Summer Semester 2015

Software Engineering Design & Construction

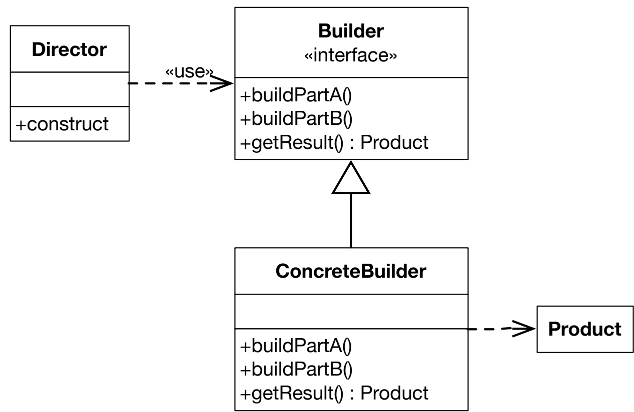
Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Builder Pattern

The Builder Pattern

Divide the construction of multi-part objects in different steps, so that different implementations of these steps can construct different representations of object

Builder - Structure



Builder defines the individual steps of the construction of Product.

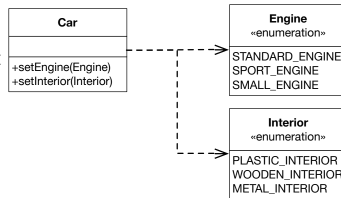
Director knows in which order to construct Product.

ConcreteBuilder implements the steps of construction.

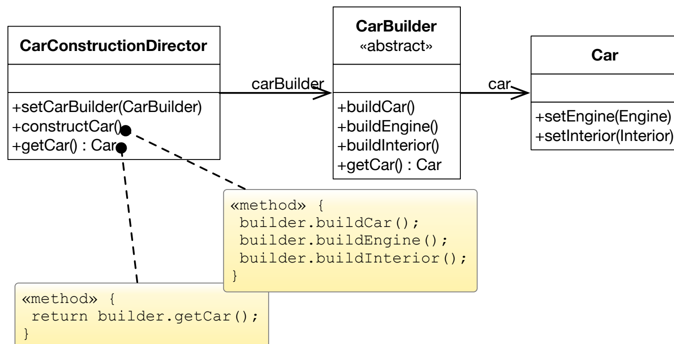
Example
Car Builder

Builder - A Car Builder

- We want to construct different types of cars.
- In this example, cars have an engine and an interior.



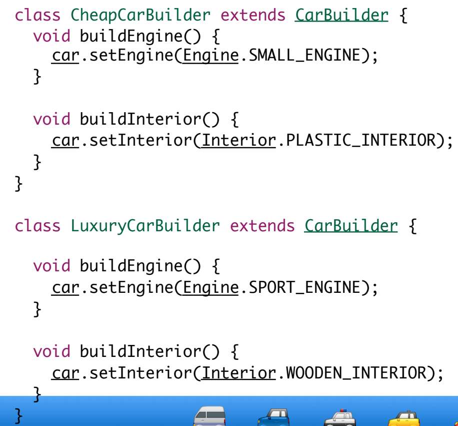
Builder - A Car Builder



CarBuilder defines the methods to construct car parts. Concrete builders must implement these methods. For convenience, the instantiation of cars (buildCar()) is implemented in CarBuilder.

CarConstructionDirector is configured with a CarBuilder and calls the construction methods in the correct order.

Two Possible Car Builders



Assessment of the Builder Pattern

Advantages:

- Creation of objects can be configured at runtime.
- Concrete builders can use complex logic.
  E.g. a car builder creating cars depending on available parts in storage.
- Good way to create composite structures.

Disadvantages:

- May yield many classes.
- Only works if all objects can be constructed using the same order.

Example
Collections

The map operation

- Takes the elements of a collection and applies a given function `f` to create the elements of the target collection.

```
Collection[E]{
   def map[T](f: E=>T):Collection[T] = {…}
}
```

- Let's assume that we want/have to transform the elements and the type of the collection at the same time.
  E.g., we want to map from a List of Strings to an Array of type `Array[SHA256]` in one step as shown in the following example:

```
val hashes : Array[SHA256] =
   List("a","b","c") map {e => SHA256(e)}
```


1. Apply the Builder Pattern

```
trait Builder[T,C[T]] {
   def add(t : T) : Unit
   def build : C[T]
}

class ListBuilder[T] extends Builder[T,List] {
   private var l : List[T] = List.empty
   def add(t : T) : Unit = l ::= t
   def build : List[T] = l
}

case class Singleton[T](t : T) {
   def map[X,C[X]](f : T => X) (builder : Builder[X,C]) : C[X] = {
      builder.add(f(t))
      builder.build
   }
}

Singleton(100).map(_.toString)(new ListBuilder)
```


Given the current solution we are now able to determine the type of the target collection that is created while we map the elements.

However, the current solution always requires that we explicitly pass in an instance of `Builder`.

2. Automatic “Selection” of the Builder
   (Using `implicit` Factories for Builders)

```
// The existing Builders are kept

trait BuilderFactory[C[_]] {
   def create[T]() : Builder[T,C]
}

implicit val lbf = new BuilderFactory[List] {
   def create[T]() = new ListBuilder[T]
}

case class Singleton[T](t : T) {
   def map[X,C[X]](f : T => X) (implicit bf : BuilderFactory[C]) : C[X] = {
      val builder = bf.create[X]()
      builder.add(f(t))
      builder.build
   }
}

Singleton(100).map(_.toString) // the (only) builder is automatically selected
```


This solution now enables us to perform a **`map`** without having to specify the target collection; however, the result type “List[String]” is not as expected. Adding further implicitly available factories and builders; e.g.

```
implicit val setbf = new BuilderFactory[Set] { def create[T]() = new SetBuilder[T] }
implicit val singletonbf = new BuilderFactory[Singleton] { def create[T]() = new SingletonBuilder[T] }
```

would only lead to the problem that the compiler is no longer able to automatically select the Factory, because the selection is ambiguous.

Final Solution

```
trait Builder[T,C[T]] {
   def add(t : T) : Unit
   def build : C[T]
}
class ListBuilder[T] extends Builder[T,List] {
   private var l : List[T] = List.empty
   def add(t : T) : Unit = l ::= t
   def build : List[T] = l
}
class SetBuilder[T] extends Builder[T,Set] {
   private var l : Set[T] = Set.empty
   def add(t : T) : Unit = l += t
   def build : Set[T] = l
}

trait BuilderFactory[C[_]] { def create[T]() : Builder[T,C] }

val lbf = new BuilderFactory[List] {
   def create[T]() = new ListBuilder[T]
}

case class Singleton[T](t : T) {
   def map[X,C[X]](
            f : T => X
      ) (    implicit bf : BuilderFactory[C]) : C[X] = {
      val builder = bf.create[X]()
      builder.add(f(t))
      builder.build
   }
}
```

```
class SingletonBuilder[T] extends Builder[T,Singleton] {
   private var l : Singleton[T] = null
   def add(t : T) : Unit = if (l != null) throw new
IllegalStateException else l = Singleton(t)
   def build : Singleton[T] = l
}

trait LowPriorityImports {
   implicit val lbf = new BuilderFactory[List] {
      def create[T]() = new ListBuilder[T]
   }
   implicit val sbf = new BuilderFactory[Set] {
      def create[T]() = new SetBuilder[T]
   }
}
object HighPriorityImports extends LowPriorityImports{
   implicit val singletonbf = new BuilderFactory[Singleton] {
      def create[T]() = new SingletonBuilder[T]
   }
}
import HighPriorityImports._

Singleton(100).map[String,List](_.toString)

Singleton(100).map(_.toString) // =>  : Singleton[String]
```


By prioritizing the implicitly available *BuilderFactories* we now have found a solution that enables a very natural usage of the map function, but still provides all possibilities!

Takeaway

- Use **Abstract Factory** for creating objects depending on finite numbers of factors you know in advance.
  E.g. if there are only three kinds of cars.
- Use **Builder** for creating complex objects depending on unbound number of factors that are decided at runtime.
  E.g. if cars can be configured with multiple different parts.

Builder vs. Abstract Factory Pattern

- Abstract Factory focuses on creating multiple objects of a common family.
  - Abstract Factory knows what object to create.
  - Configuration is fixed after deployment of the software.
- Builder focuses on creating complex objects step by step.
  - The director knows how to construct the object.
  - Configuration is chosen at runtime via the concrete builder.