Introduction to Agile and Scala

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

September 2, 2025

Outline

- Waterfall and Agile design style
- Introduction to Scala
- Functional programming in Scala
- Links to further reading and labs

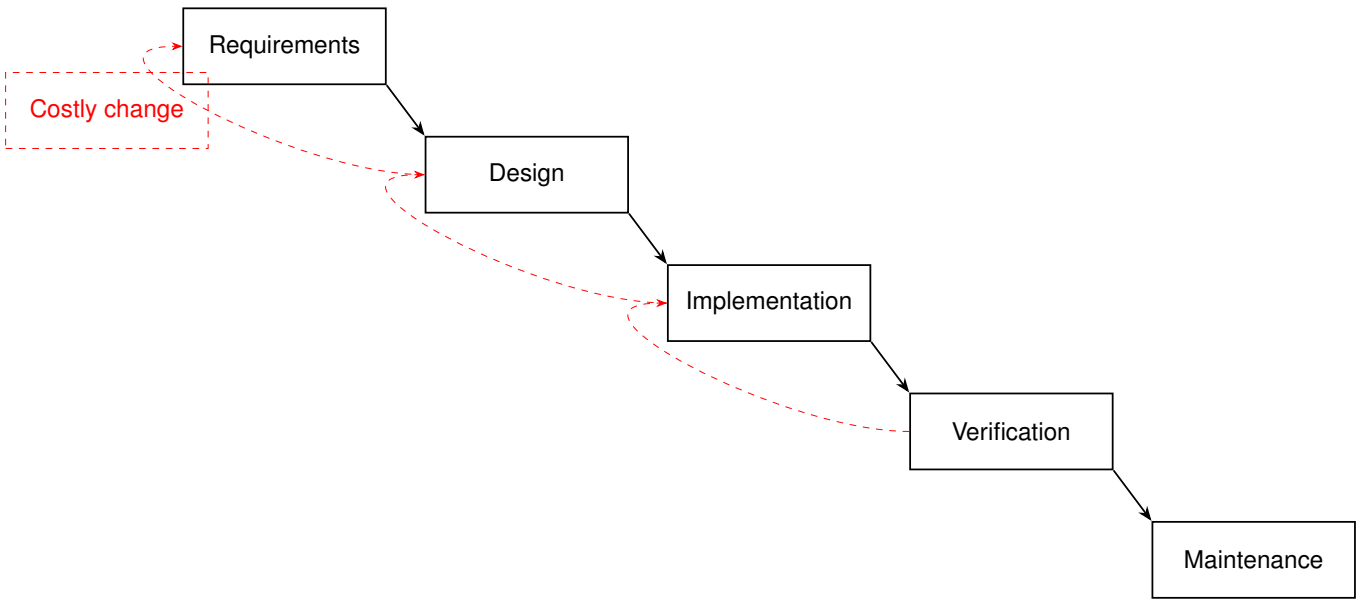
Course Overview

- This is a new course
  - Please be patient
  - I will adapt it on the go (to your needs)
  - This is agile at work
- 2 hours lecture + 2 hours lab
- Javad is TA
- We have a Discord server for discussions and questions
- Final project with a README as documentation
- Group work, explore Scrum
- Several presentations of the project development

The Classic Waterfall Model

- Traditional project management approach for software and hardware
- Development phases are sequential:
  1. Requirements
  2. Design
  3. Implementation
  4. Verification
  5. Maintenance
- Each phase must be completed before moving to the next
- Changes are costly once early phases are complete
- I worked (long time ago) in such a team: was very boring

Waterfall Model Diagram



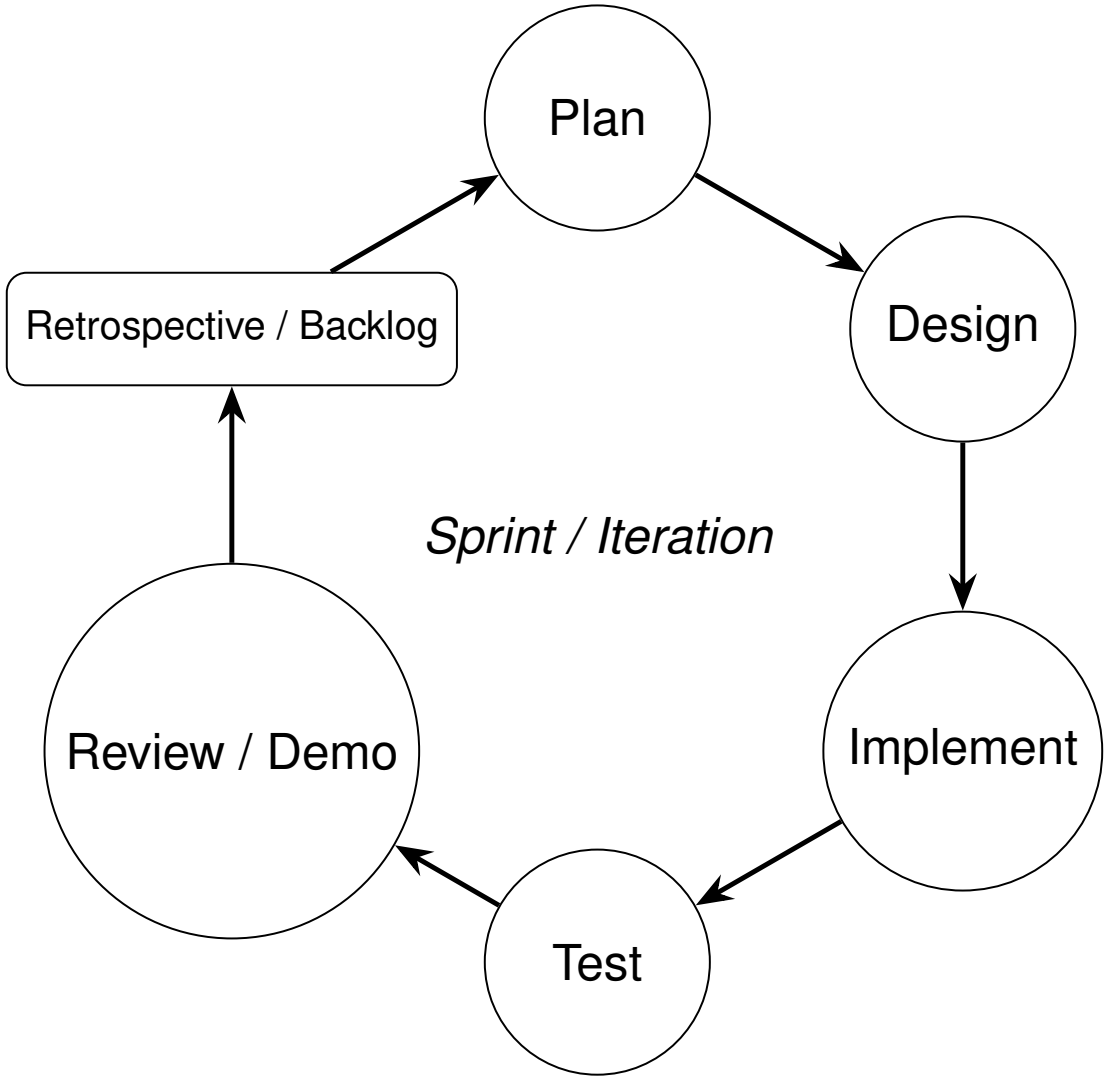
Limitations of the Waterfall Model

- Assumes requirements are fixed at the start
- Poor adaptability to changing customer needs
- Testing and feedback happen late in the process
- High risk of discovering major flaws late
- Not well-suited for rapid prototyping or exploratory projects
- Worked at Compaq for a banking SW in this style

Why Agile for Hardware?

- Traditional hardware development: long, rigid cycles
- Software has embraced Agile:
  - Quick iterations
  - Test-driven development
  - Continuous integration
- Hardware complexity is increasing $\rightarrow$ need for agility!

Agile Iteration Cycle



Agile Iteration Phases (1/3)

Plan

- Define goals for the upcoming sprint (typically 1-4 weeks)
- Select features or fixes from the product backlog
- Break down tasks into manageable user stories
- Ensure each story has clear acceptance criteria

Design

- Create a simple, implementable hardware design
- Define interfaces and parameters
- Update documentation and diagrams
- Keep the design minimal to support fast iteration
- Think: minimal viable product

Agile Iteration Phases (2/3)

Implement

- Write Chisel modules for new or updated functionality
- Commit early and often to version control
- Follow coding standards and naming conventions
- Collaborate closely to avoid merge conflicts

Test

- Use automated tests (e.g., `chiseltest`) to validate functionality
- Run both unit tests and integration tests
- Include corner cases and property-based checks
- Maintain high test coverage throughout development

Agile Iteration Phases (3/3)

Review / Demo

- Present completed work to the team or stakeholders
- Demonstrate working features in simulation or on FPGA
- Gather feedback on design choices and implementation

Retrospective / Backlog Refinement

- Reflect on what worked well and what can be improved
- Adjust processes, tools, and team coordination as needed
- Update the product backlog with new ideas or changes
- Prepare for the next sprint cycle

Hardware Design Today

- VHDL / Verilog = rigid, low-level
- Hard to reuse and test modularly
- Long simulation cycles
- Not optimized for iteration or testing early
- Often follows a version of the waterfall model

Software-Inspired Hardware Flow

1. Write modular, parameterized designs
2. Test-first using simulation
3. Version control and continuous integration (CI)
   - For example, using GitHub actions
4. Frequent review and refactor
   - With enough tests, refactoring is safe
5. Code reviews and pull requests

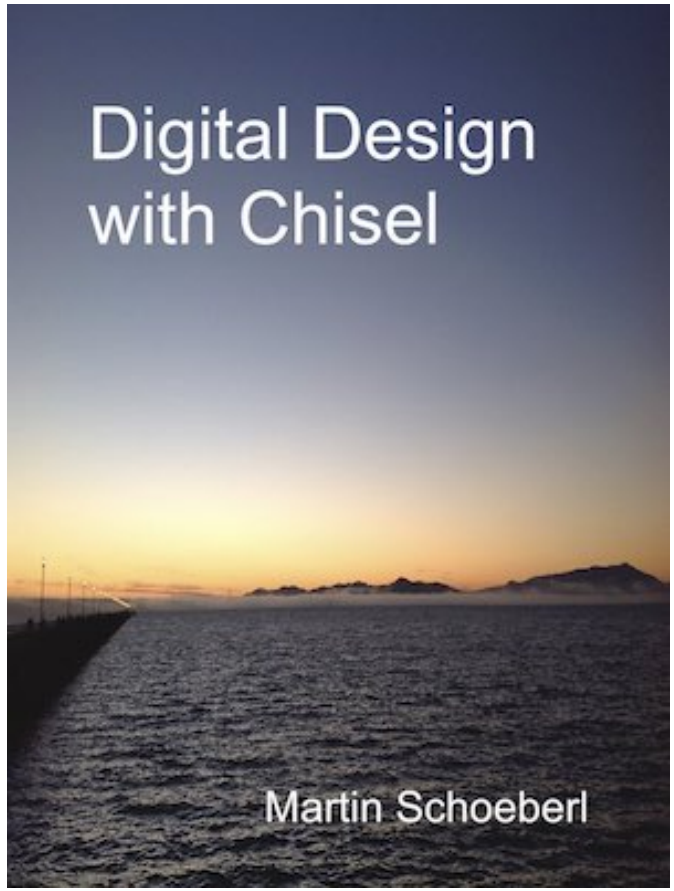
What Language do You Already Know?

- Python
- Java
- C
- Scala
- VHDL
- Verilog
- Chisel
- Haskell
- Clash
- Other

On Chisel

- The course will use Chisel and Scala
- Next week, 1 hour intro to Chisel
- If you know it already, join at 14:00
- Start reading the Chisel book

A Chisel Book



- Available in open access (as PDF)
  - Optimized for reading on a tablet (size, hyperlinks)
- Amazon can do the printout

Scala

- Object-oriented
- Functional
- Strongly typed with very good type inference
- Runs on the Java virtual machine
- Can call Java libraries
- Consider it as Java++
  - Can almost be written like Java
  - With a more lightweight syntax
  - Compiled to the JVM
  - Good Java interoperability
  - Many libraries available
- https://docs.scala-lang.org/tour/tour-of-scala.html

Scala Hello World

```
object HelloWorld extends App {
  println("Hello, World!")
}
```

- Compile with `scalac` and run with `scala`
- Or with `sbt run`
- You can even use Scala as a scripting language
- `scala-cli` is a generic Scala runner
- Show both
- Use `scala-cli` locally along the examples presented

The “Real” Hello World

```
object Hello {
  def main(args: Array[String]): Unit = {
    println("Hello, world!")
  }
}
```

- Every program starts with an `object` and a `main` method
- Use `println` to print to the console
- Access to command line arguments via `args`
- Similar ot the Java static `main` function

Scala Values and Variables

- Scala distinguishes between immutable (`val`) and mutable (`var`)
- By default, use `val` (immutability is preferred)

```
// A value is a constant
val i = 0
// No new assignment; this will not compile
i = 3

// A variable can change the value
var v = "Hello"
v = "Hello World"

// Type usually inferred, but can be declared
var s: String = "abc"
```

Basic Types

- Common Scala types:

```
val a: Int = 42
val b: Double = 3.14
val c: Boolean = true
val d: String = "Scala"
```

- Type inference: Scala often figures out the type for you

```
val x = 100      // Int
val y = "hi"     // String
```

Simple Loops

```
// Loops from 0 to 9
// Automatically creates loop value i
for (i <- 0 until 10) {
  println(i)
}
```

Conditions

```
for (i <- 0 until 10) {
  if (i%2 == 0) {
    println(i + " is even")
  } else {
    println(i + " is odd")
  }
}
```

Expressions and Functions

- Everything in Scala is an expression (returns a value)

```
val sum = 1 + 2   // 3
val cond = if (sum > 2) "big" else "small"
```

- Defining a function:

```
def add(a: Int, b: Int): Int = {
  a + b
}
```

Scala Arrays and Lists

```
// An integer array with 10 elements
val numbers = new Array[Integer](10)
for (i <- 0 until numbers.length) {
  numbers(i) = i*10
}
println(numbers(9))
```

```
// List of integers
val list = List(1, 2, 3)
println(list)
// Different form of list construction
val listenum = 'a' :: 'b' :: 'c' :: Nil
println(listenum)
```

Scala Collections

- Scala has a powerful collection library
- `Seq` is an ordered collection of elements (also called a sequence)
- The default implementation is immutable
- We index into a `Seq` with `()`, with zero-based indexing
- Collections work well with functional programming

```
val numbers = Seq(1, 15, -2, 0)
val second = numbers(1)
```

Lists

- Lists are common in Scala
- Default list is immutable

```
val nums = List(1, 2, 3, 4)

// head and tail
println(nums.head)  // 1
println(nums.tail)  // List(2, 3, 4)

// append
val nums2 = nums :+ 5
```

- When you append to a List it creats a new list

Scala Classes

```
// A simple class
class Example {
  // A field, initialized in the constructor
  var n = 0

  // A setter method
  def set(v: Integer) = {
    n = v
  }

  // Another method
  def print() = {
    println(n)
  }
}
```

Scala (Singleton) Object

```
object Example {}
```

- For *static* fields and methods
  - Scala has no static fields or methods like Java
- Needed for `main`
- Useful for helper functions

Tuples

- Scala has the notion of tuples
- Can hold a sequence of different types
- Fields are then accessed with `._n`, starting with 1
- Easy option to return more than one value from a function

```
val city = (2000, "Frederiksberg")
val zipCode = city._1
val name = city._2
```

Functional Programming

- Functional programming (FP) treats computation as the evaluation of functions
- Functions are first-class objects
- Can be a parameter of a function
- Can be returned from a function
- Avoid mutable state
- Recursion is *not* the main point of FP
- Higher-order functions take functions as arguments

First-Class Functions

- In Scala, functions can be assigned to variables, passed as arguments, or returned

```
// A normal function definition
def double(x: Int): Int = x * 2

println(double(5))   // prints 10

// A function that takes another function
def applyTwice(f: Int => Int, v: Int): Int = {
  f(f(v))
}

println(applyTwice(double, 3)) // prints 12
```

Function Literals (Anonymous Functions)

- A function literal is a shorthand for defining a function ”inline”
- by the => symbol
- Does not require a name (`def`)
- Often used with higher-order functions

```
// Normal function
def square(x: Int): Int = x * x

// Function literal (anonymous function)
val squareFn = (x: Int) => x * x

// Using a literal directly
val nums = List(1, 2, 3, 4)
val squares = nums.map(x => x * x)
println(squares) // List(1, 4, 9, 16)
```

Higher-Order Functions

- Functions that take other functions as arguments
- What we just used before
- Useful for hardware generators (map, filter, etc.)

```
val nums = List(1, 2, 3, 4)
val squares = nums.map(x => x * x)
println(squares) // List(1, 4, 9, 16)
```

Immutability

- FP encourages immutable values (using `val`)
- No side effects: easier to reason about hardware generation

```
val a = 5
// a = 6   // ERROR: reassignment not allowed

val b = a + 1   // OK, creates a new value
```

- In Chisel, `val` describes structure, not time-varying state
- Registers (`Reg`) are explicit when you need mutable hardware state

Mapping Over Collections

- `map` applies a function to each element of a collection
- Produces a new collection of the same size

```
val nums = List(1, 2, 3, 4)
def square(x: Int): Int = x * x

val squares = nums.map(square)
println(squares) // List(1, 4, 9, 16)
```

Filtering Collections

- `filter` keeps only elements that satisfy a condition

```
val nums = List(1, 2, 3, 4, 5, 6)

def isEven(x: Int): Boolean = (x % 2 == 0)
val evens = nums.filter(isEven)

println(evens) // List(2, 4, 6)
```

Reducing Collections

- `reduce` combines all elements using a binary function
- Useful for sums, products, and combining signals

```
val nums = List(1, 2, 3, 4)

def add(x: Int, y: Int): Int = x + y
val sum = nums.reduce(add)

println(sum) // 10
```

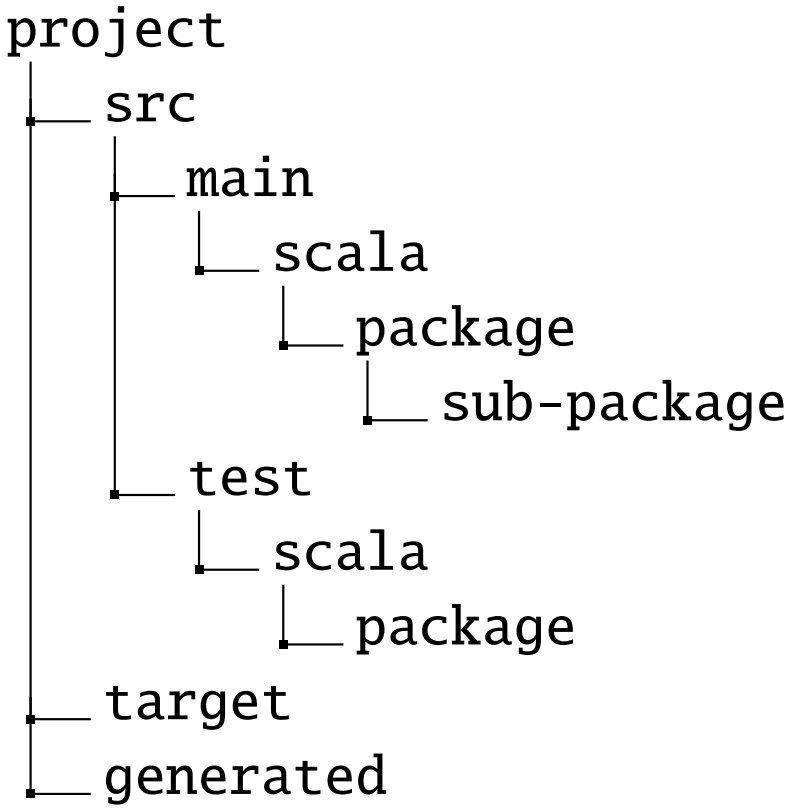
- Hardware analogy: adding a set of signals
- Use `reduceTree` for a balanced reduction tree

Scala Build Tool (sbt)

- Downloads Scala compiler if needed
- Downloads dependent libraries (e.g., Chisel)
- Compiles Scala programs
- Executes Scala programs
- Does a lot of magic, maybe too much
- Compile and run with:

```
sbt run
```

File Organization in Scala/Chisel



ScalaTest

- Testing framework for Scala and Java
- `sbt` understands ScalaTest
- Add library to `build.sbt`

```
libraryDependencies += "org.scalatest" %%
   "scalatest" % "3.1.4" % "test"
```

- Run all tests with:

```
sbt test
```

- When all (unit) tests are ok, the test passes
- A little bit funny syntax
- ChiselTest is based on ScalaTest

ScalaTest Hello World

```
import org.scalatest._
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers

class ExampleTest extends AnyFlatSpec with
    Matchers {
  "Integers" should "add" in {
    val i = 2
    val j = 3
    i + j should be (5)
  }

  "Integers" should "multiply" in {
    val a = 3
    val b = 4
    a * b should be (12)
  }
}
```

Further Reading and Web Resources

- Scala defined hardware generators for Chisel
  - Journal article with generator examples
- Chisel book website
  - Information on Chisel, a bit of Scala
  - Download the free PDF
- Digital design course at DTU
  - Slides on digital design with Chisel
- Digital design lab at DTU
  - Lab material for the digital design course
  - Option to train a bit on Chisel

Hardware Exercise

- Some of you come with different hardware design expertise
- You shall test yourself with a small exercise
- Exercise description
- Use a hardware description language of your choice
- Upload your solution to DTU learn till the end of the week (Sunday), including a test bench
- I will not grade it, but give you feedback on your solution

Tool Setup for Different OSs

- Windows
  - Use the installers from the websites
- macOS
  - `brew install sbt`
  - For the rest, use the installer from the websites
- Linux/Ubuntu
  - `sudo apt install openjdk-8-jdk git make gtkwave`
  - Install sbt
  - IntelliJ as from the website
- Instruction details: https://github.com/schoeberl/agile-hw/blob/main/Setup.md

An IDE for Chisel

- IntelliJ
- Install the Scala plugin
- For IntelliJ: File - New - Project from Existing Sources..., open build.sbt
- Show it
- Visual Studio Code with the Scala plugin (Metals)

Scala CLI

- scala-cli
- A tool for compiling and running Scala code/scripts
- show it: REPL, scripts, full-blown Scala apps (in tmp)

Lab 0: Hello World in Chisel

- Install all tools, see Setup.md
- Clone or download the repository from:
  - `https://github.com/schoeberl/agile-hw`
- Start IntelliJ and follow the instructions from the lab0 page
- `sbt test`
- Explore the code, maybe change the example

Lab 1

- Functional programming in Scala
- First part with Scala REPL
- The lab contains tests, run with `sbt test`
- Code is in lab1

Next Week

- Quick poll on Chisel
- Introduction into Chisel (2x 1 hour lecture, 1 hour lab)
- If you know Chisel, you can stay at home
- Or better contribute as TA to help out (starting 14:00)

Summary

- Processors do not get much faster – we need to design custom hardware
- We need a modern language for hardware/systems design for efficient/fast development
- Chisel builds on the power of object-oriented and functional Scala
- We shall write hardware generators