Winter Semester 15/16

# Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

---

Programming Languages and Design Principles

---

# Making Code Look Like Design

# “Designing” with Pseudo-Assembler

What does the following program do?

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

# “Designing” with Pseudo-Assembler

What does the following program do?

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

Style can only be recommended, not enforced!

# Designing with Structured Programming Languages

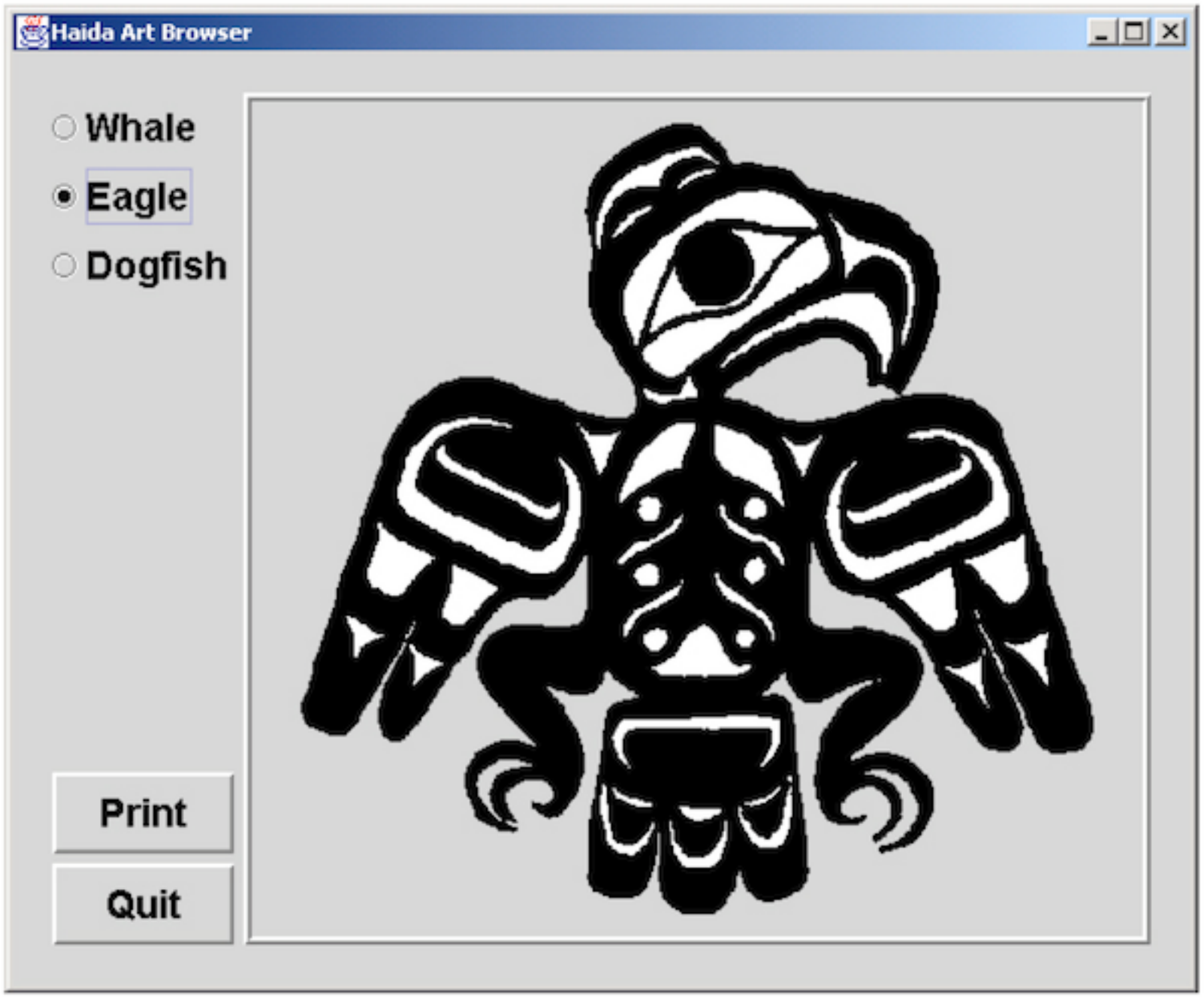
What does the following program do?

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

Style gets enforced!

# Better languages, More challenging tasks…

A simple image browser with structured programming



# Code for Image Browser Structured into Procedures

Try to identify which method calls which method!

```
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)
}
```

# Structured Programming with Style

```
main()
```

```
gui_radio_button(n)

gui_button_menu(labels)

gui_radio_menu(labels)
```

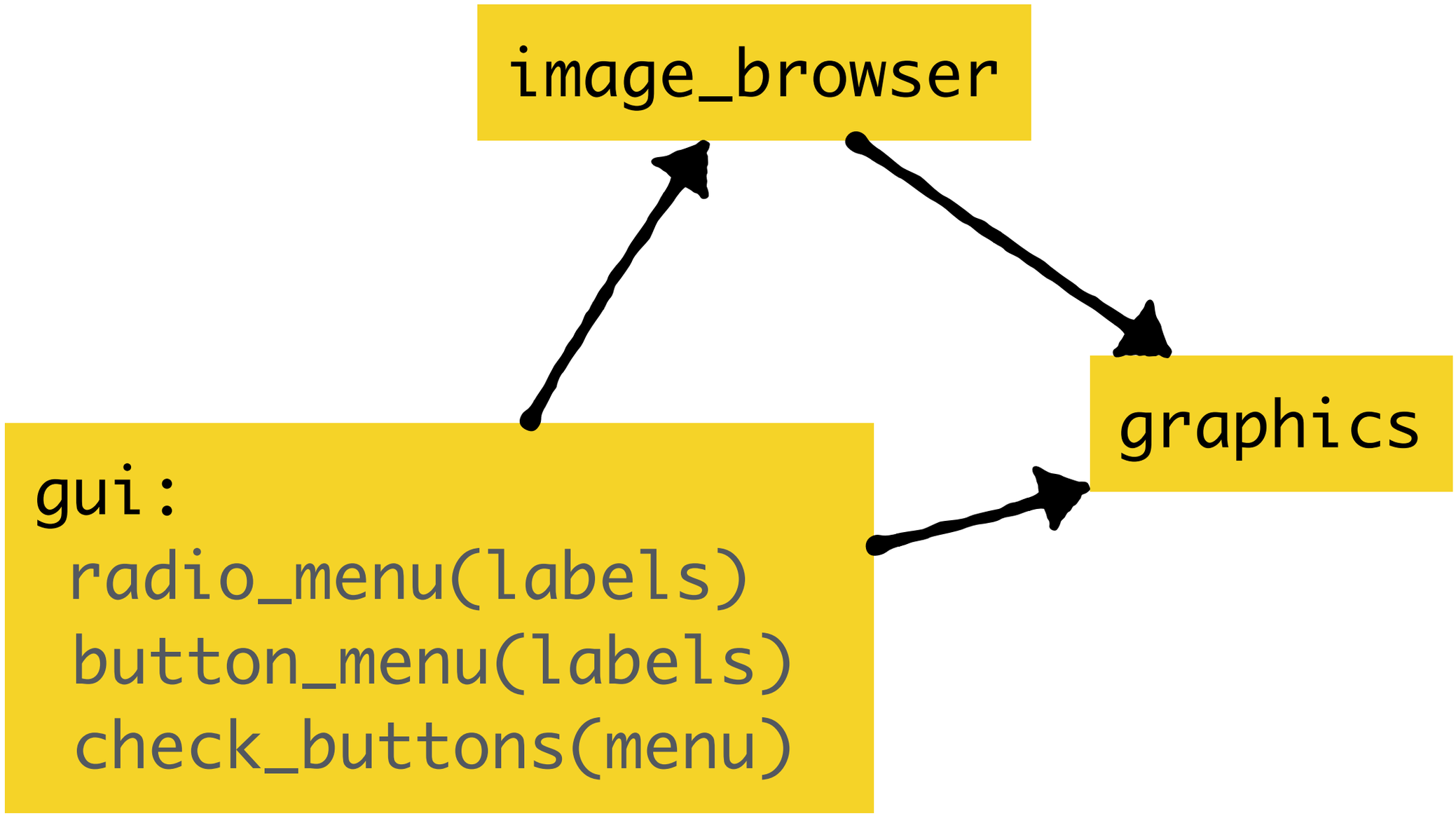
```
graphic_draw_image (img)

graphic_draw_circle (x, y, r)

graphic_draw_label (string)
```

```
state_set_y (y)

state_get_y ()

state_set_x (x)

state_get_x ()
```

# Designing with Modular Programming Languages

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

# Module-based Abstraction



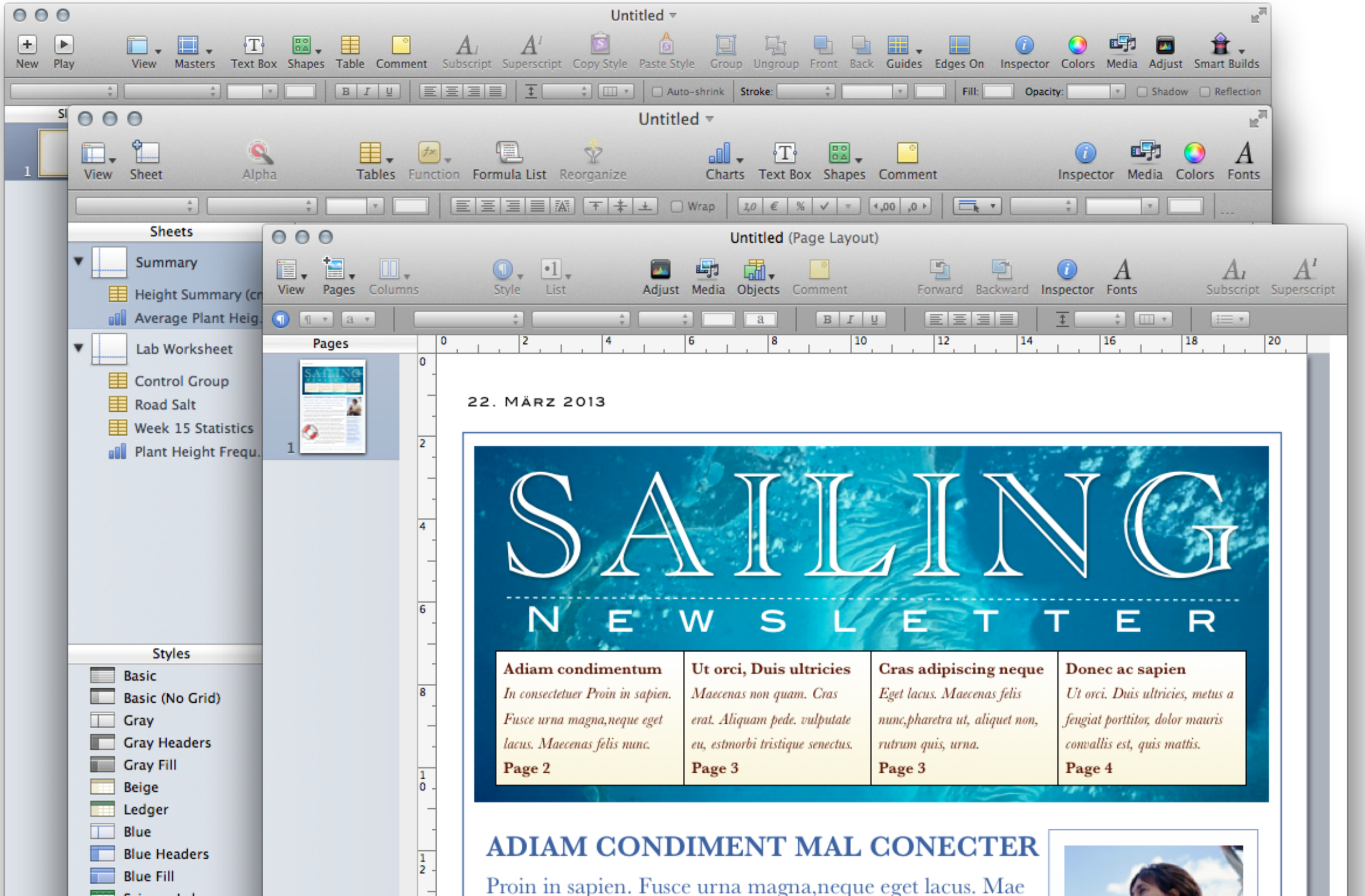
# Abstraction mechanisms enable us to code and design simultaneously!

"Write what you mean."

*"The significant problems we face cannot be solved at the same level of thinking we were at when we created them."*

–Einstein

# Let’s “develop” application families with sophisticated GUIs with uniform look and feel with modular programming…



# Designing with Object-Oriented Programming Languages

Object-oriented programming languages introduce new abstraction mechanisms:

- classes
- inheritance
- subtype polymorphism
- virtual methods

The roots of object-oriented programming languages are in the sixties.

Allan Kay,
Smalltalk 70 - 80

Dahl and Nygaard,
Simula 64, 68

*[…] 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

# Designing with Functional, Object-Oriented Programming Languages

Fill an array with n Person objects where each Person has a unique id.

Code:

```
case class Person(id : Int)

var ids = 0
def nextId() : Int = { val id = ids ; ids+= 1; id }

Array.fill(15){ new Person(nextId()) }
```

Result:

```
=> Array[Person] = Array(Person(0), …, Person(14))
```

# Designing with Functional, Object-Oriented Programming Languages **with a Flexible Syntax**

Creating an abstraction to express that we want to repeat something n times.

```
def repeat[T: scala.reflect.ClassTag](times: Int)(f: ⇒ T): Array[T] = {
  val array = new Array[T](times)
  var i = 0
  while (i < times) { array(i) = f; i += 1 }
  array
}
```

Now, we can express that we want to read in two values from the command line using our new control-abstraction.

```
val values = repeat(2) { System.in.read() }
```

# Designing with Functional, Object-Oriented Programming Languages with a Flexible Syntax vs. Explicit Language Features

## Java's native try-with-resources statement

```
File tempFile = File.createTempFile("demo", "tmp");
try (FileOutputStream fout = new FileOutputStream(tempFile)) {
  fout.write(42);
}
```

## Using Scala's language features enables us to define a new control structure that resembles Java's try-with-resources statement.

```
def process[C <: Closeable, T](closable: C)(r: C ⇒ T): T = {
    try { r(closable) }
    finally { if (closable != null) closable.close() }
}

val tempFile = File.createTempFile("demo", "tmp");
process(new java.io.FileOutputStream(tempFile)) { fout ⇒
  fout.write(42);
}
```

# Programming Languages with notable Features:

- RUST avoids buffer errors statically (based on ownership)
  Graydon Hoare, 2009

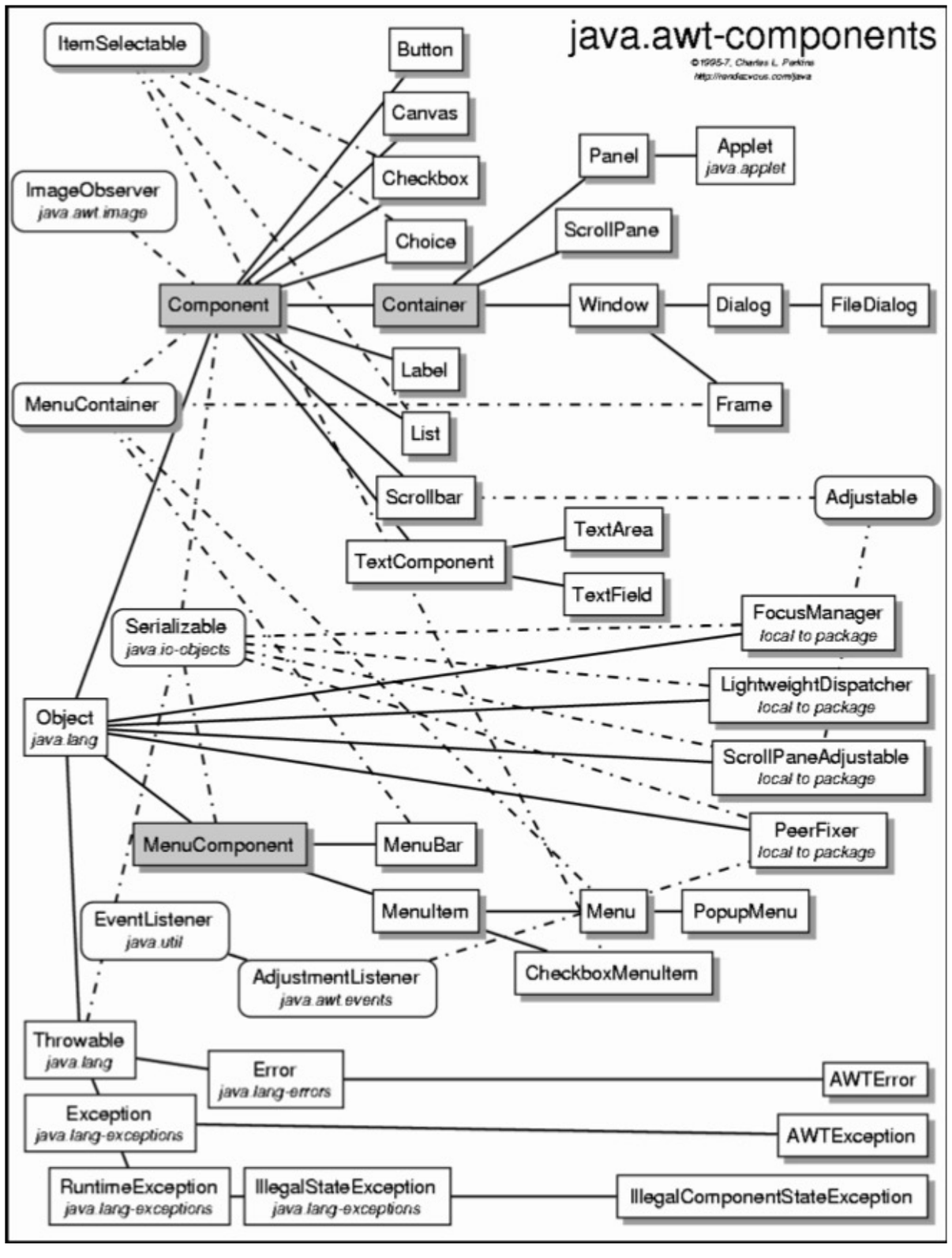
- Checked C avoids buffer errors statically and dynamically (introduces new checked pointer types)
  David TardiF, June 2016 (v 0.5)

- Perl (3) implements a taint mode to avoid injections dynamically
  Larry Wall, 1987

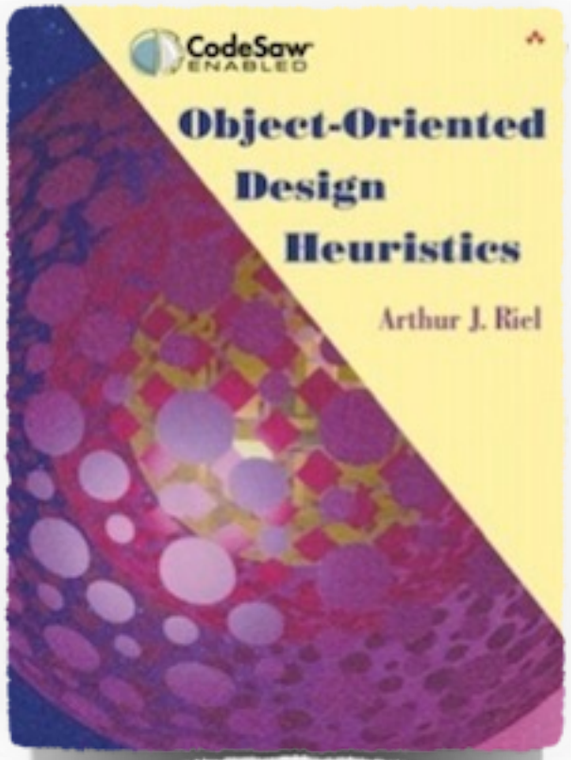
- Java made first steps to avoid cryptographic issues with the “Cryptography Architecture”

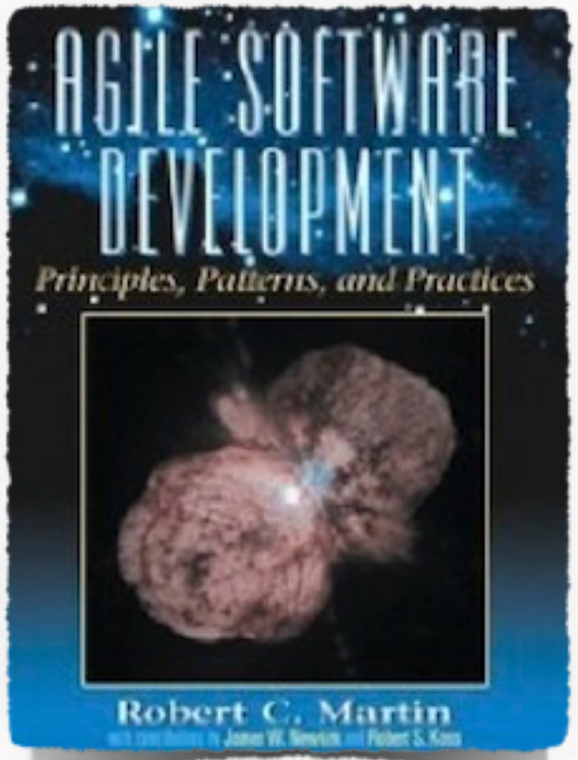
- GO, Erlang,… have advanced support for concurrency

# Programming Languages are not a Panacea

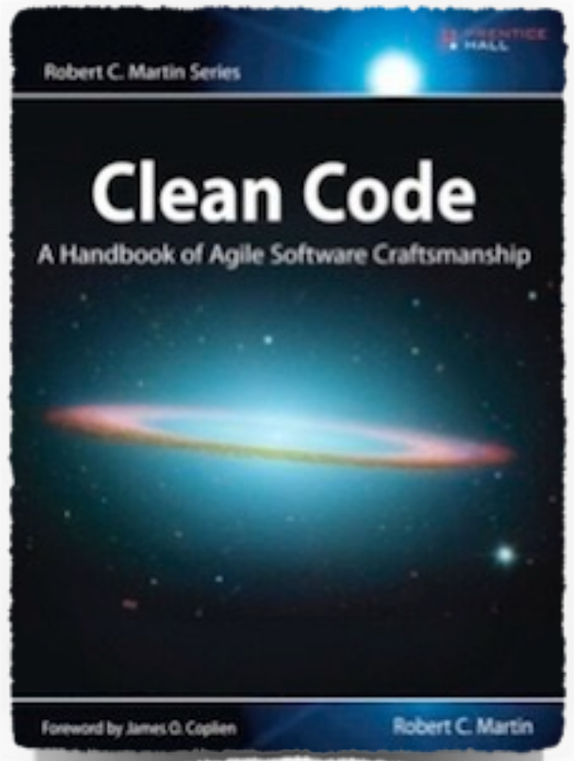


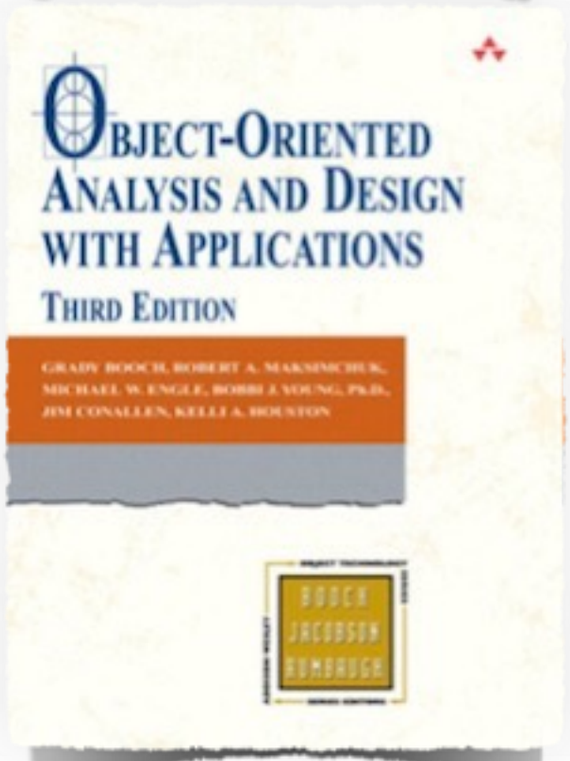
# We need good style to cope with complexity!

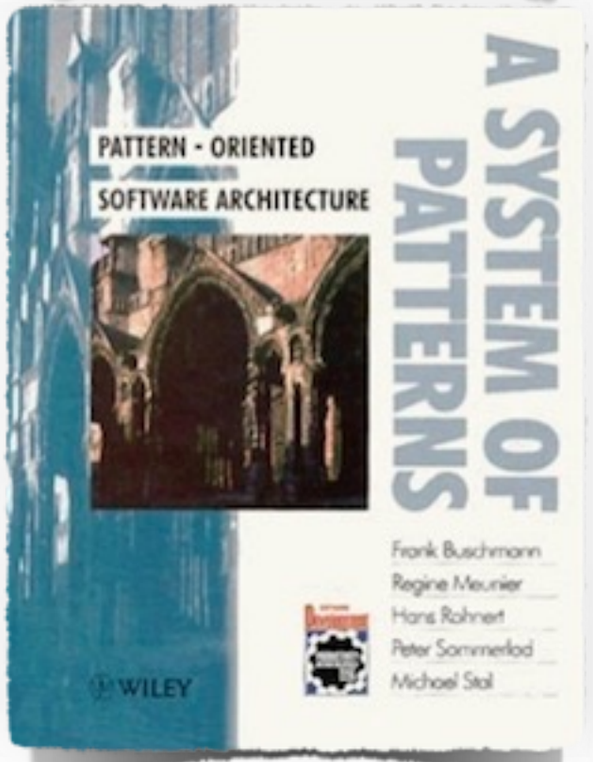


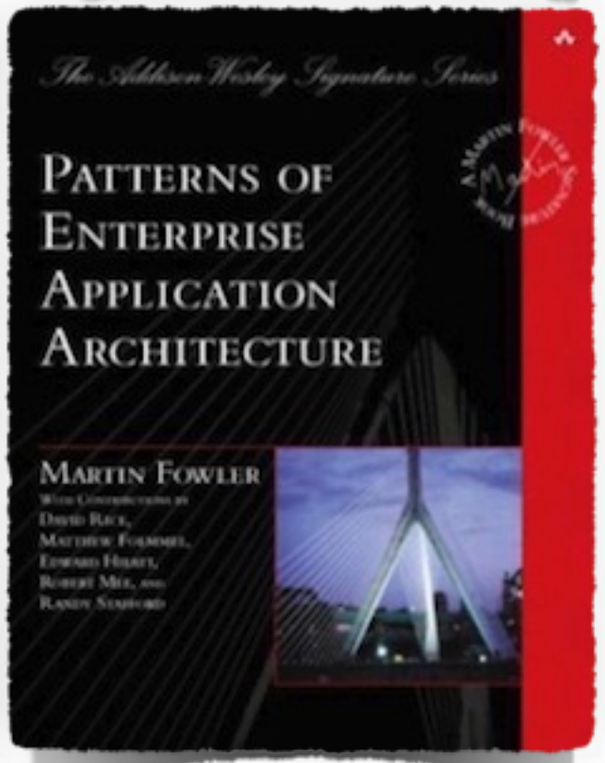








# General Design Principles

The following principles apply at various abstraction levels!

- Keep it short and simple
- Don't repeat yourself (also just called "DRY-Principle")
- High Cohesion
- Low Coupling
- No cyclic dependencies
- Make it testable
- Open-closed Design Principle
- Make it explicit/use Code
- Keep related things together
- Keep simple things simple
- Common-reuse/Common-closure/Reuse-release principles

# Object-Oriented Design Principles

- Liskov Substitution Principle
- Responsibility Driven Design
- ...

# Design Constraints

- **Conway's Law**
  A system's design is constrained by the organization's communication structure.