

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

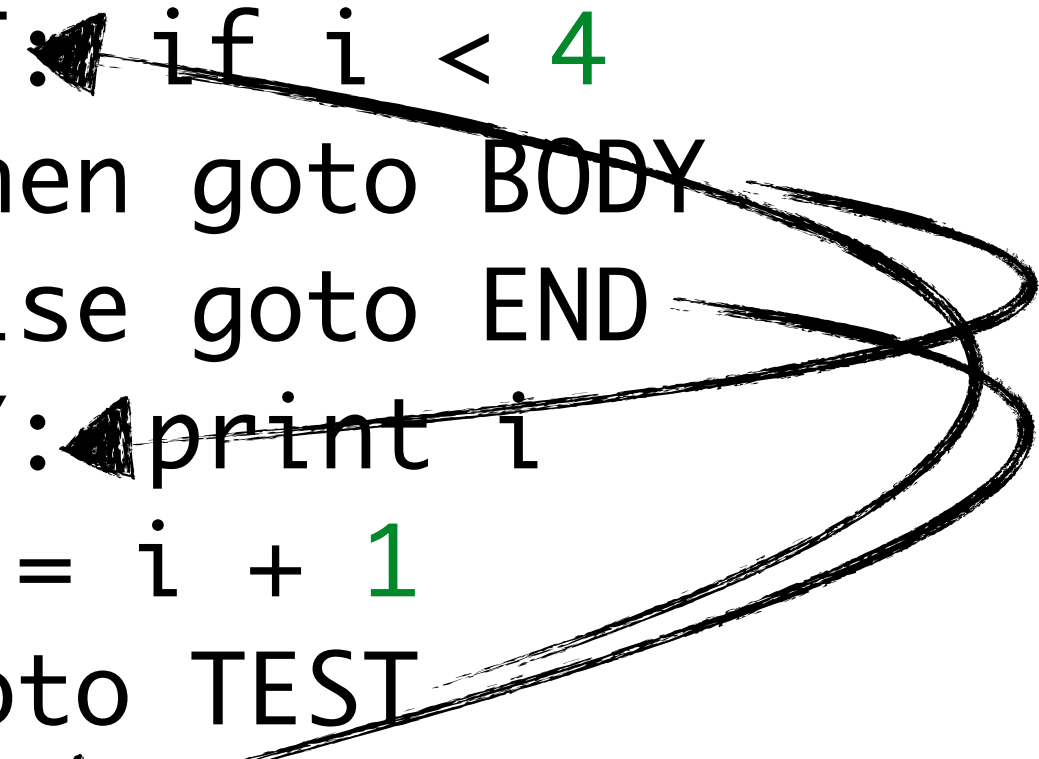
Programming Languages and Design Principles

Making Code Look Like Design

“Designing” with Pseudo-Assembler

What does the following program do?

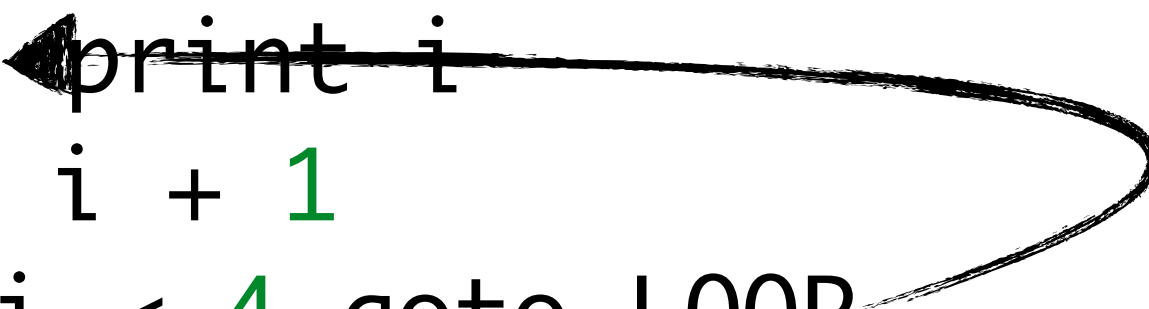
```
i = 1
TEST: if i < 4
      then goto BODY
      else goto END
BODY: print i
      i = i + 1
      goto TEST
END:
```

A hand-drawn control flow diagram is overlaid on the code. It consists of several curved arrows. One arrow starts from the 'if' statement in the TEST block and points to the 'goto BODY' statement. Another arrow starts from the 'goto TEST' statement in the BODY block and loops back to the 'if' statement in the TEST block. A third arrow starts from the 'goto END' statement in the TEST block and points to the 'END:' label. There are also some scribbles and lines connecting the labels TEST, BODY, and 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

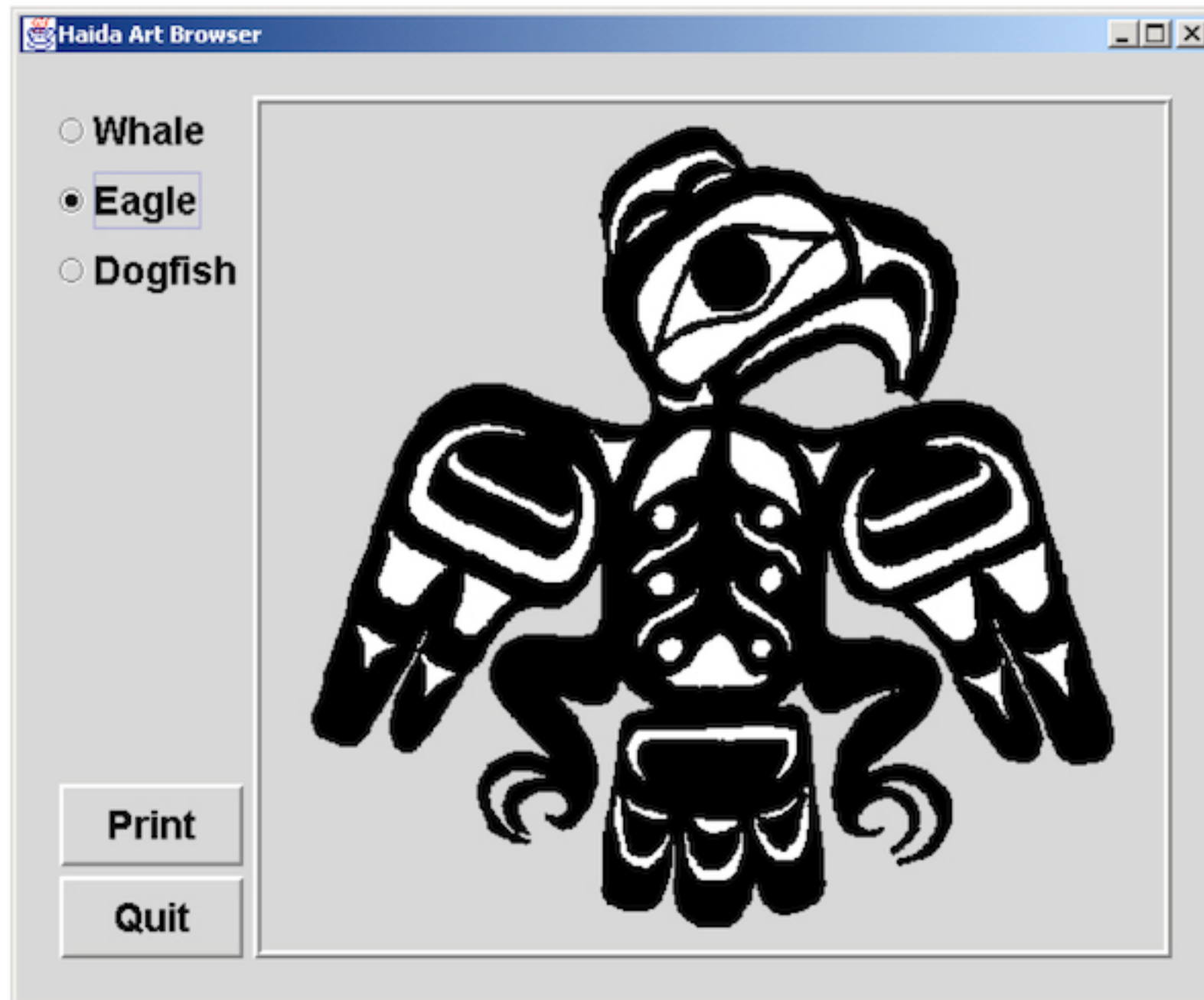
What does the following program do?

```
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 ()
```

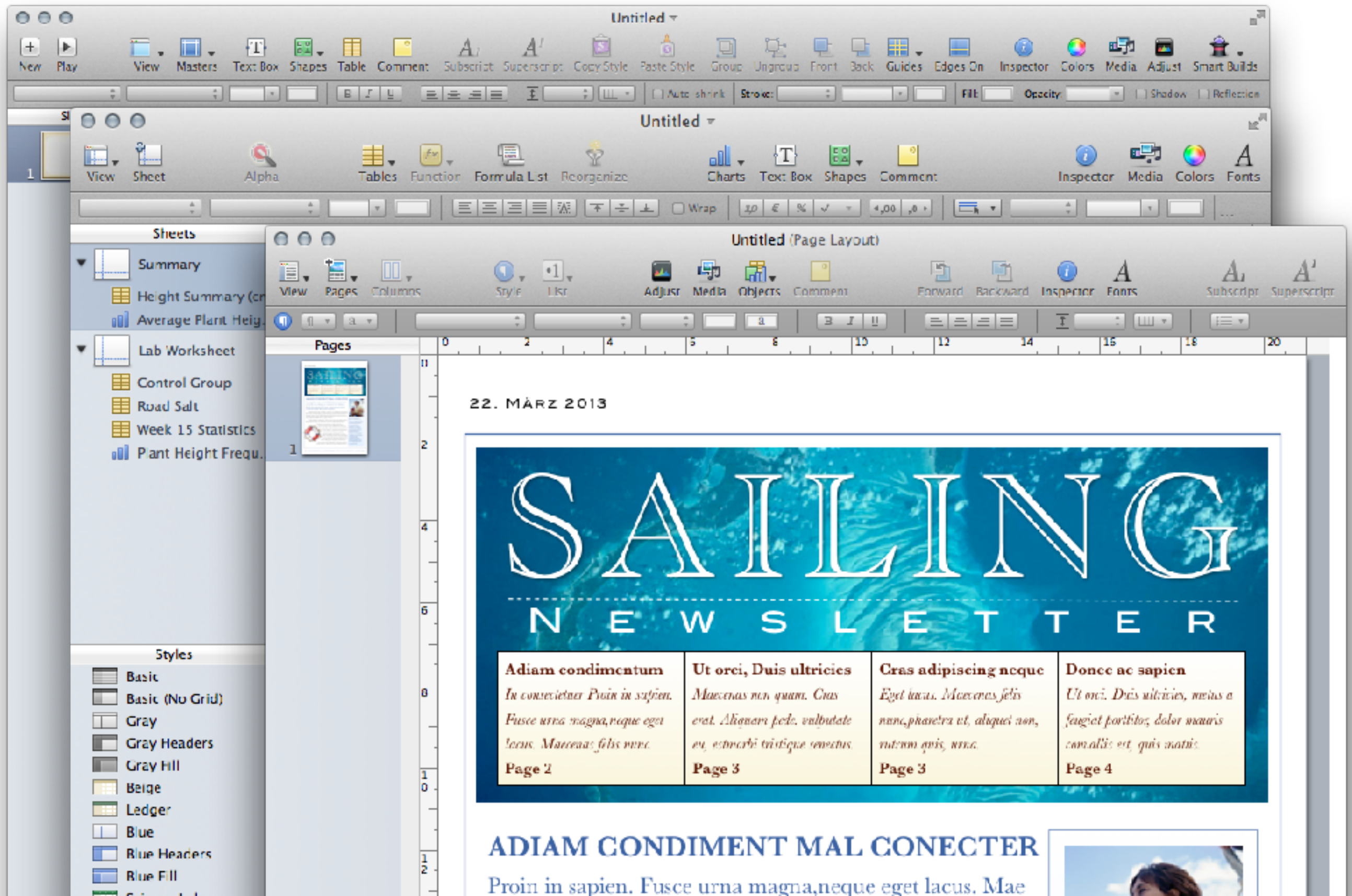
```
state_set_x (x)
```

```
state_get_x ()
```

Abstraction mechanisms
enable us to code and
design simultaneously!

"Write what you mean."

Let's “develop” application families with sophisticated GUIs with uniform look and feel with structured/modular programming...



Designing with Object-Oriented Programming Languages

Object-oriented programming languages introduce new abstraction mechanisms:

- classes
- inheritance
- subtype polymorphism
- virtual methods

(Still) Dominating
Programming Paradigm

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

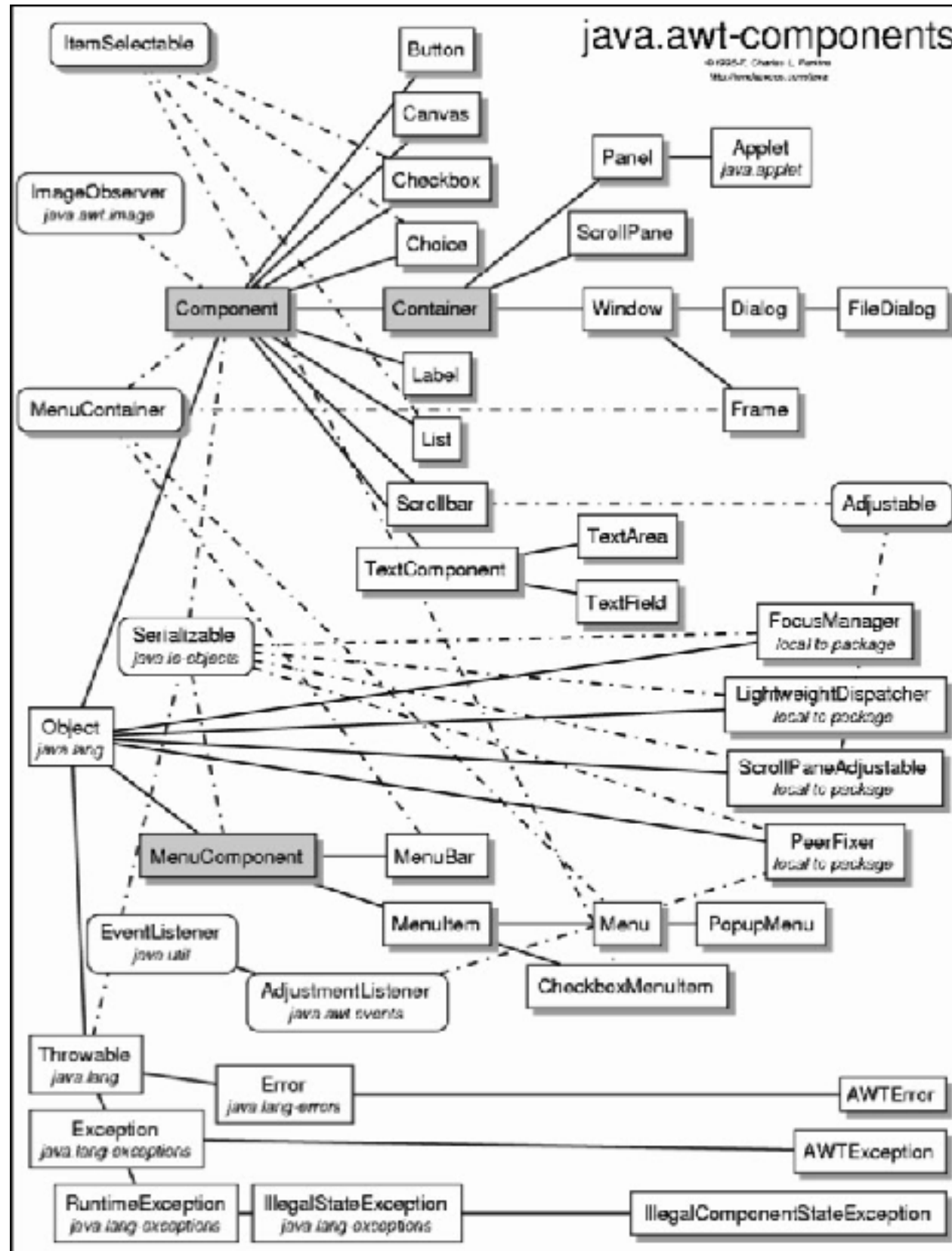


Dahl and Nygaard,
Simula 64, 68



Allan Kay,
Smalltalk 70 - 80

Programming Languages are not a Panacea



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

–Einstein

[...] 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

Read the next n objects from an ObjectInputStream.

Code:

```
class Person(id : Int)

val in : java.util.ObjectInputStream = ...
val n/*count*/ = in.read() // the number of stored object
scala.Array.fill(n){ in.readObject() }
```

Result:

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

Design Challenge: Getting rid of Global State

Typical - screwed up - design!

```
import java.util.HashMap;

public class GlobalState {
    public static Object theMutex; // = ...
    public static HashMap<Object,Object> theCache; // = ...
}

class Main {
    public static void init() {
        synchronized(GlobalState.theMutex) {
            GlobalState.theCache.put("year", new Integer(2017));
            printIt()
        }
    }

    public static void printIt() {
        System.out.println(GlobalState.stateToString());
    }

    public static void main(String[] args) {
        GlobalState.theMutex = new Object();
        GlobalState.theCache = new HashMap<>();

        init();
        doIt();
    }
}
```

Design Challenge: Getting rid of Global State

```
import java.util.HashMap;

public class TheState {
    public Object theMutex; // = ...
    public HashMap<Object,Object> theCache; // = ...
}

class Main {
    public static void init(TheState theState) {
        synchronized(theState.theMutex) {
            theState.theCache.put("year",new Integer(2017));
            printIt(theState)
        }
    }

    public static void printIt(TheState theState) {
        System.out.println(theState);
    }

    public static void main(String[] args) {
        TheState theState = new TheState();
        theState.theMutex = new Object();
        theState.theCache = new HashMap<>();

        init(theState);
        doIt(theState);
    }
}
```

No global State,
but we now have to pass around the state.

Scala - Implicit Parameters

- If a method with implicit parameters misses arguments for them, such arguments will be automatically provided.
- Eligible are all identifiers x that can be accessed at the point of the method call without a prefix and that denote an implicit parameter.
- [... more details will follow later]

Design Challenge: Getting rid of Global State

```
import java.util.HashMap;
```

```
class State {  
    val theMutex: Object = new Object()  
    val theCache: HashMap[Object, Object] = new HashMap()  
}
```

```
class Main {  
  
    def init(implicit s: State): Unit = {  
        s.theMutex.synchronized {  
            s.theCache.put("year", new Integer(2017))  
            printIt()  
        }  
    }  
}
```

```
def printIt(implicit s: State): Unit = {  
    System.out.println(s)  
}
```

```
implicit val s = new State()  
s.theMutex = new Object()  
s.theCache = new HashMap <> ()
```

```
init()  
doIt()
```

```
}
```

No global State,
the state is implicitly passed around.

Avoid that objects are used in the wrong context.

- an object which stores the result of an exam should be bound to the respective exam and should not be compared with some other object storing the results of some other exam
- The value encapsulating the result of the analysis of a method should only be used w.r.t. the specific data-flow analysis that was used to create it.

Traits in Scala

```
trait Table[A, B] {  
  def defaultValue: B  
  def get(key: A): Option[B]  
  def set(key: A, value: B) : Unit  
  def apply(key: A) : B = get(key) match {  
    case Some(value) => value; case None => defaultValue  
  }  
}
```

```
class ListTable[A, B](val defaultValue: B) extends Table[A, B] {  
  private var elems: List[(A, B)] = Nil  
  def get(key: A) : Option[B] = elems collectFirst { case (`key`, value) => value }  
  def set(key: A, value: B) : Unit = elems = (key, value) :: elems  
}
```

```
trait SynchronizedTable[A, B] extends Table[A, B] {  
  abstract override def get(key: A): Option[B] =  
    this.synchronized { super.get(key) }  
  abstract override def set(key: A, value: B) : Unit =  
    this.synchronized { super.set(key, value) }  
}
```

```
object MyTable extends ListTable[String, Int](0) with SynchronizedTable[String, Int]
```



mixin
composition

Traits in Scala (Continued)

```
trait LoggingTable[A, B] extends Table[A, B] {  
  abstract override def get(key: A): B = {  
    println("Get Called"); super.get(key)  
  }  
  abstract override def set(key: A, value: B) = {  
    println("Set Called"); super.set(key, value)  
  }  
}
```

```
class MyTable  
  extends ListTable[String, Int](0)  
  with LoggingTable  
  with SynchronizedTable
```

mixin
composition
(Order matters!)

Mixin Composition in Scala

- In Scala, if you mixin multiple traits into a class the inheritance relationship on base classes forms a directed acyclic graph.
- A linearization of that graph is performed.

The Linearization (Lin) of a class C (`class C extends C1 with ... with Cn`) is defined as:

$$\underline{Lin(C) = C, Lin(Cn) \gg \dots \gg Lin(C1)}$$

where \gg concatenates the elements of the left operand with the right operand, but elements of the right operand replace those of the left operand.

$$\begin{aligned} \{a, A\} \gg B &= a, (A \gg B) \text{ if } a \notin B \\ &= (A \gg B) \text{ if } a \in B \end{aligned}$$

Mixin Composition in Scala

```
abstract class AbsIterator extends AnyRef { ... }  
trait RichIterator extends AbsIterator { ... }  
class StringIterator extends AbsIterator { ... }  
class Iter extends StringIterator with RichIterator { ... }
```

- The linearization of class `Iter` is:

$\{a, A\} \gg B = a, (A \gg B) \text{ if } a \notin B$
 $= (A \gg B) \text{ if } a \in B$

- $\{ \text{Iter}, \text{Lin}(\text{RichIterator}) \gg \text{Lin}(\text{StringIterator}) \}$
- $\{ \text{Iter}, \text{Lin}(\text{RichIterator}) \gg \{ \text{StringIterator}, \text{Lin}(\text{AbsIterator}) \} \}$
- $\{ \text{Iter}, \text{Lin}(\text{RichIterator}) \gg \{ \text{StringIterator}, \text{AbsIterator}, \text{AnyRef} \} \}$
- $\{ \text{Iter}, \{ \text{RichIterator}, \text{AbsIterator}, \text{AnyRef} \} \gg \{ \text{StringIterator}, \text{AbsIterator}, \text{AnyRef} \} \}$
- $\{ \text{Iter}, \text{RichIterator}, \text{StringIterator}, \text{AbsIterator}, \text{AnyRef}, \text{Any} \}$

2nd Rule

The order is relevant!

Traits in Scala (Continued)

```
trait LoggingTable[A, B] extends Table[A, B] {  
  abstract override def get(key: A): B = {  
    println("Get Called"); super.get(key)  
  }  
  abstract override def set(key: A, value: B) = {  
    println("Set Called"); super.set(key, value)  
  }  
}
```

```
class MyTable  
  extends ListTable[String, Int](0)  
  with LoggingTable  
  with SynchronizedTable
```

mixin
composition
(Order matters!)

Abstract Types in Scala

1. `class Food`

3. `class Grass extends Food`

5. `abstract class Animal {`

6. `type SuitableFood <: Food`

7. `def eat(food: SuitableFood) : Unit`

8. `}`

Abstract Type

10. `abstract class Mammal extends Animal`

12. `class Cow extends Mammal {`

13. `type SuitableFood = Grass`

14. `override def eat(food: Grass) : Unit = {}`

15. `}`

Path-dependent types in Scala

```
class DogFood extends Food
```

```
class Dog extends Animal {  
  type SuitableFood = DogFood  
  override def eat(food: DogFood) : Unit = {}  
}
```

```
scala> val bessy = new Cow  
bessy: Cow = Cow@10cd6d  
scala> val lassie = new Dog  
lassie: Dog = Dog@d11fa6  
scala> lassie eat (new bessy.SuitableFood)  
<console>:13: error: type mismatch;  
found    : Grass  
required: DogFood  
lassie eat (new bessy.SuitableFood)
```

Path-dependent types in Scala

```
class Food
```

```
abstract class Animal {  
  type SuitableFood <: Food  
  def createFood : SuitableFood  
  def eat(food: this.SuitableFood) : Unit  
}
```

```
class Cow extends Animal {  
  class Grass extends Food  
  type SuitableFood = Grass  
  def createFood = new Grass  
  override def eat(food: this.SuitableFood) : Unit = {}  
}
```

This cow only wants to eat food especially created for it!

```
val cow1 = new Cow  
val cow2 = new Cow  
cow1.eat(cow1.createFood)  
cow1.eat(cow2.createFood)  
cmd47.sc:1: type mismatch;  
found    : $sess.cmd45.cow2.Grass  
required: $sess.cmd44.cow1.SuitableFood  
         (which expands to)  $sess.cmd44.cow1.Grass
```

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

Can we further improve the comprehensibility:
`scala.Array.fill(n){ in.readObject() }`

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 a value *x* times and create an Array which stores the values using our new control-abstraction.

```
repeat(n){ in.readObject() }
```


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, I](closable: C)(r: C => I): I = {
    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);
}
```


Implementing

```
trait Col[X]{map[T](f:(X)=>T){...}}
```

This is only a first approximation of the method's signature.

Try to **implement** the classical **map** function, which performs a mapping of the values of a collection using a given function, **only once for all collection classes**.

The function should be defined by the “top-level” class (e.g., **Collection**).

The type of the collection with the mapped values should correspond to runtime type of the source collection. If this is not possible, a *reasonable* other collection should be created. (The function should not fail!)

Implementing `Col[X] {map[T](f : (X) => T) {...}}`

Initial Draft

```
trait Col[X] { def map[T](f : (X) => T) : Col[T] = {...} }  
class List[X] extends Col[X] { /*does not override map!*/ ...}  
class BitSet extends Col[Int] { /*does not override map!*/ ...}
```

```
val l = List(1,2,3)
```

```
l.map(i => i + 1) // should result in List[Int](2,3,4)
```

```
val b = BitSet(1,2,3)
```

```
b.map(i => i + 1) // should result in BitSet[Int](2,3,4)
```

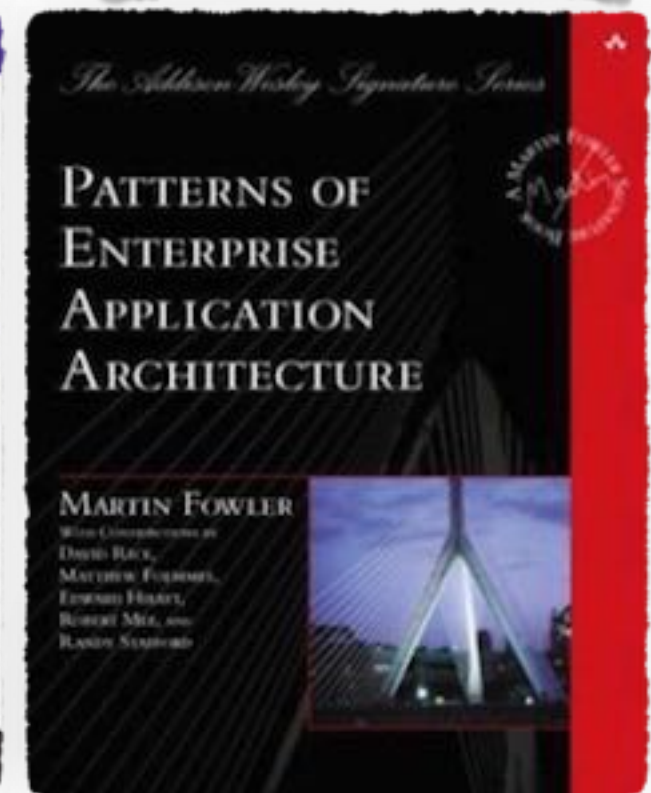
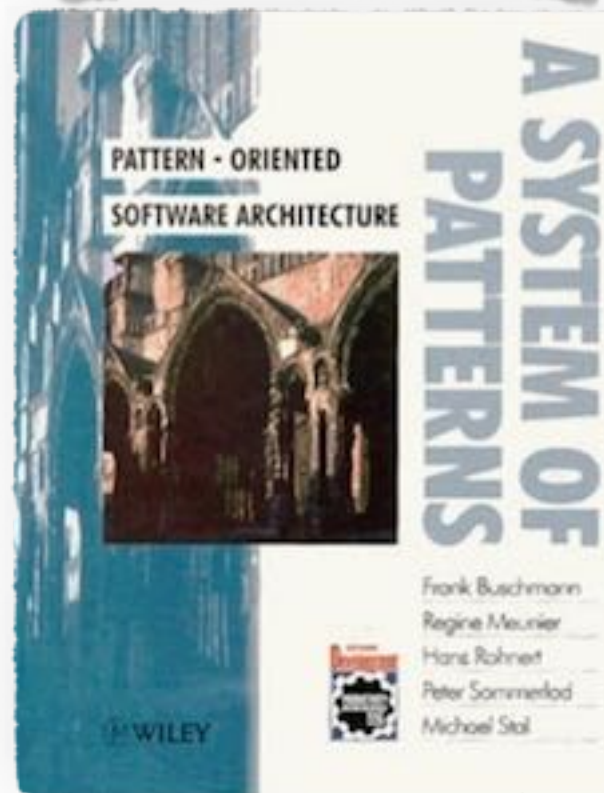
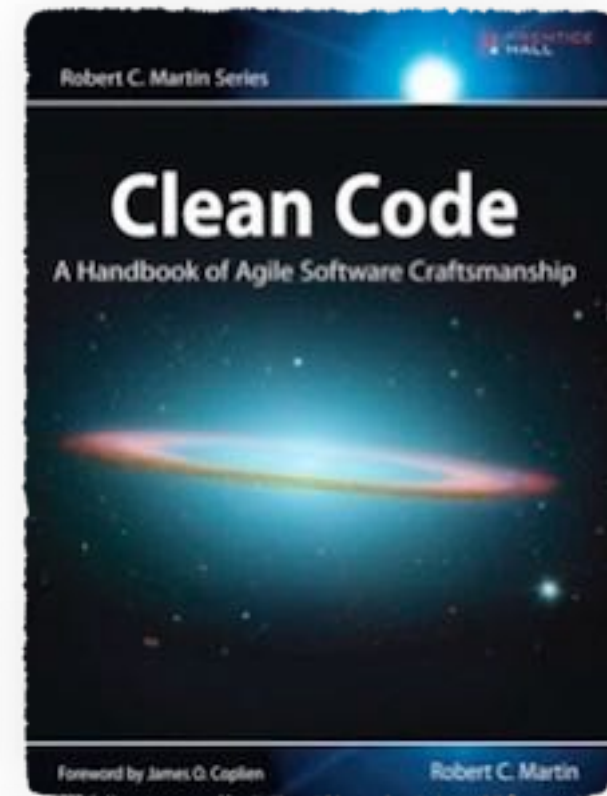
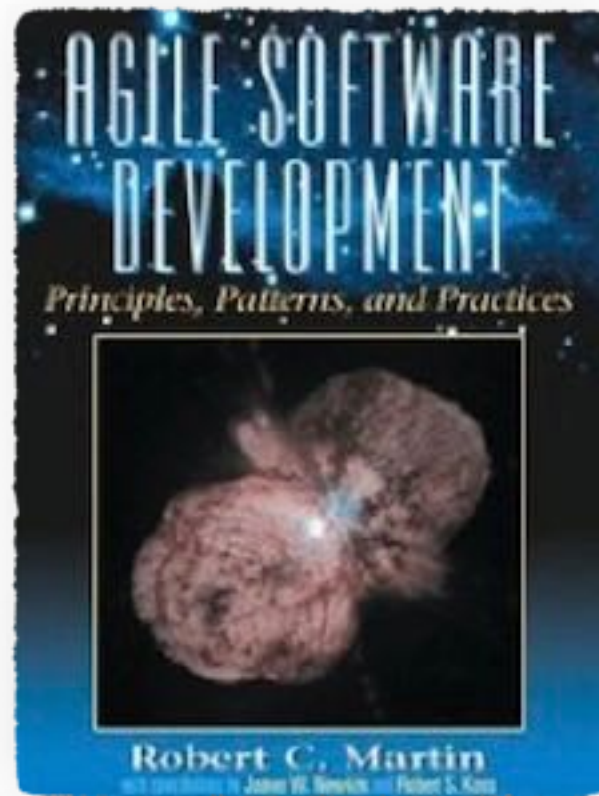
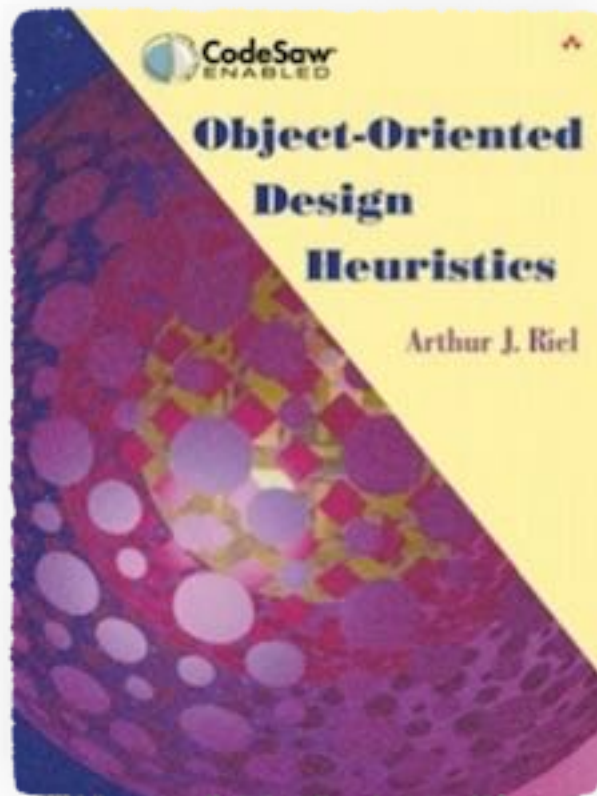
```
b.map(i => "l:" + i) // should result in ???("l:1","l:2","l:3")
```

Will be solved later!

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

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.