REENGINEERING

Successful software product must …

- Satisfy the stakeholder's requirements.
  - both functional and non-functional requirements
- Be developed on time and on budget.
- Be _____.

"All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version."

-- Ivar Jacobson

Lehman's Laws

A classic study by Lehman and Belady [Lehm85a] identified several "laws" of system change.

Continuing change

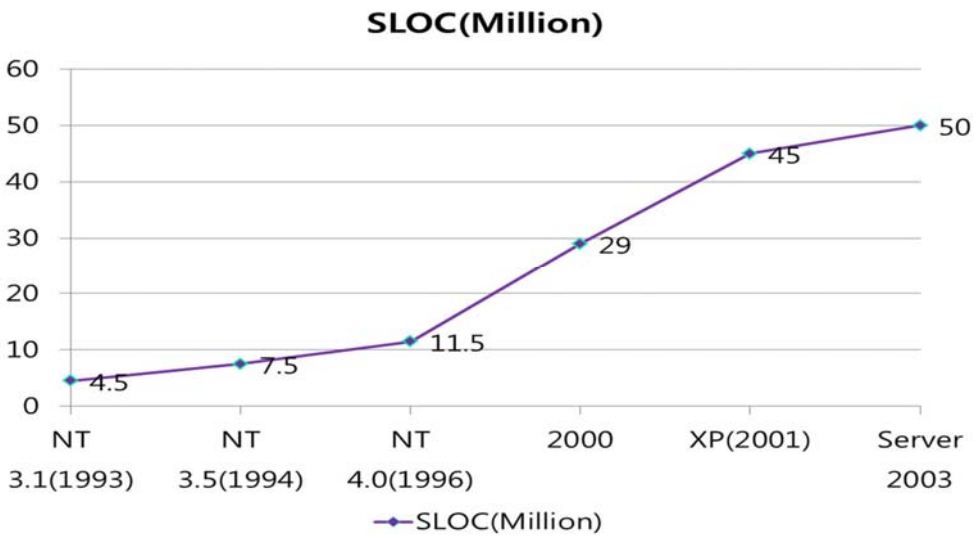
- A program that is used in a real-world environment *must change*, or become progressively less useful in that environment.

Increasing complexity

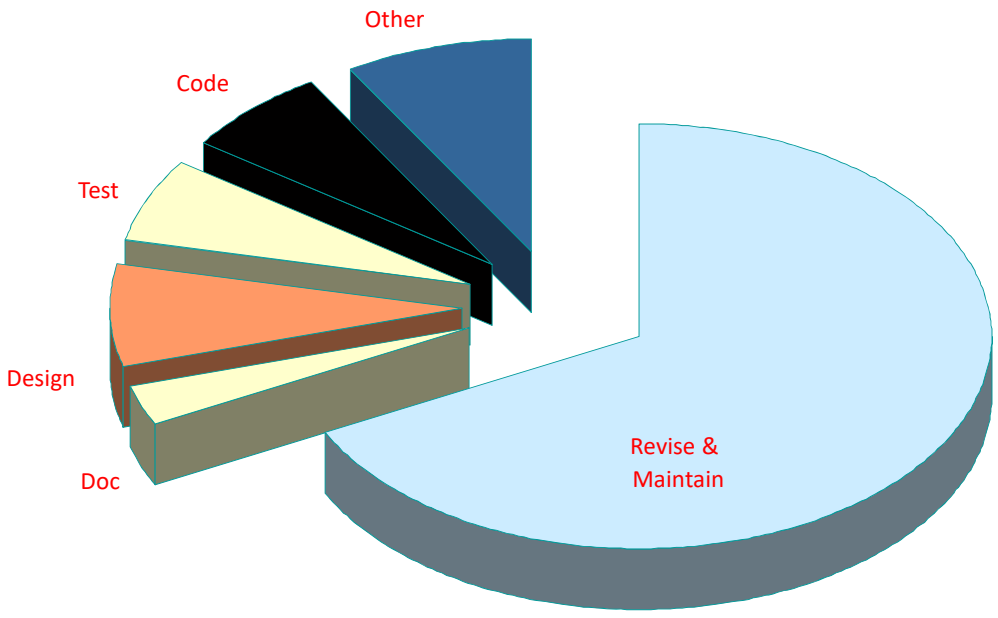
- As a program evolves, it becomes *more complex*, and extra resources are needed to preserve and simplify its structure.

slide from Marinescu

Growing Complexity: Windows



Strategic rational system development plans are based on the complete cost of a system, not solely on development costs



Between 50% and 75% of global effort is spent on maintenance!

How to embrace change, reduce cost, increase productivity?

- Paradigm shift
  - Object-oriented Paradigm
    - Enabling technology to cope with complexity.
  - Functional Paradigm or Both?
- Innovation of development Process
  - ***Iterative and incremental***, architecture-centric, use case-driven process
  - Component-based development (CBD)
  - Software product line engineering (SPLE)
  - Reuse-based software engineering

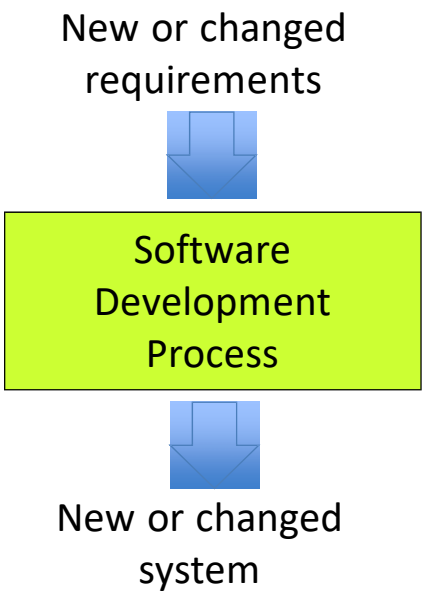
The real power and advantage of OT is *its capacity to tackle complex systems* and *to support easily adaptable systems*, lowering the cost and time of change

The Corporate Use of OT, Dec 1997, Cutter Group.
Prioritized reasons for adopting OT:

1. Ability to take advantage of new operating systems and tools
2. Elegantly tackle complexity & create easy adaptability
3. Cost savings
4. Development of revenue-producing applications
5. Encapsulation of existing applications
6. Improved interfaces
7. Increased productivity
8. Participation in "the future of computing"
9. Proof of ability to do OO development
10. Quick development of strategic applications
11. Software reuse

A development process defines who is doing what, when and how to reach a certain goal

In software engineering,
the goal is
to build a software product, or
to enhance an existing one.



Basic Disciplines in a Process

Analysis

What are the (functional/non-functional) requirements?
-Domain analysis
-Requirements gathering/analysis/spec.

Design

How to devise a logical solution to fulfill the requirements?
-System Architecture, Internal Designs
-UI, Database designs etc.

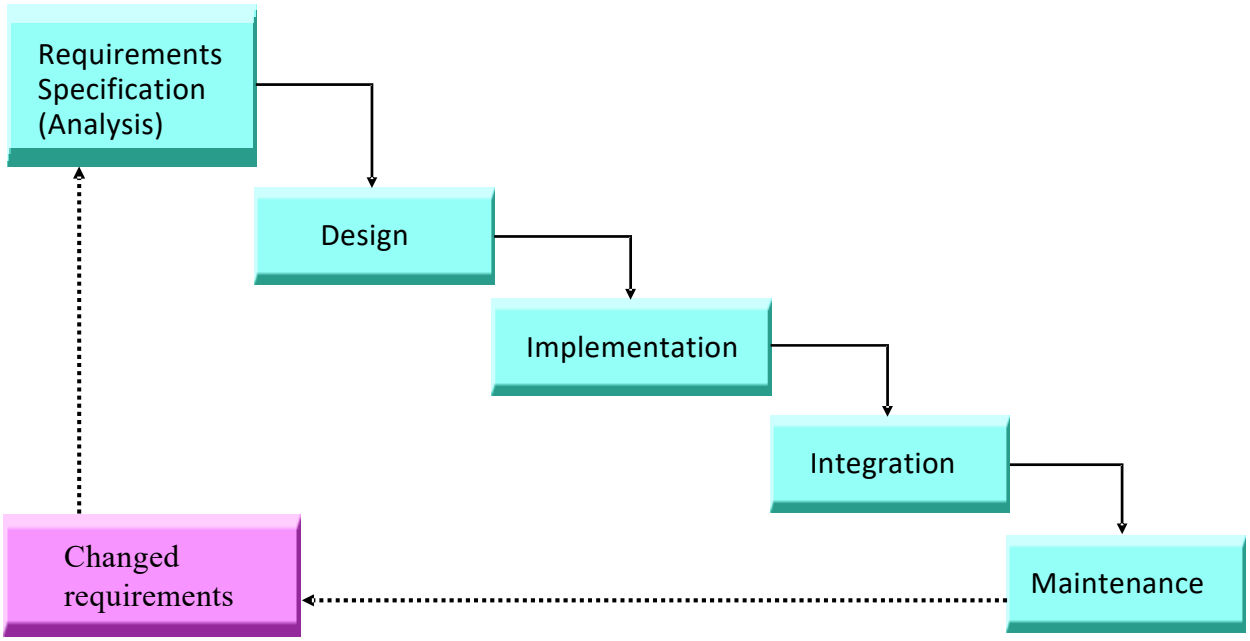
Implementation

Coding of the logical solution

Testing

- Unit test, Integration test, Regression Test
- Acceptance Test
  Does the system do what it was meant to do?

Waterfall Process



Iterative & Incremental Process

Iterative

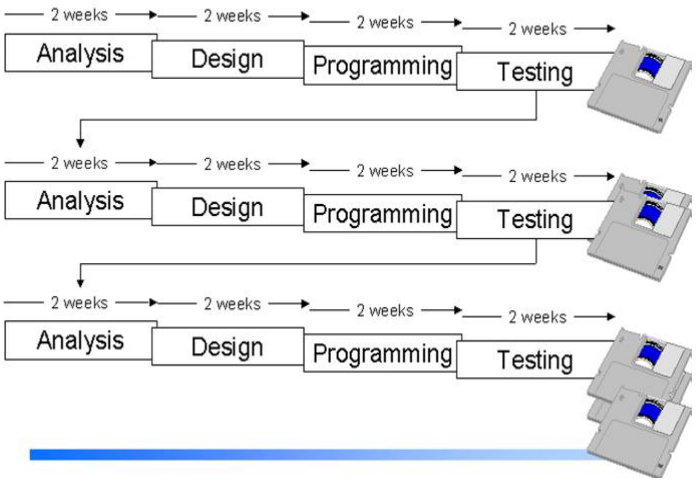
Instead of building the entire system as one go, the project has a few or many builds

A build includes only a subset of the entire functionality

Incremental

Software is developed on top of previous build

Make small but noticeable improvements in each iteration



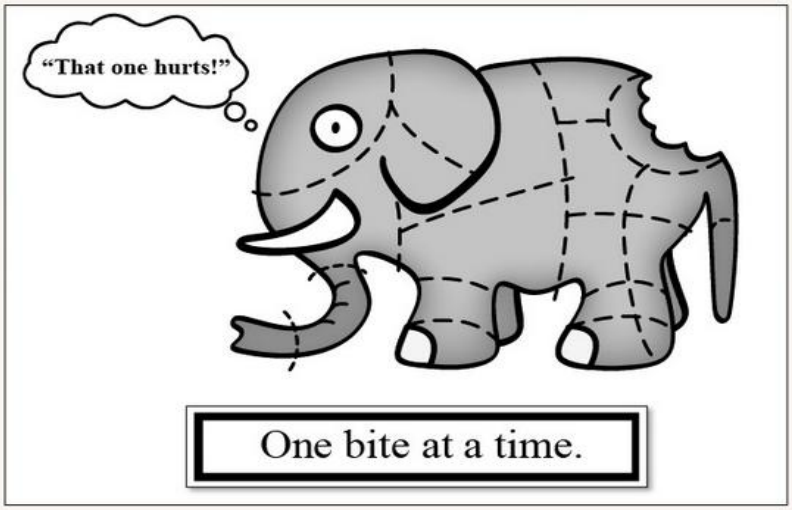
Small steps, ***feedback*** and refinement and ***adaptation***

Time-boxed

Aka. Evolutionary or spiral

You should use iterative development only on projects that you want to succeed!

Martin Fowler



Short quick development steps, feedback, and adaptation to clarify the requirements and design

Forward Engineering

- Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.
  - **Requirements** (specification of the problem being solved, including objectives, constraints, and business rules)
  - **Design** (specification of the solution)
  - **Implementation** (coding, testing, and delivery of the operational system)

What is Legacy Code?

- Old Code
- Ugly Code
- Spaghetti Code
- Complicated Code
- Gibberish Nonsense
- Code Written by Someone Else, Not by Me?
- …

Legacy System

A legacy system is a piece of software that:

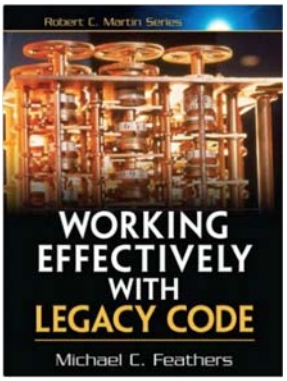
- you have inherited, and
- is *valuable* to you.
  - Often business-critical
  - Huge amount of money already invested
  - Has been tested (hopefully) and runs
  - Does (mainly) what it should do

Typical Problems

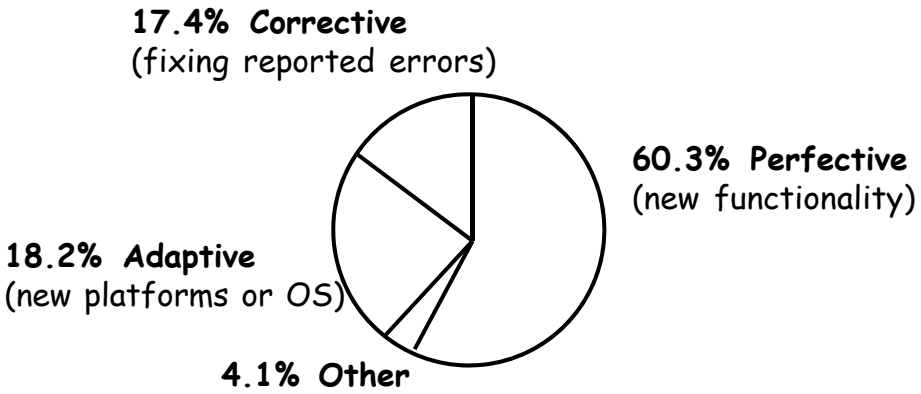
- Existing software often modified in an ad-hoc manner (quick fixes): Lack of time, resources, money, etc.
- Initial good design not maintained
  - Spaghetti code, copy/paste programming, dependencies are introduced, no tests, etc.
- Missing or outdated documentation
- Original developers no longer available

"Code with no _____"

⇒ *so, further evolution and development may be prohibitively expensive*



Maintenance Cost Due to Change Request



The bulk of the maintenance cost is due to
new functionality
⇒ even with better requirements, it is hard to predict new functions

slide from Marinescu

What about Objects ?

- Object-oriented legacy systems
  - Successful OO systems whose architecture and design no longer responds to changing requirements.
- Compared to traditional legacy systems
  - The symptoms and the source of the problems are the same
    - ravioli code instead of spaghetti code ;)
  - The technical details and solutions may differ.
- OO techniques promise better
  - flexibility,
  - reusability,
  - maintainability
  - …

⇒ they do not come for free

slide from Marinescu

Common Symptoms

Process symptoms

- *Too long* to turn things over to production
  - simple changes take too long
- Need for *constant bug fixes*
- *Maintenance dependencies*
- *Difficulties separating* products

Code symptoms

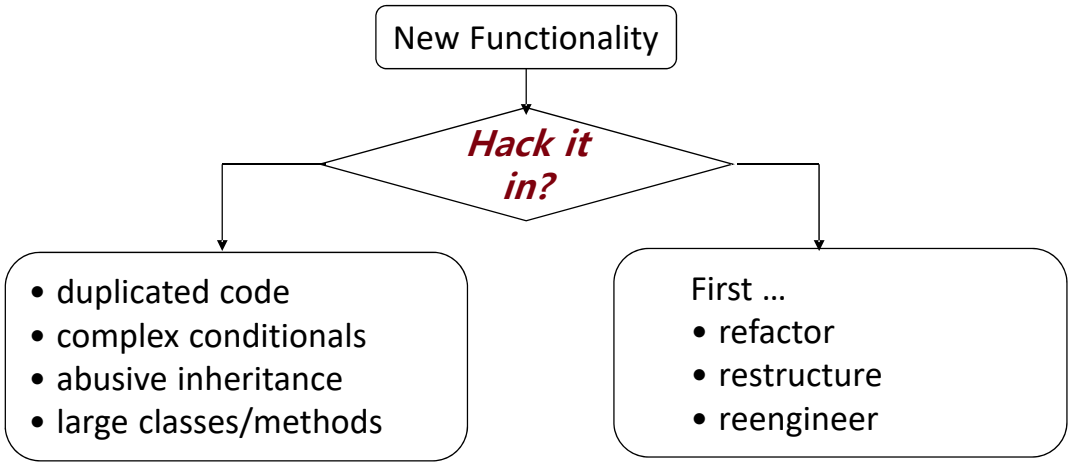
- *Big build times*
- *Duplicated code*
  - cut, paste & edit
- *Duplicated functionality*
  - similar functionality by separate teams
- *Code smells*

Common Problems

- Architectural Problems
  - Insufficient *documentation*
    - non-existent or out-of-date
  - Improper *layering*
    - too few are too many layers
  - Lack of *modularity*
    - strong coupling
- *Design Problems*
  - misuse of inheritance, missing inheritance and misplaced operations, etc.

How to deal with Legacy ?

New or changing requirements will gradually *degrade original design*
… unless extra development effort is spent to adapt the structure



Take a loan on your software
⇒ pay back via reengineering

Investment for the future
⇒ paid back during maintenance

slide from Marinescu

Reverse Engineering

- It is a process of *examination*, not a process of change or replication.
  - the process of gaining enough design-level understanding about a product to help with its maintenance, enhancement, or replacement.
- To create high-level abstractions of a system
- To identify system components and their interrelationships

"Reverse Engineering and Design Recovery: A Taxonomy" by E. J. Chikofsky and J. H. Cross II, which appeared in *IEEE Software* 7(1), 13-17. Copyright IEEE 1990.

Subareas of Reverse Engineering (I)

Redocumentation

- Redocumentation is the creation or revision of a semantically equivalent representation (i.e. views) within the same relative abstraction level.
- The "re-"prefix implies that the intent is to recover documentation about the subject system that existed or should have existed.
- Pretty printers, diagram generators, and cross-reference listing generators.

The intent is mostly to recover lost or non-existent documentation about the system.

Subareas of Reverse Engineering (II)

Design Recovery

- Design recovery ***recreates design abstractions*** from a combination of code, existing design documentation (if available), personal experience, and general domain knowledge.
- Design recovery must reproduce all of the information required for a person to fully understand
  - what a program does,
  - how it does it,
  - why it does it, and so forth.

Goals of Reverse Engineering

- Generate *alternative views*
  - automatically generate different ways to view systems
- Recover *lost information*
  - extract what changes have been made and why
- Detect *side effects*
  - help understand ramifications of changes
- Synthesize *higher abstractions*
  - identify latent abstractions in software
- Facilitate *reuse*
  - detect candidate reusable artifacts and components

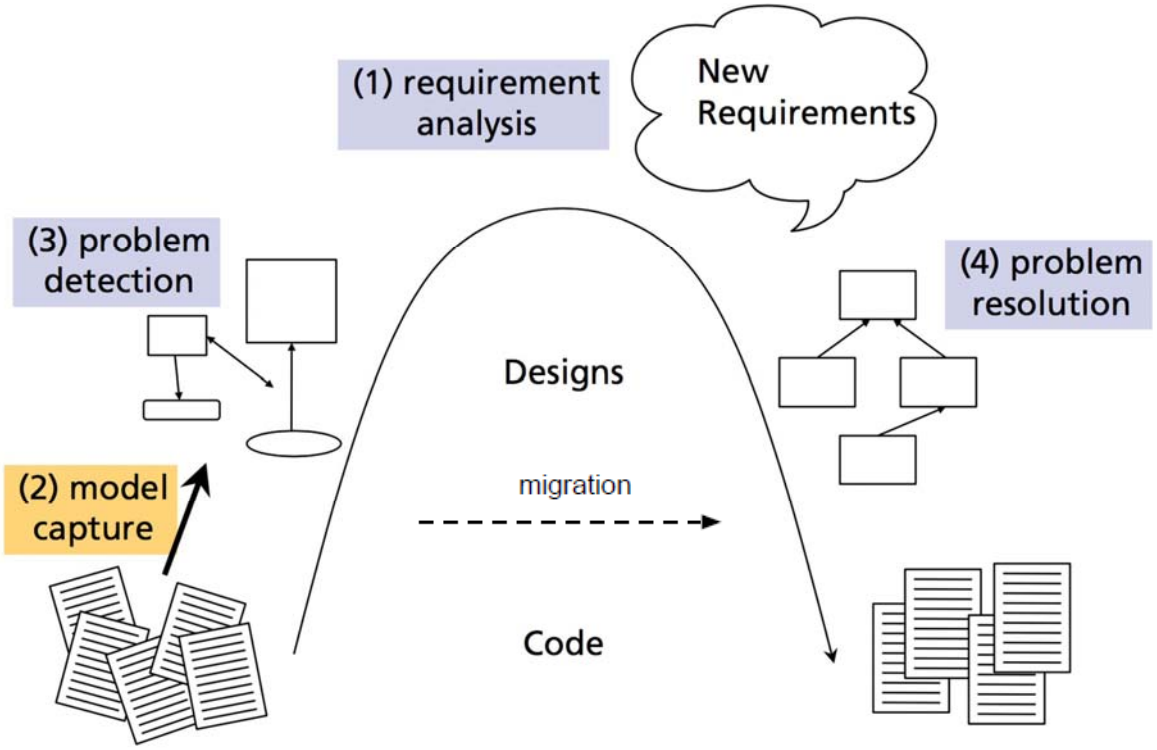
Restructuring (or Refactoring)

- Restructuring is the *transformation* from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics).
  - Code-to-code transformation
  - Data normalization
  - Reshaping requirement structures
- It may lead to structure that facilitates changes to meet the new requirements and environment constraints.
  - a form of *preventive maintenance*

Reengineering

- Reengineering is the *examination and alteration* of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.
- Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring.
  - May include modifications with respect to new requirements not met by the original system.

The Reengineering Life-Cycle



Goals of Reengineering

- *Unbundling*
  - split a monolithic system into parts that can be separately marketed
- *Performance*
  - "first do it, then do it right, then do it fast"
    - experience shows this is the right sequence!
- *Design refinement*
  - to improve maintainability, portability, etc.
- *Port* to other Platform
  - the architecture must distinguish the platform dependent modules
- Exploitation of *New Technology*
  - i.e., new language features, standards, libraries, etc.