Software Design & Programming Techniques

Introduction

Prof. Dr. Klaus Ostermann
University of Tübingen

Based on slides by Prof. Dr. Mira Mezini

Introduction

- 1.1 This Course in a Broader Context
- 1.2 Software versus Hardware Engineering
- 1.3 Consequences of the Cheap Build
- 1.4 This Course in a Nutshell

1.1 This Course in a Broader Context

1.1 This Course in a Broader Context

- 1.1.1 Why Software Engineering

1.1.1 Why Software Engineering

- The term Software Engineering was coined in 1968
  At the NATO Software Engineering Conferences.
  1968 (Garmisch, Germany) and 1969 (Rom, Italy)

The NATO Software Engineering Conference

http://homepages.cs.ncl.ac.uk/brian.randell/NATO/N1968/index.html

Software Crisis

Discussion triggered by the **Software Crisis** in the 60th.
Projects were running over-budget or never delivered at all.
Software was inefficient and often did not meet requirements.

*[The **major cause of the software crisis** is] that **the machines have become several orders of magnitude more powerful!** To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*

Edsger Dijkstra, 1972

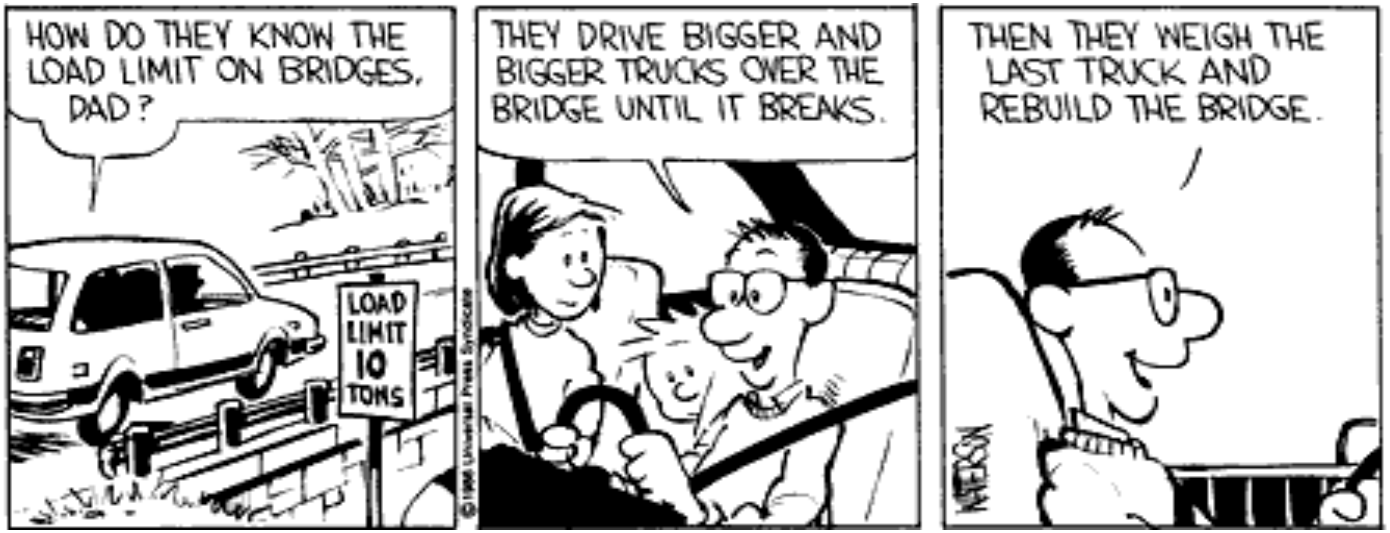
Software Crisis

- Since computers have become more powerful, more complex software is possible and **more complex programs (**more requirements**) are requested** to be developed!

- **Experience in creating big software systems was not available!**
  - How to keep code maintainable?
  - How to satisfy extensive and changing requirements?
  - How to work on code as a team?

Software Construction as Engineering

- Rather than ad-hoc problem solving…



- Goal of NATO Software Engineering Conferences:
  Adopt Hardware *) Engineering approaches for Software Engineering

*) In this lecture: Hardware is a physical, non-software product of engineering.

Software Construction as Engineering

CMU SEI (Software Engineering Institute) defines "Software Engineering" relative to "Engineering":

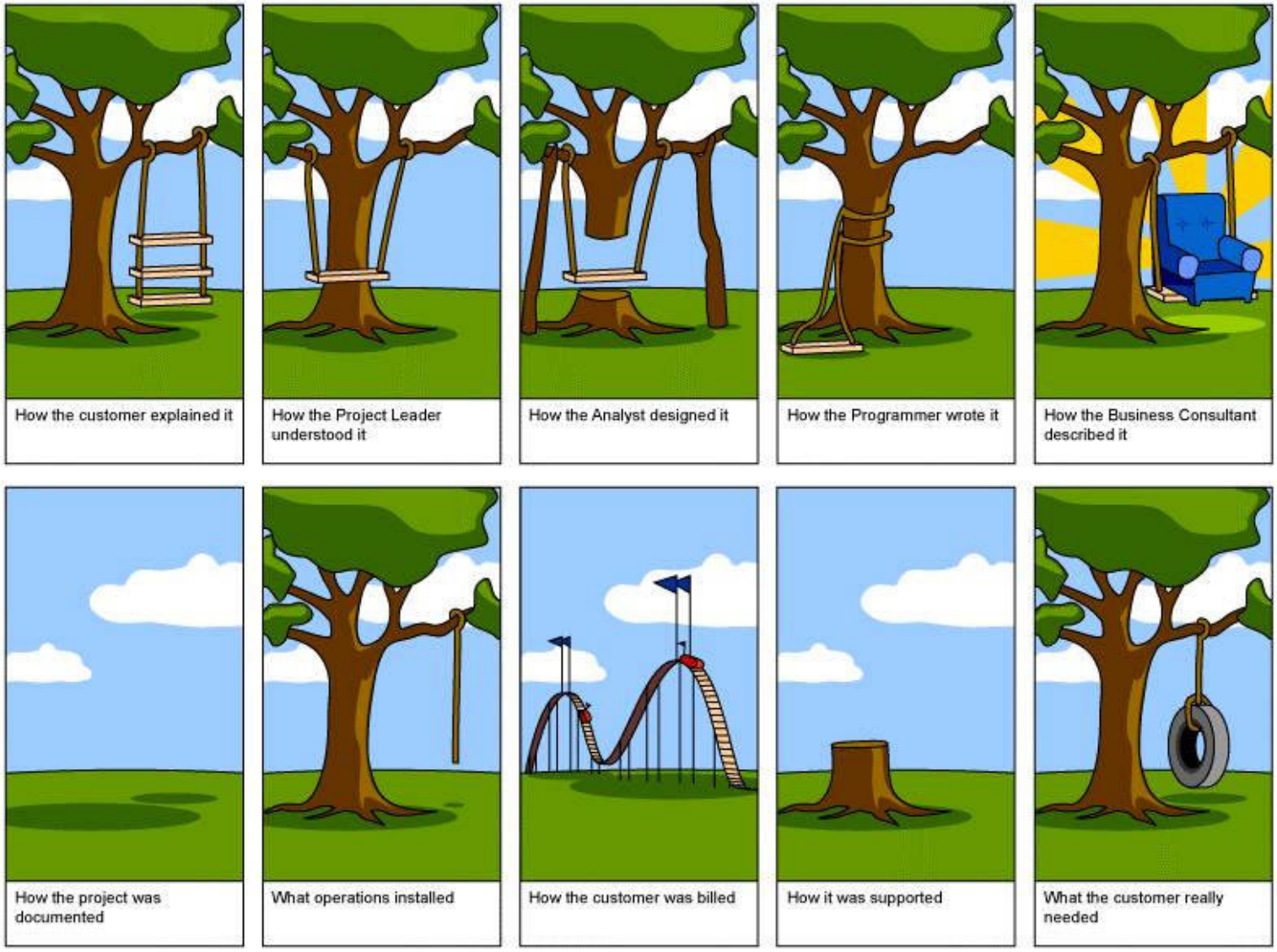
- Engineering is the systematic application of scientific knowledge in creating and building cost-effective solutions to practical problems in the service of mankind.
- **Software engineering** is that **form of engineering** that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems.

Software Engineering Today

- Software Engineering methods have made huge progress since the 60th.

- Today we have:
  - Sophisticated methods and tools for requirement engineering
  - Best practices, high-level architecture styles
  - Formal modeling techniques and visual and design notations
  - Potent high-level programming languages
  - Potent quality assurance methods, techniques and tools
  - Advanced process and project management methodologies
  - Advanced development environments
  - …

Software Engineering Today ... ☹



Software Engineering Today

- Software Engineering methods have made huge progress since the 60th.
- But: Advances in technology drive more complex requirements and business agility!
  Multimedia, Networking, Security, Embedded Systems, Usability, Reusability, Accessibility, Concurrency ...
  Continuously rapidly changing requirements…
- No wonder, we still observe the symptoms of the Software Crisis!
- A crisis that continues for 50 years is not a crisis but the normality!
- **Studying and advancing SE foundations, methods, notations, tools, is extremely important.**

1.2 Software versus Hardware Engineering

Hardware Engineering (Simplistic View)

- **Clear distinction between designing and building** (manufacturing) a product.
  Design and build done by different teams with different skills.

- When a **design effort is complete**, the **design documentation** is **turned over to** the **manufacturing** team.

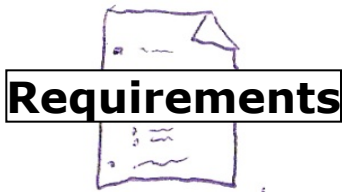
  Manufacturing team can proceed to build (lots of) the product, without further intervention of designers.

- **Manufacturing is a labor intensive expensive process**.

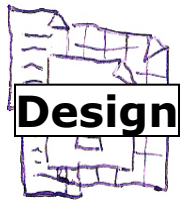
- Considerable time is spent in **validating and refining designs before they are build**.
  Simulations based on theoretical models are used to ensure the quality of the design.

Hardware Engineering (Simplistic View)

1st Phase: Analyzing
The problem to solve is analyzed and documented.



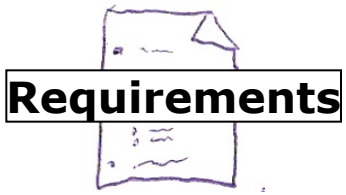
2nd Phase: Designing and Validation
Engineers translate the *requirements* into a detailed description of the solution using models and rigorously validate these models.



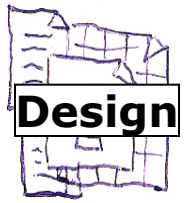
3rd Phase: Building
Workers build the *design* using appropriate tools and materials.

Example: Building a House

1st Phase: Analyzing
Customer specifies location and properties of the desired house.



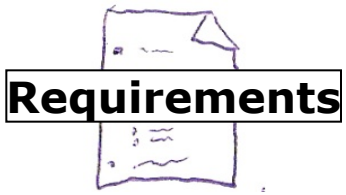
2nd Phase: Designing
Architect designs the house on paper using geometry and statics.



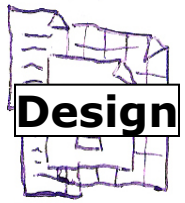
3rd Phase: Building
Construction Workers build the house.

Software Eng. Derived from Hardware Eng.

1st Phase: Analyzing
The problem to solve is analyzed and documented.



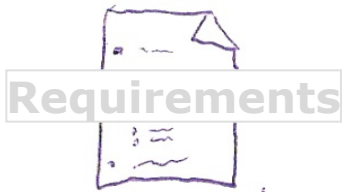
2nd Phase: Designing
Software Engineers design the software using abstract formal or semi formal models (formal specification, UML-based designs, etc).



3rd Phase: Building
Programmers implement the design using a programming language.

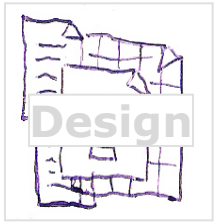
Software Eng. Derived from Hardware Eng.

1st Phase: Analyzing
The problem to solve is analyzed and documented.

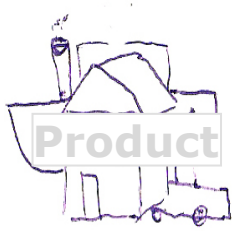


2nd Phase: Design
Software Engineers create abstract formal or semi-formal models (formal specification, UML-based designs, etc).

What do you think?



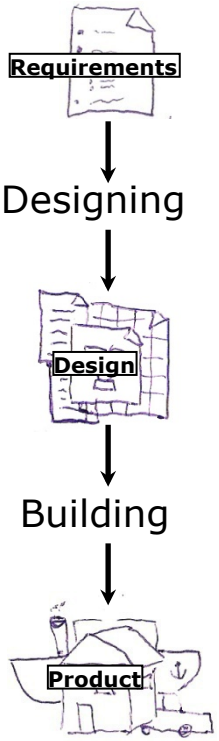
3rd Phase: Building
Programmers implement the design using a programming language.



To Code is Also to Design

Hardware

- Product is a physical object
- Built (mostly) by people
- Building
  - is expensive
  - labor & material intensive
  - slow
  - hard to redo



Software

- The product is the binary
- Built by compilers and linkers
- Building
  - is extremely cheap
  - automatic, no materials
  - very fast
  - easily to redo

Conclusions

- The executable code is the ultimate design document!
- Programming is about designing.
- Simulating software is unnecessary or even impossible.
- Agile techniques like trial-and-error are possible.

Code as THE Software Engineering Document …

Out of all the documentations that software projects normally generate, is there anything that can truly be considered an engineering document?

…I concluded that the only software documentation that actually seems to satisfy the criteria of an engineering design ***is the source code listings****.*

…there is one consequence of considering code as software design that completely overwhelms all others. It is so important and so obvious that it is a total blind spot for most software organizations. ***This is the fact that software is cheap to build****.*

Programming is not about building the software. Programming is about designing software.

Jack Reeves, To Code is to Design, C++ Report 1992

Peter Naur on the Design-Production Separation …

The distinction between design and production is essentially a practical one, imposed by the need for a division of the labor. In fact, there is ***no essential difference between design and production****, since even the production will include decisions which will influence the performance of the software system, and thus properly belong in the design phase.*

Peter Naur, NATO Software Engineering Conference 1968 http://homepages.cs.ncl.ac.uk/brian.randell/NATO/

Interim Conclusion

- The SW industry is not likely to find solutions to its problems by trying to blindly emulate HW developers.

  BTW, complex HW engineering is not as free of bugs…bridges collapsed, airliners fallen out of the sky, consumer products have been recalled …

- As CAD and CAM systems have helped HW designers to create more and more complex designs, as software is becoming more and more important, system engineering is becoming more and more like SW development.

1.3 Consequences of the Cheap Build

1.3 Consequences of the Cheap Build

- 1.3.1 Complexity and Change are Invariants
- 1.3.2 Everything is Part of the Design
- 1.3.3 Auxiliary Documentation
- 1.3.4 Development Processes Should be Agile
- 1.3.5 Programming Languages as Design Languages

1.3.1 Complexity and Change are Invariants

- **SW designs** tend to be incredibly **large and complex**
  - Typical commercial SW products have designs that consist of hundreds of thousands of lines. Many run into the millions.
  - More complex designs are targeted every day.

- **SW designs** are **constantly evolving**
  - The current design may only be a few thousand LOC … many times that may actually have been written over the life of the product.
  - A software product spends around 80% of its lifetime in maintenance.

Designing for organizing complexity and facilitating change is the key to support maintainability

1.3.2 Everything is Part of the Design

- There are different levels of design
  - High-level architectural design
  - Class-level design
  - Low(implementation)-level detailed design

- We need good SW design at all levels.
- The better the early design, the easier to do detailed design, the better its quality.

- Yet, the **high-level design is not a complete software design but just a structural framework for the detailed design.**

The Final Design is Code

- **SW design is not complete until it is coded and tested!**

- Designers should use anything that helps: user stories, Z specs, UML diagrams, etc.
- These are all useful notations for facilitating the design process and as auxiliary documentation.
- **Yet, they are not a software design!**

- **The real design** will be created using some programming language … ultimately the final design **is code**!

The Final Design is Code

- Designs should be coded as they are derived and be refined when necessary.
  - The process of rendering the design in code reveals oversights and the need for additional design effort.
  - The earlier it occurs, the better the design will be.

- The detailed design will ultimately influence (or should be allowed to influence) the high-level design.
- Refining all the aspects of a design is a process that should be happening throughout the design cycle.

Testing and Debugging is Design

- If we consider source code as design, we see that software engineers (just like other engineers) also do a considerable amount of validating and refining their designs.

- Most SW designers do not call it engineering … rather testing and debugging, i.e., do not consider testing and debugging as real "engineering" due to refusal of SW industry to accept code as the actual design.

Testing is not just concerned with getting the current design correct, it is part of refining the design.

1.3.3 Auxiliary Documentation

… Obviously the actual design documents (entailed in code) are the most important, but not the only one that must be produced.

Auxiliary documentation (directly associated with the design process) is as important for a SW project as it is for a HW project.

Auxiliary Documentation Should …

- … **capture information from the problem space** that did not make it directly into the design.
  - Inventing SW concepts to model concepts in a problem space requires an understanding of the essential problem space concepts.
  - Usually this process involves information that does not directly end up being modeled in the SW space.

- … **document** those **aspects of the design that are difficult to extract directly from the design itself**.
  - Many of these aspects are best depicted graphically. This makes them hard to include as comments in the source code.
  - This is not an argument for a graphical software design notation instead of a programming language.
  - Not different from the need for textual descriptions to accompany the design documents of other engineering disciplines.

Up-to-Date Auxiliary Documentation

- Keeping auxiliary documentation up to date manually is difficult.
  - This is an argument for more expressive programming languages.
  - It is an argument for keeping auxiliary documentation to a minimum and as informal as possible until as late in the project as possible.
- Auxiliary documentation written after code is more accurate.
  - Only the design reflected in code has been refined during the build/test cycle!
  - The probability of the initial design being unchanged during this cycle is inverse to the number of modules and programmers on a project.
- Ideally, software tools that post-process a source code design and generate auxiliary documentation should be available.

1.3.4 Development Processes Should be Agile

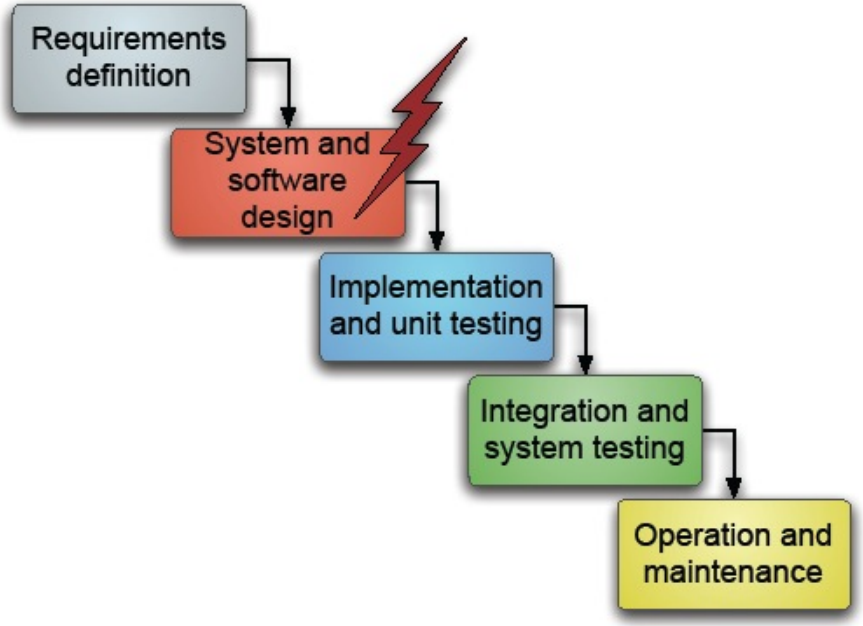
TO CODE IS TO DESIGN.

Which doesn't mean that you should start coding right away!

"Traditional" Software Development Processes

- Software development process (just process when context is clear): The set of activities and associated results that produce a software product

- Traditional process segregate different phases
  - High-level design completed and frozen before any code is written
  - Testing and debugging just to weed out construction mistakes



"Traditional" Software Development Processes

- In between are programmers – the construction workers of the SW industry …

  "… if we could just get programmers to quit "hacking" and "build" the designs as given to them (and in the process, make fewer errors) then software development might mature into a true engineering discipline"

- Not likely to happen as long as the process ignores the engineering and economic realities.

Instead of forcing SW development to conform to an incorrect process model, we need processes that help rather than hinder efforts to produce better software.

A Good Development Process …

- … **recognizes programming as a design activity** and **does not hesitate to code** when coding makes sense.
- … **recognizes testing and debugging as design activities** - the SW equivalent of design validation and refinement of other engineering disciplines - and **does not shorten the steps.**
- … **recognizes that there are other design activities** - top level design, module design, class design, etc. - and deliberately includes the steps.
- … recognizes that all design activities interact and **allows the design to change as various steps reveal the need.**

1.3.5 Programming Languages as Design Languages

If to code is to design, then
Programming Languages are Design Languages too!

Programming Languages are Design Languages

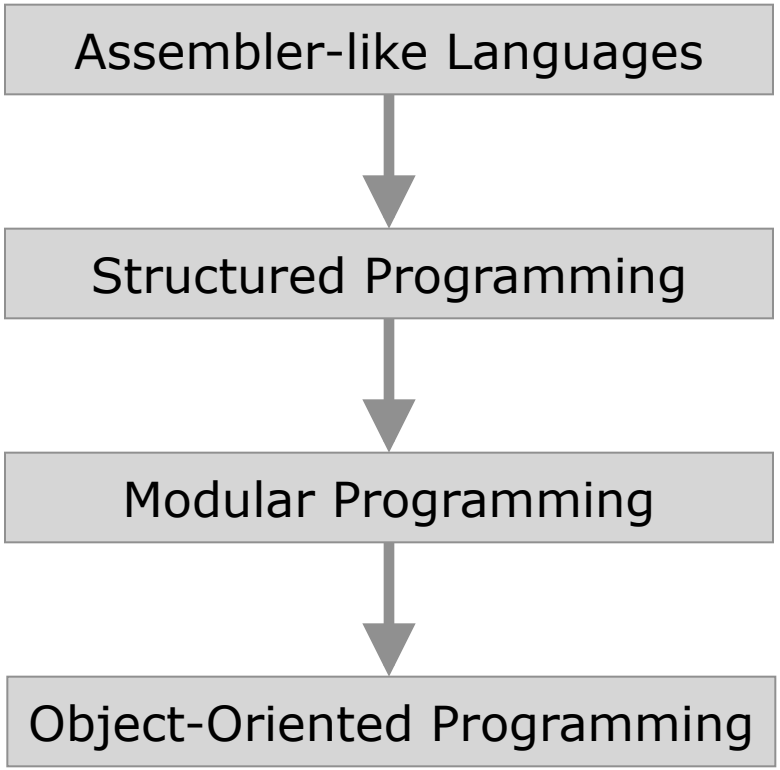
- Programming languages **outperform other design notations** in their capability to **express both high-level and detailed design**
- Hence, high-level design notations have to be translated into the target programming language before detailed design can begin.
- The translation is time consuming and error prone since a design notation may not map cleanly into the programming language of choice
- Programmers often go back to the requirements and redo the high-level design, coding it as they go.

Desired:

Unified design notation suitable for all levels of design, i.e., programming languages that are suitable for capturing high-level design.

Making Code Look Like Design

In the following we examine the development of programming languages as a process that was driven by the need to make programming languages capable of capturing higher-level designs …



1.3.5.1 "Designing" in Pseudo-Assembler

```
i = 1
TEST:  if i< 4
       then goto BODY
       else goto END
BODY:  print i
i = i + 1
       goto TEST
END:
```

What does this program do?

It just prints "123" out!

```
i = 1
LOOP:  print i
i = i + 1
       if i< 4 goto LOOP
END:
```

What does this program do?

It just prints "123" out!

GOTOs with Style …

```
i = 1                          INITIALIZE COUNTER
LOOP:  print i                 LOOP START
       i = i + 1               INCREASE COUNTER
       if i < 4 goto LOOP      LOOP CONDITION
END:
```

- **Good style**!
  - Clear structure. No crossing gotos. Better names.
  - Code structure reflects closer to what we want to express.
    "Print out i, i smaller than 4"
  - Code is easier to understand, debug, change.

- But: **Style can only be recommended, not enforced!**

1.3.5.2 Structured Programming

- Developed in the 60th to support better structuring of code.
- `Gotos` replaced by loops (`while, for`) and conditionals (`if/else`).
- New words, new grammars, **new abstractions** enable to directly express **looping/conditional** computations, instead of emulating them by jumps.

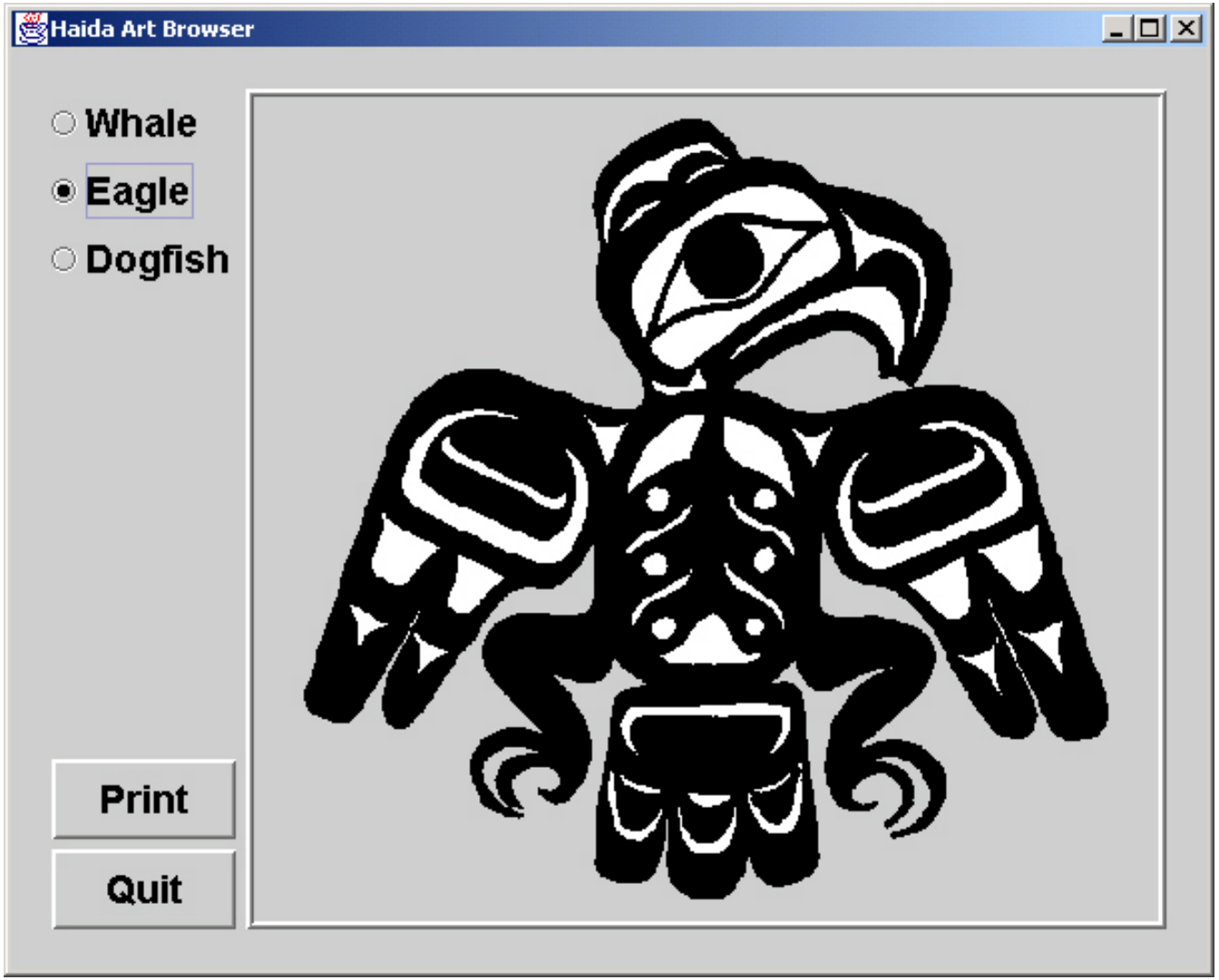
```
i = 1
while ( i < 4 ) {
        print(i)
        i = i + 1
}
```

Style gets enforced!
There is no way to write crossing gotos!

- Further, **procedures** were introduced to support user-defined abstractions.

Better Languages, More Challenging Tasks…

Let's "develop" a simple image browser with structured programming…



Structured Code for Image Browser

```
main () {
draw_label("Art Browser")
   m = radio_menu(
      {"Whale", "Eagle", "Dogfish"})
   q = button_menu({"Quit"})
   while ( ! check_buttons(q) ) {
      n = check_buttons(m)
draw_image(n)
   }
}
```

```
set_x (x) {
   current_x = x
}
```

```
draw_circle (x, y, r) {
   %%primitive_oval(x, y, 1, r)
}
```

```
set_y (y) {
   current_y = y
}
```

```
radio_menu(labels) {
   i = 0
   while (i < labels.size) {
      radio_button(i)
      draw_label(labels[i])
      set_y(get_y()
         + RADIO_BUTTON_H)
      i++
   }
}
```

```
radio_button (n) {
   draw_circle(get_x(),
      get_y(), 3)
}
```

```
get_x () {
   return current_x
}
```

```
get_y () {
   return current_y
}
```

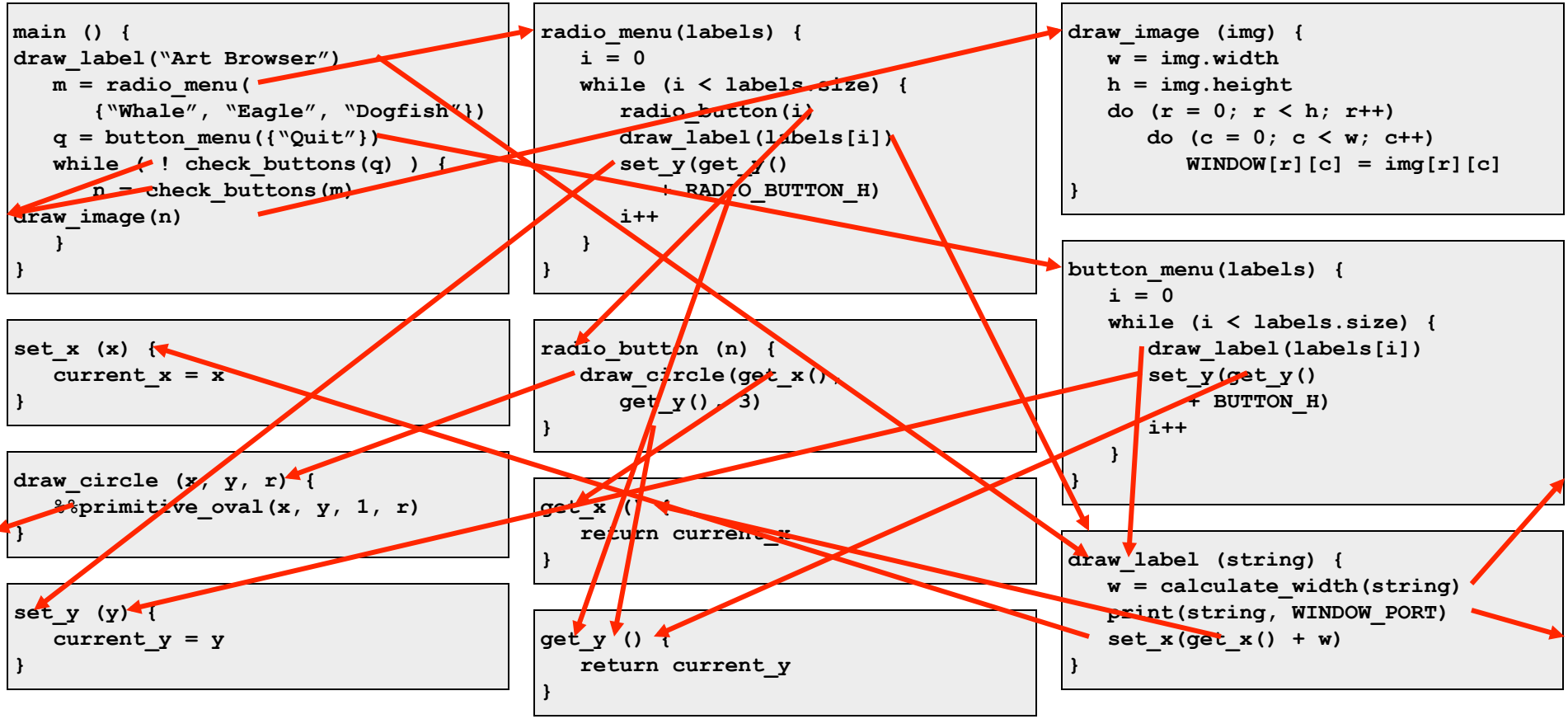
```
draw_image (img) {
   w = img.width
   h = img.height
   do (r = 0; r < h; r++)
      do (c = 0; c < w; c++)
         WINDOW[r][c] = img[r][c]
}
```

```
button_menu(labels) {
   i = 0
   while (i < labels.size) {
      draw_label(labels[i])
      set_y(get_y()
         + BUTTON_H)
      i++
   }
}
```

```
draw_label (string) {
   w = calculate_width(string)
   print(string, WINDOW_PORT)
   set_x(get_x() + w)
}
```

Code structured into procedures.

Retracting the Structured Code



Code is structured, but procedures are not!

Structured Programming with Style

```
main ()
```

```
gui_radio_button(n)
gui_button_menu(labels)
gui_radio_menu(labels)
```

```
graphic_draw_image (img)
graphic_draw_circle (x, y, r)
graphic_draw_label (string)
```

```
state_set_y (y)
state_get_y ()
state_set_x (x)
state_get_x ()
```

- Group procedures according to the functionality they implement and the state that they access
  - E.g. by naming conventions …
  - The code is closer to what we want to express.
    "main calls gui, gui calls graphic to draw, …"
  - The code is easier to understand, debug and change.

- And again: **Can only be recommended, not enforced!**

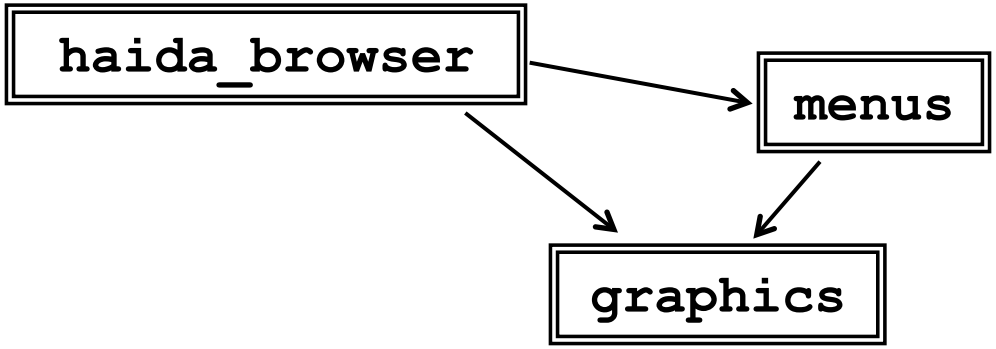
1.3.5.3 Modular Programming

- Introduced **modules,** higher-level units.
- Programming language mechanisms for supporting information hiding: Interface hides module internals.

```
module gui {
    exports:
        radio_menu(labels)
        button_menu(labels)
        check_buttons(menu)
}
```

Module-Based Abstraction

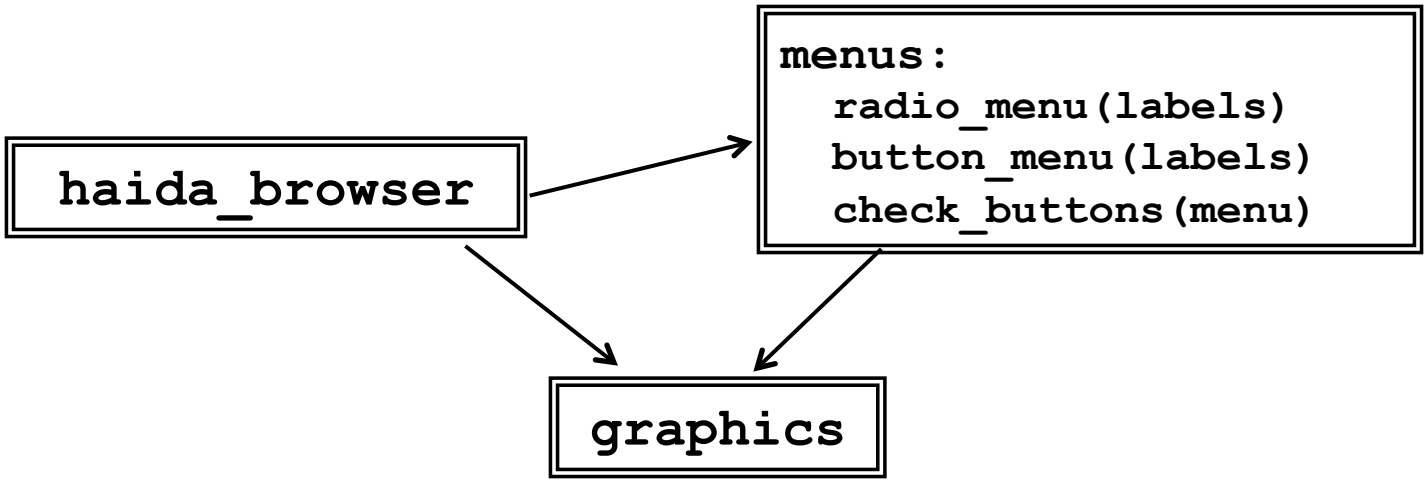
- Modules introduce higher-level abstractions!
- One can handle a whole module as if it was its interface.



- Abstraction enables to:
  - look at the overall structure of the system (architectural thinking)

Module-Based Abstraction

- Modules introduce higher-level abstractions!
- One can handle a whole module as if it was its interface.

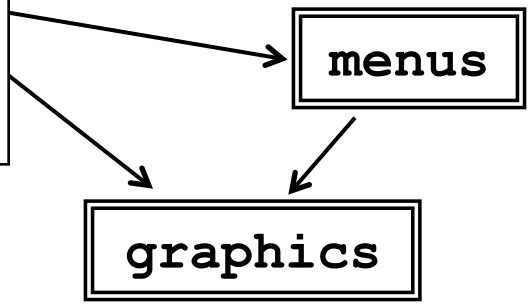


- Abstraction enables to:
  - look at the overall structure of the system (architectural thinking)
  - zoom in individual units as needed

Module-Based Abstraction

- Modules introduce higher-level abstractions!
- One can handle a whole module as if it was its interface.

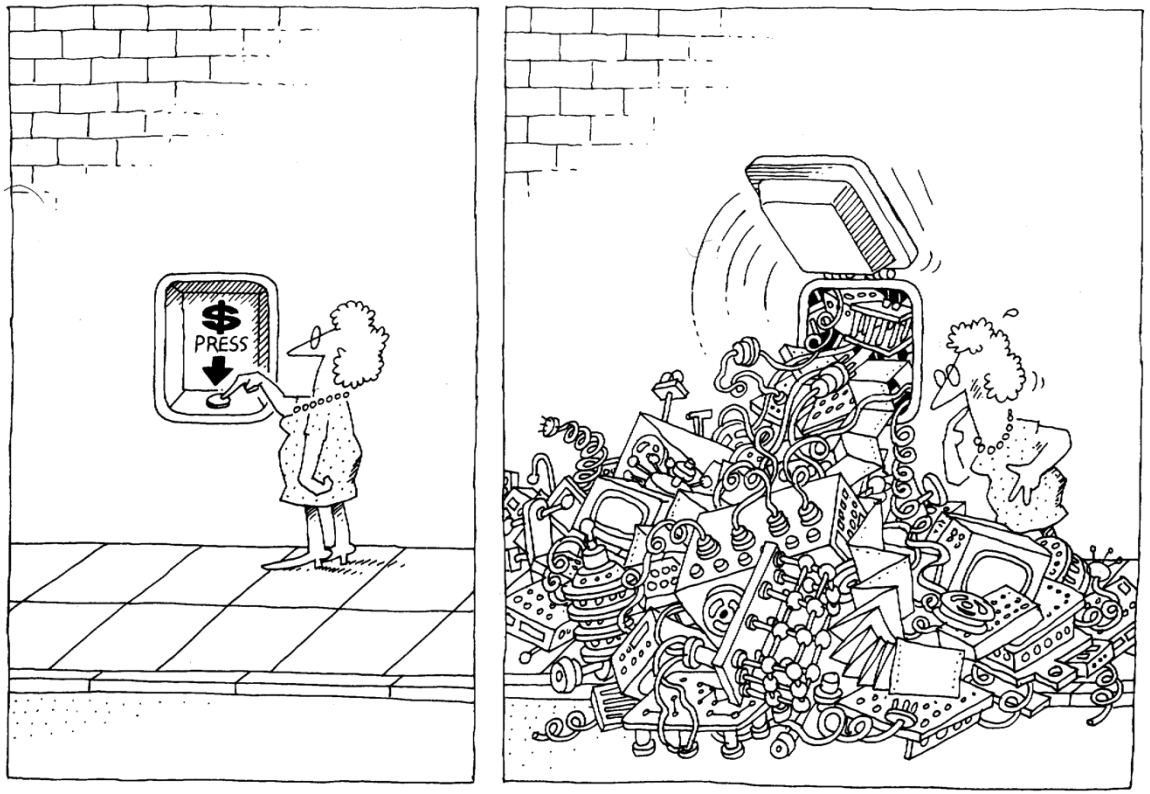
```
haida_browser:
   main () {
    draw_label("Haida Browser");
    m = radio_menu(
          {"Whale", "Eagle", "Dogfish"});
    q = button_menu({"Quit"});
    while ( ! check_buttons(q) ) {
      n = check_buttons(m);
      draw_image(n);
    }
  }
```



- Abstraction enables to:
  - look at the overall structure of the system (architectural thinking)
  - zoom in individual units as needed
  - with more or less details

Abstraction is the Key to Managing Complexity

- Abstraction means ignoring detailed information to focus on relevant information.
- Abstraction mechanisms of programming languages enable us to compose primitive units into more complex ones, give the latter a name and handle them as being primitives in the next composition level.



The task of the software development team is to engineer the illusion of simplicity.

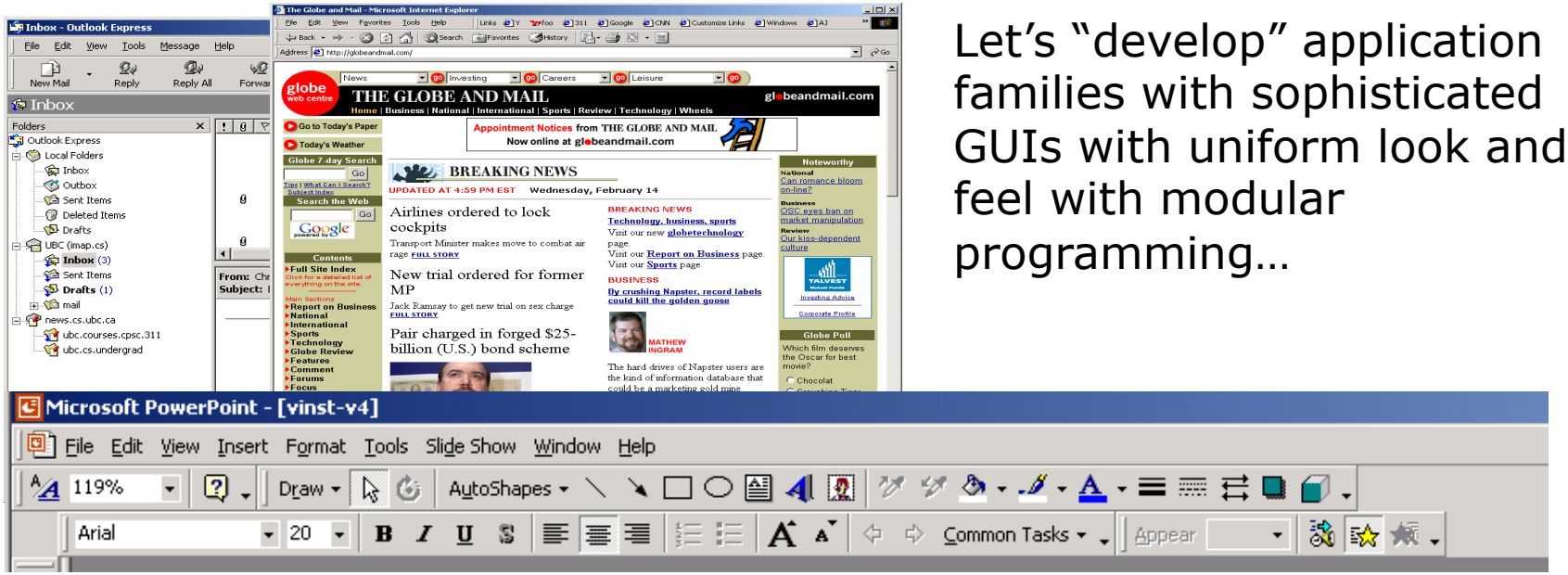
Abstraction is the Key to Managing Complexity

- Makes the code easier to understand, debug and change.
  - Allows structured organization of code.
  - Ability to ignore details.

- Makes the code closer to what we want to express.
  "Write what you mean"

- In essence: **Abstraction mechanisms enable us to code and design simultaneously!**

Better Languages, More Challenging Tasks…



Let's "develop" application families with sophisticated GUIs with uniform look and feel with modular programming…

Great variability: checkbox button, toolbar button, toggle button, radio button …

Modeling variability with modular programming appeared complex…

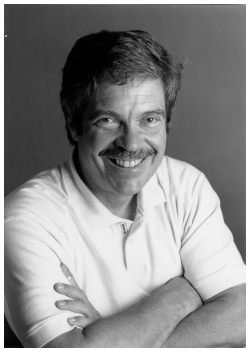
The need to better model variability eventually lead to object-oriented programming.

1.3.5.4 Object-Oriented Programming

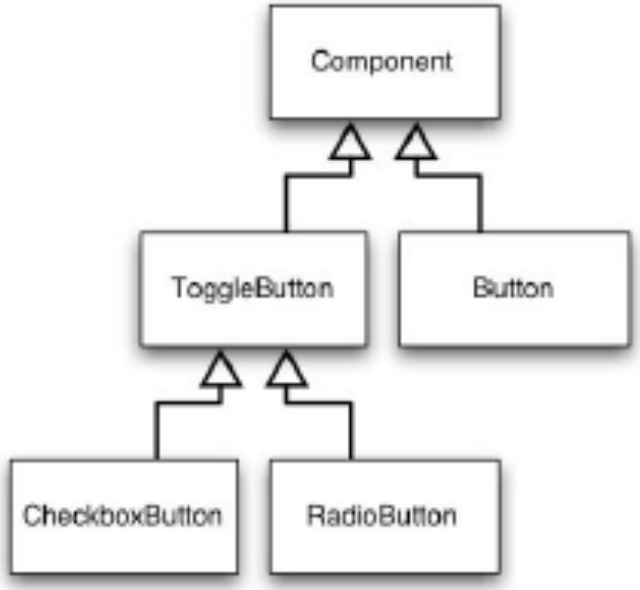
- Introduces new abstractions mechanisms:
  Classes, Inheritance, Subtype polymorphism.

- Roots in the 60ties!

Simula
Dahl & Nygaard
64, 68



Smalltalk
Alan Kay
70 – 80



- Leading programming paradigm today!
  - Recently in combination with functional programming

Popularity of OO Languages

- OO languages are popular because they make it easy to **design software and program at the same time.**

- They allow us to more directly express high level information about design components abstracting over differences of their variants
- Makes it easier to produce the design, and easier to refine it later
- With stronger type checking, they also help in the process of detecting design errors.
- This results in a more robust design, in essence a better engineered design.

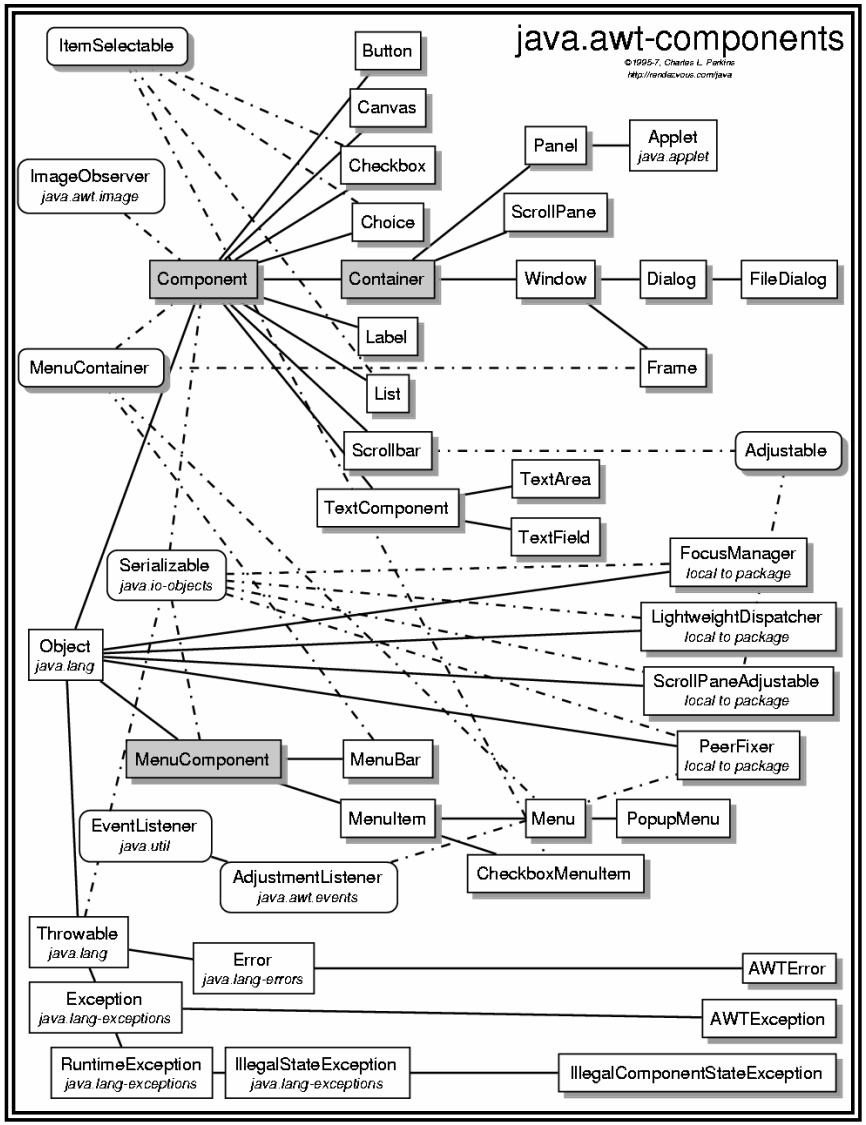
1.3.5.5 Summary

... improvements in programming techniques and programming languages in particular are overwhelmingly more important than anything else in the software business …
… programmers are interested in design … when more expressive programming languages become available, software developers will adopt them.

Jack Reeves, To Code is to Design, C ++ Report 1992

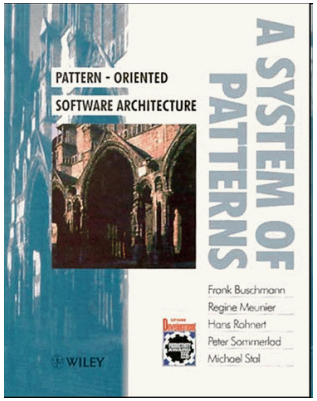
Programming Paradigms are not a Panacea

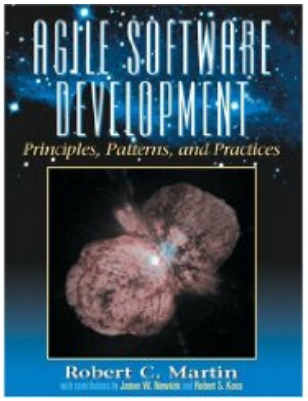
- Accessibility of object-oriented programming drives more complex designs!
- Programming languages are powerful tools, but cannot guarantee good designs.
- Programming still needs to be done properly to result in good code.
- Human creativity remains the main factor.

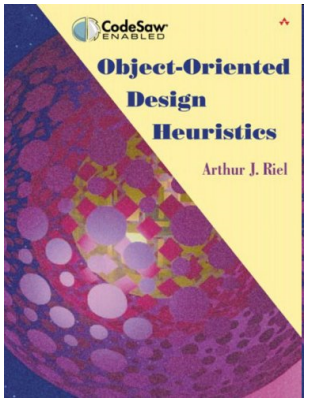


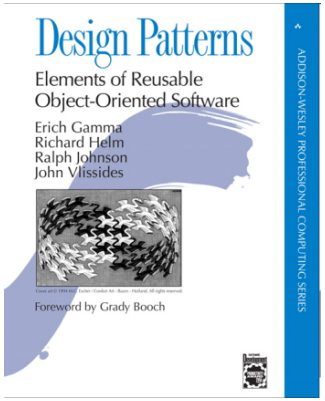
The Imperative of Good Style

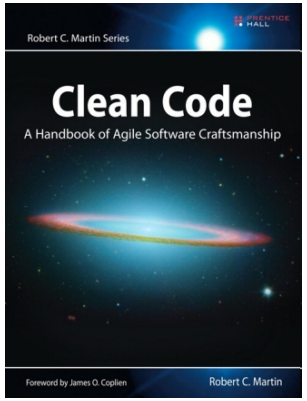
- **We still need good style to cope with complexity!**
- Help is provided through established practices and techniques, design patterns and principles.

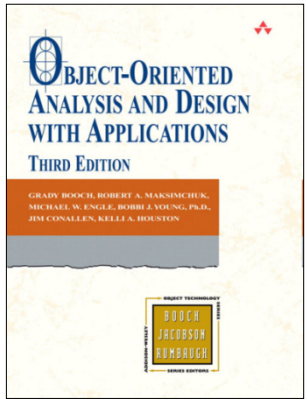


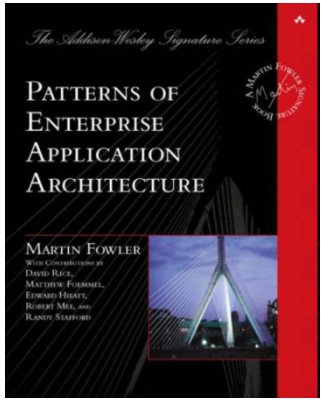












The Imperative of Good Style

- But, keep in mind…

- **Good style can only be recommended, not enforced!**
- **Style rules will be more effective if they can be expressed and abstracted over within the code.**
  - **Styles and patterns as drivers for language design**

1.4 This Course in a Nutshell

Consequences of the Cheap Build for this Course

- Designing means for us structuring code in modular way so as to support managing complexity and continuous change.

  We will actually talk about design expressed in object-oriented and functional languages (mostly Java, with some Scala and Haskell thrown in).

- We will adopt an agile design process to accommodate change in the design process.

- We will adopt test-driven development, as we consider testing to be part of design!

- Languages as design notations will be in focus as much as design principles and styles.

Tentative Content Overview

- Class Design Principles
- Design Patterns
- Frameworks
- Package Design Principles
- Aspect-Oriented Programming
- Generic Programming
- Domain-Specific Languages

Organization

- Tentative Homepage: https://github.com/klauso/SDPT2014
- Exercises
  - Start next week (Friday, Oct 24)
  - Exercises have to be submitted regularly to be admitted for the exam
  - Details will be determined next week
- Lecture
  - No lecture next week; next lecture is on Wed, Oct 29

On Questions

- **Please** ask questions during the lectures!
  - The main advantage of the lecture over books is that I can answer your questions!
  - I'll get an impression of whether I'm too slow or too quick.
  - Questions slow me down. They make lectures more interesting to both you and me.
  - It is your time. Don't waste it in the lecture if you rather want to do something else.
  - Get the most out of this course!

Summary

- Software engineering (SE) is about systematic software development.
- SE compared to other engineering disciplines:
  The cheap build process as the distinguishing feature.
- Consequences of the cheap build process:
  SW designs are complex and change is a constant,
  Everything is part of the design, to code/to test is to design,
  Development processes should be agile,
  Programming languages are design languages too.
- Goals of the course
  Learn programming techniques, design principles and patterns for structuring code to manage complexity and accommodate change.
  Get to know about new trends in language technology beyond OO.