Winter Semester 17/18

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Programming Languages and Design Principles

In the following, we will discuss the development of programming languages as a means to improve their ability to capture the software design at ever increasing abstraction levels. Or, from another point of view, we discuss why advances in programming language technology are driven by the need to make programming languages capable of capturing higher-level designs.

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:
```


Though both programs just print out "123" the second one is easier to read and comprehend. It has a better style:

- Clear structure
- No crossing gotos
- Better names
- Code structure closer to what we want to express.
  "Print out i, i smaller than 4“

Hence, the second variant, though functionally identical, is easier to understand, debug, change.

Style can only be recommended, not enforced!

5

Enforcing a specific style is always very laborious and (without proper tool support) most often fails for large(r) (distributed) groups.

Designing with Structured Programming Languages

What does the following program do?

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

Style gets enforced!

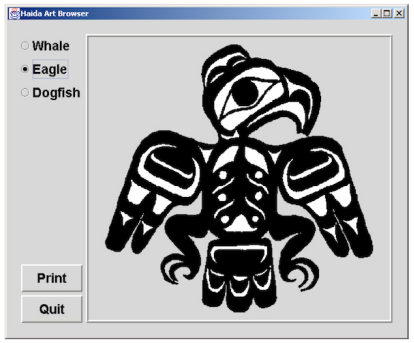
6

In the 1960th programming language support for better structuring of code emerged. **`Goto`s were replaced by loops (`while`)** and conditionals (`if`/`else`). Furthermore, procedures were introduced to support user-defined abstractions.

New words, new grammars, new abstractions enable developers to directly express looping/ conditional computations, instead of emulating them by jumps. Using a – by then – modern structured programming language, it was no longer possible to write crossing `goto`s!

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
}
```

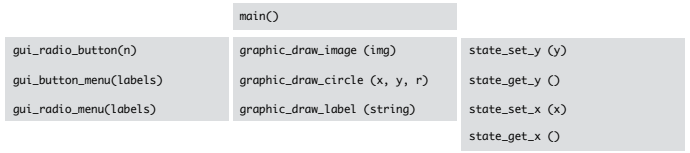
```
draw_image (img) {
   w = img.width
   h = img.height
   do (r = 0; r < h; r++)
      do (c = 0; c < w; c++)
         WINDOW[r][c] = img[r][c]
}

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)
}
```


In this case, the code is structured, but the procedures are not! It is hard, if not nearly impossible, to maintain or even extend this code.

Structured Programming with Style



Group procedures by the functionality they implement and the state they access, e.g. by naming conventions …

Advantages:

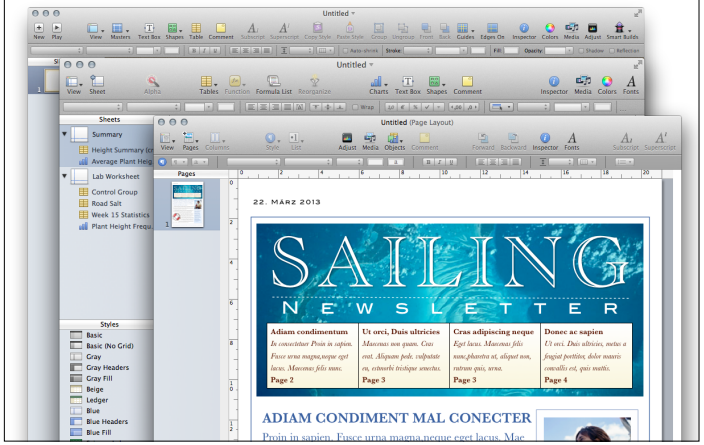
- The code is closer to what we want to express.
  "main calls gui, gui calls graphic to draw, …“
- The code is easier to understand, debug and change.

Abstraction mechanisms enable us to code and design simultaneously!

"Write what you mean."

- 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.

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



Modeling variability with modular programming languages appeared complex…

Designing with Object-Oriented Programming Languages

Object-oriented programming languages introduce new abstraction mechanisms:

- classes
- inheritance
- subtype polymorphism
- virtual methods

(Still) Dominating Programming Paradigm

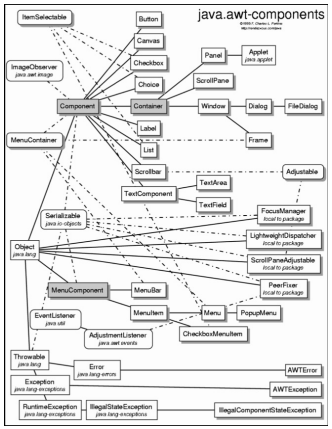
The roots of object-oriented programming languages are in the sixties.

Allan Kay,
Smalltalk 70 - 80

Dahl and Nygaard,
Simula 64, 68

- Object-oriented languages are popular because they make it **easier 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.
- Make it easier to produce the design, and easier to refine it later.
- **With stronger type checking, they also help the process of detecting design errors.**
- Result in a more **robust design**, in essence a better engineered design.

Programming Languages are not a Panacea



- Accessibility of object-oriented programming drives more complex designs!
- Programming languages are powerful tools, but cannot and will never guarantee good designs.
- Programming always needs to be done properly to result in good code.
- Human *creativity* remains the main factor.

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

–Einstein

E.g., trying to fix security problems in programs written in language X which are (conceptually) due to the design of language X is not going to work.

Another example are issues related to memory access, such as out-of memory issues, access violations or buffer overflows are to a great extend solved by avoiding/hiding pointers and by introducing automatic garbage collection etc.

[…] 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.ObjectIntputStream = …
val n/*count*/ = in.read() // the number of stored object
scala.Array.fill(n){ in.readObject() }
```

Result:

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


By fusing object-oriented and functional programming we are provided with further means to raise our abstraction level. This enables us to better express our intention.

Design Challenge: Getting rid of Global State


```
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();
    }
}
```


Code which makes use of Global State is generally hard(er) to comprehend and often (much) hard(er) to test! Try to avoid global state whenever possible!

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);
    }
}
```


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]

From the language spec (Full definition):

A method with implicit parameters can be applied to arguments just like a normal method. In this case the implicit label has no effect. However, if such a method misses arguments for its implicit parameters, such arguments will be automatically provided.

The actual arguments that are eligible to be passed to an implicit parameter fall into two categories:

- *First, eligible are all identifiers x that can be accessed at the point of the method call without a prefix and that denote an implicit definition or an implicit parameter.*
- *Second, eligible are also all members of companion modules of the implicit parameter's type that are labeled implicit.*

Design Challenge: Getting rid of Global State

No global State, the state is implicitly passed around.

```
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()
}
```


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

In Scala, traits are a unit of code reuse that encapsulate abstract and concrete method, **field and type** definitions. Traits are reused by mixing them into classes. Multiple traits can be mixed into a class (mixin composition).
Unlike classes, traits cannot (yet) have constructor parameters. **Traits are always initialized after the superclass is initialized.**

One major difference when compared to multiple inheritance is that the target method of `super` calls is not statically bound as in case of (multiple) inheritance. **The target is determined anew whenever the trait is mixed in.** This (the dynamic nature of super calls) makes it possible to **stack multiple modifications on top of each other.**

The following code snippets are taken from:

- Scala for the Impatient; Cay S. Horseman
- Programming in Scala 1.1; Martin Odersky, Lex Spoon, Bill Venners
- Scala in Depth; Joshua Suereth
- The Scala Specification

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!)

Do we first want to acquire the lock or first log the call? The desired behavior is controlled by the mixin order!

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:
  Lin(C) = C, Lin(Cn) ≫ ... ≫ Lin(C1)
  where ≫ concatenates the elements of the left operand with the right operand, but elements of the right operand replace those of the left operand.
  $\{a,A\} \gg B \quad = a,(A \gg B)$ if $a \notin B$
  $= (A \gg B)$ if $a \in B$

Recall: The result of the linearization determines the target of `super` calls made in traits, but also determines the initialization order.

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:
  - { Iter, Lin(RichIterator) ≫ Lin(StringIterator) }
  - { Iter, Lin(RichIterator) ≫ { StringIterator, Lin(AbsIterator) } }
  - { Iter, Lin(RichIterator) ≫ { StringIterator, AbsIterator, AnyRef } }
  - { Iter, { RichIterator, AbsIterator, AnyRef } ≫ { StringIterator, AbsIterator, AnyRef } } — 2nd Rule
  - { Iter, RichIterator, StringIterator, AbsIterator, AnyRef, Any } — The order is relevant!

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

In case of classical multiple inheritance, the method called by a super call is statically determined based on the place where the call appears. With traits, the called method is determined by the linearization of the class. In a way, `super` is much more flexible.

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!)

In this case we will first acquire the lock (by the SynchronizedTable) before the call is logged.

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.  }

10. abstract class Mammal extends Animal

12. class Cow extends Mammal {
13.    type SuitableFood = Grass
14.    override def eat(food: Grass) : Unit = {}
15. }
```

Abstract Type

An **abstract type declaration is a placeholder for a type that will be defined concretely in a subclass.** In the given example, SuitableFood refers to some type of Food (Food is an upper bound) that is still unknown. Different subclasses can provide different realizations of SuitableFood - depending on the needs of the respective animal.

Remark: Generics and abstract types can sometimes be used interchangeably. However, if you have many type definitions abstract types are more scalable (e.g., the abstract type only does not need to be redeclared in every abstract subtype; only in the (first) concrete one.)

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)
```


- In Scala objects can have types as members.
- The meaning of a type depends on the path you use to access it.
- The path is determined by the reference to an Object.
- Different paths give rise to different types.
- In general, a path-dependent type names an outer object.

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 = {}
}

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
```

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

`Grass` is an inner class of `Cow`, therefore, to create an instance of Grass we need a `cow` object and this `cow` object determines the path; therefore two `cow` objects give rise to two different paths!

(In Java an instance of a (non-static) inner class also always has a reference to its outer class and you need an instance of the outer class to instantiate the inner class, but this does not affect the type system; i.e., two “Map.Entry” objects have the same type independent of the underlying Map instance.)

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() }
```


By fusing object-oriented and functional programming and also providing a more flexible syntax we are provided with further means to raise our abstraction level. In this example, we demonstrate how to define our own “control-abstraction”! Defining a new control structure like this is not *reasonably possible* in Java 7 or older. In Java >= 8; the situation gets better due to closures. However, the syntax still doesn’t look like a native structure.

However, with power comes responsibility and it is easy to overdo!

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);
}
```


Java 7’s try-with-resources statement is more powerful/safe. However, it is an explicit language feature that was only added years after its need was identified.

Design Challenge

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!)

`trait Col[X]` defines a generic type with the type variable X. Furthermore, it defines the generic method map which defines the type parameter T and which gets a function f that maps a value of type X to a value of type T.
(In this context, loosely comparable to a generic abstract class in Java which has a type parameter and defines a concrete, generic method map.)

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 => "I:"+i) // should result in ???("I:1","I:2","I:3")
```

Will be solved later!

This is a typical design challenge when defining new libraries: providing rich libraries with "minimal" effort and a high level of customizability.

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

35

(Some languages avoid dead locks or race conditions.)

We need good style to cope with complexity!

36

Help is provided through established practices and techniques, design patterns and principles.

Good style can only be recommended, not enforced!

Eventually style rules will have to be turned into language features to be really effective.

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

Code/projects that adheres to these design principles generally have improved maintainability.

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.