
Adam Hasselbalch Hansen

Department of Computer Science
University of Copenhagen

Feb. 22, 2008
Outline

Introduction

Essential Difficulties

Past Breakthroughs
  High-level Languages
  Time-sharing
  Unified Programming Environments

Hopes For The Silver
  High-level and OO-Programming
  Artificial Intelligence
  Automatic and Graphical Programming
  Program Verification
  Environments and Tools

Promising Attacks on the Conceptual Essence
  Buy Vs. Build
  Requirement Refinement and Prototyping
  Great Designers

Summary
Outline

Introduction

Essential Difficulties

Past Breakthroughs
  High-level Languages
  Time-sharing
  Unified Programming Environments

Hopes For The Silver
  High-level and OO-Programming
  Artificial Intelligence
  Automatic and Graphical Programming
  Program Verification
  Environments and Tools

Promising Attacks on the Conceptual Essence
  Buy Vs. Build
  Requirement Refinement and Prototyping
  Great Designers

Summary
The What Now?

- A magic weapon to kill a mythical creature of mysterious origin
The What Now?

- A magic weapon to kill a mythical creature of mysterious origin
- Software projects becoming monsters of missed deadlines, blown budgets and flawed products
The What Now?

- A magic weapon to kill a mythical creature of mysterious origin
- Software projects becoming **monsters** of missed deadlines, blown budgets and flawed products
- Looking forward, *no silver bullets* can be seen.
The What Now?

► A magic weapon to kill a mythical creature of mysterious origin
► Software projects becoming **monsters** of missed deadlines, blown budgets and flawed products
► Looking forward, *no silver bullets* can be seen.
► No startling breakthroughs ahead, but there are several smaller ones.
Outline

Introduction

Essential Difficulties

Past Breakthroughs
  High-level Languages
  Time-sharing
  Unified Programming Environments

Hopes For The Silver
  High-level and OO-Programming
  Artificial Intelligence
  Automatic and Graphical Programming
  Program Verification
  Environments and Tools

Promising Attacks on the Conceptual Essence
  Buy Vs. Build
  Requirement Refinement and Prototyping
  Great Designers

Summary
Essential Difficulties I

- Software is not hardware
Essential Difficulties I

- Software is not hardware
- No two-fold gains every 18 months — no Moore’s Law
Essential Difficulties I

- Software is not hardware
- No two-fold gains every 18 months — no Moore’s Law
- No other technology with six O.o.M. performance increase in history. Ever.
Essential Difficulties II

Essence: The difficulties inherent in the nature of software
Essential Difficulties II

**Essence:** The difficulties inherent in the nature of software

**Accidents:** Difficulties that are not inherent, but problems nonetheless
Complexity

- Software is inherently complex
Complexity

- Software is inherently complex
- Similar parts are made into subroutines
Complexity

- Software is inherently complex
- Similar parts are made into subroutines
- Scaling software up (or down) does not linearly alter complexity
Complexity

- Software is inherently complex
- Similar parts are made into subroutines
- Scaling software up (or down) does not linearly alter complexity
- Simplified “scale models”, as in math or physics, won’t work
Complexity

- Software is inherently complex
- Similar parts are made into subroutines
- Scaling software up (or down) does not linearly alter complexity
- Simplified “scale models”, as in math or physics, won’t work
- So managing software projects is complex, and creates an *understanding burden*
No Silver Bullet

Adam Hasselbalch Hansen

Introduction

Essential Difficulties

Past Breakthroughs

High-level Languages
Time-sharing
Unified Programming Environments

Hopes For The Silver

High-level and OO-Programming
Artificial Intelligence
Automatic and Graphical Programming
Program Verification
Environments and Tools

Promising Attacks on the Conceptual Essence

Buy Vs. Build
Requirement Refinement and Prototyping
Great Designers

Summary

Conformity

- Software must be “intuitive” and “easy to use”
Conformity

- Software must be “intuitive” and “easy to use”
- Different designers create different interfaces
Conformity

- Software must be “intuitive” and “easy to use”
- Different designers create different interfaces
- Confirmation from a human standpoint is one thing ...
Conformity

- Software must be “intuitive” and “easy to use”
- Different designers create different interfaces
- Confirmation from a human standpoint is one thing...
- ... but interoperability is a whole other!
Changeability

Unlike physical products (like a building), software can be changed easily.
Changeability

- Unlike physical products (like a building), software can be changed easily
- New uses for which no one designed the software, emerges
Changeability

- Unlike physical products (like a building), software can be changed easily.
- New uses for which no one designed the software, emerges.
- New interoperability requirements emerges.
Invisibility

- No such thing as a “program blueprint”
Invisibility

- No such thing as a “program blueprint”
- Many visualization representations exist
Invisibility

- No such thing as a “program blueprint”
- Many visualization representations exist
- but none gives a clear view.
Invisibility

- No such thing as a “program blueprint”
- Many visualization representations exist
- But none gives a clear view.
- Flow control, data control, dependency patterns, data relationships, etc.
Outline

Introduction

Essential Difficulties

Past Breakthroughs
  High-level Languages
  Time-sharing
  Unified Programming Environments

Hopes For The Silver
  High-level and OO-Programming
  Artificial Intelligence
  Automatic and Graphical Programming
  Program Verification
  Environments and Tools

Promising Attacks on the Conceptual Essence
  Buy Vs. Build
  Requirement Refinement and Prototyping
  Great Designers
High-level Languages

- A factor five increase in productivity
High-level Languages

- A factor five increase in productivity
- Eliminates unnecessary complexity
High-level Languages

- A factor five increase in productivity
- Eliminates unnecessary complexity
- However, creates tool mastery burden
Time-sharing

- Preserves immediacy
Time-sharing

- Preserves immediacy
- Essentially: Code along while you compile
Time-sharing

- Preserves immediacy
- Essentially: Code along while you compile
- System response time shortened
Unified Programming Environments

- Integrated programming environments increase productivity
Unified Programming Environments

- Integrated programming environments increase productivity
- Easier to use programs together
Unified Programming Environments

- Integrated programming environments increase productivity
- Easier to use programs together
- E.g. UNIX pipes, unified file formats
Unified Programming Environments

- Integrated programming environments increase productivity
- Easier to use programs together
- E.g. UNIX pipes, unified file formats
- Each tool applies to all others without extra work
Outline

Introduction

Essential Difficulties

Past Breakthroughs
  High-level Languages
  Time-sharing
  Unified Programming Environments

Hopes For The Silver
  High-level and OO-Programming
  Artificial Intelligence
  Automatic and Graphical Programming
  Program Verification
  Environments and Tools

Promising Attacks on the Conceptual Essence
  Buy Vs. Build
  Requirement Refinement and Prototyping
  Great Designers

Summary
High-level and OOP

- Ada made a substantial difference in its day
High-level and OOP

- Ada made a substantial difference in its day
- With OOP comes abstract and hierarchial data types (e.g. C++ Templates)
High-level and OOP

- Ada made a substantial difference in its day
- With OOP comes abstract and hierarchial data types (e.g. C++ Templates)
- But this requires the problem at hand to rely heavily on these thing
High-level and OOP

- Ada made a substantial difference in its day
- With OOP comes abstract and hierarchial data types (e.g. C++ Templates)
- But this requires the problem at hand to rely heavily on these thing
- Only if nine tenths of a program is type specification underbrush will this have an O.o.M. gain.
High-level and OOP

- Ada made a substantial difference in its day
- With OOP comes abstract and hierarchial data types (e.g. C++ Templates)
- But this requires the problem at hand to rely heavily on these thing
- Only if nine tenths of a program is type specification underbrush will this have an O.o.M. gain.
- Which is doubtful...
Artificial Intelligence

- Using “intelligent programs” with generalized inference engines
Artificial Intelligence

- Using “intelligent programs” with generalized inference engines
- Using a rule base, it yields conclusions and offers advice
Artificial Intelligence

- Using “intelligent programs” with generalized inference engines
- Using a rule base, it yields conclusions and offers advice
- The essential prerequisite for such a system is a teacher
Automatic programming: State the problem, and the program writes itself, based on that problem.
Automatic and Graphical Programming

Automatic programming: State the problem, and the program writes itself, based on that problem

- It is, however, the solution method, not the problem, that needs to be described.

Automatic programming: State the problem, and the program writes itself, based on that problem

- It is, however, the solution method, not the problem, that needs to be described.
Automatic and Graphical Programming

**Automatic programming:** State the problem, and the program writes itself, based on that problem

▶ It is, however, the solution method, not the problem, that needs to be described.

**Graphical Programming:** Draw the program
Automatic and Graphical Programming

**Automatic programming**: State the problem, and the program writes itself, based on that problem

- It is, however, the solution method, not the problem, that needs to be described.

**Graphical Programming**: Draw the program

- Programmers draw flowcharts *after* writing programs
Automatic and Graphical Programming

**Automatic programming:** State the problem, and the program writes itself, based on that problem

- It is, however, the solution method, not the problem, that needs to be described.

**Graphical Programming:** Draw the program

- Programmers draw flowcharts *after* writing programs
- Constraints in screen real estate
Program Verification

- A lot of effort goes into testing and repairing bugs
Program Verification

- A lot of effort goes into testing and repairing bugs
- Program verification does not mean error-proof programs
A lot of effort goes into testing and repairing bugs.

Program verification does not mean error-proof programs.

It’s relatively easy to verify a program. The hard part is making sure it’s properly designed to begin with.
Program Verification

- A lot of effort goes into testing and repairing bugs.
- Program verification does not mean error-proof programs.
- It’s relatively easy to verify a program. The hard part is making sure it’s properly designed to begin with.
- “Beware of bugs in the above code; I have only proved it correct, not tried it.” – Donald E. Knuth, 1977
Environments and Tools

- Using proper tools to facilitate development
Environments and Tools

- Using proper tools to facilitate development
- Language specific editors, versioning systems, etc.
Environments and Tools

- Using proper tools to facilitate development
- Language specific editors, versioning systems, etc.
- Giving developers a nice working environment
Environments and Tools

- Using proper tools to facilitate development
- Language specific editors, versioning systems, etc.
- Giving developers a nice working environment
- Powerful workstations are not “the golden egg” (or indeed the Silver Bullet)
Outline

Introduction
Essential Difficulties
Past Breakthroughs
  High-level Languages
  Time-sharing
  Unified Programming Environments
Hopes For The Silver
  High-level and OO-Programming
  Artificial Intelligence
  Automatic and Graphical Programming
  Program Verification
  Environments and Tools
Promising Attacks on the Conceptual Essence
  Buy Vs. Build
  Requirement Refinement and Prototyping
  Great Designers

Summary
Buy Vs. Build

Before, business practices needed specialized software
Buy Vs. Build

- Before, business practices needed specialized software
- Now, business practices adapt to the available software
Buy Vs. Build

- Before, business practices needed specialized software
- Now, business practices adapt to the available software
- Generalized software, like spreadsheets, are extremely powerful and versatile
The client *does not know* what he wants
Requirement Refinement and Prototyping

▶ The client does not know what he wants
▶ Prototype it!
Requirement Refinement and Prototyping

- The client *does not know* what he wants
- Prototype it!
- *Grow* software. Don’t build it.
Requirement Refinement and Prototyping

- The client *does not know* what he wants
- Prototype it!
- *Grow* software. Don’t build it.
- A working system, all the way though
Great Designers

- Poor designs vs good designs can be caused by faulty process
Great Designers

- Poor designs vs good designs can be caused by faulty process
- Good designs vs great designs can not
Great Designers

- Poor designs vs good designs can be caused by faulty process
- Good designs vs great designs can not
- Take care of designers. Give them ideal working conditions. Travel funds, office size. Grow them.
There are no silver bullets in the foreseeable future.
Summary

▶ There are no silver bullets in the foreseeable future
▶ However, combining several methods into a sound software business strategy gets us a long way!
Questions