Software Design

Arquiteturas e Padrões de Software

Hugo Paredes

What is Software Design ?

Definição coletiva de Software Design

https://www.menti.com/i29b6soffv

O que é software design?

26 responses

A Proposal for a Formal Definition of the Design Concept

Paul Ralph[1] and Yair Wand[1]

[1]Sauder School of Business
University of British Columbia
Canada
{paulralph@gmail.com, yair.wand@ubc.ca}

Abstract: A clear and unambiguous definition of the design concept would be useful for developing a cumulative tradition for research on design. In this article we suggest a formal definition of the concept *design* and propose a conceptual model linking concepts related to design projects. The definition of design incorporates seven elements: agent, object, environment, goals, primitives, requirements and constraints. The design project conceptual model is based on the view that projects are temporal trajectories of work systems that include human agents who work to design systems for stakeholders, and use resources and tools to accomplish this task. We demonstrate how these two conceptualizations can be useful by showing that 1) the definition of design can be used to classify design knowledge and 2) the conceptual model can be used to classify design approaches.

Keywords: design, information systems design, software design project, requirements, goals, science of design

1 Introduction

There have been several calls for addressing design as an object of research. Freeman and Hart call for a comprehensive, systematic research effort in the science of design: "We need an intellectually rigorous, formalized, and teachable body of knowledge about the principles underlying software-intensive systems and the processes used to create them," [1, p.20]. Simon [2] calls for development of a "theory of design" and gives some suggestions as to its contents. Yet, surprisingly, it seems no generally-accepted and precise definition of design as a concept is available.[1]

[1] As an anecdotal note – we have asked colleagues in several conferences to suggest a definition of design (in the software and IS context) and often the responses indicated IS academics did not have a well-defined notion of the concept.

Software design is the process by which an agent creates a specification of a software artifact, intended to accomplish goals, using a set of primitive components and subject to constraints.

"all the activities involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems" or "the activity following requirements specification and before programming, as ... [in] a stylized software engineering process"

A Science of Design for Software-Intensive Systems

Computer science and engineering needs an intellectually rigorous, analytical, teachable design process to ensure development of systems we all can live with.

LISA HANEY

More than 200 years after George Washington was elected the first U.S. president, the November 2004 presidential election will usher in a new experiment in democracy—the first widespread use of electronic voting technology in a national election. Whatever your attitude about e-voting systems and whatever you may have read about them in recent months, they are an example of the software-intensive systems that every industrialized society increasingly depends on for their basic operations. Software-intensive systems help protect and manage commercial air travel, operate the electric power grid, file tax returns, and may one day even uphold our democratic foundations.

Yet today, in spite of more than three decades of effort, we still lack what could be called a solid scientific basis for building any kind of large-scale system that includes software elements. Our software-intensive systems are so complex we often find ourselves struggling to understand and control them. New systems are created and adopted with high hopes, crossed fingers, and even controversy, as in the case of e-voting. Security breaches, privacy violations, and emergent (unexpected) properties are some of the regular, often unwelcome, results.

For computer scientists, being told that creating software-intensive systems is difficult is not news. Even attendees at the 1968 NATO Software Engineering Conference in Garmisch, Germany, recognized the need to build, for example, a foundation for the systematic creation of software-intensive systems [3]. Efforts have continued since then to develop a scientific basis and process for creating them.

Living in a world in which the number and diversity of devices, amount of software, and degree of connectivity in complex systems are all increasing by orders of magnitude, it is essential that we have a science of design on which to base our efforts to create these systems. Changing this situation won't happen overnight, but we also can't wait another three decades to begin in earnest.

While it's easy to assert the need for a science of design, defining what such a science entails and understanding what design really means are much greater challenges.

By way of analogy, the design of a large building is based on many scientifically discovered and validated facts, along with volumes of codified experience. A similar scientific basis exists for many engineered artifacts, ranging from integrated circuits to airplanes. Professional disciplines like architecture and engineering have been working on issues of design for far longer than a few decades. In general, they have a stronger scientific footing, and computer scientists can learn where that footing overlaps with computing. (The National Science Foundation has funded at least one ongoing project to research how information technology might better assist the architectural design process.)

On the other hand, software-intensive systems are complex, being composed of software, people, computers, and other devices. Though the other components' connections to the software and their role in the overall design of the system are critical, the core consideration for a software-intensive system is the software itself, and other approaches to systematizing design have yet to solve the "software problem"—which won't be solved until software

What you do when you "do SW design" ?

What things are part of a SW design ?

Who might use design "outputs" and for what ?

Maybe different ways to think about it?

- Goals
- Activities
- Inputs, Outputs
- Techniques, Skills
- Principles
- Descriptions

Qualities?
Principles or rules?

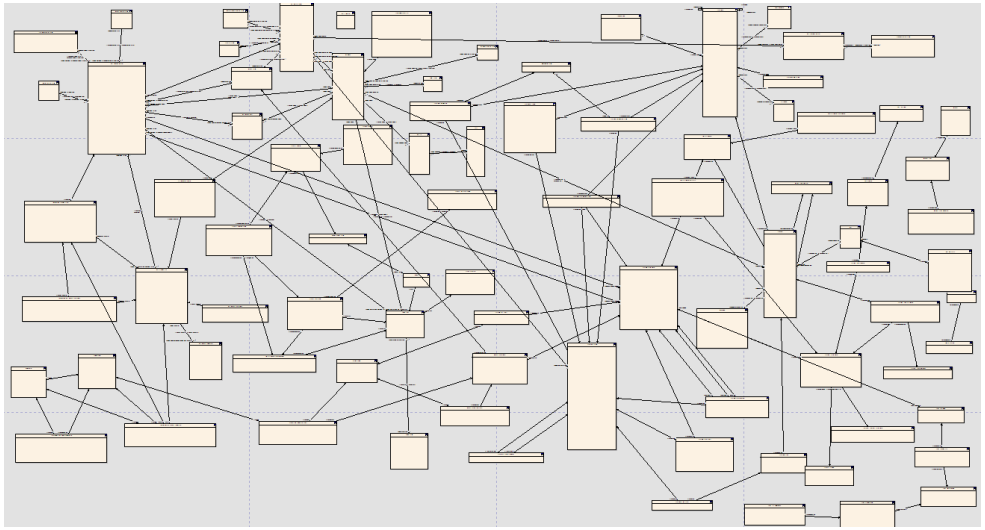
Design activities:

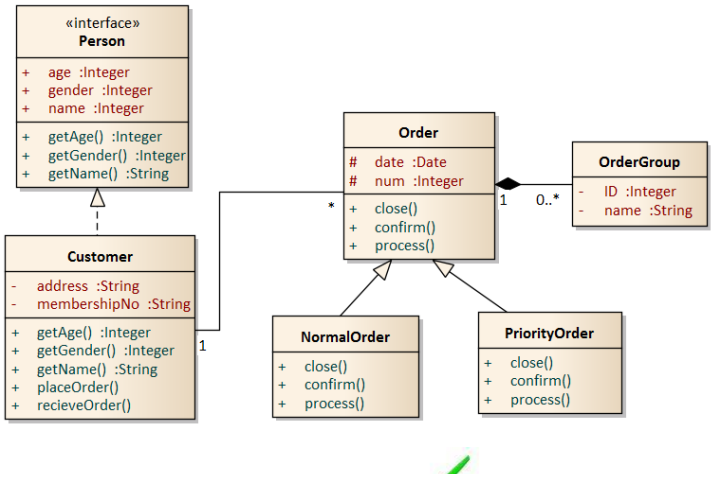
1. Identify purpose and features
2. Select classes
3. Sketch a user-interface (UI)
4. Work out dynamics with scenarios
5. Build a class diagram

Coad, Peter, and Mark Mayfield. *Java design: building better apps and applets*. Yourdon Press, 1997.

Design Principles: Keep it Simple (KIS)

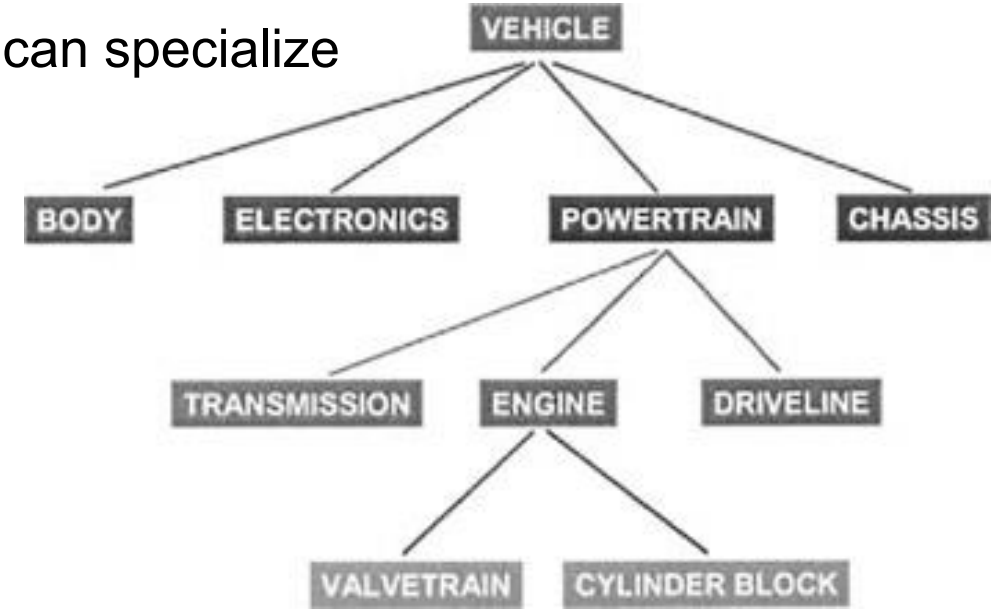
- Simplicity is a great virtue but it requires hard work to achieve it
  and education to appreciate it.
- And to make matters worse: complexity sells better.





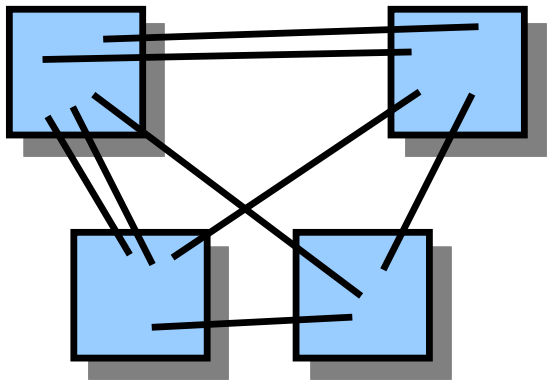
Design Principles: Decomposition

- Breaking problem into independent smaller parts
  - Each individual component is smaller, and therefore easier to understand
  - Parts can be replaced or changed without having to replace or extensively change other parts.
  - Separate people can work on separate parts
  - An individual software engineer can specialize

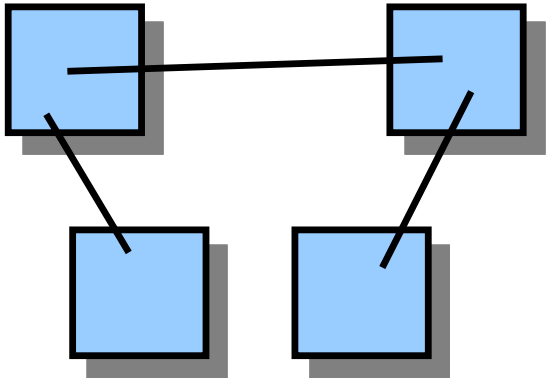


Design Principles: Coupling

- Coupling is a measure of interdependency between modules.



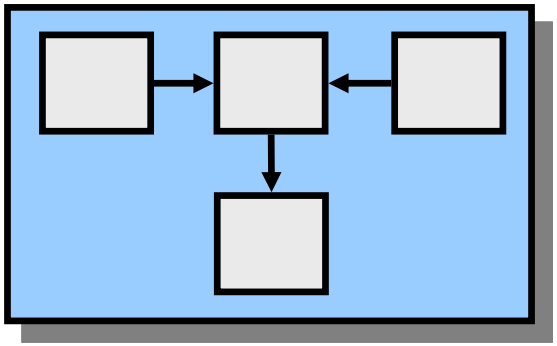
high coupling



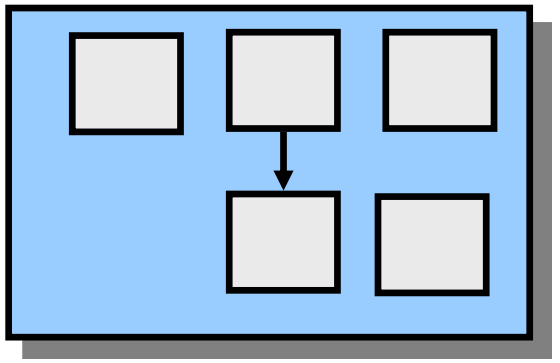
low coupling

Design Principles: Cohesion

- Cohesion is concerned with the relatedness within a module.



high cohesion

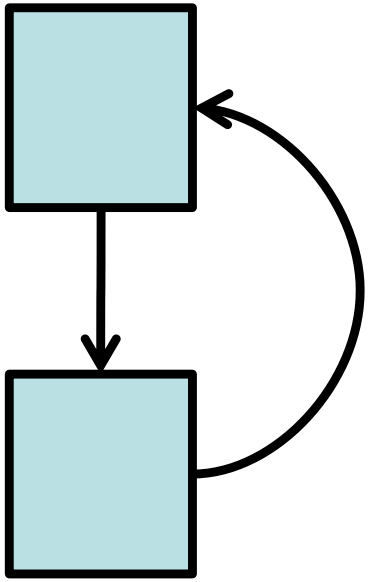


low cohesion

Design Principles: Information Hiding

- What is inside, must stay inside.

Design Principles: No Circular Dependencies



Callers must depend on callee, not vice versa

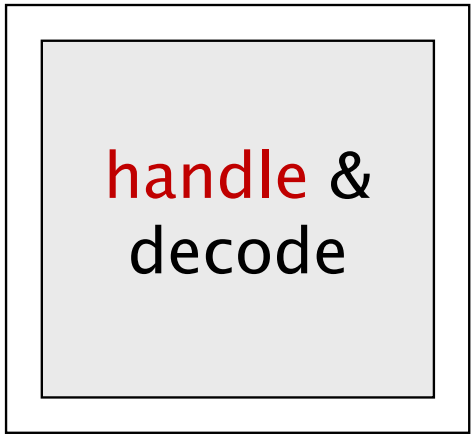
This violates an earlier design advice: Decomposition

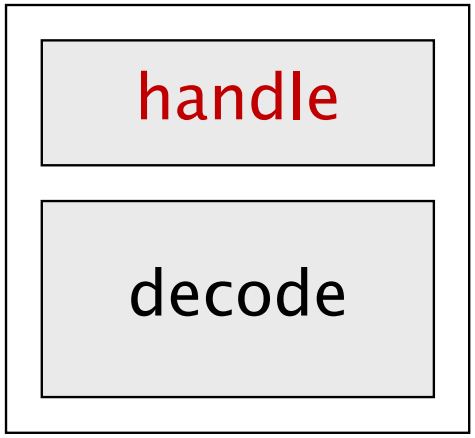
Design Principles: Separation of Concerns

- Issues that are not related should be handled in different components

Telecom protocol:

decode1 ; handle1 ; decode2 ; handle2 ; decode3





Design Principles: Open/Closed

- Entities should be open for extension, but closed for modification.

Design principles

1. Design with composition rather than inheritance
2. Design with interfaces
3. Design in interfaces
4. Design with notification

Coad, Peter, and Mark Mayfield. *Java design: building better apps and applets*. Yourdon Press, 1997.

Abstraction

Control Hierarchy

Refinement

Software Architecture

Modularity

Design concepts

Structural Partitioning

Information Hiding

Data Structure

Software Procedure

Designers need to think about OO in a "new", "fresh", "non-traditional" way
in particular, about three terms:
objects, encapsulation, inheritance

What are objects?

- We learn early that they're:
  - Data with methods
  - Visibility: public, private, etc
  - Supports information hiding, encapsulation
- Good match for:
  - domain concepts
  - data objects

Shalloway, Alan, and James R. Trott. *Design patterns explained: a new perspective on object-oriented design*. Pearson Education, 2004.

S&T's “New” View of Objects

- An object is:

  An entity that has responsibilities

  - may include providing and managing information
- Focus here on what it does, not how
- An external view
- We can design using objects defined this way (without other details)

Shalloway, Alan, and James R. Trott. *Design patterns explained: a new perspective on object-oriented design*. Pearson Education, 2004.

S&T's New View of Encapsulation

- Old/Beginner/Implementation view of encapsulation:
  hiding data inside an object
- New view: **hiding anything**, including:
  - implementation
  - derived classes
  - design details
  - instantiation rules

Shalloway, Alan, and James R. Trott. *Design patterns explained: a new perspective on object-oriented design*. Pearson Education, 2004.

Really a New View of Encapsulation?

Information hiding: (from Wikipedia)

Information hiding is the principle of segregation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed.

The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

Read: http://en.wikipedia.org/wiki/Information_hiding

Information Hiding and David Parnas

- Term introduced in a famous paper:
  David Parnas, "On the Criteria to Be Used in Decomposing Systems Into Modules."
  Comm. of the ACM, 1972.
- Parnas is an important figure in SW Engin.
- Keep in mind about *Information [sic] Hiding*:
  - "Information" does not mean just data encapsulation
  - Not just limited to OO

Design considerations:

- Compatibility
- Extensibility
- Fault-tolerance
- Maintainability
- Modularity
- Reliability
- Reusability
- Robustness
- Security
- Usability
- Performance

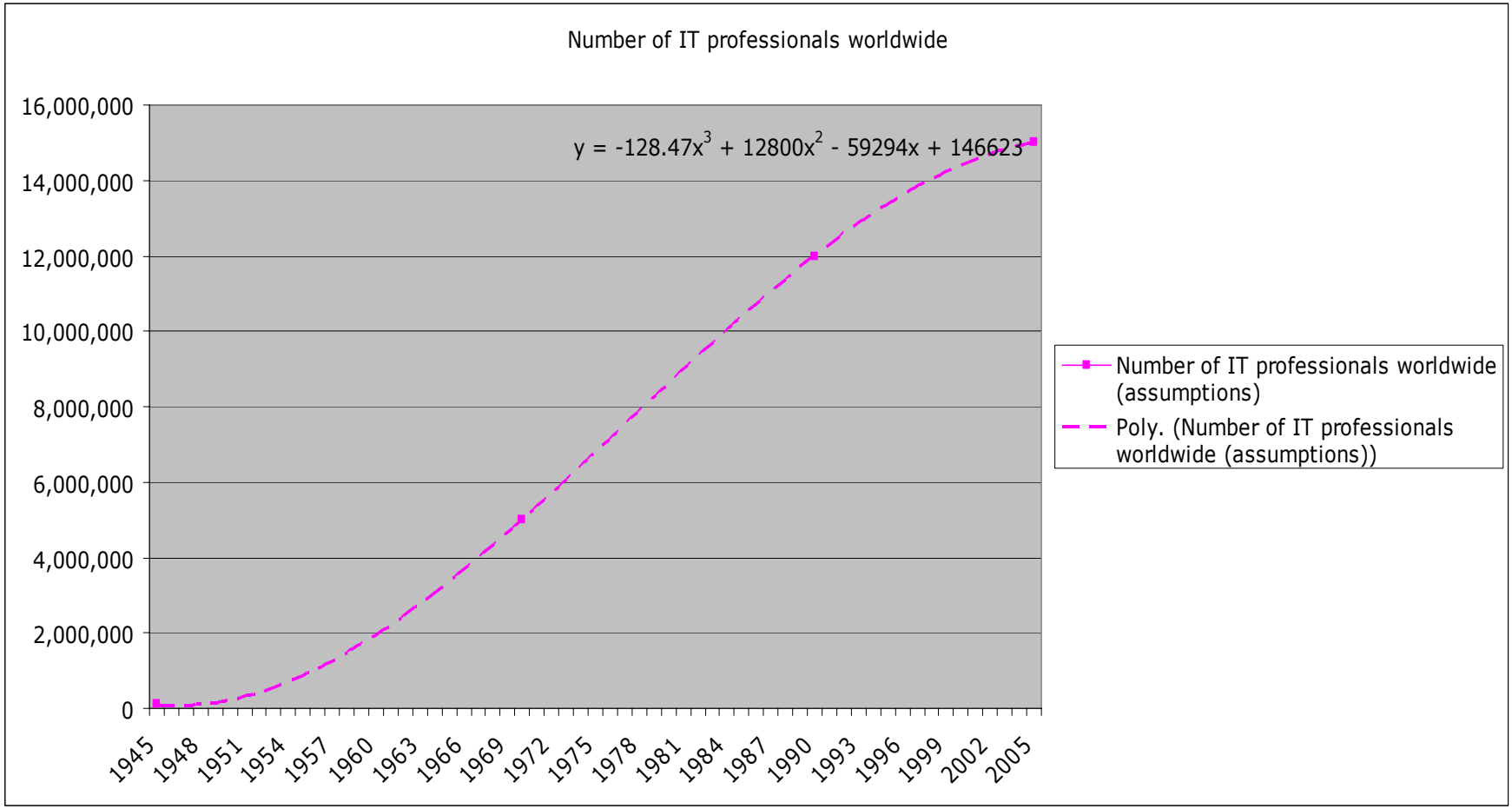
The role of software

- **Our civilization runs on software**
  - Innovation to capture new value
  - Improving productivity of resources deployed
- **The privilege and responsibility of the software professional**

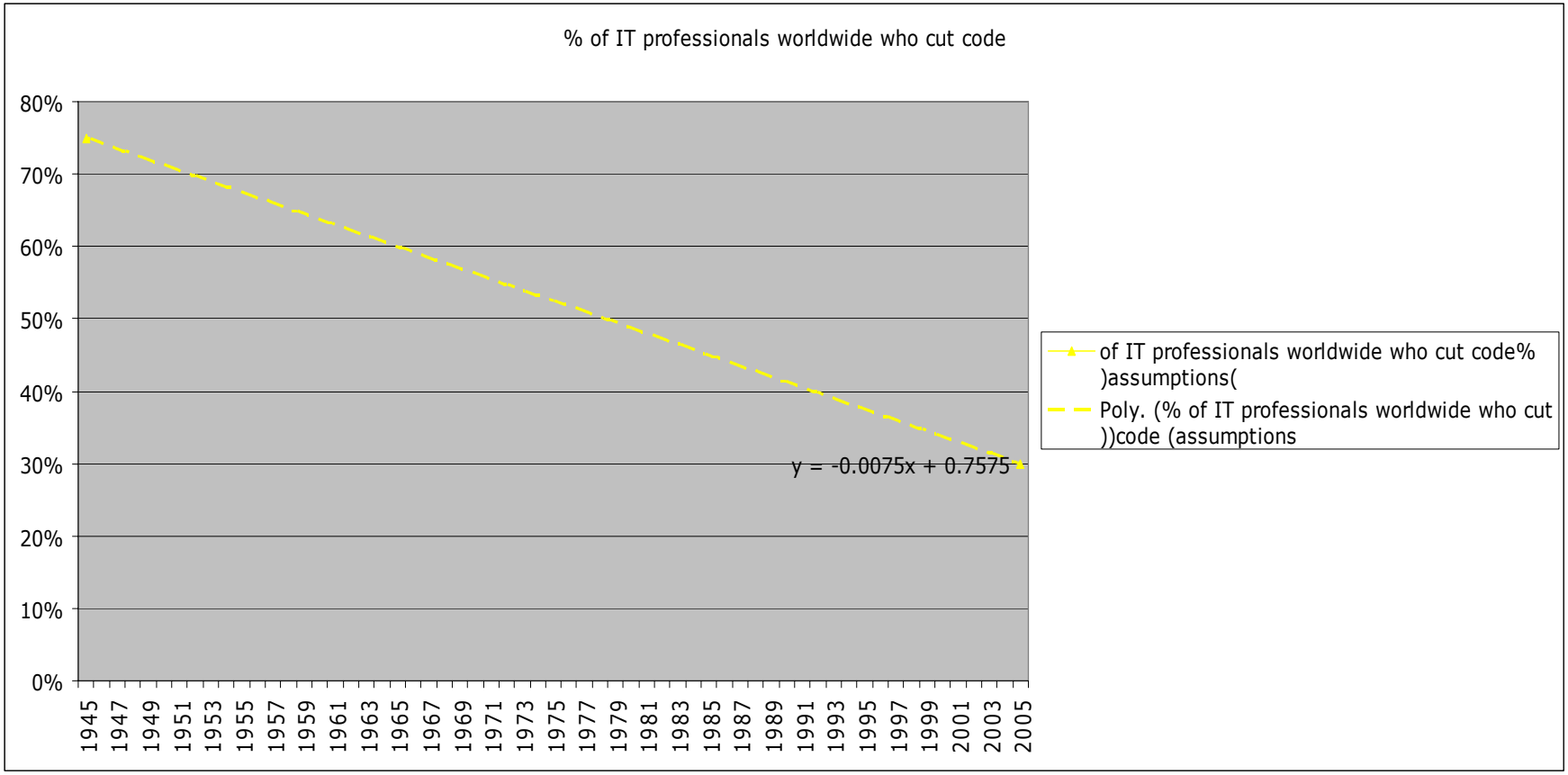
How much software exists in the world?

- **SLOC is a measure of labor (not of value)**
  - Old code never dies (you have to kill it)
  - Some code is DOA
- **Some assumptions**
  - 1 SLOC = 1 semicolon
  - Number of software professionals worldwide
  - %of software professionals who cut code
  - SLOC/developer/year
  - $100US/SLOC

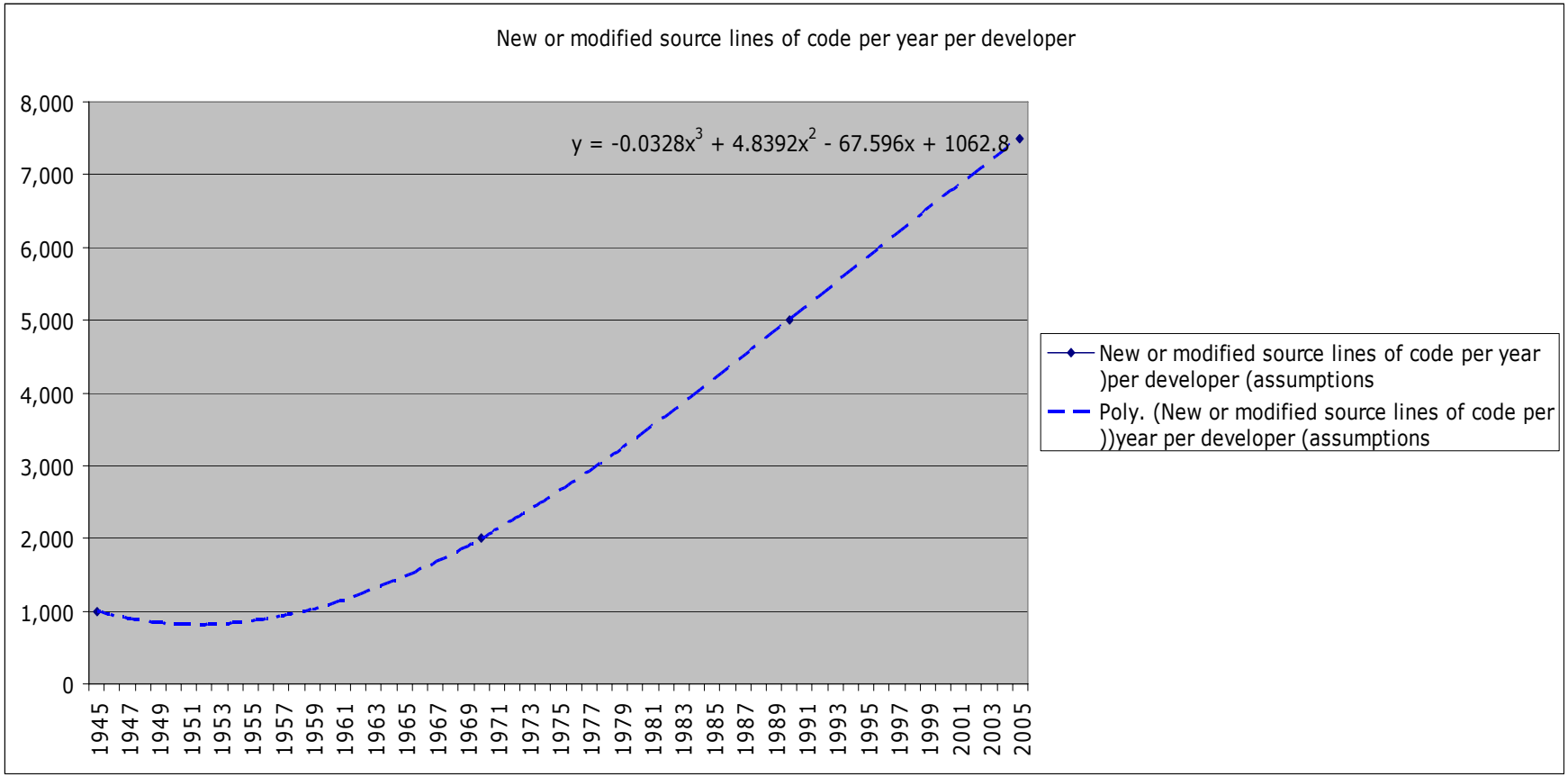
Number of software professional worldwide



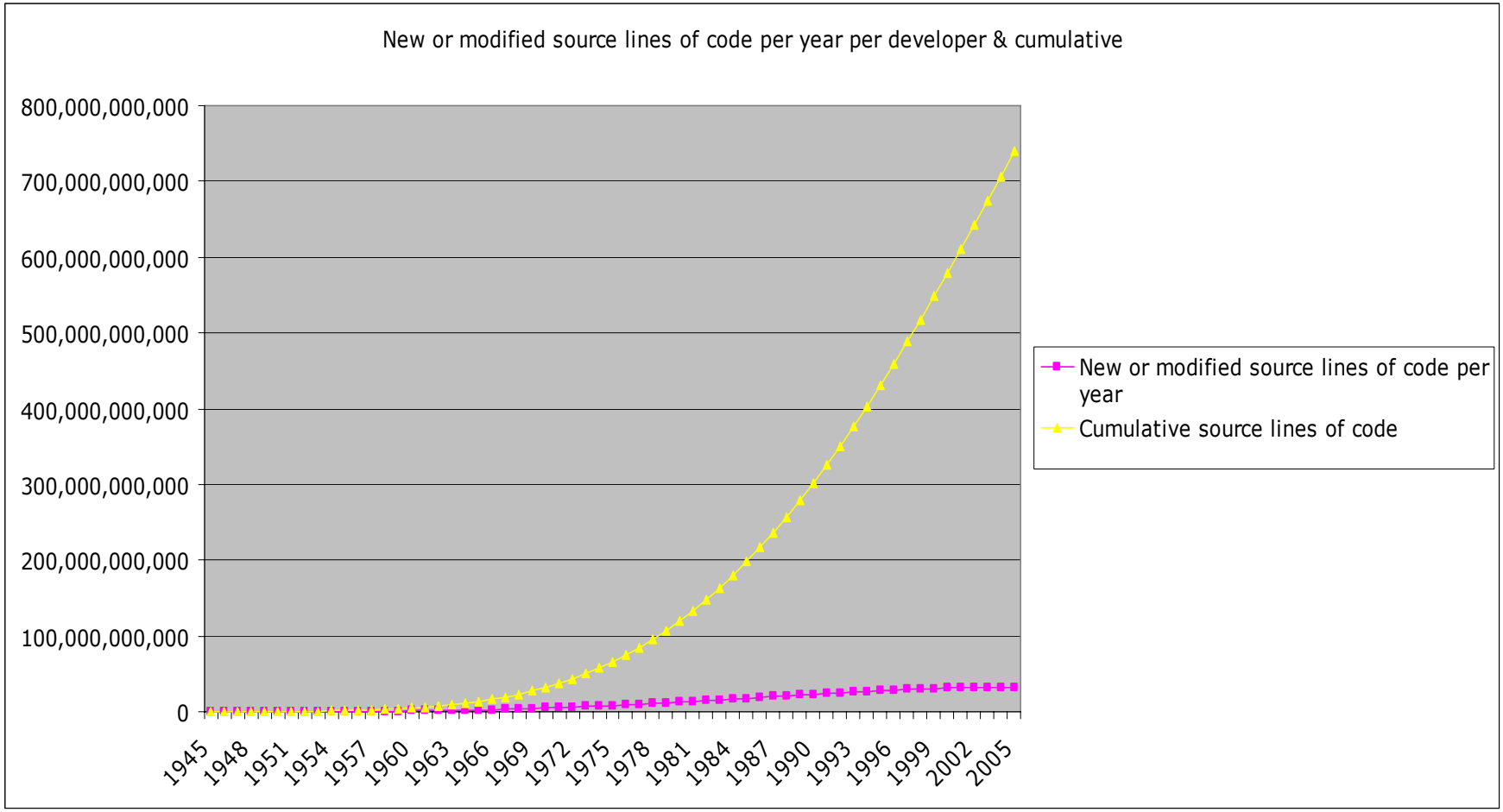
% of software professionals who cut code



SLOC/developer/year



New or modified SLOC/year and cumulative

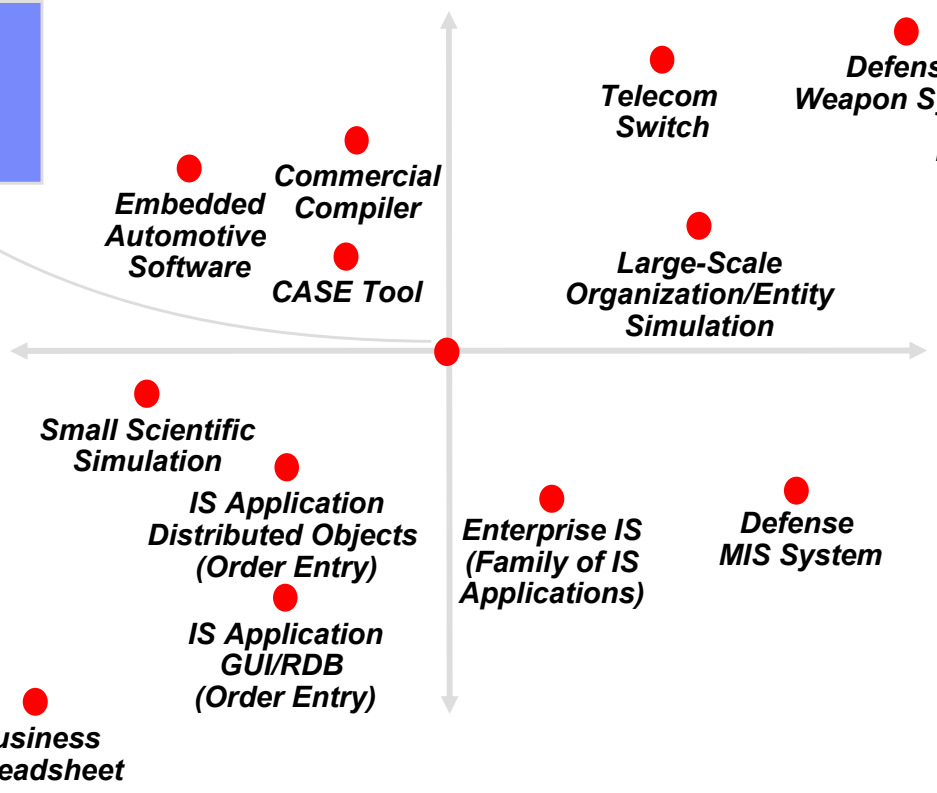


Dimensions of software complexity

Higher technical complexity

- Embedded, real-time, distributed, fault-tolerant
- Custom, unprecedented, architecture reengineering
- High performance

An average software project
- 5-10 people
- 3-9 month duration
- 3-5 external interfaces
- Some unknowns & risks



Lower management complexity

- Small scale
- Informal
- Single stakeholder
- "Products"

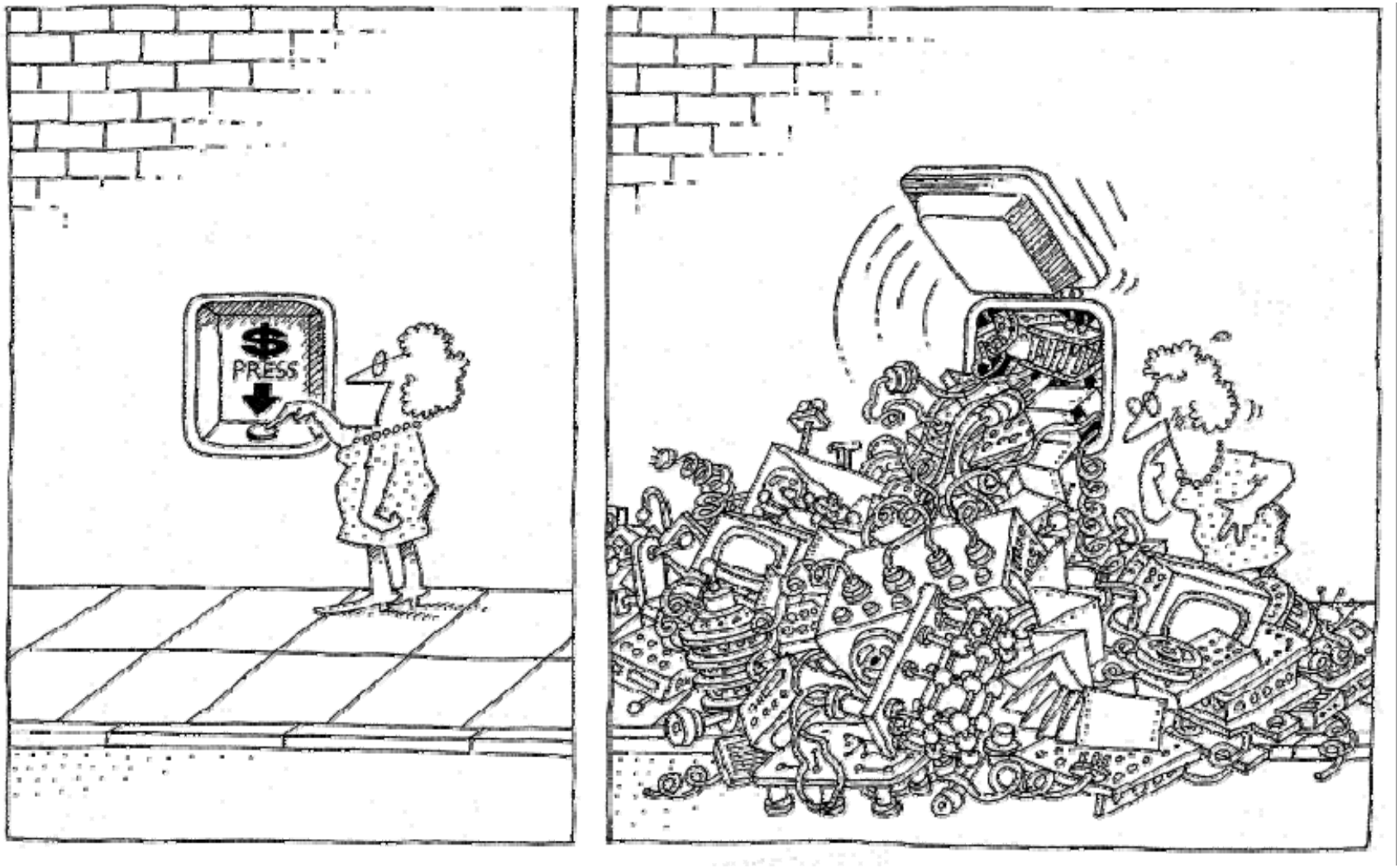
Higher management complexity

- Large scale
- Contractual
- Many stake holders
- "Projects"

Lower technical complexity

- Mostly 4GL, or component-based
- Application reengineering
- Interactive performance

Creating the illusion of simplicity



The entire history of software engineering
Is one of rising levels of abstraction

Languages: Assembly -> Fortran/COBOL -> Simula -> C++ -> Java
Platforms: Naked HW -> BIOS -> OS -> Middleware -> Domain-specific
Processes: Waterfall -> Spiral -> Iterative -> Agile
Architecture: Procedural -> Object Oriented -> Service Oriented
Tools: Early tools -> CLE -> IDE -> XDE -> CDE
Enablement: Individual -> Workgroup -> Organization