Learning from failure

Failure and understanding failure is a key factor for successful design!

[Henry Petroski]
**Warning:** This presentation is about pragmatic software development projects. It only contains very few scientific insights.

Or as an unknown scientist called Einstein once said: In theory, theory and practice are the same. In practice, they are not.

But I will provide the first Perpetuum Mobile Silver Bullet for optimizing systems:

---

There actually IS a silver bullet for boosting software engineering productivity: shoot the right person.
### One Side of the Coin: System & Software Architecture Design

You could design the best architecture if you knew **everything** in advance – i.e., we could anticipate change.

In that case, the Waterfall model would be a perfect fit.

Unfortunately, the real world is not perfect. **It is changing in unanticipated ways.** That's what we call evolution.

But for sustainable software architecture we need at least a stable base, i.e. core design that does only change rarely.

### Can we balance both sustainable architecture design and need for change?

---

© Prof. Dr. Michael Stal, 2012
Nothing is Permanent except Change
QCon London 2012

Agenda

- A bit of architecture change
- Requirements Engineering and its impact on design
- Some other architecture viewpoints
- And now for something completely different: testing
- More architecture issues I forgot to cover so far
- Premature refactoring
- But here the talk is over, maybe 😃
- Did I already mention change?
Change and Architecture?  - Two Ends of the Spectrum

Architecture design from day 1 in a total agile approach  Architecture with Big Design Up Front

Unstable architecture!  Nice but either unsuitable or overengineered architecture!

Quo Vadis, Architect?

- Is agile architecture a kind of magic only gurus and wizards can master?

As designers we like to get the best of both worlds: obtain a stable, sustainable core, AND embrace change.

Can we achieve this objectives and, if yes, how would we proceed?

Yes, we can!
Let's dig deeper

The iterative incremental life cycle evolutionary model releases created as prototypes in an iterative approach based on accurately defined rules.

An iteration

- a sequence of activities or tasks performed within a period of time (4-8 weeks)
- typically requirements analysis (at least partially), design, implementation, integration, testing (partially)

An increment is a

- stable, executable and testable software release
- provides new functionality that moves the product closer to its completion

Iterative means re-do:
- a rework scheduling strategy which helps to improve the (quality of the) product

Incremental means add onto:
- a staging and scheduling strategy which helps to improve the process and feature set by avoiding a big-bang Integration

→ We should use both approaches together, i.e. we use the benefits of both!!!
Introducing a Change-Based Quality Feedback Loop

Define and realize a software architecture using an iterative, risk-driven, requirements-driven, and test-driven development process, in which:

- **An iterative, time-boxed** approach provides continuous feedback.
- **Risk- and requirements orientation** ensures that the most important aspects of the system’s realization are addressed first.
- A **test-driven** approach provides concrete feedback on the quality of the architecture and its realization.

The goal of each iteration is to produce **product quality and less risk**, so that the next iteration can be taken on safe ground.

---

Why we should care about Architecture

*If you think good architecture is expensive, try bad architecture*  
[Brian Foote and Joseph Yoder]
Architecture versus Design (according to Len Bass)

"Design is a continuous activity of making decisions beginning with a collection of decisions that have broad system wide scope and moving to a collection of decisions that have very narrow scope.

I would characterize a decision as architectural if it has one or more of the following properties:
- it has system wide impact
- it affects the achievement of a quality attribute important to the system"

“Architecture is about the important things” [Martin Fowler]
“Architecture is about everything costly to change” [Grady Booch]

Strategic and Tactical Design

Strategic design focuses on global system scope
At the beginning consider only strategic requirements, i.e., requirements with systemic and strategic impact:
- All functional requirements
- All operational requirements

Tactical Design encompasses all local design decisions with non-systemic impact
Tactical requirements are requirements with local scope such as developmental requirements (e.g., modifiability)

„Stable“ Strategy but Tactical Adaptations
The Art of Architecture

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

[C.A.R Hoare]

Some Advice from an Expert: Frederik P. Brooks

Design comprises three phases:
1. Formulation of conceptual constructs
2. Implementation in real media
3. Interaction with real users

Iterative evolutionary design is essential
In the first months requirements engineering and architecture design should go hand in hand, because
While architects don’t fully understand the requirements, customers don’t fully understand the design
Write down all assumptions about users and their uses in the beginning
Learn from your predecessors
Maintain a sketch book with your ideas, concepts
Nothing is Permanent except Change
QCon London 2012

An Architecture Process from 10000 feet that addresses Change

Determine Forces
Preparation
Create Architecture Baseline

Feedback Loop
Refine & Assess Architecture
Refactor Architecture

Executable Increment
Complete?
no
yes
Executable Architecture

Piecemeal Growth

Architecture approach in detail

Perform User Requirements Elicitation
Create Domain Model
Model dynamics of scenarios
Determine Scope & Boundaries
Create first conceptual draft

Structure the Baseline Architecture
Introduce Deployment Views
Define Principles & Guidelines
Plan and Realize Increments
The Onion Model – Two Core Principles: Priority Driven & Strategic before tactical design

Architecture design follows an onion model:
- start with the inner core
- incrementally continue with outer layers

Functional Core
Distribution & Concurrency
Infrastructure
Strategic Qualities
Prio high … low
Tactical Qualities
Prio high … low

No Deep Dive - The Pyramid Model

How deep do we need to go?

The baseline architecture must be as complete as necessary to govern the subsequent software development, but it must also be as simple as possible, otherwise it cannot be communicated.

*Three levels of detail to limit depth*

*A focus on architecturally significant requirements and corresponding architecture views to limit breadth*
Scenario-based Approaches
- Example: Design for Change

**Source:** End user, developer, system admin

**Stimulus:** Wishes to add/delete/modify/vary functionality, quality attribute

**Artifact:** System User Interface, platform, environment

**Environment:** @ runtime, compile time, build time, design time

**Response:** Locates places in Arch. To be modified (without affecting other functionality)

Response measure:
Cost in terms of elements affected

---

Design for Change

**Goal**
Changes Arrive

Tactics to Control Change

Changes Made, Tested and deployed within time and budget

---

**Changeability**

**Localize Changes**
- Semantic Coherence
- Anticipate Expected Changes
- Generalize Module
- Limit possible options
- Abstract common services

**Prevention of Ripple Effect**
- Hide Information
- Maintain existing Interface
- Restrict Communication Paths
- Use an intermediary

**Defer Binding Time**
- Runtime Registration
- Configuration Files
- Polymorphism
- Component Replacement
- Adherence to defined protocols

---

**But ... mind the caveat of overdoing it**
(Re-)Use is Essential

Human beings, who are almost unique in having the ability to learn from the experience of others, are also remarkable for their apparent disinclination to do so.

[Douglas Adams. 1952-2001. Last Chance to See]

Some Words about Dealing with Technology and Hardware Changes

Use feasibility prototypes for technologies that are mission critical
   Especially to check for quality attributes
Use simulations if you are in a system engineering / embedded system context:
   Simulate what is missing or difficult to test otherwise
   Do not only test at Q Gates
An Architect’s Framework

The mindset, activities, practices, methods, and technologies for defining and realizing software architectures form a best practice framework for software architects to...

- Specify and implement a software architecture systematically and in a timely fashion
- Check and ensure the appropriate architectural quality
- Respond to changes of all kinds, such as changing requirements and priorities
- Deal with problems that arise during the definition and realization of the software architecture

Panta rhei - Evolutionary Design embraces Change

There is nothing permanent except change

[Heraclitus, 535–475 BC]
Nothing is Permanent except Change
QCon London 2012

Page 29

Design erosion is the root of evil

In the lifecycle of a software system
changes are the rule and not the
exception

Unsystematic approaches
(“workarounds”) cure the symptom
but not the problem

After applying several workarounds,
software systems often suffer from
design erosion

Such systems are doomed to fail
(negative impact on operational &
developmental properties)

How do we know we must improve?

Lack of Internal or External Quality

Quality Attributes (use methods like
ATAM), and,

Structural quality indicators which
include

Economy
Visibility
Spacing
Symmetry
Emergence

Consequently, the goal of
architecture improvement is to
achieve or meet such qualities

Page 30
There is a strange Smell

If it stinks, there must be something we need to clean up

Architecture smells
- Duplicate design artifacts
- Hammer & Nail syndrome
- Unclear roles of entities
- Inexpressive or complex architecture
- Everything centralized
- Home-grown solutions instead of best practices
- Over-generic design
- Asymmetric structure or behavior
- Dependency cycles
- Design violations (such as relaxed instead of strict layering)
- Inadequate partitioning of functionality
- Unnecessary dependencies

Example of Architecture Problem

A true story: In this example architects introduced Transport Way as an additional abstraction. But can't we consider transport ways as just as another kind of storage? As a consequence the unnecessary abstraction was removed, leading to a simpler and cleaner design.
Possible Refactoring Pattern

Context
Eliminating unnecessary design abstractions

Problem
Minimalism is an important goal of software architecture, because minimalism increases simplicity and expressiveness
If the software architecture comprises abstractions that could also be considered abstractions derived from other abstractions, then better remove these abstractions

General solution idea
Determine whether abstractions / design artifacts exist that could also be derived from other abstractions
If this is the case, remove superfluous abstractions and derive from existing abstractions instead

Caveat
Don’t generalize too much (such as introducing one single hierarchy level: “All classes are directly derived from Object”)

Yet another example

Monitors need to access agents to control the network
Agents change their internal state and create events
To have a common clock architects decided to enhance the monitor
... which leads to a cycle
Yet Another Refactoring Pattern

Context
Cyclic dependencies between subsystems

Problem
System reveals at least one dependency cycle between subsystems
Subsystem A may either depend directly or indirectly on subsystem B (e.g., A depends on C which depends on B) which is why we always need to consider the transitive hull
Dependency cycles make systems less maintainable, changeable, reusable, testable, understandable

General solution idea
Get rid of the dependency cycle by removing or inverting dependencies

Change needs Refactoring

Refactoring is integrated into the architecture design process:
- It improves the structure
- It supports a risk-, requirements- and test-driven approach
After refactoring check for correctness

To check the correctness of refactorings, we should use a test-driven approach. Available options:

- **Formal approach**: Prove semantics and correctness of program transformation
- **Implementation approach**: Leverage unit and regression tests to verify that the resulting implementation still meets the specification
- **Architecture analysis**: Check the resulting software architecture for its equivalence with the initial architecture (consider requirements)

Use at least the latter two methods to ensure quality

Frequently discussed Obstacles to Refactoring

- **Organization / management**
  - Featuritis: Considering improvement by refactoring as less important than features
  - “Organization drives architecture” problem

- **Process support**
  - No refactoring activities defined in process
  - Refactorings not checked for correctness, test manager not involved

- **Technologies and tools**
  - Unavailability of tools: refactoring must be done manually
  - Unavailability of refactoring catalog

- **Applicability**
  - Refactoring used instead of reengineering
  - Wrong order of refactorings
Reengineering – when and how to use it

Use Reengineering when

- The system's documentation is missing or obsolete
- The team has only limited understanding of the system, its architecture, and implementation
- A bug fix in one place causes bugs in other places
- New system-level requirements and functions cannot be addressed or integrated appropriately

Process

**Phase I: Reverse engineering**
- Analysis / recovery: determine existing architecture (consider using CQM)
- SWOT analysis
- Decisions: what to keep, what to change or throw away

**Phase II: Forward engineering**

Rewriting in a Nutshell

Rewriting is a radical and fresh restart: existing design and code is trashed and replaced by a whole new design and implementation. Depending on focus:

- Improves structure regarding:
  - Simplicity, visibility, spacing, symmetry, emergence
  - Maintainability, readability, extensibility
  - Bug fixing
- Provides new functionality
- Improves its operational qualities
- Improves design and code stability

As a consequence, rewriting addresses all types of software quality: functional, operational, and the various developmental qualities
Refactoring, reengineering, and rewriting comparison (1)

Refactoring, reengineering, and rewriting are complementary approaches to sustain architecture and code quality.

**Start with refactoring** – it is cheap and (mostly) under the radar.

**Consider reengineering when refactoring does not help** – but it is expensive.

**Consider rewriting when reengineering does not help** – but it is expensive and often risky.

---

Refactoring, reengineering, and rewriting comparison (2)

<table>
<thead>
<tr>
<th></th>
<th>Refactoring</th>
<th>Reengineering</th>
<th>Rewriting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scope</strong></td>
<td>Many local effects</td>
<td>Systemic effect</td>
<td>Systemic or local effect</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>Structure transforming</td>
<td>Behavior / semantics preserving</td>
<td>Disassembly / reassembly</td>
</tr>
<tr>
<td><strong>Results</strong></td>
<td>Improved structure</td>
<td>Identical behavior</td>
<td>New system</td>
</tr>
<tr>
<td><strong>Improved qualities</strong></td>
<td>Developmental (might change Operational Quality)</td>
<td>Functional</td>
<td>Operational</td>
</tr>
<tr>
<td><strong>Drivers</strong></td>
<td>Complicated design / code evolution</td>
<td>Refactoring is insufficient</td>
<td>Bug fixes cause rippling effect</td>
</tr>
<tr>
<td><strong>When</strong></td>
<td>Part of daily work</td>
<td>Requires a dedicated project</td>
<td>Requires dedicated effort or a dedicated project, depending on scope</td>
</tr>
</tbody>
</table>
Mind your Architecture Governance

Without Architecture Governance the System is subject to uncontrolled Change and Extension

Introduce countermeasures, e.g.,:
- Architecture Guidelines and Policies as well as their Enforcement
- Means to ensure Requirements Traceability
- No Checking-in without other Persons reviewing Code and Documents
- Test-Driven-Design
- Risk-Based Analysis & Test

Software Architect’s Dilemma

Life must be understood backwards; but ... it must be lived forward

[Søren Aabye Kierkegaard, Danish philosopher and theologian, 1813-1855]
Reviews help finding the Bad Smells

Quantitative Architecture Reviews
- Code quality assessment
- Simulations
- Prototypes

Qualitative Architecture Reviews
- Scenario-based approaches
- Experience-based approaches

An Architecture Assessment or Review should not be considered an afterthought.

It is a means to check a system regularly and find problems early

Visualization Tools help keeping the system in good Shape

In many projects the responsibility for internal code and design quality is not well defined.
The software architect has to ensure that the required CQM activities are established.

The software architect should be the protector of the quality of the software system!

Use Visualization Tools at least in larger code bases

By the way: this is a real system
“Preventive Maintenance”

Experts solve problems, geniuses avoid them

[Albert Einstein]

Architecture Quality is also influenced by other aspects

Involvement of Software Architects in different phases and disciplines

- Business & Strategy
- Rollout & Maintenance
- Requirements Engineering
- Test & Quality
- Design
- Integration & Implementation
Nothing is Permanent except Change
QCon London 2012

Architects & Requirements – Problem 1:
Understanding the Requirements

A program which perfectly meets a lousy specification is a lousy program

[Cem Kaner, Software Engineering Professor and Consumer Advocate]
Problem 3: Sources of Requirements

The system needs to support timezones
The system should be faster than light
The system should offer an interface
Did I tell them we have only 4 weeks
The system should be cheap AND offer 24 x 7

=> Requirements must have high Quality

Quality of Requirements determines Quality of Software Architecture

Cohesive
Complete
Consistent
Correct
Current
Externally Observable
Feasible
Unambiguous
Mandatory
Verifiable

Upon frequent change, quality of requirements is essential!
No Risk – No Fun?

The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in; we're computer professionals. We cause accidents.

Nathaniel Borenstein, US Programmer

Needless to say, ad-hoc changes imply higher accidental complexity.

Mind all Risks and conduct a Risk Analysis early

Approach for risk analysis according to Christine Hofmeister (“Applied Software Architecture”):

- **Description of risk**: e.g., dependence on persistence layer
- **Influential factors that lead to this risk**: e.g., requirement to decouple business from persistence layer, not enough technology skills in team
- **Solution approach**: e.g., introduce data access layer
- **Possible strategies**: e.g., give subproject to external company, use open source solution, use platform-specific solution
- **Related topics and strategies**: e.g., decoupling business logic from other backend layers
Knowing the expectations is essential

At least at project begin,
Architects don’t understand requirements very well
Customers tell what they want, not what they need
Architects may even not know the implicit requirements

Hence,
Keep in touch with Customers
Apply a KANO Analysis
Understand your Business Goals
Develop Design and Requirements in parallel

Testing as a never ending story

Testing is an infinite process of comparing the invisible to the ambiguous in order to avoid the unthinkable happening to the anonymous

[James Bach, Test Guru]
Testing is about Safety Nets not about Control

Observation (Peter Zimmerer): *Software products are never released – they escape!*

Consequence: Mind the testing necessities during architecture design:
- Test Driven Design
- Test Exit Criteria
- Code Quality Management
- Appropriate Test Methods

Risk-Based Test Strategies are a good approach

---

Risk Based Test Strategy

- Evaluate risks:
  - What is the risk
  - Which part of the system does it affect
  - How likely is the risk
  - How big is the possible damage
  - What priority does the risk have
  - Can it be tested? If yes, when and using what method
  - Can the test be automated
  - Which resources (budget, time, ---) are required
Design for testability – Practical definition

Visibility / observability
What you see is what you test
Ability to observe the outputs, states, internals, resource usage, and other side effects of the software under test
Interaction with the system under test through observation points

Control(lability)
The better we can control the software, the more testing can be automated and optimized
Ability to apply inputs to the software under test or place it in specified states (for example reset to start state)
Interaction with the system under test through control points

Communication is essential

Software Development is a collaborative game

[Alistair Cockburn]
Change-based Design requires Effective Agile Communication

Leadership, and communication and interaction with other roles in software development, are probably the most time-intensive and most important responsibilities of a Software Architect.

The roles with whom the architect interacts, the topics about which they interact with these roles, and the intensity of the interaction depend on the concrete development workflow and activity performed in a software project.

Conclusions

Architecture Change should be considered in the whole lifecycle, not only at the end - it is a crosscutting concern.

The Development and the Architecture Design Process must support change.
There must be a balance between change and architectural stability.
Piecemeal Growth needs to be combined with Architecture Assessment.
Test Driven Design introduces Safety Nets.
Change requires Agile Communication.

This is what Agile Architecture is about.
A departing thought

Each problem that I solved became a rule which served afterwards to solve other problems.

[René Descartes, 1596–1650, in "Discours de la Methode"]