SIEMENS
Ingenuity for life

PLM and Innovation Excellence

Learning Campus

Your partner for Business Learning

Siemens Core Learning Program

Architecture Erosion & Refactoring

Authors: Prof. Dr. Michael Stal | Dietmar Schütz | Wieland Eckert | Jörg Bartholdt

Rotten Software

Sometimes the developers manage to maintain this purity of design through the initial development and into the first phase.

More often something goes wrong.

The software starts to rot like a piece of bad meat.

[Robert C. Martin (Uncle Bob):
“Agile Software Development”]

Refactoring

Learning objectives

- Understand design erosion and how to avoid it
- Learn principles of refactoring
- Know activities and best practices refactoring
- Understand how reengineering and rewriting differ from refactoring

Refactoring

Agenda

Design Erosion & Smells

Refactoring / Reengineering / Rewriting

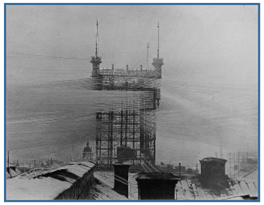
Summary

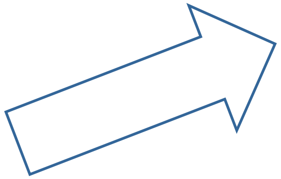
Architecture Erosion

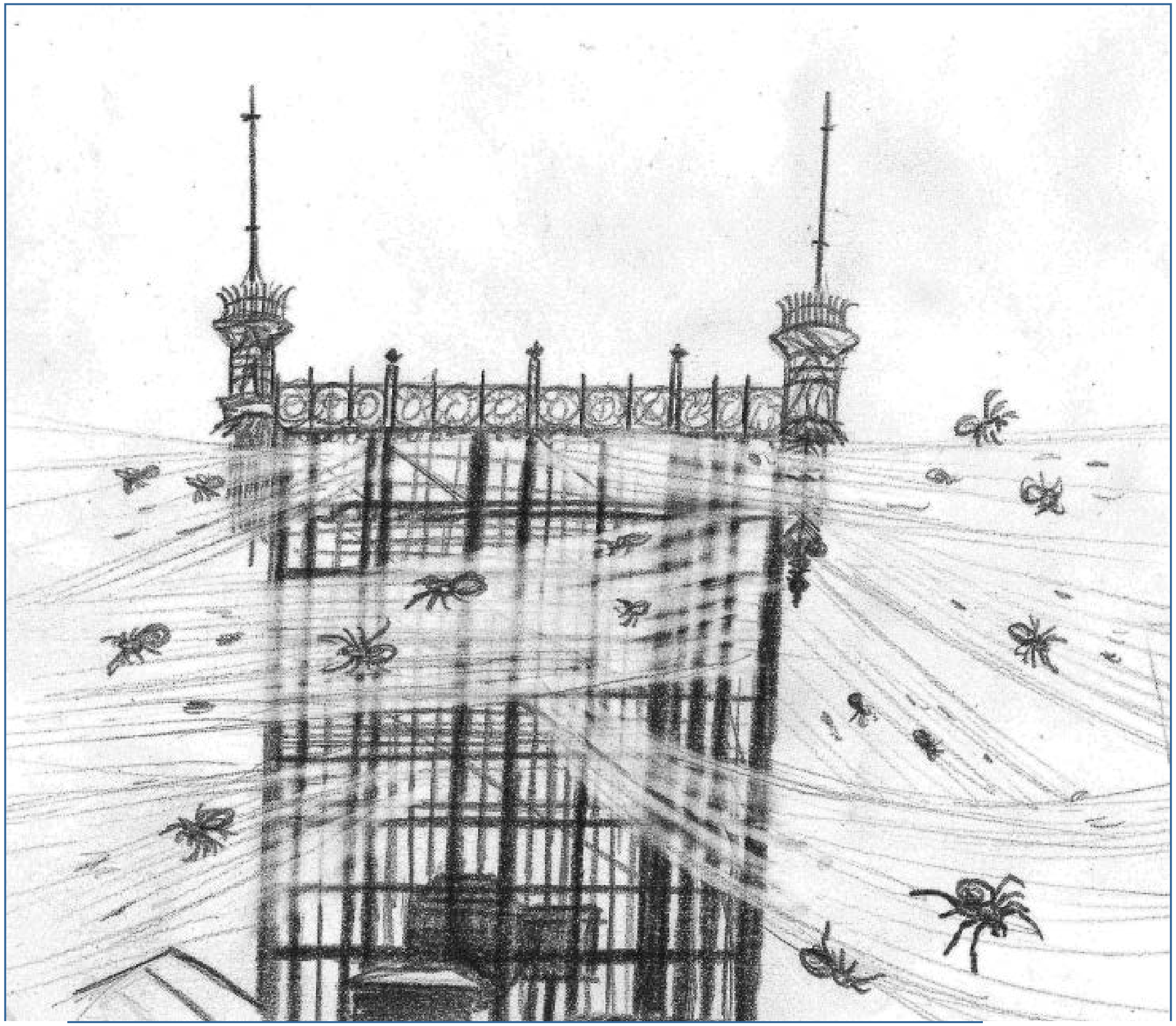
The gap observed between planned and actual architecture: **technical debt**

Implementation decisions

- diverge from the architecture-as-planned, or
- otherwise violate constraints or principles of that architecture





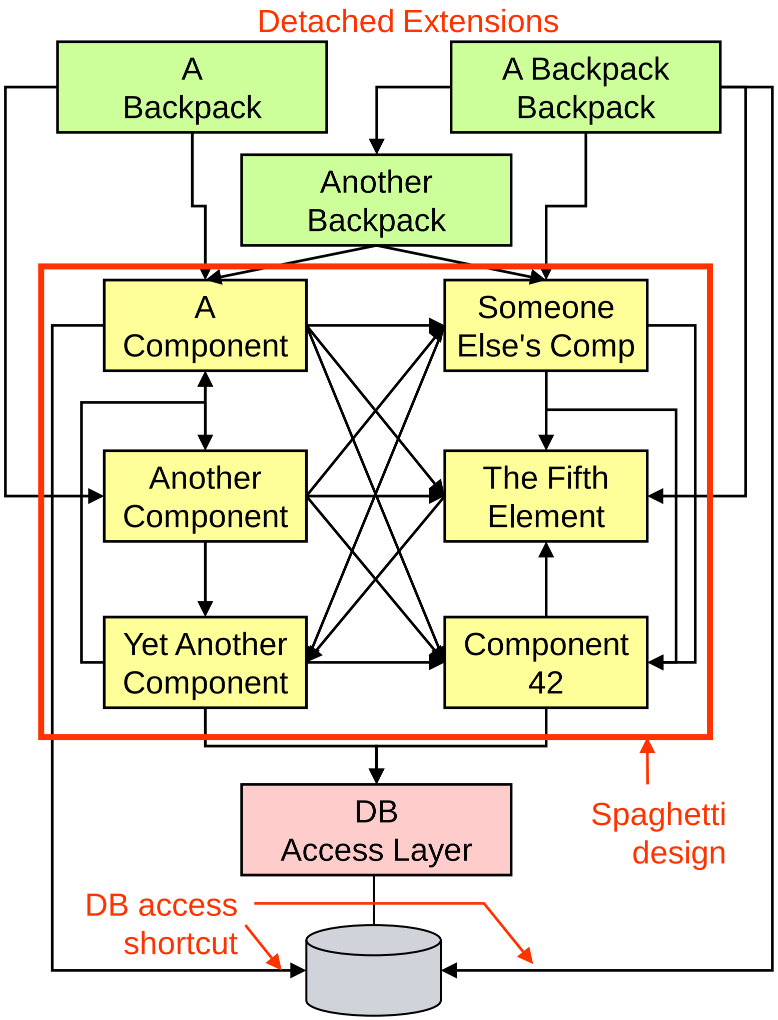


Source: http://www.thisiscolossal.com/2014/09/telefontornet-stockholm/

Source: imgur.com

Design erosion is the root of all evil

- In the lifecycle of a software system **changes are the rule and not the exception**
  - New requirements or increments imply modifications or extensions
  - Engineers must adapt their solutions to new technologies
  - Changes in business force changes in IT
  - Bug fixes require patches or local corrections
- **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** as workarounds have a **negative impact on operational and developmental properties**



Erosion always happens

Some reasons for erosion

- Prototypes become products
- Hacks / workarounds / shortcuts
- Lack of understanding “architecture-as-planned”
- Time pressure
- …

Some types of SW erosion

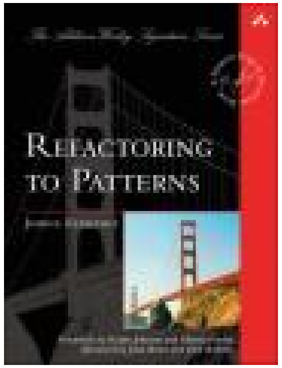
- Architectural rule violations
- Cyclic dependencies
- Dead code
- Code clones
- Metric outliers
- …

IF IT CAN'T BE FIXED WITH DUCT TAPE
THEN YOU'RE NOT USING ENOUGH DUCT TAPE
?

Code smells – Code refactoring

- Kent Beck's grandmother's saying:
  If it stinks, change it
- Thus, identify **bad smells** such as:
  - Code that is duplicated
  - Methods that span several dozen lines
  - Subclasses introducing the same method
  - Usage of temporary variables
  - Usage of switch statements

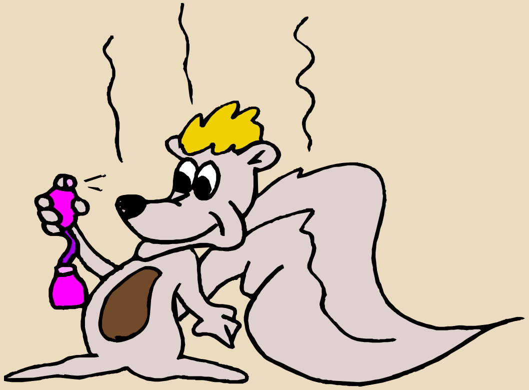
➔ ***“Refactoring to Patterns”***



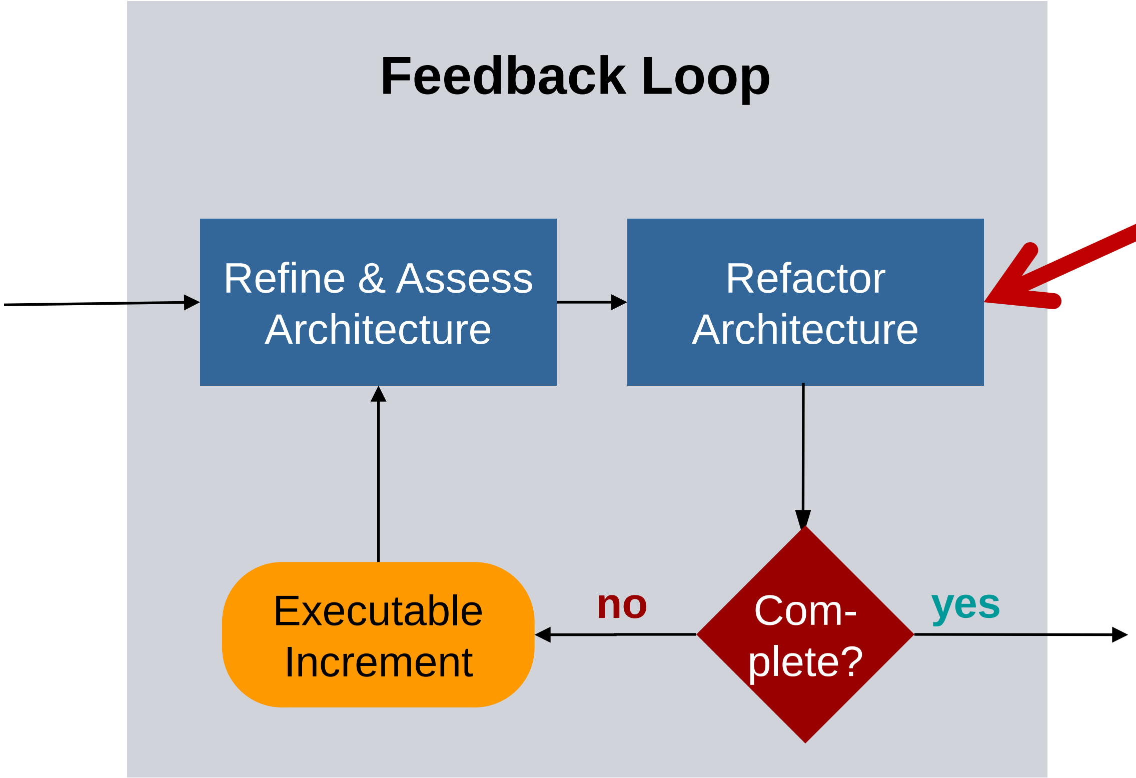
Architecture refactoring – "Smells"

Architecture smells

- Duplicate design artifacts
- Unclear roles of entities
- Inexpressive / complex architecture
- Everything centralized
- Home-grown solutions
- Over-generic design
- Asymmetric structure or behavior
- Dependency cycles
- Design violations (such as relaxed instead of strict layering)
- Inadequate partitioning of functionality
- Unnecessary dependencies
- ...



Refactoring is part of the architecture design process



Refactoring is integrated into the iterative-incremental architecture design process:

- **It improves the structure**
- **It supports a risk-, requirements- and test-driven approach**

Refactoring

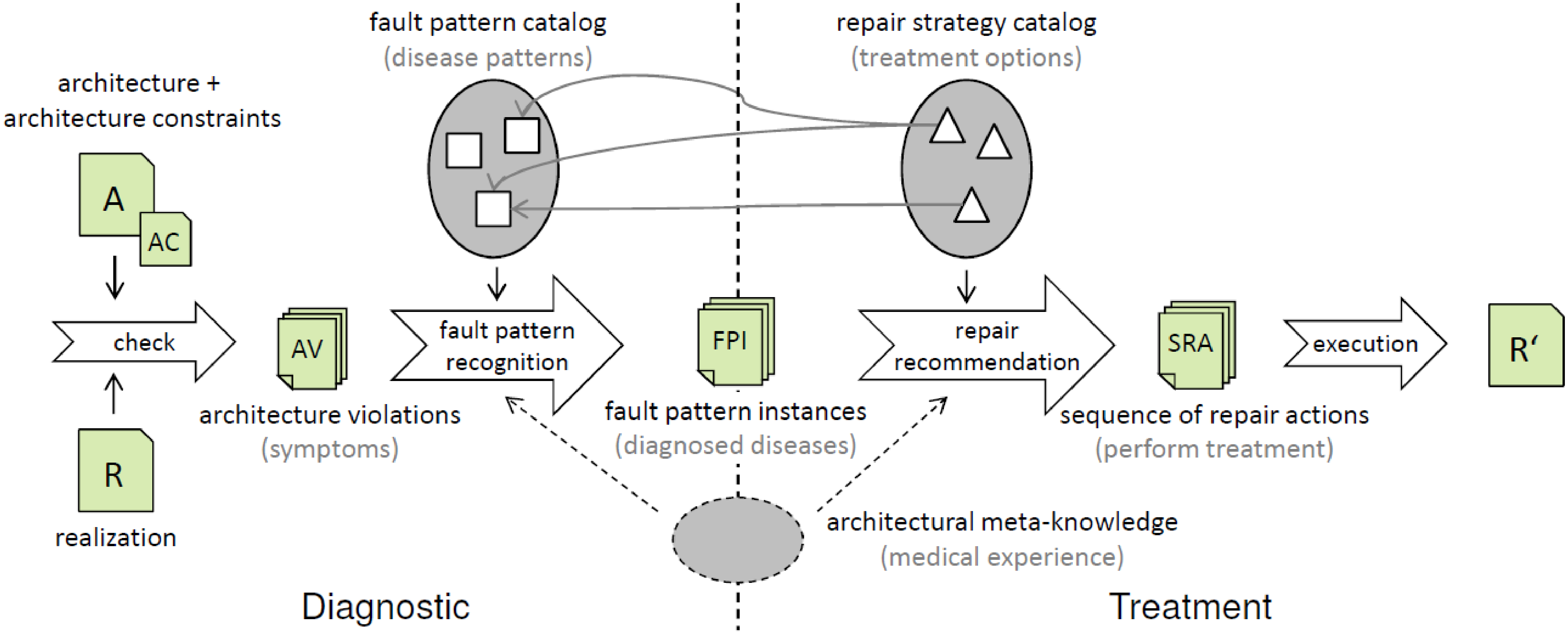
Agenda

Design Erosion & Smells

Refactoring / Reengineering / Rewriting

Summary

Architecture Erosion: Treating the Patient



Source: Mair et. al. *Towards Flexible Automated Software Architecture Erosion Diagnosis and Treatment*

Refactoring – what is it?

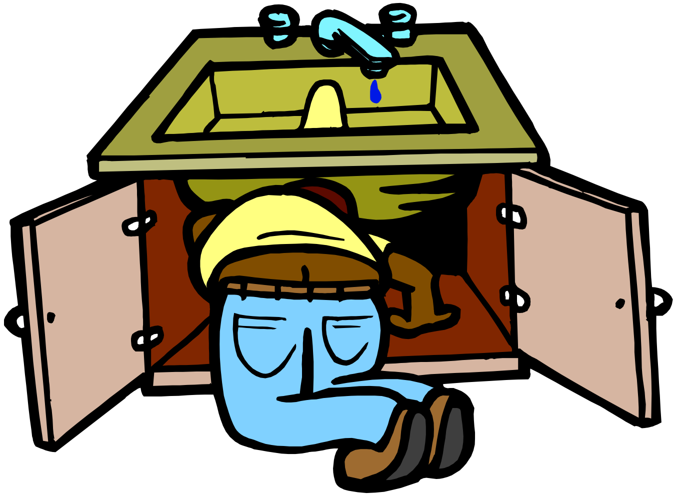
- "Code refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [Martin Fowler]

- Put more generally: **Refactoring** is the process of changing a software system or process in such a way that it…

  does not alter the external behavior, yet

  improves its internal structure

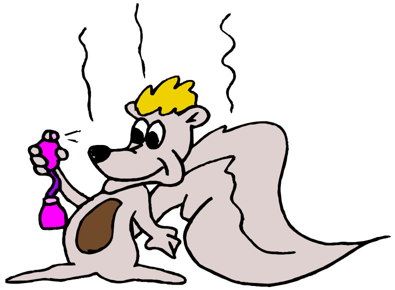
Note: External interfaces remain unchanged!



Architecture refactoring - Definition

- Architecture refactoring is about the **semantic-preserving transformation** of a software **design**
- It **changes structure** but not behavior
- It applies to **architecture-relevant** design artifacts such as UML diagrams, models, DSL expressions, aspects
- Its goal is to **improve architecture and design quality**.
- You got an "architectural smell"? Use an architecture refactoring pattern to solve it!

Note:
A smell is only an ***indicator*** of a possible problem, not a proof

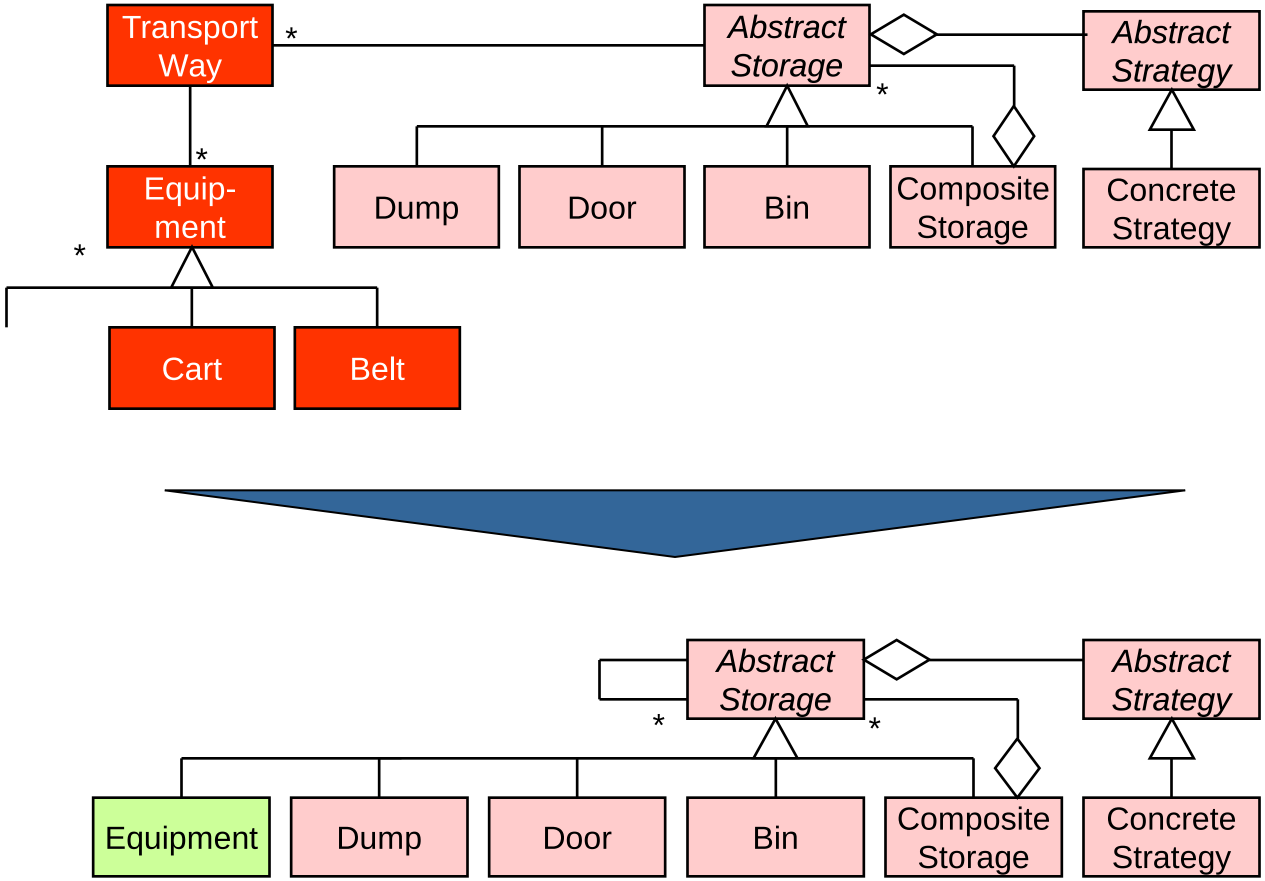


SIEMENS

Ingenuity for life

Example: Remove unnecessary abstractions (1)

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.

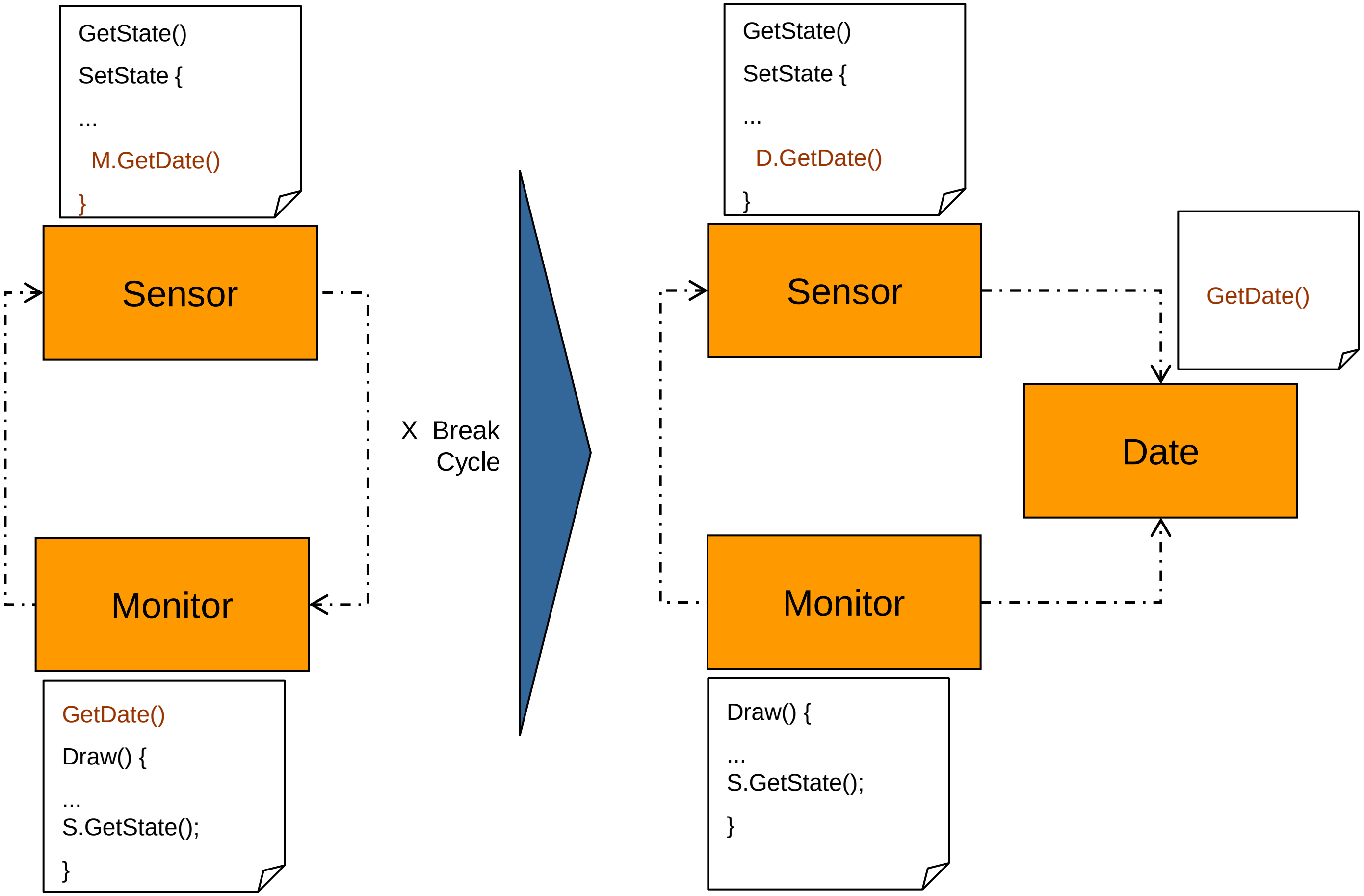


Example: Remove unnecessary abstractions (2)

- 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 it is better to 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 dependent from other existing abstractions
- Caveat
  - Don't generalize too much (such as introducing one single hierarchy level: "All classes are directly derived from Object")

Example – Break dependency cycles (1)

In this example, the monitor invokes the state getter / setter methods but also provides GetDate() to the sensor, lea-ding to a simple dependency cycle. Providing this method to monitors was a bad design decision, anyway. Introducing a separate date object solves the problem.

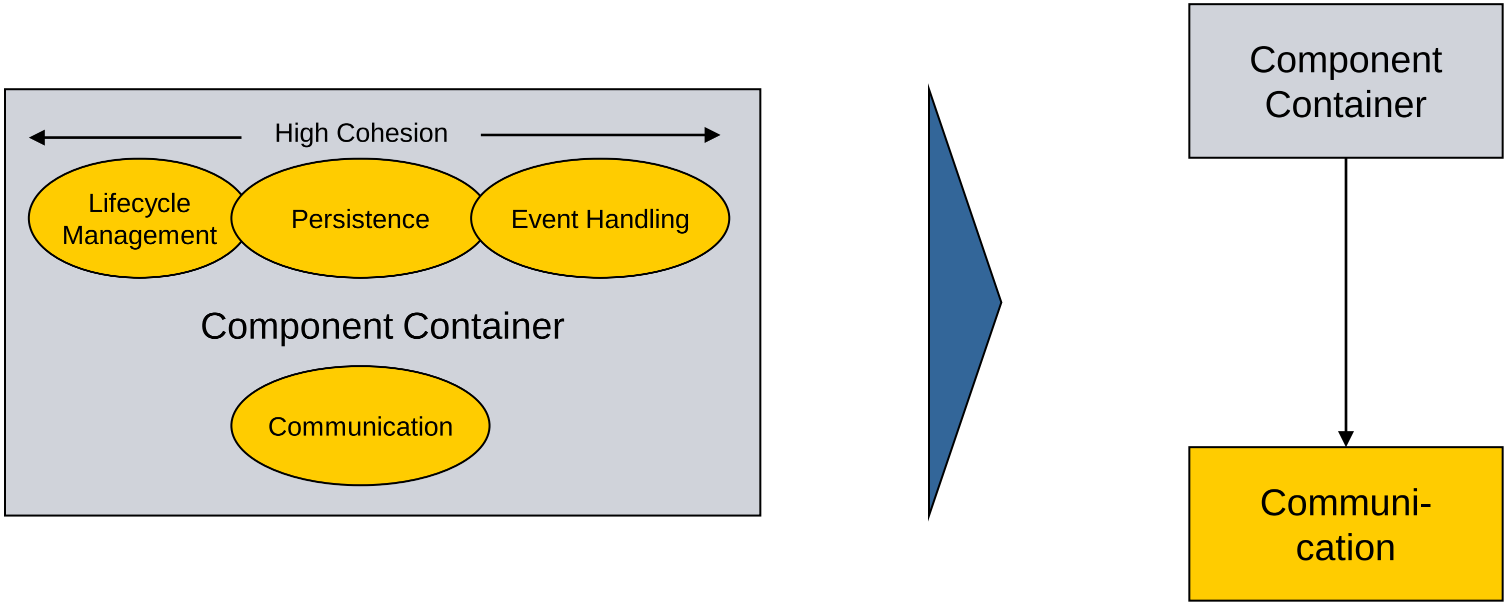


Example – Break dependency cycles (2)

- Context
  - Dependencies between subsystems
- Problem
  - Your 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
  - Thus, dependency hierarchies should form DAGs (directed acyclic graphs)
- General solution idea
  - Get rid of the dependency cycle by removing one of the dependencies

Split subsystems (1)

Example: When analyzing interdependencies between entities in a middleware subsystem, two (or more) sets of components could be determined. Within each of these sets there was high cohesion; between these sets only low cohesion. Thus, the subsystem was split into two parts.



Special variant: Split layer in a layered system

Split subsystems (2)

- Context
  - Cohesion within a subsystem
- Problem
  - Within a subsystem the interdependencies (cohesion) should be high
  - Between two subsystems in a software architecture, the degree of coupling should be rather loose
  - If the cohesion between some parts is loose, then some design decisions seem to be questionable
  - It is recommendable to change this to obtain better modularization and understandability
  - Another potential problem is subsystems/components with too many responsibilities
- General solution idea
  - Loose cohesion within a subsystem implies that the functionality can be split into multiple subsystems
  - Thus, determine areas with high cohesion in a subsystem. All those areas with low cohesion are candidates for becoming subsystems of their own

Checking correctness

Available options to check the compliance with the initial architecture:

- **Formal approach**: Prove **semantics and correctness** of program transformation
- **Implementation approach**: Leverage **unit and regression tests** (follow test-driven methods)
- **Architecture analysis:** Use an **architecture or design review**

Use at least the **latter two methods** to ensure quality (if implementation is already available).

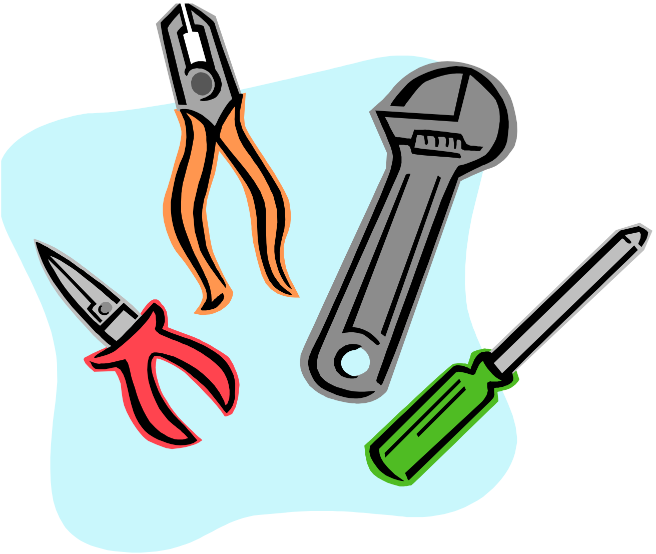
SIEMENS

Ingenuity for life

Application of architecture refactorings

Architecture refactoring shall be done in a **systematic and controlled** way.

It is the architect's responsibility to



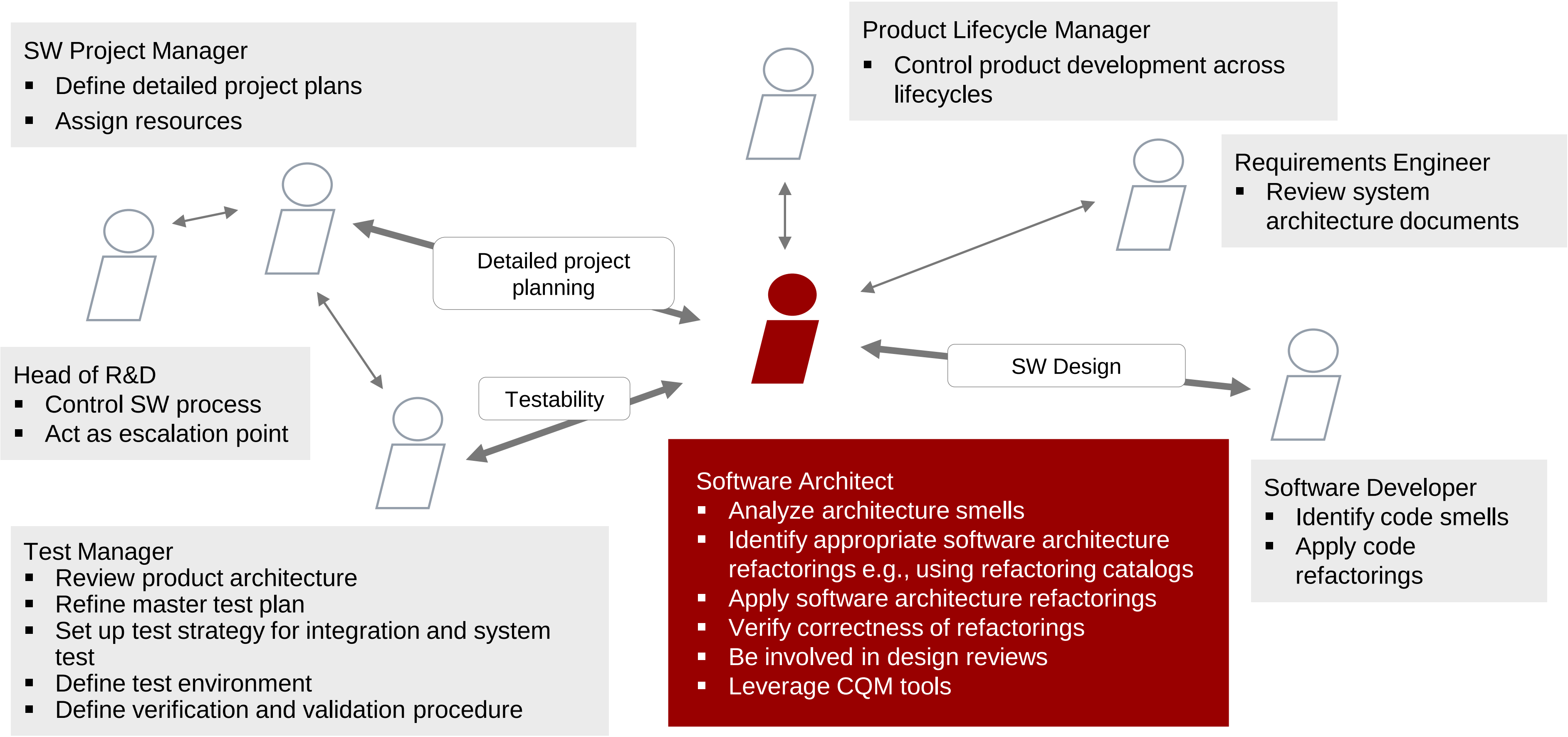
- Check the **applicability of refactorings** e.g.
  - impact to requirements
  - proper scope
  - proper solution
- Define the **order** of refactoring
  - Strategic before tactical aspects
  - Priorities of qualities
- **Apply** the refactorings
- **Ensure the quality** after the refactoring (in conjunction with the test manager).

Obstacles to refactoring

- **Organization / management**
  - Providing new features is considered more important
  - “Organization drives architecture” problem
- **Process support**
  - No steps / activities / responsibilities defined
  - Results are not checked for correctness
  - Test manager not involved
- **Technologies and tools**
  - Only manual refactoring (tools unavailable)
  - Refactoring not documented
- **Applicability**
  - Refactoring used instead of reengineering and vice versa.
  - Wrong order

Refactoring – Responsibilities and communication

The process of refactoring requires communication with testers and developers



SW Project Manager

- Define detailed project plans
- Assign resources

Product Lifecycle Manager

- Control product development across lifecycles

Requirements Engineer

- Review system architecture documents

Head of R&D

- Control SW process
- Act as escalation point

Test Manager

- Review product architecture
- Refine master test plan
- Set up test strategy for integration and system test
- Define test environment
- Define verification and validation procedure

Software Architect

- Analyze architecture smells
- Identify appropriate software architecture refactorings e.g., using refactoring catalogs
- Apply software architecture refactorings
- Verify correctness of refactorings
- Be involved in design reviews
- Leverage CQM tools

Software Developer

- Identify code smells
- Apply code refactorings

Where to obtain architecture refactorings?

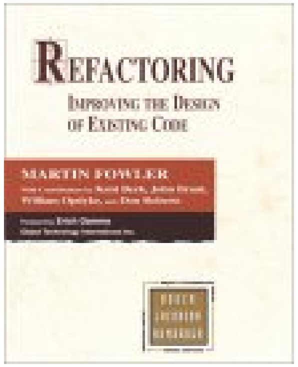
A whole catalog of architecture refactorings is provided as a starting point in your course folder

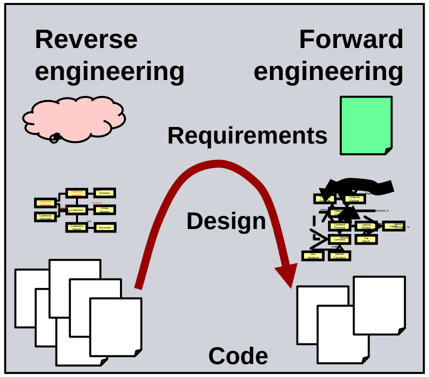
1. Rename Entities
2. Remove Duplicates
3. Introduce Abstraction Hierarchies
4. Remove Unnecessary Abstractions
5. Substitute Mediation with Adaptation
6. Break Dependency Cycles
7. Inject Dependencies
8. Insert Transparency Layer
9. Reduce Dependencies with Facades
10. Merge Subsystems
11. Split Subsystems
12. Enforce Strict Layering
13. Move Entities
14. Add Strategies
15. Enforce Symmetry
16. Extract Interface
17. Enforce Contract
18. Provide Extension Interfaces
19. Substitute Inheritance with Delegation
20. Provide Interoperability Layers
21. Aspectify
22. Integrate DSLs
23. Add Uniform Support to Runtime Aspects
24. Add Configuration Subsystem
25. Introduce the Open/Close Principle
26. Optimize with Caching
27. Replace Singleton
28. Separate Synchronous and Asynchronous Processing
29. Replace Remote Methods with Messages
30. Add Object Manager
31. Change Unidirectional Association to Bidirectional

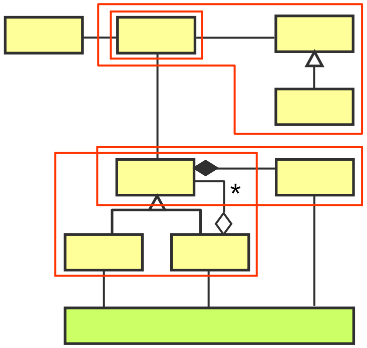
Refactoring, reengineering, and rewriting comparison (1)

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
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER





Refactoring, reengineering, and rewriting comparison (2)

SIEMENS

Ingenuity for life

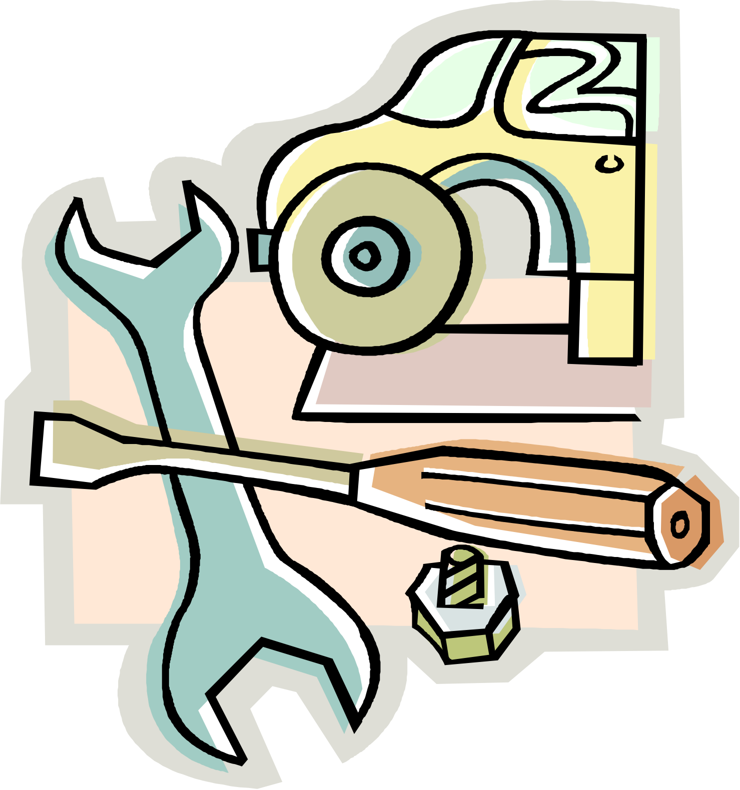
	Refactoring	Reengineering	Rewriting
Scope	▪ Many local effects	▪ Systemic effect	▪ Systemic or local effect
Process	▪ Structure transforming ▪ Behavior / semantics preserving	▪ Disassembly / reassembly	▪ Replacement
Results	▪ Improved structure ▪ Identical behavior	▪ New system	▪ New system or new component
Improved qualities	▪ Developmental ▪ Operational	▪ Functional ▪ Operational ▪ Developmental	▪ Functional ▪ Operational ▪ Developmental
Drivers	▪ Complicated design / code evolution ▪ When fixing bugs ▪ When design and code smell bad	▪ Refactoring is insufficient ▪ Bug fixes cause rippling effect ▪ New functional and operational requirements ▪ Changed business case	▪ Refactoring and reengineering are insufficient or inappropriate ▪ Unstable code and design ▪ New functional and operational requirements ▪ Changed business case
When	▪ Part of daily work ▪ At the end of each iteration ▪ Dedicated refactoring iterations in response to reviews ▪ It is the 3rd step of TDD	▪ Requires a dedicated project	▪ Requires dedicated effort or a dedicated project, depending on scope

Backup

Backup

Reengineering – How it differs from refactoring

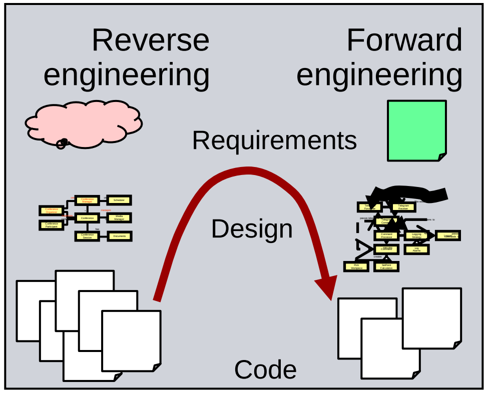
- **Scope:** Reengineering always affects the entire system; refactoring has typically (many) local effects
- **Process:** Reengineering follows a disassembly / reassembly approach; refactoring is a behavior-preserving, structure transforming process
- **Result:** Reengineering can create a whole new system – with different structure, behavior, and functionality; refactoring improves the structure of an existing system – leaving its behavior and functionality unchanged



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

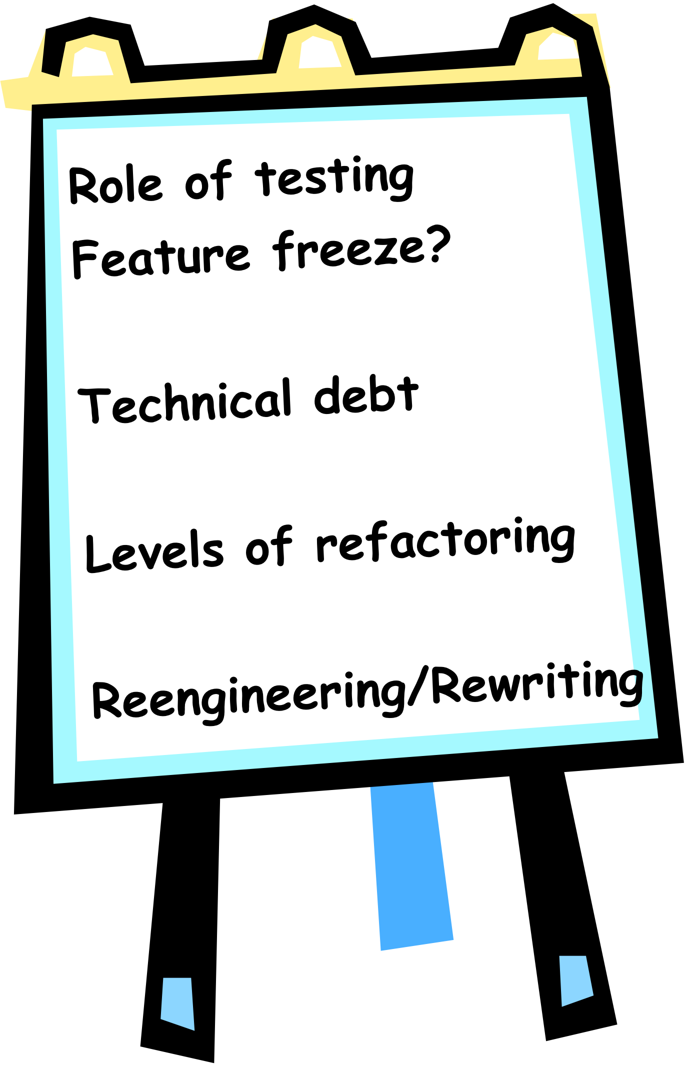
Agenda

Design Erosion & Smells

Refactoring / Reengineering / Rewriting

Summary

Refactoring aspects



Role of testing

Feature freeze?

Technical debt

Levels of refactoring

Reengineering/Rewriting

What we learned

Refactoring

- **Changes artifacts** without changing external behavior

Reengineering

- Complete redesign / restructuring, typically **changes external behavior**

Rewriting

- **Rewrite** the complete architecture

Checking correctness

- Use **testing** and architecture **inspections**

Software architect's responsibilities

- Detect architecture **smells**
- Find and apply appropriate **refactoring, reengineering, rewriting**
- Perform **quality assurance** of refactoring activities

Departing thought

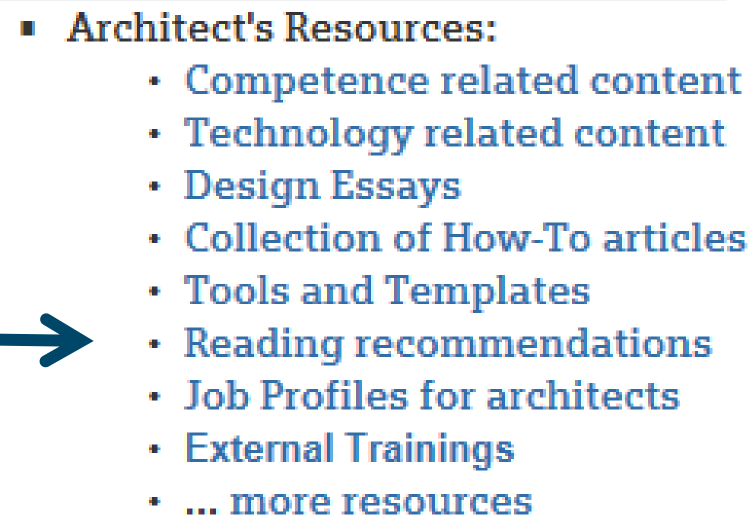
When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.

[Martin Fowler, *Refactoring: Improving the Design of Existing Code*, p. 88]

Further readings

Use the SSA Wiki :
https://wiki.ct.siemens.de/x/fReTBQ

and check the “Reading recommendations”:
https://wiki.ct.siemens.de/x/-pRgBg



- Architect's Resources:
  - Competence related content
  - Technology related content
  - Design Essays
  - Collection of How-To articles
  - Tools and Templates
  - Reading recommendations
  - Job Profiles for architects
  - External Trainings
  - ... more resources