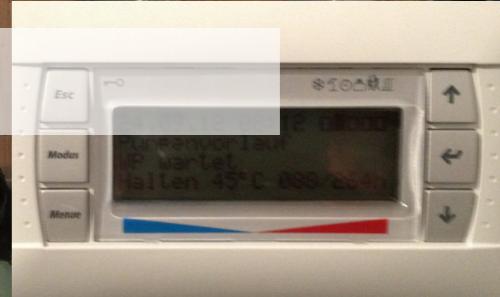


Winter Semester 16/17

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Smart Home Example



A First Sketch (I/II)

```
abstract class Location {  
    private List<Shutter> shutters; // FEATURE: DARKENING  
    private List<Light> lights; // FEATURE: LIGHTING  
  
    public Location(List<Shutter> shutters, List<Light> lights) {  
        this.shutters = shutters;  
        this.lights = lights;  
    }  
  
    public List<Shutter> shutters() { return shutters; }  
    public List<Light> lights() { return lights; }  
}  
  
abstract class CompositeLocation<L extends Location> extends Location {  
    private List<L> locations;  
  
    public CompositeLocation(List<L> locations) {  
        super(shutters(locations), lights(locations));  
        this.locations = locations;  
    }  
    private static List<Light> lights(List<? extends Location> locs) {...}  
    private static List<Shutter> shutters(List<? extends Location> locs) {...}  
  
    public List<L> locations() { return locations; }  
}
```



A First Sketch (II/II)

```
class Room extends Location {  
    public Room(List<Shutter> shutters, List<Light> lights) {  
        super(shutters, lights);  
    }  
}  
  
class Floor extends CompositeLocation<Room> {  
    public Floor(List<Room> locations) { super(locations); }  
}  
  
class House extends CompositeLocation<Floor> {  
    public House(List<Floor> locations) { super(locations); }  
}  
  
class Main {  
    public static void main(String[] args) {  
        House house = new House(null);  
        List<Floor> floors = house.locations();  
    }  
}
```



A Second Sketch (I/II)

We try to achieve feature decomposition.



```
interface Location { }

interface CompositeLocation<L extends Location> extends Location {
    abstract List<L> locations();
}

class Room implements Location { }

class Floor implements CompositeLocation<Room> {
    private List<Room> rooms;

    public List<Room> locations() { return rooms; }
}

class House implements CompositeLocation<Floor> {
    private List<Floor> floors;

    public List<Floor> locations() { return floors; }
}
```

A Second Sketch (II/II)

We try to achieve feature decomposition.

```
interface LocationWithLights extends Location {  
    List<Light> lights();  
}
```

```
class RoomWithLights extends Room implements LocationWithLights {  
    private List<Light> lights;  
    public List<Light> lights() { return lights; }  
}
```

```
abstract class CompositeLocationWithLights<LL extends LocationWithLights>  
    implements CompositeLocation<LL> {
```

```
    public List<Light> lights() {  
        List<Light> lights = new ArrayList<Light>();  
        for (LocationWithLights child : locations()) {  
            lights.addAll(child.lights());  
        }  
        return lights;  
    }
```



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] =  
        synchronized { super.get(key) }  
    abstract override def set(key: A, value: B) : Unit =  
        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)  
    }  
  
object 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:

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

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

2nd Rule

The order is relevant!

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

Abstract Type

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

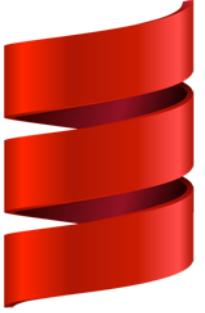
abstract class Animal {
    type SuitableFood <: Food
    def createFood : SuitableFood
    def eat(food: this.SuitableFood) : Unit
}

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

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!





A Third Sketch

(Let's start with the translation of the Java Code)

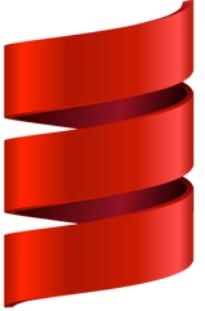
```
trait Shutter
```

```
trait Light
```

```
abstract class Location {  
    def shutters: List[Shutter]  
    def lights: List[Light]  
}
```

```
abstract class CompositeLocation[L <: Location] extends Location {  
    def lights: List[Light] = locations.flatMap(_.lights)  
    def shutters: List[Shutter] = locations.flatMap(_.shutters)  
    def locations: List[L]  
}  
class Room(  
    val lights: List[Light],  
    val shutters: List[Shutter]) extends Location  
class Floor(val locations: List[Room]) extends CompositeLocation[Room]  
class House(val locations: List[Floor]) extends CompositeLocation[Floor]
```

```
object Main extends App {  
    val house = new House(new Floor(new Room(Nil, Nil) :: Nil) :: Nil)  
    val floors: List[Floor] = new House(Nil).locations  
}
```



A Third Sketch (Base)

```
trait Building {
```

```
  trait TLocation {}  
  type Location <: TLocation
```

Enable the refinement of TLocation!

```
  trait TRoom extends TLocation  
  type Room <: TRoom with Location  
  def createRoom(): Room
```

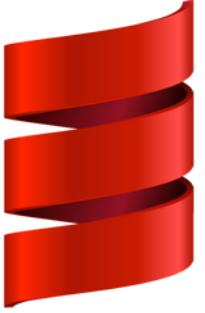
We need a Factory method to create
(yet unknown) rooms.

```
  trait CompositeLocation[L <: Location] extends TLocation {  
    def locations: List[L]  
  }
```

```
  trait TFloor extends CompositeLocation[Room]  
  type Floor <: TFloor with Location  
  def createFloor(locations: List[Room]): Floor
```

```
  trait THouse extends CompositeLocation[Floor]  
  type House <: THouse with Location  
  def createHouse(locations: List[Floor]): House
```

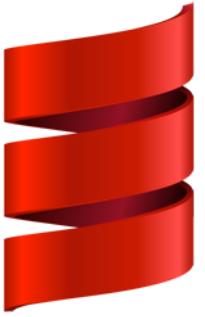
```
  def buildHouse(specification: String): House = {  
    // imagine to parse the specification...  
    createHouse(List(createFloor(List(createRoom()))))  
  }
```



A Third Sketch (Adding Lights)

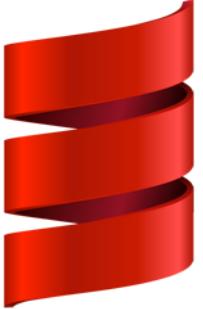
```
trait Lights extends Building {  
  
  trait TLocation extends super.TLocation {  
    def lights(): List[Light]  
    def turnLightsOn = lights.foreach(_.turnOn())  
    def turnLightsOff = lights.foreach(_.turnOff())  
  }  
  type Location <: TLocation  
  
  trait TRoom extends super.TRoom with TLocation  
  type Room <: TRoom with Location  
  
  trait CompositeLocation[L <: Location]  
    extends super.CompositeLocation[L] with TLocation {  
    def lights: List[Light] = locations.flatMap(_.lights())  
  }  
  
  trait TFloor extends super.TFloor with CompositeLocation[Room]  
  type Floor <: TFloor with Location  
  
  trait THouse extends super.THouse with CompositeLocation[Floor]  
  type House <: THouse with Location  
}
```

The trait Shutters is comparable!



A Third Sketch (Lights And Shutters)

```
trait LightsAndShutters extends Lights with Shutters {  
  
trait TLocation  
  extends super[Lights].TLocation  
  with super[Shutters].TLocation  
type Location <: TLocation  
  
trait TRoom extends super[Lights].TRoom with super[Shutters].TRoom with TLocation  
type Room <: TRoom with Location  
  
trait CompositeLocation[L <: Location]  
  extends super[Lights].CompositeLocation[L]  
  with super[Shutters].CompositeLocation[L]  
  with TLocation  
  
trait TFloor extends super[Lights].TFloor with super[Shutters].TFloor  
  with CompositeLocation[Room]  
type Floor <: TFloor with Location  
  
trait THouse extends super[Lights].THouse with super[Shutters].THouse  
  with CompositeLocation[Floor]  
type House <: THouse with Location  
}
```



A Third Sketch (Usage)

```
object BuildingsWithLightsAndShutters extends LightsAndShutters with App {  
  
    type Location = TLocation  
    type Room = TRoom  
    type Floor = TFloor  
    type House = THouse  
  
    def createRoom(): Room = new Room {  
        var lights = List.empty[Light];  
        var shutters = List.empty[Shutter]  
    }  
    def createFloor(rooms: List[Room]): Floor =  
        new Floor { val locations = rooms }  
    def createHouse(floors: List[Floor]): House =  
        new House { val locations = floors }  
  
    val h = buildHouse("three floors with 6 rooms each")  
    h.lights  
    h.shutters  
    h.locations  
    h.turnLightsOn  
}
```