something you invented yourself. The main purpose of a development methodology is to organize work into manageable units and to enable you to track the progress. It also helps with risk assessment and management by identifying the most risky and/or difficult parts of the project. Usually those are the ones you want to address first.

Another advantage is that modern methodologies have formalized points of communication where team members can address problems and discuss solutions for those problems on a regular base.

If you don't have a process you might want to have a look at agile processes like Scrum or Kanbas. Nowadays almost everybody agrees that your development process should be an iterative process with iterations not lasting longer than 4 weeks. At the end of every iteration there should be a presentable result in form of implemented project features.

Page 1



HELLO2MORROW

The Value of Architecture

Alexander von Zitzewitz

hello2morrow Inc.

"If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization." - Gerald Weinberg

Overview

Software architecture has value in itself and is a critical factor determining the total cost, maintainability and success of a software development project. But in reality many software projects fail or never reach their true potential due to the erosion or lack of architecture.

After describing minimal requirements for designing and maintaining an architecture this paper will highlight the areas where architecture provides real value for a software project. It will also look at software architecture in the context of an agile project.

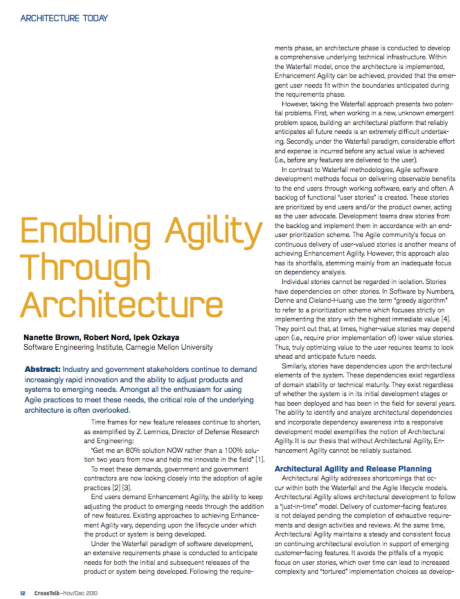
In the scope of this paper I define (software-) architecture as the decomposition of a software system into smaller manageable units (called architectural artifacts) and establishing rules defining allowed and forbidden dependencies between those artifacts. The artifacts on the highest level can be decomposed again into smaller lower-level artifacts and this process should go on recursively until the typical size of an artifact is small enough so that it can be easily maintained and understood by a single person. A good software architecture always tries to minimize the number of allowed dependencies between artifacts and never allows cyclic dependencies between artifacts. The architecture therefore describes the large-scale structure of a software system.

Designing an Architecture

The next logical step after gathering the initial requirements for a project is the design of an initial architecture. Like it is often impossible to gather all requirements at the beginning of a project, it is also not necessary to have a complete architecture that describes every detail and aspect of the system before coding begins. But a couple of important questions have to be answered upfront:

- What are the major components of my system and how do they depend on each other? (These are your architectural artifacts on the highest level)
- How do I organize my code?
- How do I build my system?
- What will be the artifacts created by the build?
- How do I organize cross-cutting or general functionality like persistence, logging, authentication etc. ?
- What is my general strategy for technical layering, e.g., where do I put

Page 1



ARCHITECTURE TODAY

Enabling Agility Through Architecture

Nanette Brown, Robert Nord, Ipek Ozkaya

Software Engineering Institute, Carnegie Mellon University

Abstract: Industry and government stakeholders continue to demand increasingly rapid innovation and the ability to adjust products and systems to emerging needs. Amongst all the enthusiasm for using Agile practices to meet these needs, the critical role of the underlying architecture is often overlooked.

Time frames for new feature releases continue to shorten, as exemplified by Z. Lemnios, Director of Defense Research and Engineering:

"Get me an 80% solution NOW rather than a 100% solution two years from now and help me innovate in the field" [1].

To meet these demands, government and government contractors are now looking closely into the adoption of agile practices [2] [3].

End users demand Enhancement Agility, the ability to keep adjusting the product to emerging needs through the addition of new features. Existing approaches to achieving Enhancement Agility vary, depending upon the lifecycle under which the product or system is being developed.

Under the Waterfall paradigm of software development, an extensive requirements phase is conducted to anticipate needs for both the initial and subsequent releases of the product or system being developed. Following the requirements phase, an architecture phase is conducted to develop a comprehensive underlying technical infrastructure. Within the Waterfall model, once the architecture is implemented, Enhancement Agility can be achieved, provided that the emergent user needs fit within the boundaries anticipated during the requirements phase.

However, taking the Waterfall approach presents two potential problems. First, when working in a new, unknown emergent problem space, building an architectural platform that reliably anticipates all future needs is an extremely difficult undertaking. Secondly, under the Waterfall paradigm, considerable effort and expense is incurred before any actual value is achieved (i.e., before any features are delivered to the user).

In contrast to Waterfall methodologies, Agile software development methods focus on delivering observable benefits to the end users through working software, early and often. A backlog of functional "user stories" is created. These stories are prioritized by end users and/or the product owner, acting as the user advocate. Development teams draw stories from the backlog and implement them in accordance with an end-user prioritization scheme. The Agile community's focus on continuous delivery of user-valued stories is another means of achieving Enhancement Agility. However, this approach also has its shortfalls, stemming mainly from an inadequate focus on dependency analysis.

Individual stories cannot be regarded in isolation. Stories have dependencies on other stories. In Software by Numbers, Denne and Cleland-Huang use the term "greedy algorithm" to refer to a prioritization scheme which focuses strictly on implementing the story with the highest immediate value [4]. They point out that, at times, higher-value stories may depend upon (i.e., require prior implementation of) lower value stories. Thus, truly optimizing value to the user requires teams to look ahead and anticipate future needs.

Similarly, stories have dependencies upon the architectural elements of the system. These dependencies exist regardless of domain stability or technical maturity. They exist regardless of whether the system is in its initial development stages or has been deployed and has been in the field for several years. The ability to identify and analyze architectural dependencies and incorporate dependency awareness into a responsive development model exemplifies the notion of Architectural Agility. It is our thesis that without Architectural Agility, Enhancement Agility cannot be reliably sustained.

Architectural Agility and Release Planning

Architectural Agility addresses shortcomings that occur within both the Waterfall and the Agile lifecycle models. Architectural Agility allows architectural development to follow a "just-in-time" model. Delivery of customer-facing features is not delayed pending the completion of exhaustive requirements and design activities and reviews. At the same time, Architectural Agility maintains a steady and consistent focus on continuing architectural evolution in support of emerging customer-facing features. It avoids the pitfalls of a myopic focus on user stories, which over time can lead to increased complexity and "tortured" implementation choices as develop-

12 CrossTalk—Nov/Dec 2010

Download from www.hello2morrow.com

Awards and nominations

- Second prize of Jax innovation award in April 2007
- Nomination for European ICT prize 2007
- Awarded as most exciting innovation on Systems 2005

Q & A

Some of our more than 300 customers:

SIEMENS
UOB 大華銀行
KÜHNE+NAGEL
COMMERZBANK
ONB OESTERREICHISCHE NATIONALBANK
Ford
BMW
GE
Standard Chartered
sanofi aventis
DAIMLER-BENZ
Deutsche Bank
CREDIT SUISSE
Capgemini
T · ·Systems· · ·

HELLO2MORROW

Q & A

My Email Address:

a.zitzewitz@hello2morrow.com

Win a brand new IPad by visiting our booth
And watch a demo of Sonargraph