Anomalies and refactoring in software architectures

Prof. Dr. Everton Cavalcante

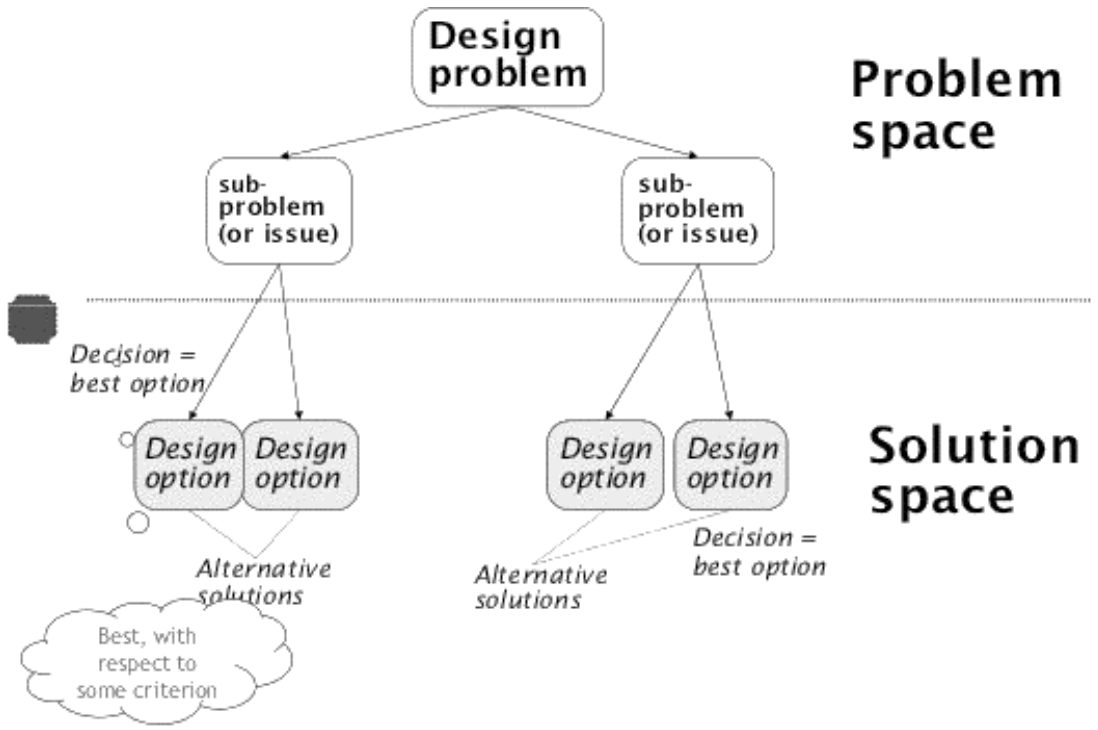
http://www.dimap.ufrn.br/~everton/

Software Architecture as an engineering discipline

The issue is on how to organize a system to simultaneously

- make the suitable decisions
- provide the required functionalities (functional requirements)
- guarantee the required quality of service (non-functional requirements)

Software Architecture as an engineering discipline



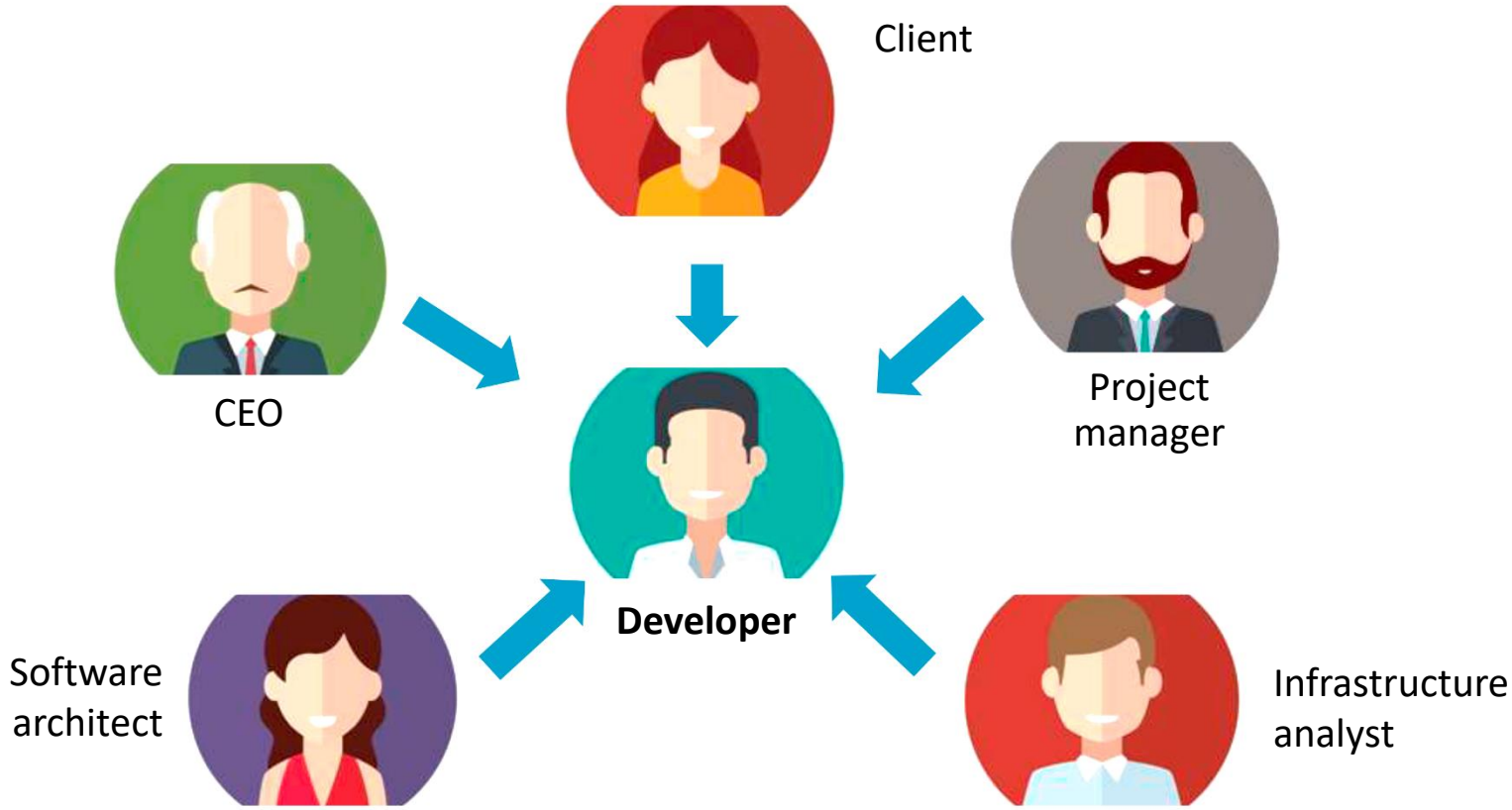
Architecting is largely about making decisions

- If the decisions are not the right ones, they potentially will impact how the system operates and/or result in failures of many types
- If one starts with the wrong architecture, the software is necessarily going to be unsuccessful and failed

A system could be successful and yet poorly architected

The life in a software development project

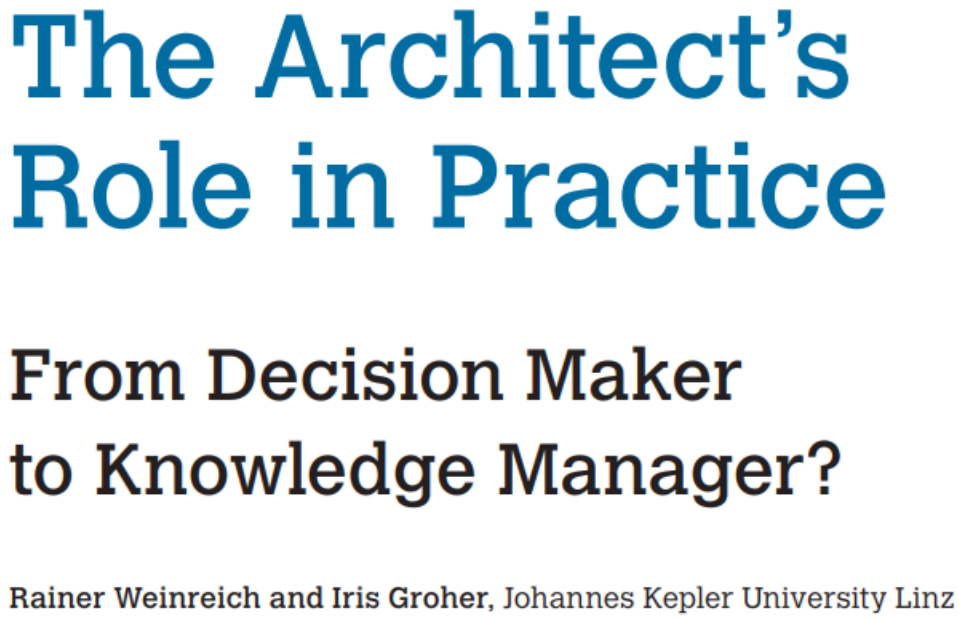
Does software quality really depend only on developers?




SOFTWARE ARCHITECT

The life in a software development project

Despite software architects' central role as designers, it is well known that their job is much more than making technical decisions during a system's initial design. Architects are involved in not only the early development phases but also system implementation and evolution, where they act as evaluators, extenders, and sustainers. […] Software architects are transforming from primary decision makers to advisors, coordinators, and knowledge managers.


The Architect's Role in Practice

From Decision Maker to Knowledge Manager?

Rainer Weinreich and Iris Groher, Johannes Kepler University Linz

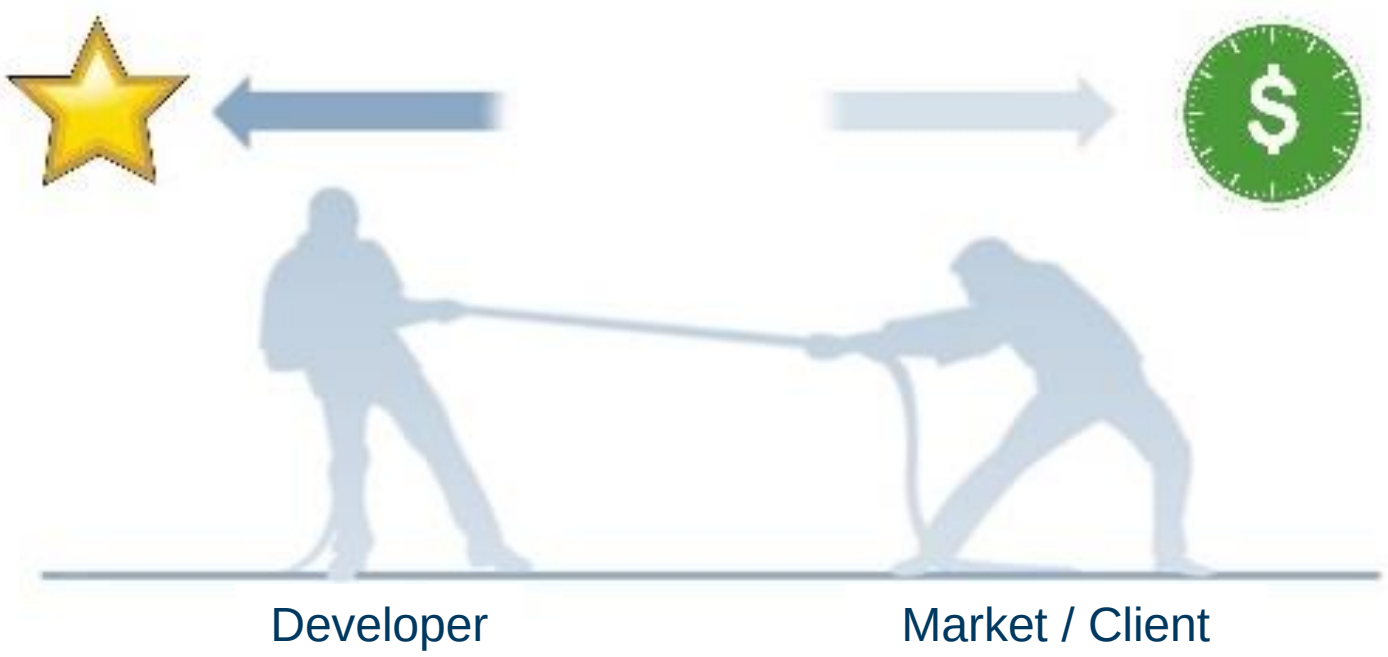
Rainer Weinreich, Iris Groher. **The architect's role in practice: From decision maker to knowledge manager?** IEEE Software, vol. 33, no. 6, November/December 2016, pp. 63-69

The life in a software development project

Things do not always go as expected

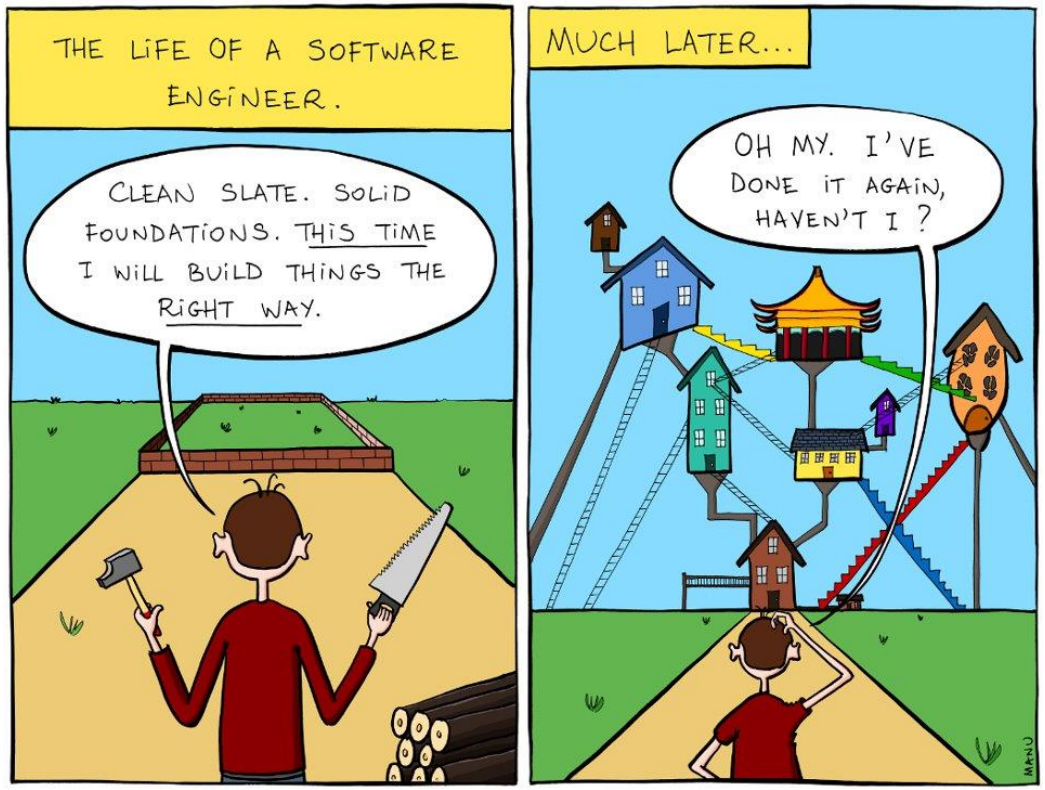
The life in a software development project

There is a constant pressure to deliver software with quality in the shortest time while seeking to reduce costs



The life in a software development project

...but it is hard to meet all of these requirements in practice



The life in a software development project

...and sometimes the development team is forced to take shortcuts

https://goo.gl/GtvK65

http://sou.gohorseprocess.com.br/

Goals

- Define the concept of technical debt and explain why it happens
- To present common problems (anomalies) that affect both source code and software architectures
- To briefly introduce refactoring process and main techniques

Technical debt

Metaphor coined by Ward Cunningham to represent

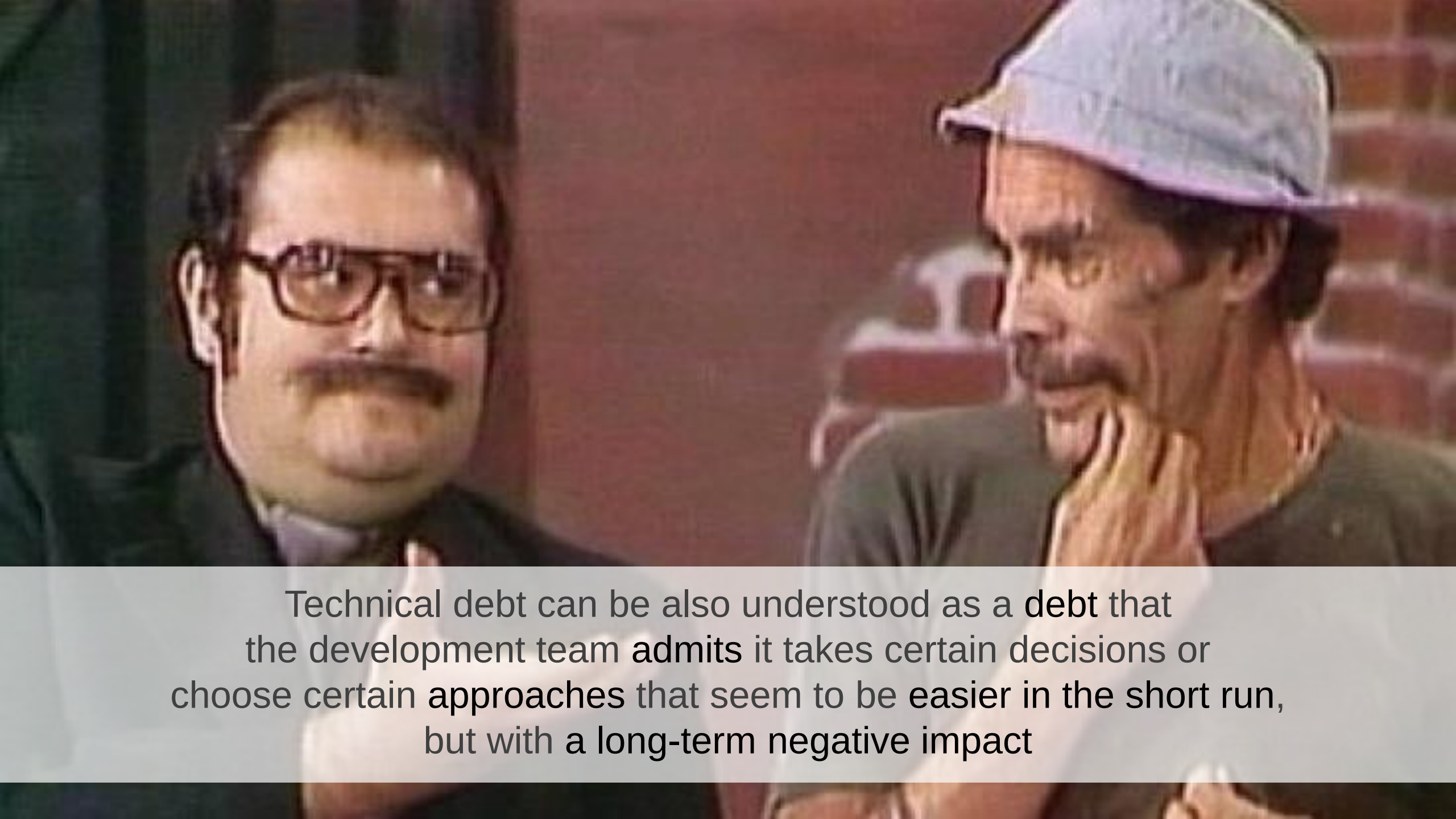
– incomplete, immature or inadequate artifacts in the software development process

– artifacts acknowledged as bad, but that are left as-is to possibly fixed later

– artifacts that may imply a future risk to software quality

Ward Cunningham. **The WyCash portfolio management system.** Addendum to the Proceedings on 1992 Object-Oriented Programming Systems, Languages, and Applications Conference (OOPSLA'92), Vancouver, BC, Canada. USA: ACM, 1992, pp. 29-30


https://www.youtube.com/watch?v=pqeJFYwnkjE



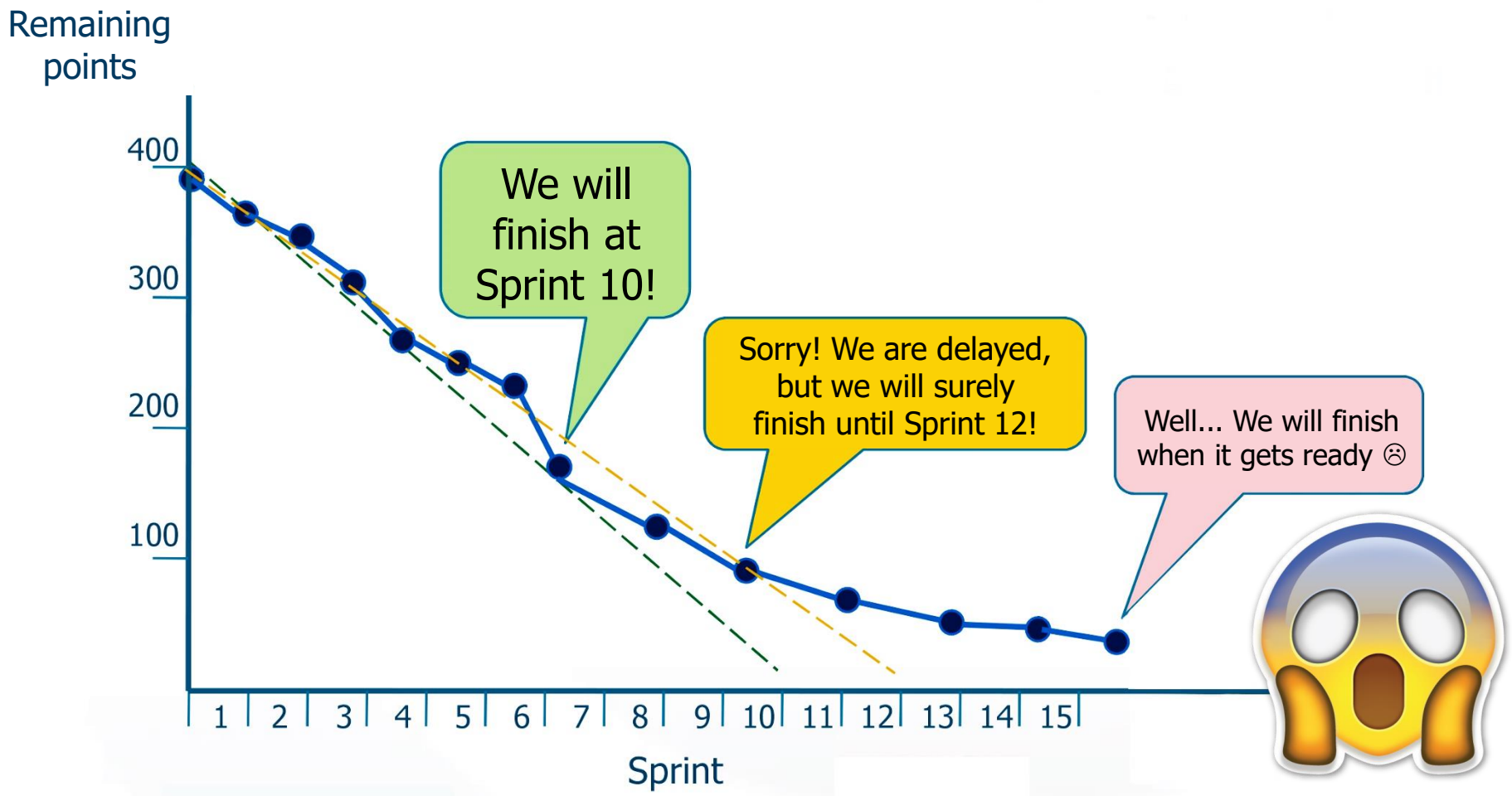
Technical debt can be also understood as a debt that
the development team admits it takes certain decisions or
choose certain approaches that seem to be easier in the short run,
but with a long-term negative impact

Characteristics of technical debt

Technical debt is often associated with

- rework or extra work by the development team
- increase in human and material costs
- delays in software delivery
- reduction of software maintainability
- reduction in productivity of the development team
- negative impact on software quality
- loss of time

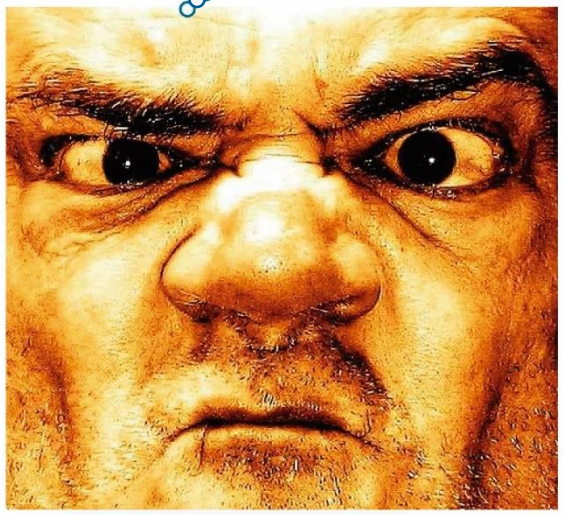
Evidence of technical debt



Evidence of technical debt

Client's feeling

Functionalities got ready quite fast!

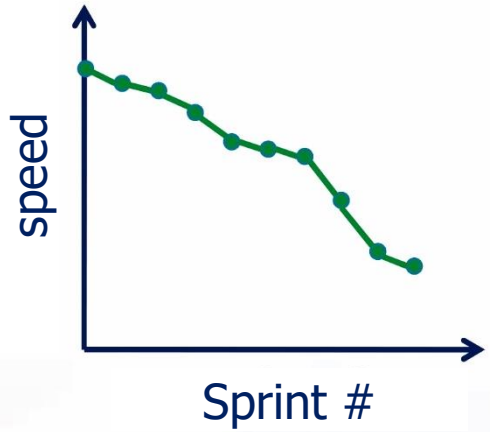


Developer's feeling

This code is a shame!

Metrics

- Duplication
- Test coverage
- Size of methods and classes
- Speed



Evidence of technical debt

- *FIXME*
- *TODO*
- *We'll fix this later*
- *Don't worry with documentation now*
- *Things must be ready for yesterday*
- *Never mind*
- *No problem*
- *What is the problem with global variables?*
- *We need to deliver it working, so let's make it work*
- *It works, so it is OK. Good!*
- *And? This code is already bad anyway...*

Causes of technical debt

Technical debt is often caused by

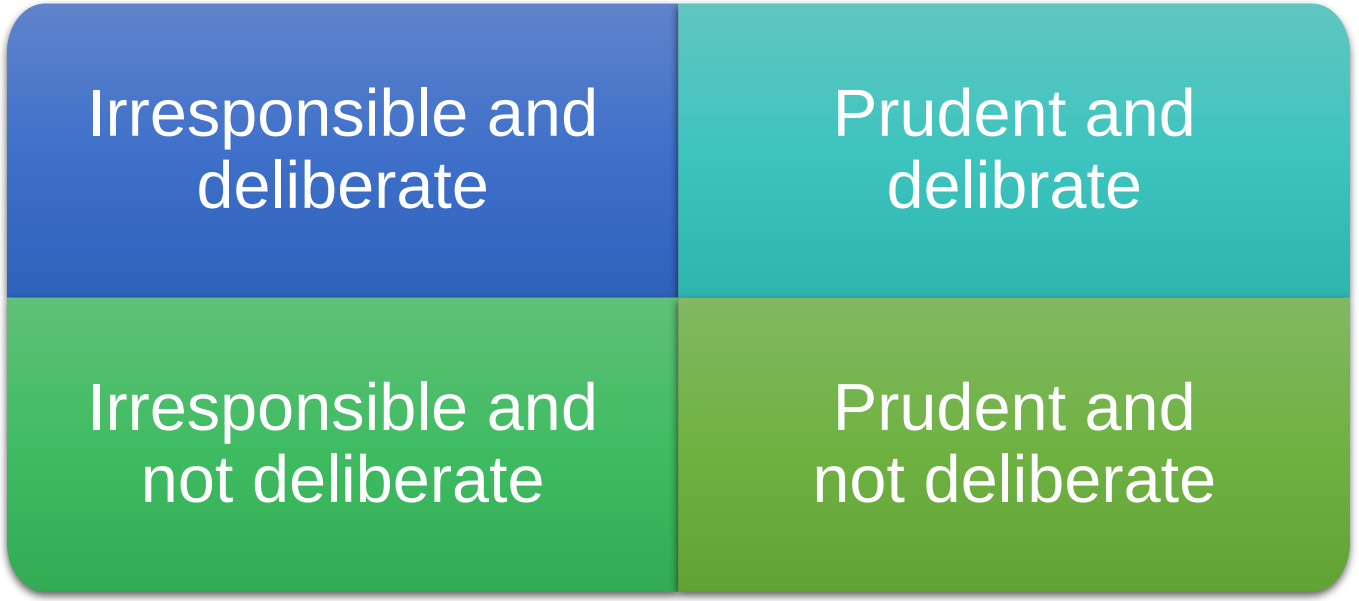
– ambiguous, poorly understood and/or unstable requirements
– adoption of bad design and architectural decisions
– adoption of bad programming practices
– negligence in testing
– insufficient, incomplete or outdated documentation
– lack of expertise in the development team
– pressure for delivery

Types of tecnical debt

The Technical Debt Quadrant (Martin Fowler)

https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

We don't have time to make this right, so let's only make it work anyway



We'll deliver it now, but we're aware of limitations and consequences

How to design a software architecture?

It works, but now we know how we should make this

Types of tecnical debt

Technical debt can appear in

Source code	Testing	Documentation	Defects
• Code anomalies • Bad programming practices • Implementation “shortcuts”	• Tests not performed • Low coverage due to deficiencies in test cases • Outdated test cases	• Absent documentation • Inadequate or incomplete documentation • Outdated documentation	• Defects identified, but not fixed

How to deal with technical debt?

- Option #1: ignore it and let the system as-is until being discontinuated because the client is not satisfied
- Option #2: remake the system from the scratch, but now right
  - Expensive, risky process, with no guarantees that the same problems will not occur again
- Option #3: fix problems on time
- Option #4: refactor in a continuous, incremental way
  - Make modifications in a systematic, incremental way during the development process

How to deal with technical debt?

- Stop accumulating it, no matter how bad it is
- Be honest and acknowledge it
- Solve the problem, not postponing its correction
  - “Repay the debt instead of continuing to pay interests”
- Make use of tools, if and when feasible
  - Do not always automate things
  - Some types of technical debt can only be detected and understood by human beings
- Correct not only the product, but also the process

EXCUSE ME, BUT... YOUR CODE SMELLS

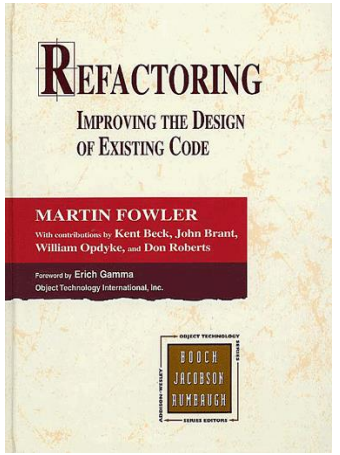
A definition

Code smell (or bad smell)

is a surface indication observed in the implementation that usually corresponds to a deeper (maintenance) problem in the system

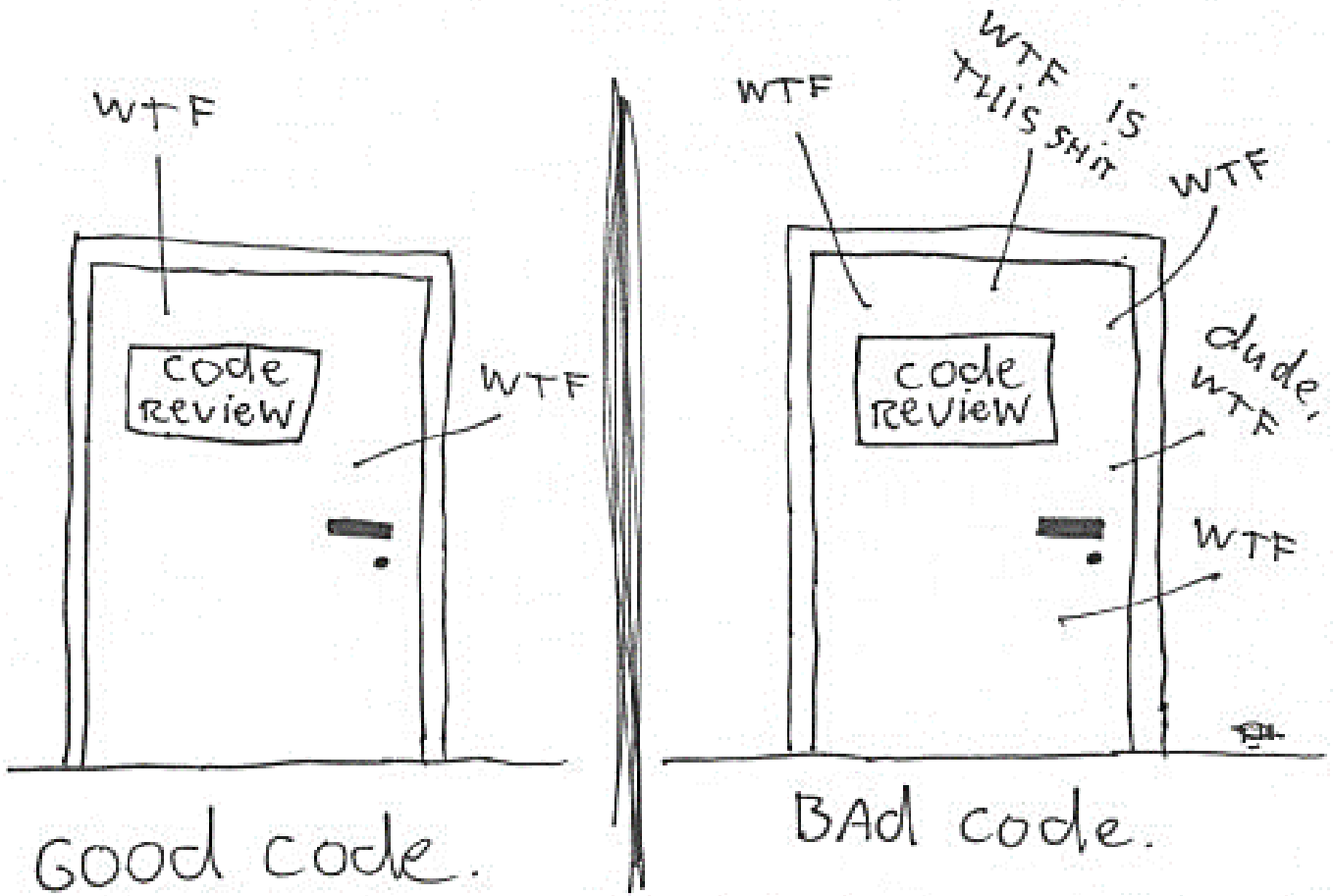
– It is an apparent symptom of a problem at source code that may indicate a deeper problem that may generate future risks

– It can be understood as a type of violation of principles and/or good programming practices



Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts.
Refactoring: Improving the design of existing code. USA: Addison-Wesley Longman, Inc., 1999

The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE



Code smell

– A code smell generally is not a defect as it does not prevent the software from working

– Code smells can hamper development or increase the risk of errors in future

Causes of code smells

Code smells are often caused by

– maintenance and/or evolution activities over the code

– refactoring activities

– stressful conditions to developers when submitted to a high workload

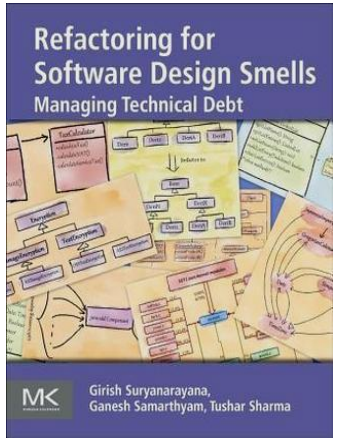
– pressure for delivering artifacts

– lack of expertise of developers

Code smells

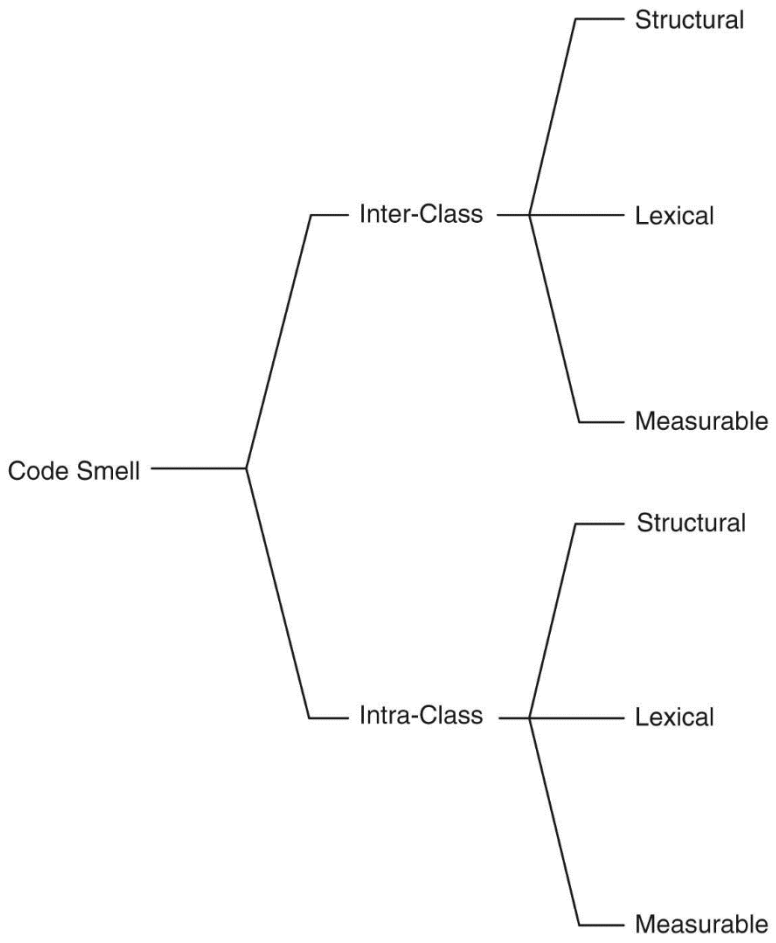
In object-oriented programming, code smells typically refer to classes and/or methods that violate fundamental principles of such a paradigm

– Clearly defined single responsibility (*Single Responsibility Principle* – SRP)
– Encapsulation
– Information hiding
– Use of few, clear interfaces
– Appropriate use of inheritance



Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma**.**
Refactoring for software design smells: Managing technical debt.
USA: Morgan-Kauffman/Elsevier, Inc., 2015

Classification of code smells



- Measurable: smell detected by one or mode code metrics on source code attributes
- Lexical: refers to the vocabulary used to express the role of a class or method, either by the class identifier or comments
- Structural: recurring pattern that is not directly observed by simple metrics of source code internal attributes

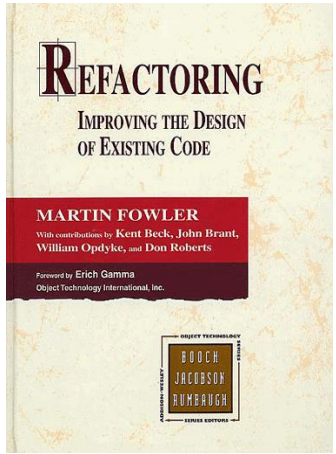
Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, Anne-Françoise Le Meur.
DECOR: A method for the specification and detection of code and design smells.
IEEE Transactions on Software Engineering, vol. 36, no. 1,
January/February 2010, pp. 20-36

Catalogue of code smells

Long Method
Large Class (God Class)
Primitive Obsession
Long Parameter List
Data Clumps
Switch Statements
Temporary Field
Refused Bequest
Alternative Classes with Different Interfaces

Divergent Change
Shotgun Surgery
Parallel Inheritance Hierarchies
Incomplete Library Class
Duplicated Code
Lazy Class
Data Class
Speculative Generality

Dead Code
Feature Envy
Inappropriate Intimacy
Message Chains
Middle Man
Comments



Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts.
Refactoring: Improving the design of existing code. USA: Addison-Wesley Longman, Inc., 1999

Classification of code smells

https://sourcemaking.com/refactoring

Bloaters: code, methods or classes that grow to the point of becoming difficult to work with

Examples: *Long Method*, *Large Class (God Class)*, *Long Parameter List*

Object-Orientation Abusers: incomplete or incorrect application of object-oriented programming principles

Examples: *Alternative Classes with Different Interfaces, Temporary Field*

Classification of code smells

https://sourcemaking.com/refactoring

Changes Preventers: if a change is needed somewhere in the code, a number of other changes elsewhere will also be necessary, thereby making development costly and complex

Examples: *Divergent Change, Shotgun Surgery*

Dispensables: anything unnecessary, superfluous whose absence would make the code cleaner, more efficient, and easier to understand

Examples: *Lazy Class, Comments, Dead Code*

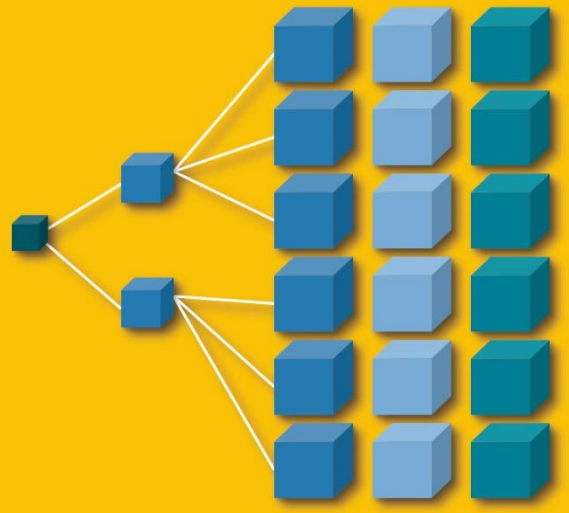
Classification of code smells

https://sourcemaking.com/refactoring

Couplers: smells that contribute to excessive coupling between classes

Examples: *Feature Envy, Innapropriate Intimacy*

How to detect code smells?



Code smells can be detected through statical code analysis, which can be manual (inspection) or automated (supported by tools)

Bad code patterns are identified. The existence of one case does not necessarily means that there are code smells. However, a number of occurrences may indicate that the code indeed has problems.

Results of detection are aggregatd. Metrics are calculated and their measures are used as indicators to determine how much and which type of risky behavior has been detected in the code.

Tools to detect code smells

- JDeodorant (Java)
  https://users.encs.concordia.ca/~nikolaos/jdeodorant/
- inFusion (C/C++, Java)
  http://www.intooitus.com/products/infusion/
- Stench Blossom (Java)
  https://goo.gl/98jj6f
- PMD (Java)
  https://pmd.github.io

Tools to detect code smells

- iPlasma (C++, Java)
  http://loose.upt.ro/reengineering/research/iplasma
- SpIRIT (Smalltalk, C++, Java)
  https://sites.google.com/site/santiagoavidal/projects/spirit
- SonarQube (20+ programming languages)
  https://www.sonarqube.org/

Tools to detect code smells

Evaluation of tools

An experience report on using code smells detection tools

Francesca Arcelli Fontana, Elia Mariani, Andrea Morniroli, Raul Sormani, Alberto Tonello.
An experience report on using code smells detection tools. Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11), Berlin, Germany. USA: IEEE Computer Society, 2011, pp. 450-457

On the evaluation of code smells and detection tools

Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, Cláudio Sant'Anna. **On the evaluation of code smells and detection tools**. Journal of Software Engineering Research and Development, vol. 5, no. 7, October 2017

A Review-based Comparative Study of Bad Smell Detection Tools

Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, Eduardo Figueiredo. **A review-based comparative study of bad smell detection tools**. Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE'16), Limerick, Ireland. USA: ACM, 2016.

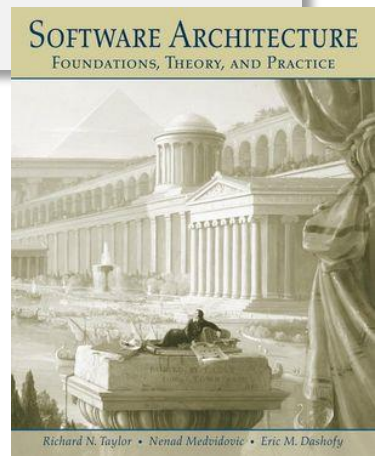
Another definition

Architectural degradation (or architectural decay)

The resulting discrepancy between a system's prescriptive [as-intended] and descriptive [as-implemented] architecture

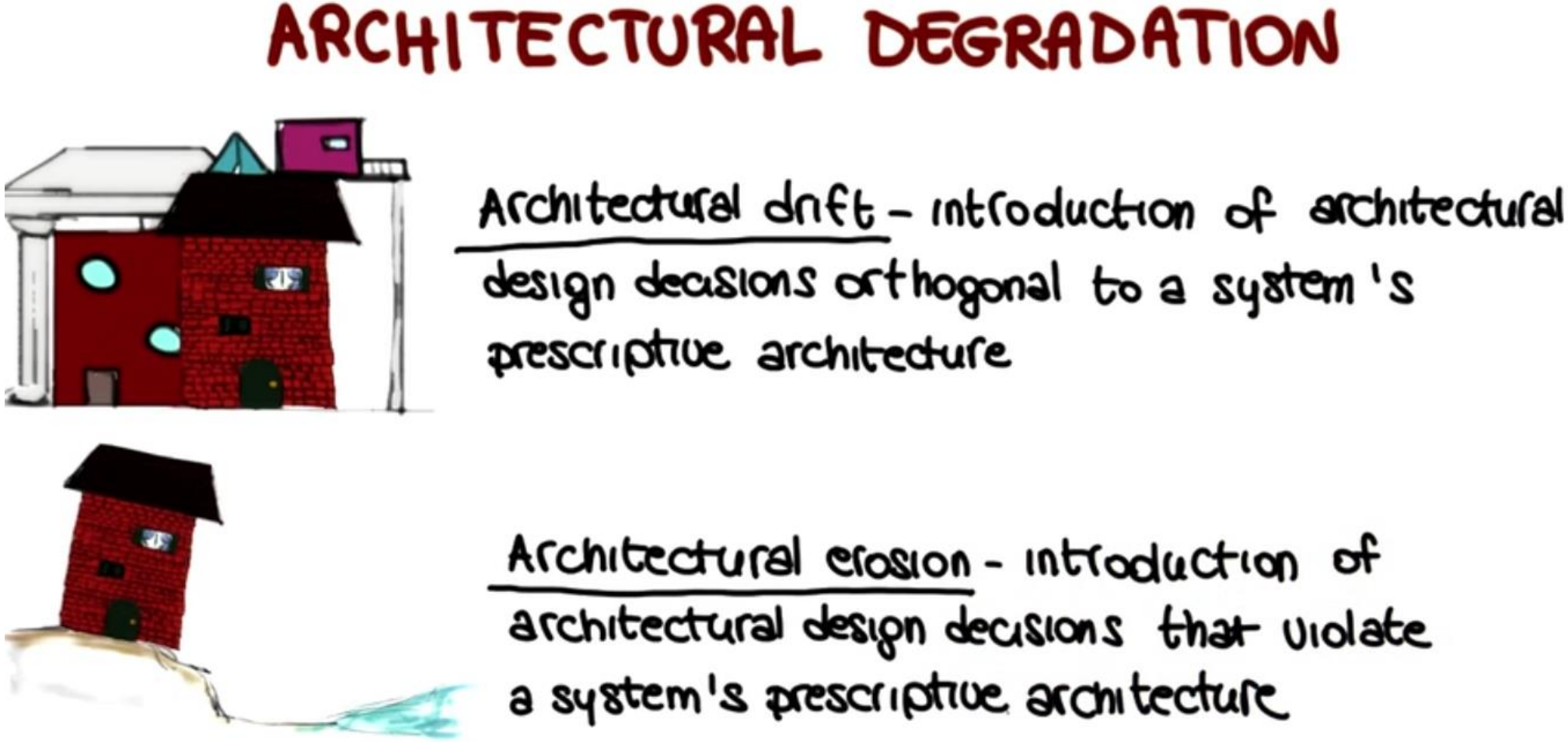
It comprises two related phenomena, namely

- architectural drift and
- architectural erosion



Richard N. Taylor, Nenad Medvidovic, Eric M. Dashofy.
Software Architecture: Foundations, theory, and practice.
USA: John Wiley & Sons, Inc., 2010

Architectural degradation

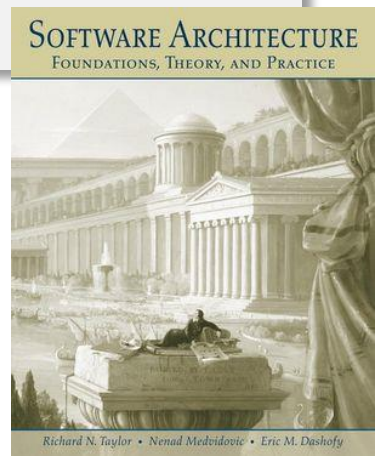


One definition more

Architectural recovery

Is the process of determining a software system's architecture from its implementation artifacts

– Process used in cases of extreme architectural degradation in which the prescriptive (or even the descriptive) architecture
  - is not recognizable anymore
  - is outdated as to be useless
  - is misleading
– Implementation artifacts can be source code, executable files, etc.



Richard N. Taylor, Nenad Medvidovic, Eric M. Dashofy.
Software Architecture: Foundations, theory, and practice.
USA: John Wiley & Sons, Inc., 2010

One definition more (!)

Architectural smell

is a commonly (although not always intentionally) used architectural decision that negatively impacts system quality

Identifying Architectural Bad Smells

Joshua Garcia, Daniel Popescu, George Edwards and Nenad Medvidovic
Computer Science Department
University of Southern California

Joshua Garcia, Daniel Popescu, George Edwards, Nenad Medvidovic. **Identifying architectural bad smells.** Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), Kaiserslautern, Germany. USA: IEEE Computer Society, 2009, pp. 255-258

Architectural smells

- As in code, design also smells and may indicate problems in a greater level of granularity (and abstraction) in the system
  - and this may impact how functional and quality requirements are achieved, mainly maintainability (among many others)
- Architectural smells are analogous to code smells because they both represent common solutions that are not necessarily faulty or errant, but still negatively impact software quality

Architectural smells

- Architectural smells may be caused by
  - applying a design solution in an inappropriate context
  - mixing combinations of design abstractions that have undesirable behaviors
  - applying design abstractions at the wrong level of granularity
- Architectural smells can be remedied by changing the internal structure of the system and/or behaviors of internal elements without changing the external behavior of the system
  - undergoing a (more complicated) refactoring process

Architectural smells

Some architectural smells may be caused by (architecturally-relevant) code smells, but not all of them

– Detecting code anomalies is useful to locate potential sources of architecture degradation

– Systematic code refactoring can contribute to address symptoms of architecture degradation

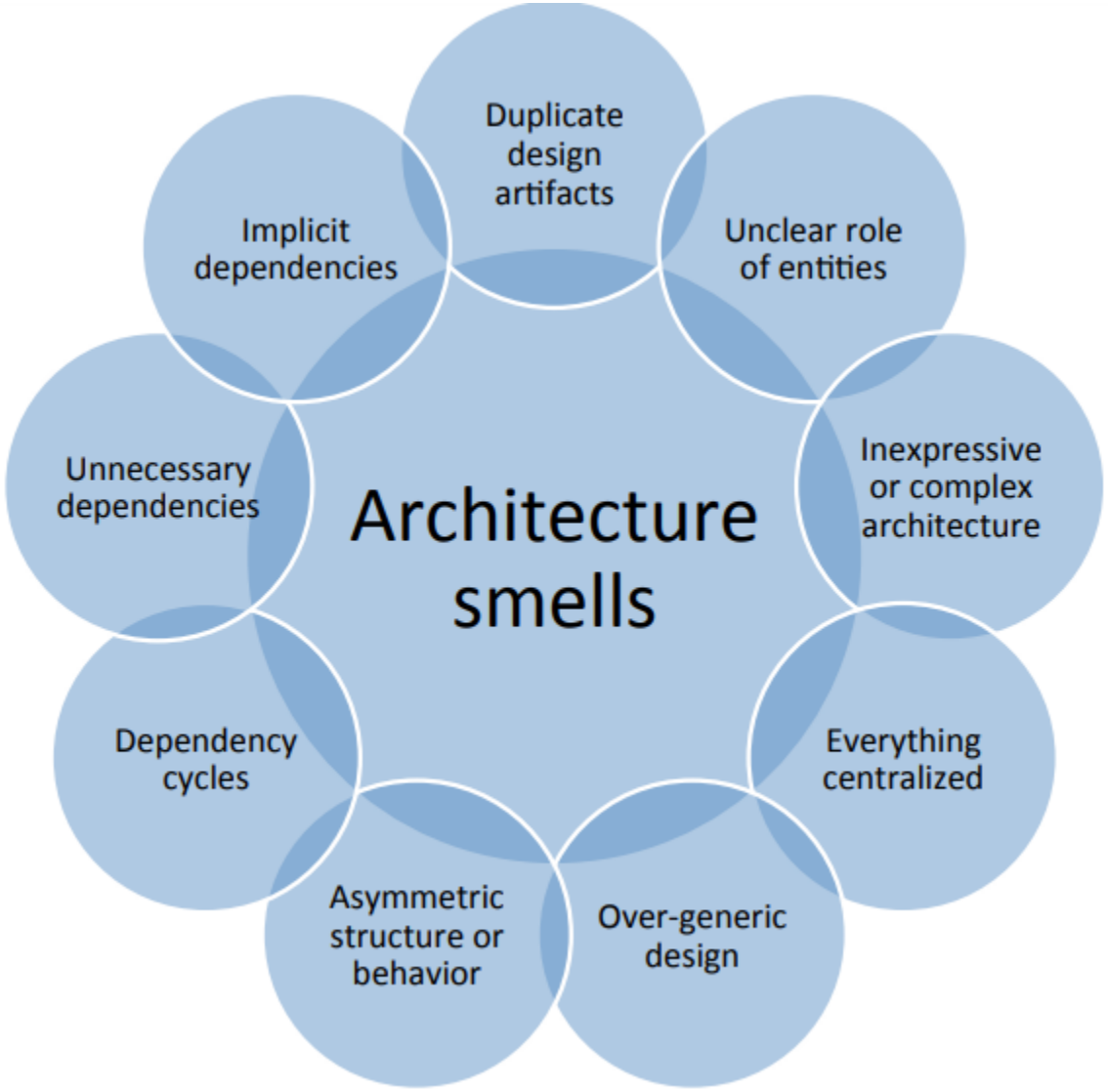
On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms

Isela Macia[1], Roberta Arcoverde[1], Alessandro Garcia[1], Christina Chavez[2], Arndt von Staa[1]
[1]Opus Group, LES, Informatics Department, PUC-Rio, RJ, Brazil
[2]Computer Science Department, Federal University of Bahia, BA, Brazil

Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, Arndt von Staa. **On the relevance of code anomalies for identifying architecture degradation symptoms**. Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012), Szeged, Hungary. USA: IEEE Computer Society, 2012, pp. 277-286

Architectural smells

Common architectural design problems manifested as architectural smells



© Ganesh Samarthyam

(A) Catalogue of architectural smells

Brick Concern Overload

Brick Use Overload

Brick Dependency Cycle

Unused Interface

Ambiguous Interface

Duplicate Component Functionality

Scattered Functionality

Component Envy

Connector Envy

Connector Chain

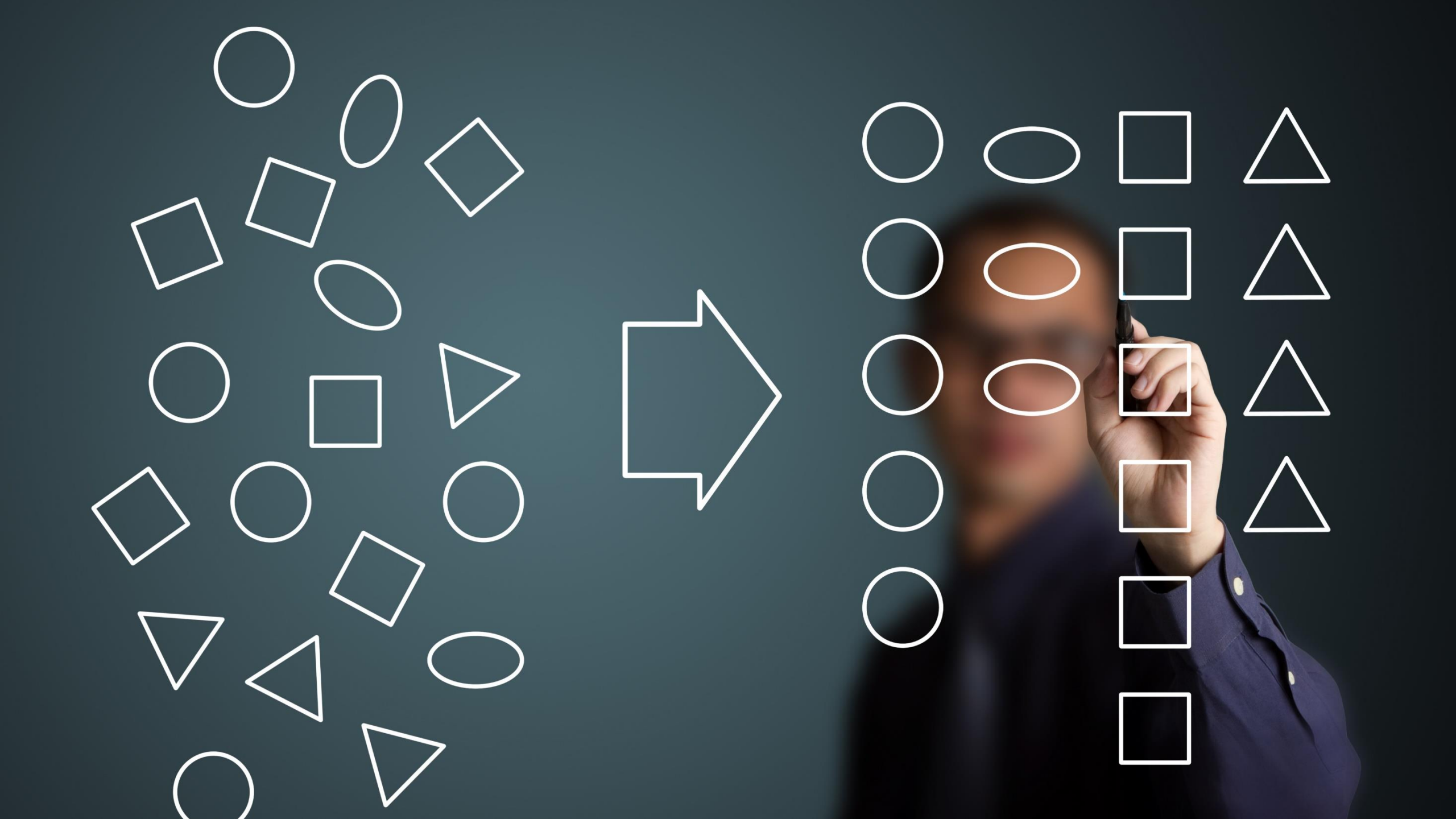
Extraneous Adjacent Connector

Toward a Catalogue of Architectural Bad Smells

Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic

University of Southern California, Los Angeles, CA, USA

Joshua Garcia, Daniel Popescu, George Edwards, Nenad Medvidovic. **Toward a catalog of architectural bad smells.** Rafaella Mirandola, Ian Gorton, Christine Hofmeister (eds.) Proceedings of the 5th International Conference on Quality of Software Architectures (QoSA 2009), East Stroudsburg, PA, USA. Lecture Notes in Computer Science, vol. 5581. Germany: Springer-Verlag Berlin Heidelberg, 2009, pp. 146-162



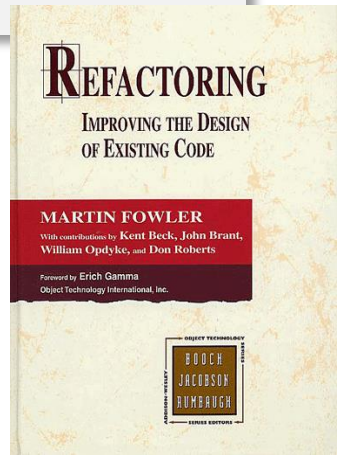
One definition more (!)

Refactoring

Systematic, disciplined process for changing the internal structure of a program aiming to increase its understanding, maintainability or other quality attributes, but without changing its visible behavior

It can be done at several moments in development

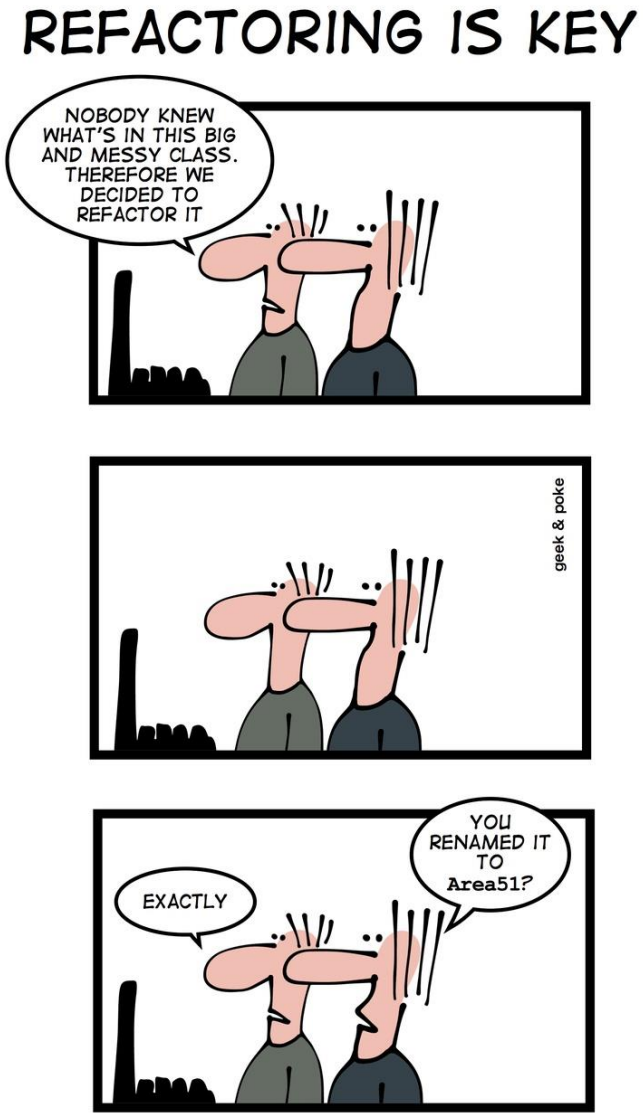
- Inclusion of new functionalities
- Testing and defect detection
- Code review
- Optimization



Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts.
Refactoring: Improving the design of existing code. USA: Addison-Wesley Longman, Inc., 1999

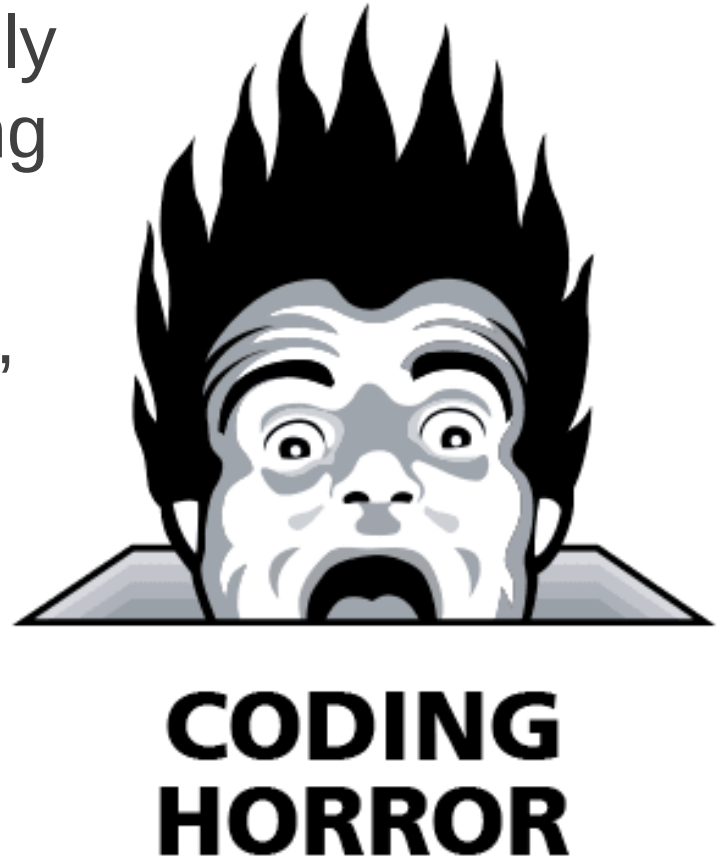
Reasons to refactor

- To increase maintainability
  - Reduction of coupling
  - Improvements on software architecture
  - Correction of code smells and errors
- To make the software design more modular
- It is typically the cheapest option
  - It is better than rewriting everything from the scratch
- To reduce stress in the development team



Reasons to not refactor

- When there are no enough means to completely test the refactored software once the refactoring process is finished
- Simply because new technologies have arisen, without a clear justification of their usefulness and value for the current software
- Software does not need changes
  - If there is no reason for changes, then there is no reason for refactoring



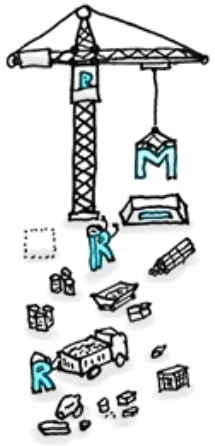
Reasons to not refactor

Developers claim that they avoid refactoring because

- the risks are high
- the Return on Investment (ROI) is not worth enough
  - or actually nobody knows what the ROI will be
- it is an activity that has a cost and may be hard to perform
- it often depends on managerial decisions

Ewan Tempero, Tony Gorschek, Lefteris Angelis.
Barriers to refactoring. Communications of the ACM, vol. 60, no. 10, October 2017, pp. 54-61

Classification of refactoring techniques



Composing Methods: aim at simplifying methods/classes that are too long and difficult to understand and maintain, besides removing eventual duplications

Examples: *Extract Method*, *Extract Variable*, *Substitute Algorithm*

Moving Features Between Objects: aim at safely moving functionalities between classes, creating new classes, and promoting greater encapsulation

Examples: *Move Method, Move Field, Extract Class*

https://sourcemaking.com/refactoring

Classification of refactoring techniques

Organizing Data: help to deal with data and simplify associations between classes, thus making them more portable, reusable

Examples: *Replace Magic Number with Symbolic Constant*

Simplifying Conditional Expressions: aim at simplifying conditional expressions and their logic

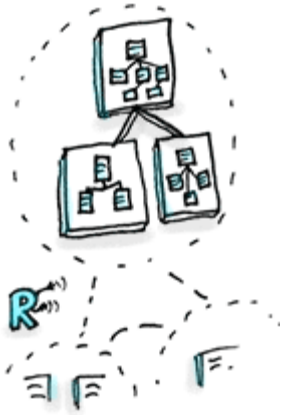
Examples: *Decompose Conditional, Remove Control Flag*

https://sourcemaking.com/refactoring

Classification of refactoring techniques

Simplifying Method Calls: aim at making method invocations simpler, easier to understand, thus simplifying interactions among classes

Examples: *Rename Method, Add Parameter, Remove Parameter*



Dealing with Generalization: aim at organizing functionalities in the class hierarchy by creating new classes and interfaces as well as appropriate inheritance relationships

Examples: *Extract Subclass, Extract Superclass, Collapse Hiearchy*

https://sourcemaking.com/refactoring

Architecture refactoring

Code refactoring vs. architecture refactoring

Code refactoring	Architecture refactoring
A module-level or class-level concern	A system level concern that cuts across modules or sub-systems
Impact of refactoring is within a team	Impact of refactoring is often across teams
Typically performed to improve the internal structure of the code	Performed for various reasons: cost, legal, security, performance, availability, …
Management buy-in typically not required	Management buy-in is typically required
Upfront planning is typically (relatively) limited	Upfront planning and co-ordination (sometimes between teams) is often required
Unit tests are important to ensure that "behaviour is preserved"	Unit tests, integration tests, system tests, NFR tests, … are required
Risk of breaking the working software is relatively low	Risk of breaking the working software is relatively high

Architecture refactoring

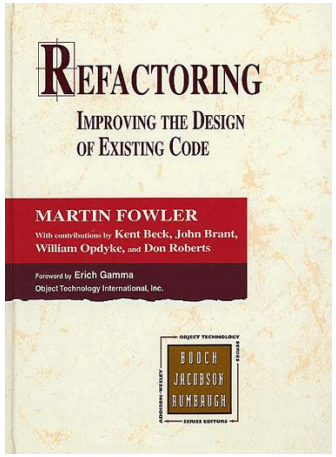
Key reasons vs. main challenges

- Increase system's quality atributes
- Reduce costs
- Address technical debt
- Address architecture degradation
- Need for modernization
- Compliance with business needs

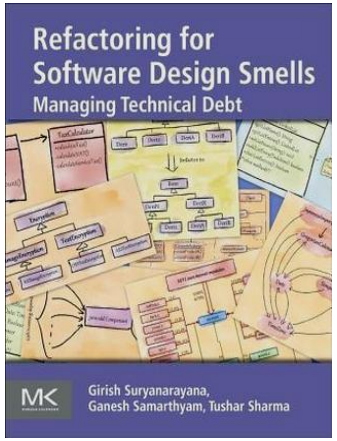
- Fear of breaking working software
- Lack of tool support
- Merge process problems
- Getting management buy-in (mainly for unclear ROI)

The take away message

Further reading



Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. Refactoring: Improving the design of existing code. USA: Addison-Wesley Longman, Inc., 1999



Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma. Refactoring for software design smells: Managing technical debt. USA: Morgan-Kauffman/Elsevier, Inc., 2015

Further reading

DECOR: A Method for the Specification and Detection of Code and Design Smells

Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, Anne-Françoise Le Meur. **DECOR: A method for the specification and detection of code and design smells.** IEEE Transactions on Software Engineering, vol. 36, no. 1, January/February 2010, pp. 20-36

Better Now Than Later: Managing Technical Debt in Systems Development

Richard E. Fairley, Mary Jane Willshire. **Better now than later: Managing technical debt in systems development.** Computer, vol. 50, no. 5, May 2017, pp. 80-87

When and Why Your Code Starts to Smell Bad

Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, Denys Poshyvanyk. **When and why your code starts to smell bad.** Proceedings of the 37th International Conference on Software Engineering (ICSE 2015), Florence, Italy. USA: IEEE, 2015, pp. 403-414

Barriers to Refactoring

Ewan Tempero, Tony Gorschek, Lefteris Angelis. **Barriers to refactoring.** Communications of the ACM, vol. 60, no. 10, October 2017, pp. 54-61

Anomalies and refactoring in software architectures

Prof. Dr. Everton Cavalcante

http://www.dimap.ufrn.br/~everton/