Software Engineering: Design & Construction

Department of Computer Science
Software Technology Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Final Exam | **Sample Solution** | **February 13, 2018**

First Name	
Last Name	
Matriculation Number	
Course of Study	
Department	
Signature	

Permitted Aids

- Everything except electronic devices
- Use a pen with document-proof ink (No green or red color)

Announcements

- Make sure that your copy of the exam is complete (21 pages).
- Fill out all fields on the front page.
- Do not use abbreviations for your course of study or your department.
- Put your name and your matriculation number on all pages of this exam.
- The time for solving this exam is 90 minutes.
- All questions of the exam must be answered in English.
- You are not allowed to remove the stapling.
- You can use the backsides of the pages if you need more space for your solution.
- Make sure that you have read and understood a task before you start answering it.
- It is not allowed to use your own paper.
- Sign your exam to confirm your details and acknowledge the above announcements.

Topic	1	2	3	4	Total
Points					
Max.	26	29	14	21	90

Name: ______________________________ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴

Topic 1: Introduction 26P

Nicht den Satz vergessen, dass wir immer eine kurze Begrüdung erfordern.

a) True or False 7.5P

1. Java 8 Interfaces with default methods enable the same kind of multiple inheritance that is represented by traits in Scala.
   “Vorschlag für Neuformulierung:” Java 8 interface inheritance and Scala trait inheritance are equally expressive.

2. The design of the code is not affected by the choice of programming language.

3. The use of object composition helps to alleviate the Fragile Base Class Problem.

4. An explicit instantiation of the strategy pattern (i.e. defining a explicit strategy interface) can be avoided with the support of higher-order functions in Scala or Java 8.

5. Can you imagine a Turing complete object oriented programming language that support inheritance where the Liskov substitution principle can never be violated?

Name: ______________________ Matriculation Number: ⊔⊔⊔⊔⊔⊔⊔

Solution

1.5P for each. For false statements, 0.5P for the ***yes / no*** and 1P for the reasoning.

1. No. Mix-ins not possible and no State.
2. No. Different language features facilitate different patterns, e.g. mixins.
3. Yes. The objects used for composition can change without TODO
4. Yes. A strategy in this case consists of a passed function.
   No - if the strategy interface would have to define multiple methods… (i.e., for simple strategies yes, for complex ones no)
5. No. Turing-completeness implies that overridden methods can have arbitrary functionality that violates the LSP.

b) Basic Knowledge 11P

1. Take a look at the following Java code:

```java
public class ButtonClicker {
   public static void main(String[] args) {
      String hello = "Hello";
      JButton button = new JButton();
      button.addActionListener(e -> {
         System.out.println(hello);
      });
      button.doClick();

      hello = "Hola";
      button.addActionListener(e -> {
         System.out.println(hello);
      });
      button.doClick();
   }
}
```

In particular recall the requirements of Lambda expressions in Java code.

This code will not compile. Shortly state why this code won't compile. Write Scala code that models the same functionality but does compile. Also shortly sketch what you would have to change in the Java code to make it work. (6P)

Solution

Scala:

```scala
object ButtonClicker{
  def main(args: Array[String]): Unit = {
    var hello = "Hello"

    val button = new JButton
    button.addActionListener(e => {
      println(hello)
    })
    button.doClick()
    hello = "Hola"
    button.addActionListener(e => {
      println(hello)
    })

    button.doClick()
  }
}
```

Java:

```
class Greeter {
   public String hello;
}


public class ButtonClicker {
    public static void main(String[] args) {
       JButton button = new JButton();
       Greeter greeter = new Greeter();
       greeter.hello = "Hello";
       button.addActionListener(e -> {
           System.out.println(greeter.hello);
        });
        button.doClick();

        greeter.hello = "Hola";
        button.addActionListener(e -> {
            System.out.println(greeter.hello);
        });
        button.doClick();
    }
}
```

Name: ________________________ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴

2. Discuss two features of Scala that help to narrow the representational gap compared to Java. (2P)

Solution

Two of the following concepts suffice.

- Higher-kinded types
- Path-depedent types
- Mixin composition
- Pattern matching
- Flexible syntax
- Embedded DSLs

3. Which part of the class `StatementSequence` has to be documented when the class is intended to be used by subclassing? Describe the relation to the Fragile Base Class Problem and give a concrete example. (3P)

```
trait ASTNode
class StatementSequence extends ASTNode {
   private[this] var statements = List.empty[ASTNode]
   def addStatement(s: ASTNode):Unit = statements :+= s
   def removeStatement(s: ASTNode):Unit = statements = statements filterNot {_ == s}

   def createNestedSubSequence():Unit = {
      val subSequence = new StatementSequence()
      statements.tail.foreach { stmt =>
         removeStatement(stmt)
         subSequence.addStatement(stmt)
      }
      addStatement(stmt)
   }
}
```

Solution

The self-call structure (1P) should be documented (moveRow calls addRow and deleteRow) (1P) The following implementation breaks if `createNestedSubSequence` is changed to an implementation not using `addStatement` or `removeStatement`. (1P)

```
case Object NOP extends ASTNode
class ExtraNOPInStatementSequence extends StatementSequence {
  override def addStatement(s: ASTNode):Unit = {
    super.addStatement(s)
    super.addStatement(NOP())
  }
}
```

Name: ____________________ Matriculation Number: ⊔⊔⊔⊔⊔⊔⊔

c) Traits and Mixin Composition 6P

Given the following class and trait definitions, give the class linearization for all traits and classes. You have to provide all intermediate steps of the linearization and you won't get any points for just writing down the final solution. The linearization of class D defined in the following class hierarchy

```
class A
trait B extends A
class C extends A
class D extends C with B
```

should look like the following:

```
Lin(A) = {A, AnyRef, Any}
Lin(B) = {B, Lin(A)}
= {B, A, AnyRef, Any}
Lin(C) = {C, Lin(A)}
= {C, A, AnyRef, Any}
Lin(D) = {D, Lin(B) ≫ Lin(C)}
= {D, {B, A, AnyRef, Any} ≫ {C, A, AnyRef, Any}}
= {D, B, C, A, AnyRef, Any}
```

Reuse linearization results if possible (e.g. the linearization of D can reuse the result of the linearization for B and C). You can use the following abbreviations for the traits and classes.

- Expr
- Print
- Eval
- And
- Or
- Not
- BinExp
- PrintBinExp

```
abstract class Expr
trait Printable extends Expr {
   def print():Unit
}
trait Evaluable extends Expr {
   def eval:Boolean
}
trait BinExp extends Expr {
   def left:Expr
   def right:Expr
}
trait PrintBinExp extends Expr with Print

class And(...) extends Expr with BinExp with Print with Eval {...}
class Or(...) extends Expr with Print with Eval {...}
class Not(...) extends Expr with Eval {...}
class Nand(...) extends Expr with PrintBinExp {...}
```

(Code excerpt to reduce page turning)

```
abstract class Expr
trait Print extends Expr {...}
trait Eval extends Expr {...}
trait BinExp extends Expr {...}
trait PrintBinExp extends BinExpr with Printable

class And(...) extends Expr with BinExp with Print with Eval {...}
class Or(...) extends Expr with Eval with Print {...}
class Not(...) extends Expr with Eval {...}
class Nand(...) extends Expr with PrintBinExp {...}
```

Solution

```
Lin(Expr)  = {Expr, AnyRef, Any} (0.5P)

Lin(Print)  = {Print, Lin(Expr)} =
= {Print, Expr, AnyRef, Any} (0.5P)

Lin(Eval)  = {Eval, Lin(Expr)}
= {Eval, Expr, AnyRef, Any} (0.5P)

Lin(BinExp) = {BinExp, Lin(Expr)}
= {BinExp, Expr, AnyRef, Any} (0.5P)

Lin(PrintBinExp) = {PrintBinExp, Lin(Print) ≫ Lin(BinExpr)}
= {PrintBinExp, {Print, Expr, AnyRef, Any} ≫ {Expr, AnyRef, Any}}
= {PrintBinExp, Print, Expr, AnyRef, Any} (0.5P)



Lin(And)  = {And, Lin(Eval) ≫ Lin(Print) ≫ Lin(BinExpr) ≫ Lin(Expr)}
= {And, {Eval, Expr, AnyRef, Any} ≫ {Print, Expr, AnyRef, Any}
≫ {BinExp, Expr, AnyRef, Any} ≫ {Expr, AnyRef, Any}}
= {And, {Eval, Print, Expr, AnyRef, Any} ≫ {BinExp, Expr, AnyRef, Any}
  ≫ {Expr, AnyRef, Any}}
= {And, {Eval, Print, BinExp, Expr, AnyRef, Any} ≫ {Expr, AnyRef, Any}}
= {And, {Eval, Print, BinExp, Expr, AnyRef, Any}} (1P)

Lin(Or)  = {Or, Lin(Print) ≫ Lin(Eval)}
= {Or, {Print, Expr, AnyRef, Any} ≫ {Eval, Expr, AnyRef, Any} ≫ {Expr, AnyRef, Any}}
= {Or, Print, Eval, AnyRef, Any} (0.5P)

Lin(Not)  = {Not, Lin(Eval) ≫ Lin(Expr)}
= {Not,  {Eval, Expr, AnyRef, Any}  ≫ {Expr, AnyRef, Any}}
= {Not, Eval, Expr, AnyRef, Any} (0.5P)

Lin(Nand) = {Nand, Lin(PrintBinExp) ≫ Lin(Exp)}
= {Nand, {PrintBinExp, Print, Expr, AnyRef, Any} ≫ {Expr, AnyRef, Any}}
= {Nand, PrintBinExp, Print, Expr, AnyRef, Any} (0.5P)
```

d) Forwarding vs Delegation 3P

Sketch a template for how to implement delegation semantics in Scala (2-3 classes).

Solution

There is an additional m method that takes the caller (self) as an argument and can therefore delegate the call back to the caller.

```
trait Base {
  def m()
  def f()
}

class Callee extends Base {
  def m(self: Base) {
    ...
    self.f
  }

  def m() {
    m(this)
  }
  ...
}

class OtherClass extends Base {
  val callee = new Callee()

  def m() {
    callee.m(this)
  }
  ...
}
```

Topic 2: Design Patterns 29P

The following tasks are related to the shown code. Please, read the tasks first.

```
case class State(line: Int, registers: IndexedSeq[Int])

abstract class Program(program: IndexedSeq[Instr]) {
    def eval(): Int = {
        var state = State(0, getInitRegisterContent())
        while (state.line < program.length) {
            state = program(state.line).eval(state.registers)
        }
        state.registers(0)
    }
```

remove empty lines between defs
(here and below...)

```
    def getInitRegisterContent(): IndexedSeq[Int]
}

trait Instr {
    def eval(registers: IndexedSeq[Int]): State

    def accept(v: Visitor): Unit
}

case class Inc(line: Int, register: Int) extends Instr {
    override def eval(registers: IndexedSeq[Int]): State = {
        val newRegisters = registers.updated(register, registers(register) + 1)
        State(line + 1, newRegisters)
    }

    override def accept(v: Visitor): Unit = v.visit(this)
}

case class CondDec(line: Int, register: Int, jumpTgtLine: Int) extends Instr {
    override def eval(registers: IndexedSeq[Int]): State = {
        val content = registers(register)
        if (content == 0) {
            State(jumpTgtLine, registers)
        } else {
            val newContents = registers.updated(register, content - 1)
            State(line + 1, newContents)
        }
    }

    override def accept(v: Visitor): Unit = v.visit(this)
}

case class GoTo(line: Int, nextLine: Int) extends Instr {
    override def eval(registers: IndexedSeq[Int]) = State(nextLine, registers)

    override def accept(v: Visitor): Unit = v.visit(this)
}
```

Name: ______________________ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴

```
trait Logger {      make it a one-liner
   def log(obj: Object): Unit
}

object JDKLoggingWrapper extends Logger {
    override def log(obj: Object): Unit = {
        val lvl = java.util.logging.Level.INFO
        java.util.logging.Logger.getGlobal.log(lvl, obj.toString)
    }
}

case class TracingInstr(instr: Instr, logger: Logger) extends Instr {
    override def eval(registers: IndexedSeq[Int]): State = {
        logger.log(s"$this($registers)")
        instr.eval(registers)
    }

    override def accept(v: Visitor): Unit = instr.accept(v)
}

trait Visitor {
    def visit(inc: Inc)
    def visit(dec: CondDec)
    def visit(goTo: GoTo)
}
```

Name: ________________________________ Matriculation Number: ___ ___ ___ ___ ___ ___ ___

a) "Pattern Recognition" 13P

Identify ***all*** design patterns used in the previous Scala code. For each pattern instance, shortly state which classes are responsible for what role in the pattern. If the pattern has methods that are characteristic, name the corresponding methods in the code. It is likely that you can not make a final decision for all pattern, e.g. because of missing instantiations or usages. In such a case name all alternative patterns and state one criteria which that is required to make a final decision about the implemented pattern.

Solution

- 2P Factory Method: all case classes have an apply method which is a factory method for creating expressions; (1P) for apply method and (1P) for the case class.
- 2P Template method: AbstractClass = Program (0.5P),TemplateMethod = eval (1P), PrimitiveOperation = getRegisterContent (0.5P), ConcreteClass = Not shown in code
- 3P Visitor: Instr = Element (0.5P), subclasses of Instr = ConcreteElement (0.5P), Visitor = Visitor (1P), accept(visitor: Visitor) = accept method (0.5P), visit(...) = visit methods (0.5P).
- 3P Decorator: Component = Instr (1P), ConcreteDecorator = Decorator = TracingInstr (1P), Operation = eval (1P)
- 3P Proxy: Subject = Instr (1P), Proxy = TracingInstr (1P), Request = eval (1P) (Alternative to Decorator, the points are not added together)
- 3P Adapter: Target = Logger (1P), Adapter = JDKLoggingWrapper (1P), Adaptee = java.util.logging.Logger (1P)
- 3P Strategy: Context = TracingInstr (1P), Strategy = Logger (1P), ConcreteStrategy = JDKLoggingWrapper (1P)

In total, we have 16 points here but only give 12 points maximum, i.e., a student can obtain all points even if he omits something from the above list.

The `TracingInstr` could be either a decorator or a proxy (0.5P) as there exists no client that the demostrate the intended usage (0.5P).

Name: ______________________________ Matriculation Number: ⌴⌴⌴⌴⌴⌴⌴

b) Assess the Design 10P

1. Give one example where the design is open for extension but closed for modification and one example where the design is open for extension but not closed for modification. Provide a concrete example for both extensions. (4P)

2. Do you see any potential to apply the Interface Segregation Principle to the trait `Instr`? Justify your answer! (2P)

3. Are there any design flaws regarding the class `TracingInstr`? Justify your answer! (2P)

4. Are there any immediate violations of the Liskov Substitution Principle? (1P)

You can assume that no

Solution

1. The design is open for extensions and closed for modification when extending `Instr` (1P) with new functionality using the visitor pattern. This does not apply for adding new instructions, since new tasks would require to modify the `Visitor` class (1P). One concrete example for each case (2P).
2. Yes. (0.5P) The method accept could be extracted to a separate trait. (1.5P)
3. Yes. (0.5P) In the presence of the visitor pattern adding new instructions like `TracingInstr` is not supported. The implementations overcomes this issue by delegating the call to `accept` to its component instruction. (1.5P)
4. No. (1P)

Name: ______________________________ Matriculation Number: ☐☐☐☐☐☐☐

c) Sortable Sets 6P

Sketch two designs for a generic `SortableBag` trait in Scala that supports sorting. Assume that you have a base class `Bag[+A]`, which is covariant in `A`. The outcome of the sorting method should be a fresh set and the order of elements depends on how elements are compared. One of your designs should use the template pattern and the other the strategy pattern to parameterize the sorting process with a comparison operation.

It is sufficient to write two Scala class definitions for the above design. You do not have to implement the sorting method, but indicate in a comment where you use the respective comparison operation.

Discuss the advantages and disadvantages of each design in not more than 4 sentences.

Solution

```
// template
trait SortableBag[+A] extends Bag[A] {
def compare[B >: A](a1: B, a2: B): Int

def sort(): SortableBag[A] = ... // using compare
}

// strategy
trait SortableBag[+A] extends Set[A] {
def sort[B >: A](cmp: (B, B) => Int): SortableBag[A] = ... // using cmp
}
```

2P for each class:

1. 0.5P for extending `Bag`
2. 0.5P for **either** using invariant `A` **or** covariant `A` and bounded `B`.
3. 1P for correct pattern design.

In the template design, comparison is a parameter of the set itself. To sort elements of an existing set with a different comparison operation, we have to create a new subclass of the bag and copy the original bag (1P). In the strategy design, comparison is a parameter of the sorting operation and the above problem does not arise (1P).

Name: ______________________ Matriculation Number: ☐☐☐☐☐☐☐

Topic 3: Reactive Programming **14P**

a) Signals vs Events 3P

Explain the difference between signals (i.e., time-varying values) and events in not more than two sentences.

For each of the following abstractions, say whether it is conceptually a signal (i.e., time-varying value) or rather an event.

1. The selected cell in a spreadsheet application.
2. The charge of a cell phone battery.
3. A brightness sensor.
4. The inventory of a video game character.

Solution

A signal is defined/holds a value at all times. (0.5P)

An event on the other hand is defined/holds a value only at concrete points in time (propagation turns). (0.5)

1. Signal
2. Signal
3. Signal
4. Signal

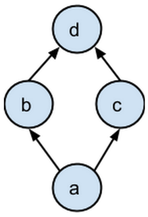
Name: ______________________________ Matriculation Number: ␣␣␣␣␣␣␣

b) Glitches 5P

1. Explain what glitches are.
2. Describe **two** reasons why glitches are not desirable in a reactive programming language.
3. Show an example of a graph configuration where a glitch can occur. Show **two** propagation orders, one with a glitch and one without.

Solution

1. Glitches are temporary spurious values due to the propagation order in the graph. They occur when a node is updated using some new and some old values in the predecessors. (1P)
2. two of them are enough: (1P each, max is 2P)
   - nodes are evaluated redundantly
   - spurious values have no meaning
   - events can erroneously fire more than once
3. glitch: a → b → d → c → d or a → c → d → b (0.5P)
   no glich: a → b → c → d or a → c → b → d (0.5P)



(1P)

c) Video animation 6P

Video animations consist of still images, called ***frames***, rendered at a high rate – ideally around 60 frames per second. For debugging and logging purposes, video games can often display how many frames per second they currently render. Since the render time can vary from frame to frame by a certain amount, the ***frames per second*** number is often averaged over the last n frames.

Given a signal

```
val frameTime: Signal[Int]
```

which holds the time in milliseconds it took to completely render the last frame. You can assume that two consecutive frames never take the same time to render. Implement a signal `fps`, which holds the current value of ***frames per second***, averaged over the last 50 frames. At the beginning of the animation, when less than 50 frames have been rendered in total, average over all previously rendered frames.

Solution

Using `last`:

```
val frameTime: Signal[Int]
val e = frameTime.changed // 1P
val window = e.last(50) // 2P
// 1P for signal constructor, 0.5P for sum apply,
// 0.5P for length apply, 1P for correct logic:
val fps = Signal { 1000.0 / (window().sum / window().length) }
   // or: Signal { 1000.0 * window().length / window().sum }
```

Using `fold` (encoding `last`) with a queue:

```
// 0.5P for using fold, 0.5P for initial value
// 0.5P for special case < or >= 50, 0.5P for queue construction
val window = e.fold(new Queue[Int]) { (acc, t) =>
  if(acc.length >= 50) acc.dequeue
  acc enqueue t
  acc
}
```

Using `fold` (encoding `last`) with a list:

```
// initial value can be Nil, even though that does not type check
// grading same as for the queue version
val window = e.fold(Nil) { (acc, t) =>
  val res = if(acc.length < 50) acc else acc.tail
  res ::: List(t)
}
```

Using `fold` to go the whole way:

```
val frameTime: Signal[Int]
val e = frameTime.change // 1P
val fps = e.fold(0) { (acc : (Double, Int, Int), t) =>
  val i = if (acc._2 < 50) acc._2+1 else 50
  val res = acc._1 -
} map (_._1)
```

Name: ____________________________ Matriculation Number: ☐☐☐☐☐☐☐

Topic 4: Software Design — 21P

Your task is to design the architecture for a smart home controller. Use design patterns and follow design principles to adhere to the description below. Your smart home controller solution should provide multiple components:

1. **Devices.** A smart home combines a variety of devices that can be controlled. For each device it should be possible to activate and de-activate the device and query the device whether it is currently active. For now, we only want to consider one kind of devices, namely lights. Within our smart home, we want to be able to control different kinds of light using the previously described interface. For example, we can have simple lights whose internal state can just be toggled

```scala
class SimpleLight {
  var turnedOn: Boolean = false

  def toggle: Unit = {
    turnedOn = !turnedOn
  }
}
```

   or we can have dimmable lights whose brightness can be adjusted in steps (`level = 0` indicates that the light is powered off, `level = 100` indicates that the light is fully powered on).

```scala
class DimmableLight {
   var level = 0
 }
```

2. **Scenarios.** Within the smart home it should be possible to define multiple scenarios. Scenarios describe which actuators are to be activated based on some pre-condition, where a condition can be evaluated to some Boolean value. The simplest base scenario we want to support is to activate a light with no pre-condition. For example, activating the scenario "Kitchen" turns on all lights in the kitchen, while the scenario "Living Room" turns on all lights in the living room. A scenario has a name and comprises a set of pre-conditions and devices and provides a way to be activated, where the latter activates all devices that are part of the scenario if the pre-condition is satisfied.

   In addition to simple scenarios as described before, we also want to support complex scenarios that can additionally contain other scenarios. For example, the scenario "Party" can contain the scenarios "Kitchen" and "Living Room". When this scenario is activated, both sub-scenarios are activated, which means that both the lights in the kitchen and in the living room are powered on.

3. **Creating Scenarios.** Finally, your implementation should provide a way to create scenarios step-by-step, by adding arbitrary many devices, pre-conditions or scenarios.

Sketch your design as Scala code. Use expressive class and method names and **shortly document the used patterns and the responsibilities of each class you add in the pattern.**

The sketch has to contain code for:

- Representing devices
- Integrating devices with variable interfaces, using the example of lights
- Representing scenarios
- Building scenarios

General Hints:

- You can omit method implementations that are not necessary for your pattern(s) by using `???`.
- For the devices it is sufficient if you show how to integrate `SimpleLight` or `DimmableLight`, however, your implementation should be open for extension with respect to other lights with different interfaces.

- For the scenarios, you can assume there is a method `isTrue` which returns true if all conditions contained in the set are satisfied

Name: ______________________ Matriculation Number: ␣␣␣␣␣␣␣

Solution

```
trait Device {
  def isActive: Boolean

  def activate(): Unit

  def deactivate(): Unit
}

trait Light extends Device

case class SimpleLightAdapter(sl: SimpleLight) extends Light {
  override def isActive: Boolean = sl.turnedOn

  override def activate(): Unit = if (!sl.turnedOn) sl.toggle

  override def deactivate(): Unit = if (sl.turnedOn) sl.toggle
}

case class DimmableLightAdapter(dl: DimmableLight) extends Light {
  override def isActive: Boolean = dl.level > 0

  override def activate(): Unit = dl.level = 100

  override def deactivate(): Unit = dl.level = 0
}

class SimpleLight {
  var turnedOn: Boolean = false

  def toggle: Unit = {
    turnedOn = !turnedOn
  }
}

class DimmableLight {
  var level = 0
}

trait Condition {
  def eval: Boolean
}

trait Scenario {
  protected var preConditions = List[Condition]()
  protected var devices = List[Device]()

  def name: String

  def activate(): Unit

  def isActive: Boolean = devices.forall(_.isActive)

  def addPreCondition(c: Condition): Unit = {
```

```
    preConditions = c :: preConditions
  }

  def addDevice(d: Device): Unit = {
    devices = d :: devices
  }
}

case class SimpleScenario(name: String) extends Scenario {
  override def activate(): Unit  = {
    println(s"Activating $name Scenario")
    devices.foreach(_.activate())
  }
}

case class ComplexScenario(name: String) extends Scenario {
  var subScenarios = List[Scenario]()

  def addScenario(s: Scenario): Unit = {
    subScenarios = s :: subScenarios
  }

  override def activate(): Unit = {
    if (preConditions.map(_.eval).forall(b => b)) {
      subScenarios foreach (_.activate())
      devices foreach (_.activate())
    }
  }
}

class ScenarioBuilder {
  var scenarios: List[Scenario] = List()
  var preConditions: List[Condition] = List()
  var devices: List[Device] = List()

  def addScenario(s: Scenario) = {
    scenarios = s :: scenarios
    this
  }

  def addPreCondition(c: Condition) = {
    preConditions = c :: preConditions
    this
  }

  def addActuator(d: Device) = {
    devices = d :: devices
    this
  }

  def build(name: String): Scenario = {
    val scenario = if (scenarios.isEmpty) ComplexScenario(name) else SimpleScenario(name)
    preConditions.foreach(c => scenario.addPreCondition(c))
    devices.foreach(a => scenario.addDevice(a))
```

```
        scenario
      }
    }
```