

Software Engineering Design & Construction

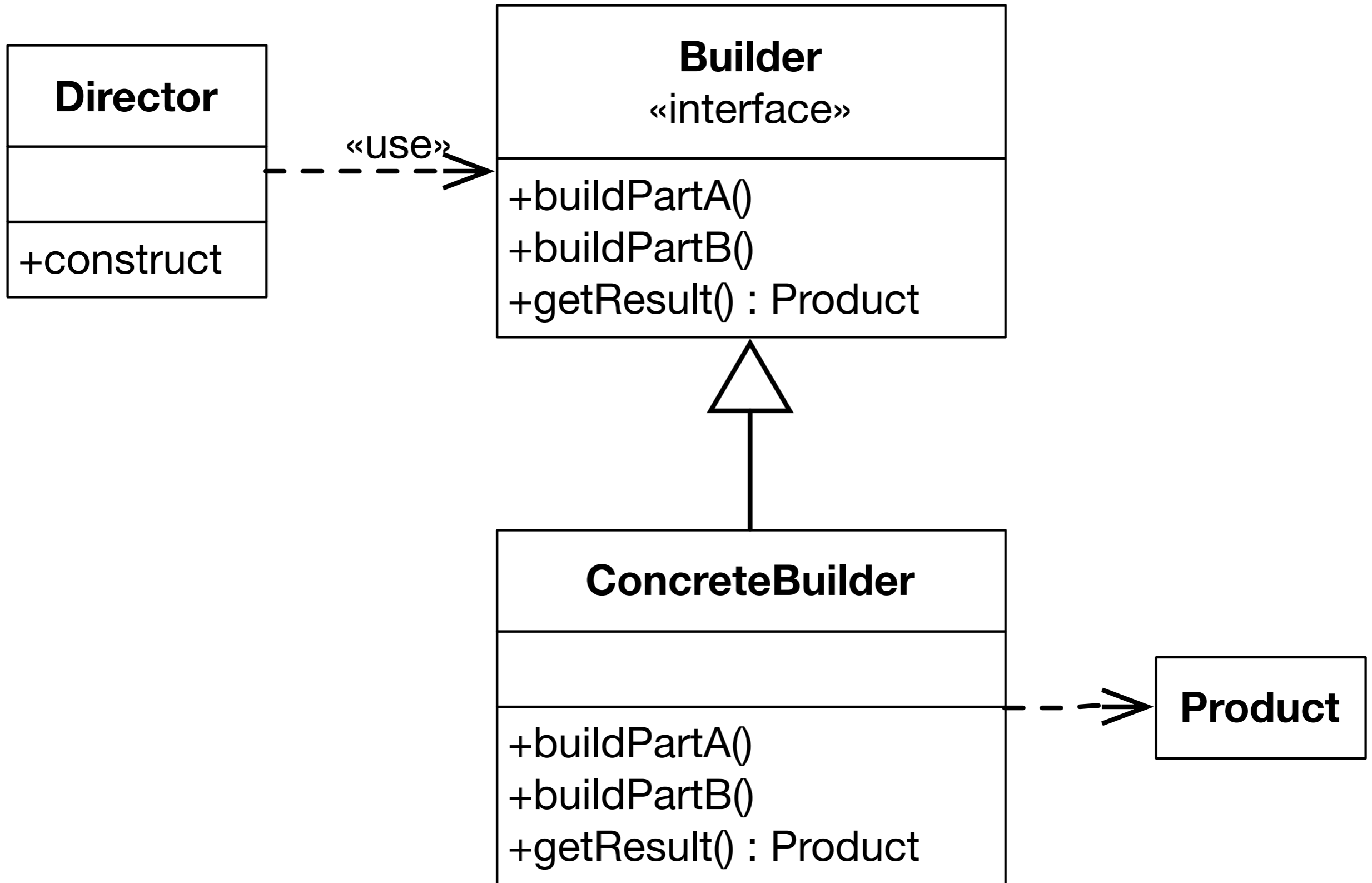
Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

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

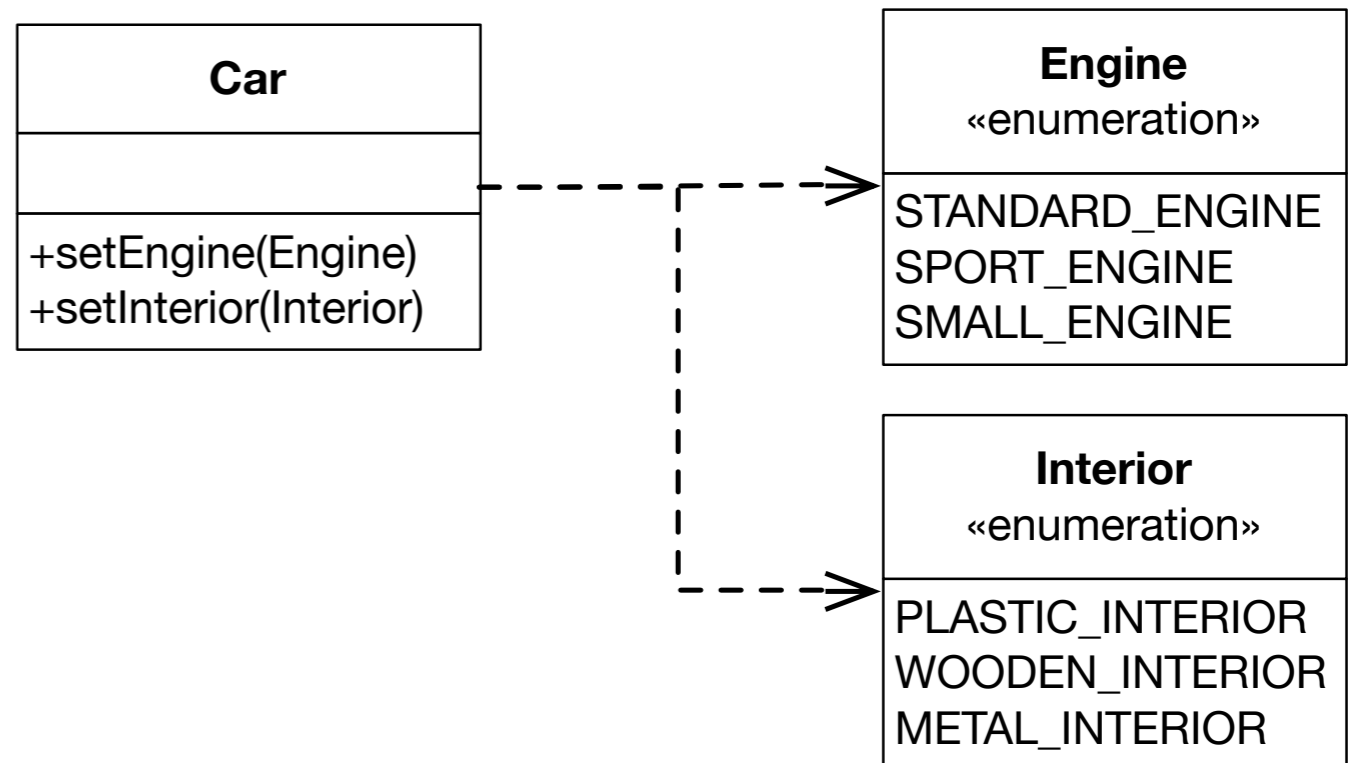


Example Car Builder

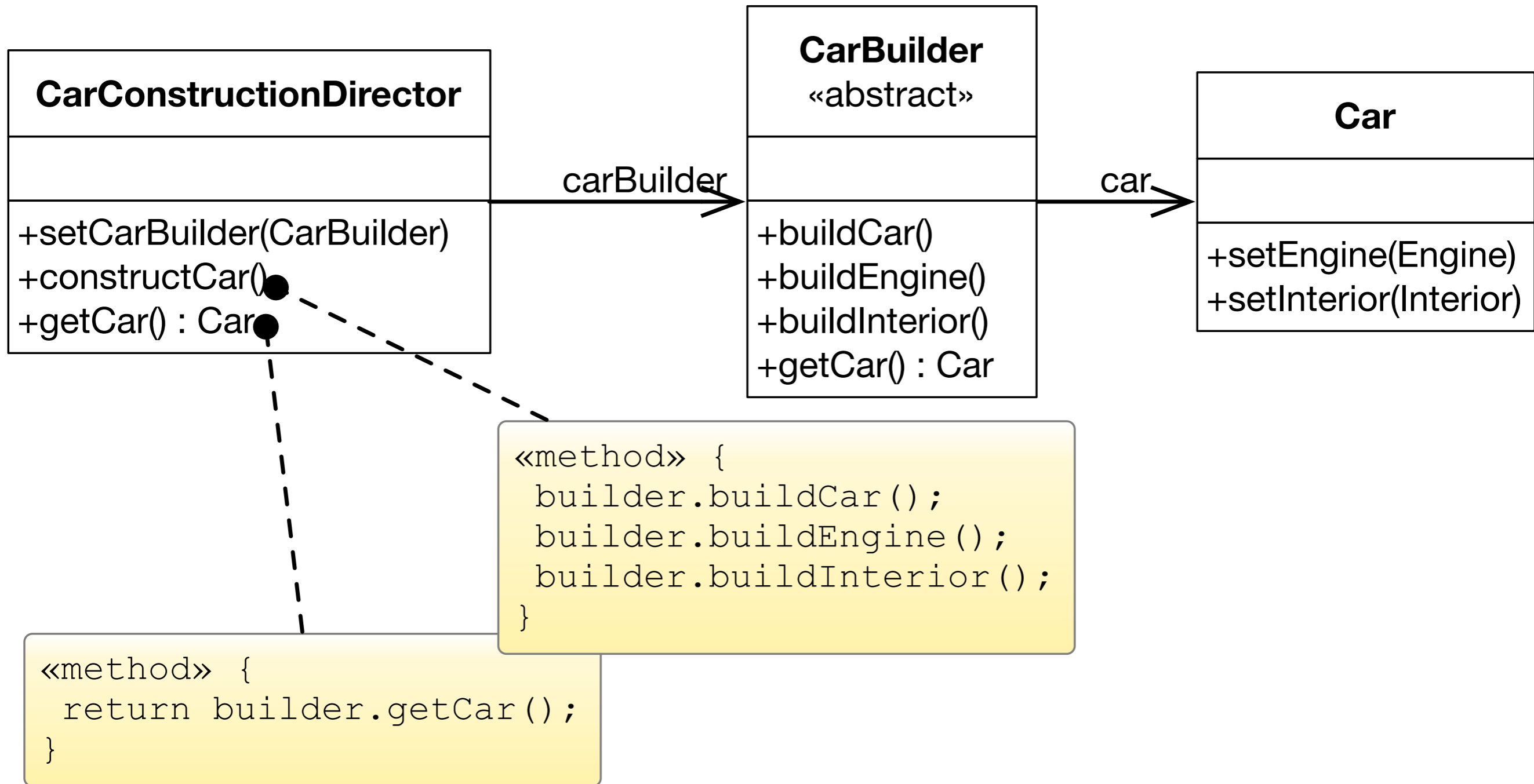


Builder - A Car Builder

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



Builder - A Car Builder



Two Possible Car Builders

```
class CheapCarBuilder extends CarBuilder {  
    void buildEngine() {  
        car.setEngine(Engine.SMALL_ENGINE);  
    }  
  
    void buildInterior() {  
        car.setInterior(Interior.PLASTIC_INTERIOR);  
    }  
}
```

```
class LuxuryCarBuilder extends CarBuilder {  
  
    void buildEngine() {  
        car.setEngine(Engine.SPORT_ENGINE);  
    }  
  
    void buildInterior() {  
        car.setInterior(Interior.WOODEN_INTERIOR);  
    }  
}
```



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)
```

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
```

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]
```

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.