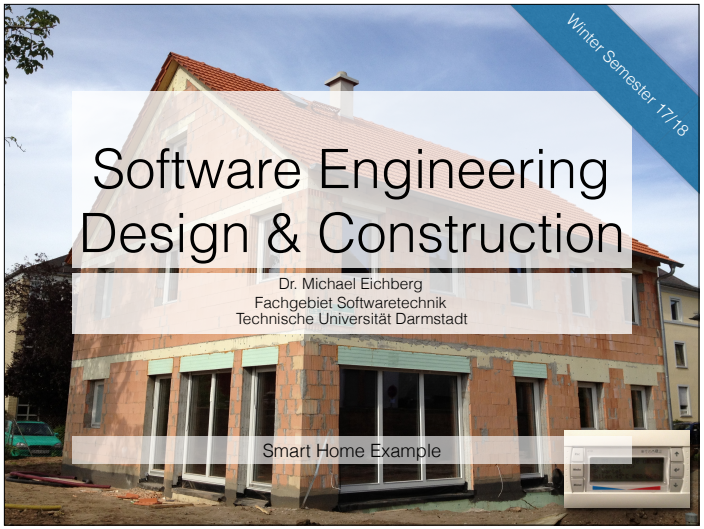

A smart home has many features that are controlled automatically:
Heating, Lighting, Shutters, …

We want to develop a very modular software that helps us to control our smart home and which only contains those parts that are actually required.

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; }
}
```

2

Location is the base class that declares the functionality that some location can offer (optionally!). **Hence, it takes multiple responsibilities.**

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();
  }
}
```

Assessment

In the prototypical solution all (optional) features are declared by the main interface (Location).

Ask yourself: Which design principle is violated?

We should split the code, if we want to be able to make functional "packages", such as heating control, lighting control, or security, optional. Consider, e.g., the case that the provider may want to sell several configurations of a smart home, each with a specific selection of features.

How to model interacting/depending features? E.g., a sensor that closes the shutters in the evening and turns on the lights.

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; }
}
```


So far we are just modeling the basic structure of a building (`House`).

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;
  }
}
```

Given the shown code/the proposed solution, we can identify several issues:

- `class FloorWithLights extends CompositeLocationWithLights and Floor`
  The class should inherit from (CompositeLocationWithLights and Floor) ? (we don't want code duplication!)
- `class HouseWithLights extends ...`
  The class should inherit from ? (we don't want code duplication!)
- Imagine that we have another additional feature; e.g., shutters and **we want to avoid code duplication**!

Ideally, we would like to have several versions of class definitions - one per responsibility - which can be "**mixed and matched**" as needed.

… In Java, we have to use a Pattern to solve the Design Problem (there is no language support!)

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 CompositeLocati
class House(val locations: List[Floor]) extends CompositeLo

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

A naive translation doesn't solve the problem!

What we want to achieve is that:

- Features that are developed independently (such as heating, cooling or lighting) can be (freely) combined
- The solution is type safe even in the presence of new optional features (which requires appropriate support by the available programming language)
- We do not duplicate code (Copy & Paste programming).

Additionally, the underlying programming language should also support separate compilation to enable us to deploy our solution independently.

Note: A house with lights is (conceptually) also a different ***type of house*** *than a house with lights and shutters and air conditioning.*

A Third Sketch (Base)

```
trait Building {

   trait TLocation {}
   type Location <: TLocation

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

   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()))))
   }
}
```

Enable the refinement of TLocation!

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

Note, that the buildHouse method constructs a House object though the concrete type is not yet known.

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 implementation of 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
}
```


Though we got the features that we wanted, the code feels like “Assembler Code” at the type level. Scala lacks support for deep, nested mixin composition (i.e., it does not support Virtual Classes/Dependent Classes).

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
}
```


Basically, in the first 4 lines we create type aliases for location, room, floor and house which "fixes" our abstract type definitions. After that we implement the factory methods as required. For the method parameter types and return types, we still use the names of the type definitions.

Example Usage

```
val r1 = BuildingsWithLightsAndShutters.createRoom()
val r0 = BuildingsWithLights.createRoom()
BuildingsWithLightsAndShutters.createFloor(List(r1, r0))
```

- For further information search for the Cake Pattern in Scala.
- More advanced language concepts such as Virtual Classes and Dependent Classes would make the solution even easier (much less boilerplate code!)