Design Patterns

宋 杰

Song Jie

东北大学 软件学院

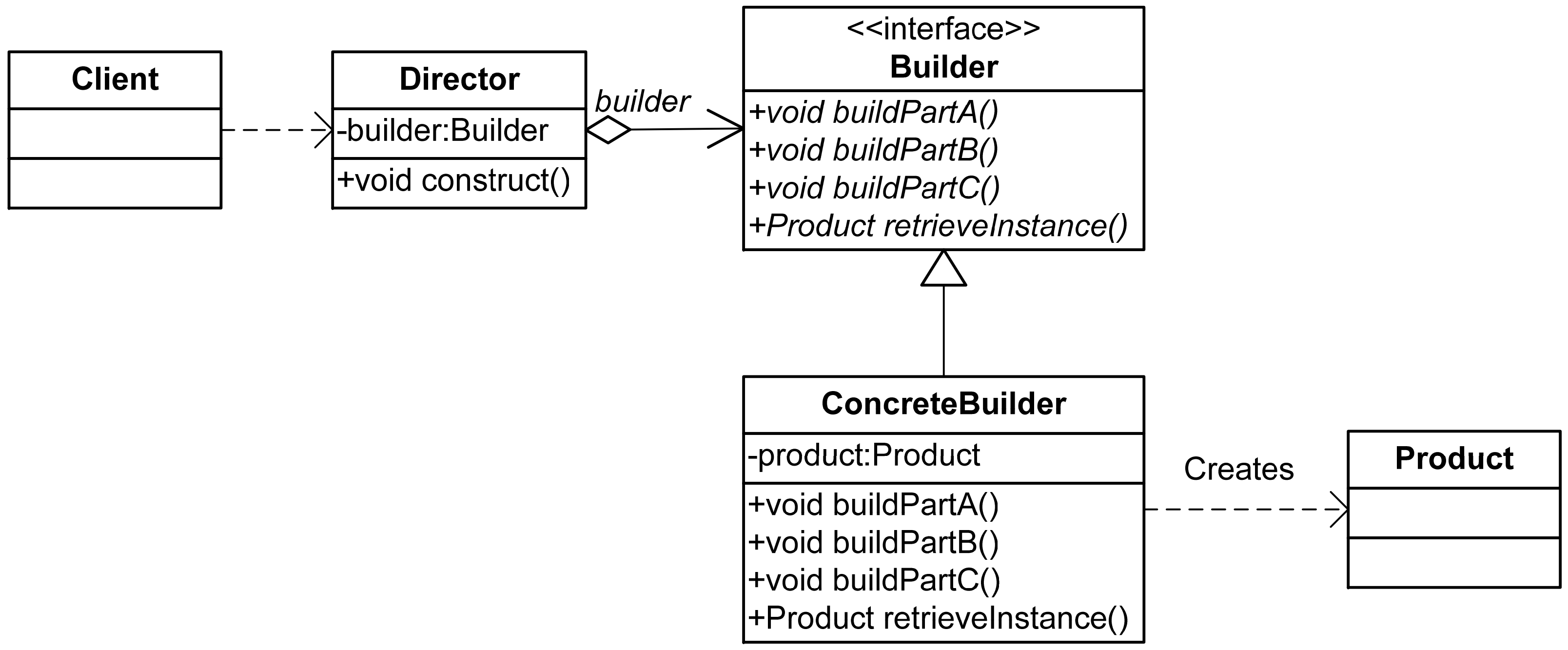
Software College, Northeastern University

4. Builder Pattern

Intent

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Representation: internal structure, compositions, required state (attribute values).
- 建造者模式将产品的结构和产品的零件建造过程对客户端隐藏起来，把对建造过程进行指挥的责任和具体建造者零件的责任分割开来，达到责任划分和封装的目的。

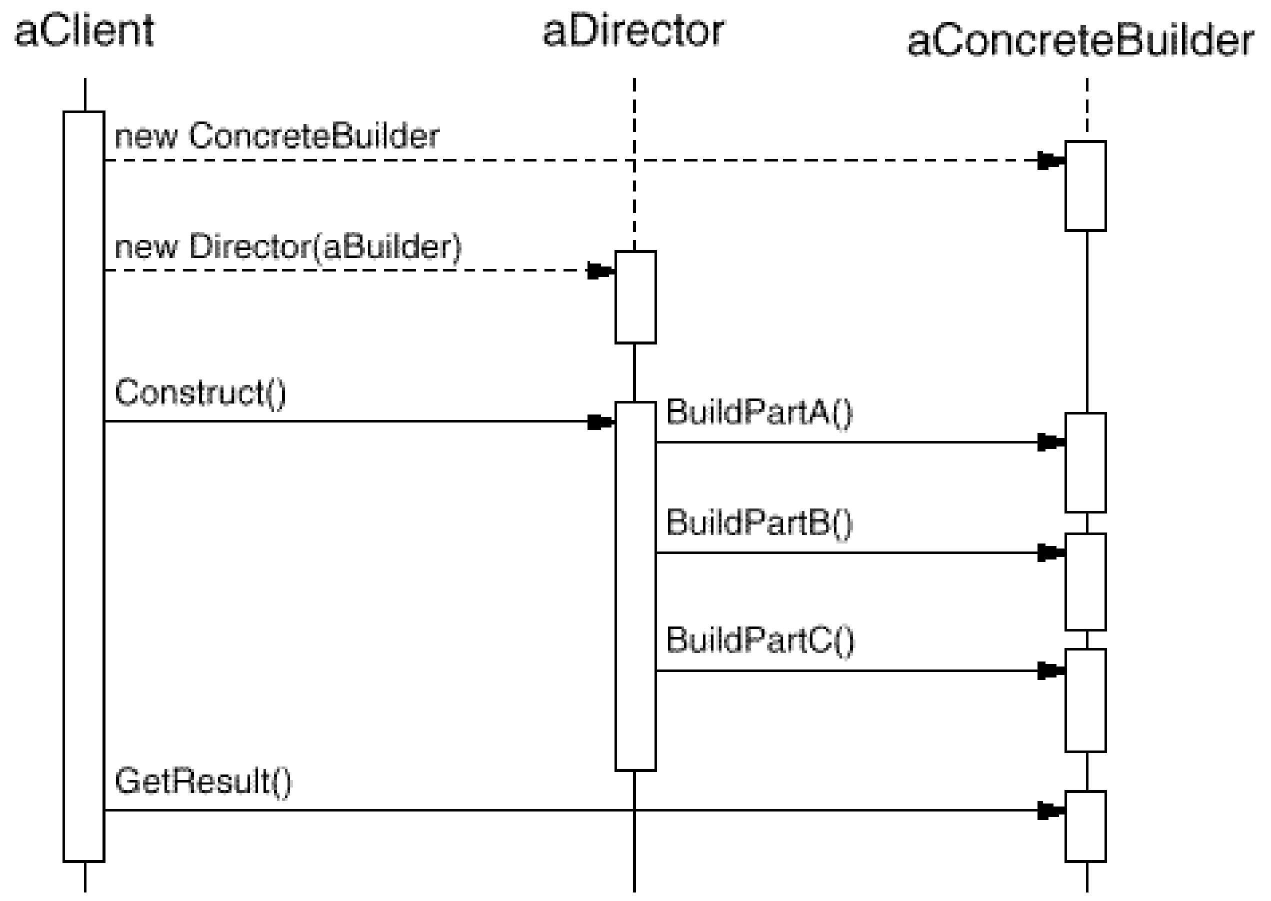
Structure


Client
Director
-builder:Builder
+void construct()
builder
<<interface>>
Builder
+void buildPartA()
+void buildPartB()
+void buildPartC()
+Product retrieveInstance()
ConcreteBuilder
-product:Product
+void buildPartA()
+void buildPartB()
+void buildPartC()
+Product retrieveInstance()
Creates
Product

Participants

- Builder: specifies an abstract interface for creating parts of a Product object.
- ConcreteBuilder:
  - Constructs and assembles parts of the product.
  - Defines and keeps track of the representation it creates.
  - Provides an interface (method) for retrieving the product.
- Director: constructs an object using the Builder interface.
- Product: represents the complex object under construction.

Collaborations



Implementation

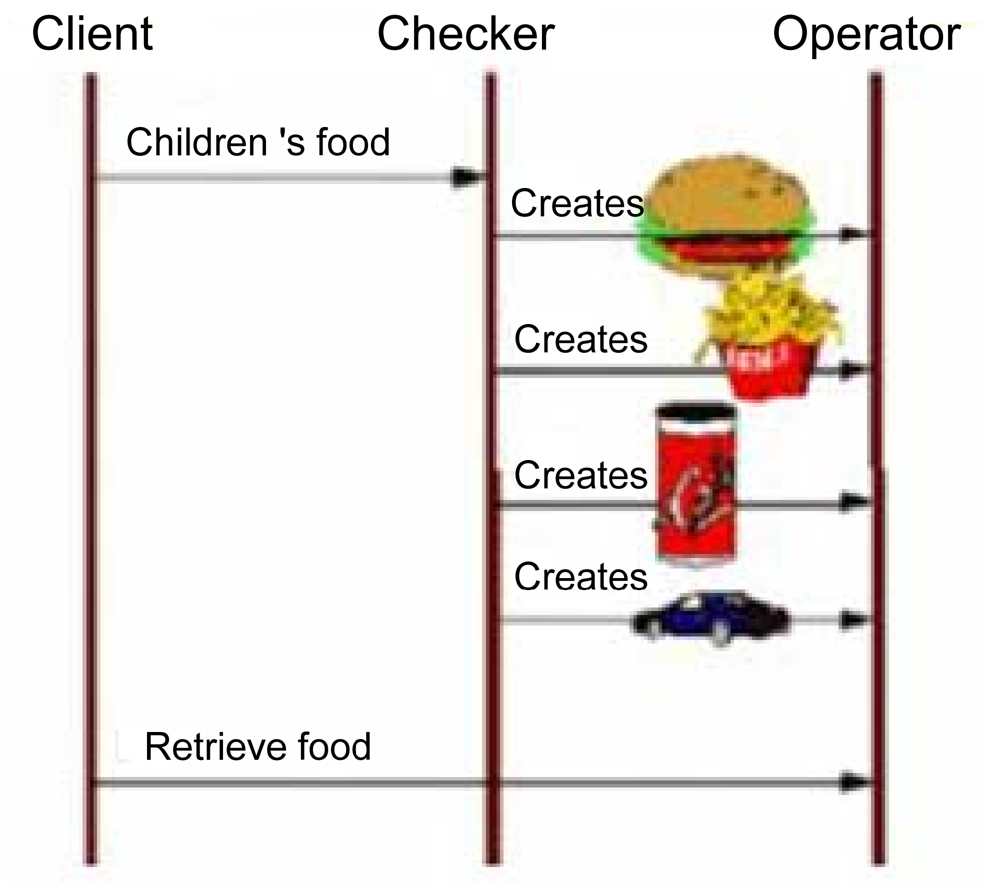
- Builder interface must be general enough to allow the construction of products for all kinds of concrete builders;
- One product correspond to one ConcreteBuilder.

Sample Code

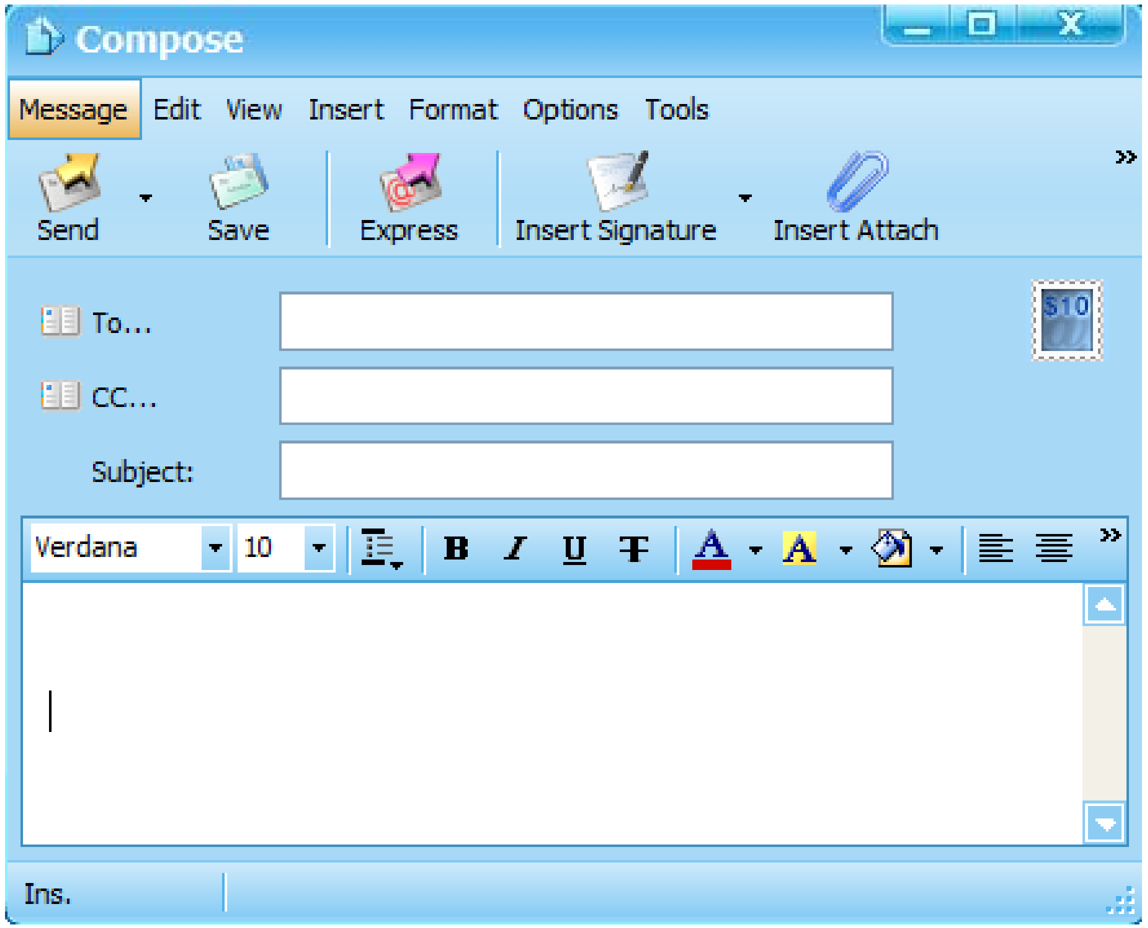
```
Builder builder = new ConcreteBuilder();
Director director = new Director( builder );

director.construct();
Product product = builder.retrieveInstance();
```

Examples 1: McDonalds



Examples 2: Email



```
class Director{
    private EmailBuilder builder;

    public Director( EmailBuilder builder){
         this.builder = builder;
    }
    public void construct(){
        builder.to("songjie@mail.neu.edu.cn");
        //more invocations
    }
}
interface EmailBuilder {
    public void to(String value);
    public void from(String value);
    public void organization(String value);
    public void plainText(String content);
    public void jpegImage(Image content) ;
    public void attachment(File file) ;
    public Email retrieveEmail() ;
}

interface Email{
}
```

Consequences

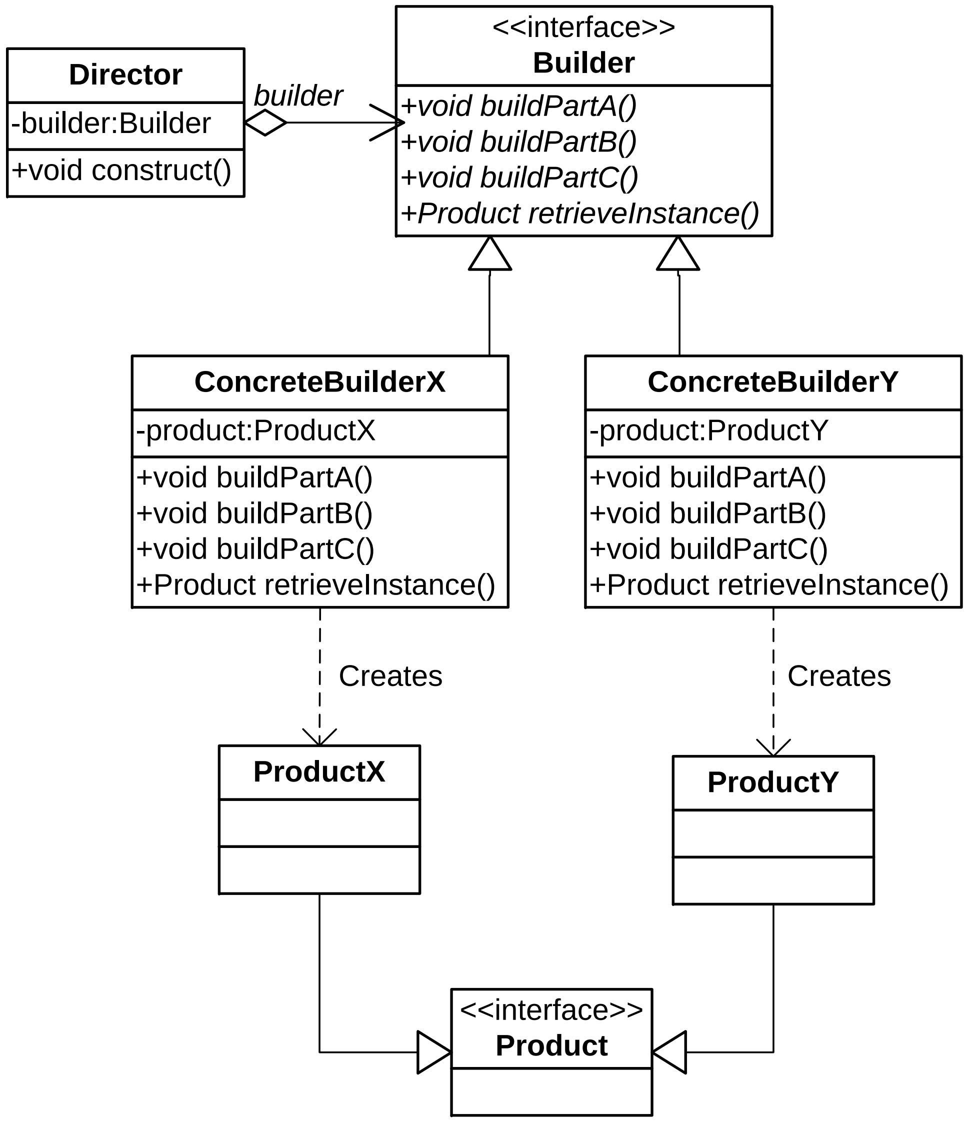
- It lets you vary a product's representation.
  - The builder pattern can provide the director with an abstract builder for constructing the product.
- It isolates code for construction and representation.
  - The builder pattern improves modularity by encapsulating the way a complex object is constructed and represented. Clients needn't know anything about the classes that define the product's internal structure; such classes don't appear in builder's interface.
- It gives you finer control over the construction process.
  - The builder pattern constructs the product step by step under the director's control. Only when the product is finished does the client retrieve it from the builder.

Applicability

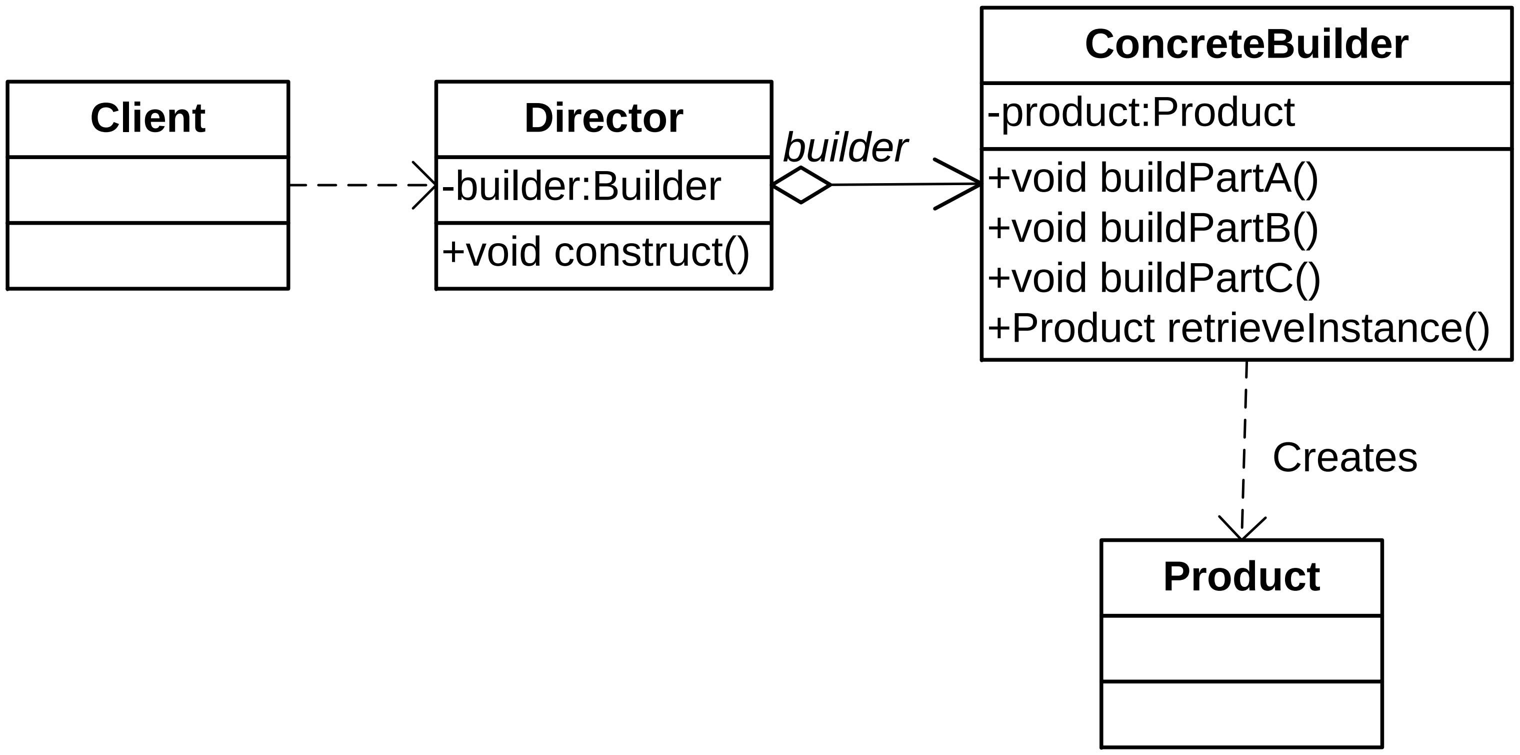
- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- The construction process must allow different representations for the object that's constructed.
  - The parts are independent and should be created orderly;
  - The parts are dependent but some parts are uncertain until they are created;
  - some parts are uneasy to get.

Variation 1: Multiple Products

- Builder is not Factory, it is used for creating a complex product with optional representation, but also suitable for multiple products when.
  - Each product satisfy the applicabilities of builder.
  - Each product have uniform interface for *retrieveInstance()*;
  - Products share compositions, thus Builder can provide uniform *buildParts()* method ;
- When products have different compositions:
  - empty implementations of unrelated *buildParts()* in ConcreteBuilder.
- When products have different interfaces:
  - Multiple *retrieveXXXInstance()* methods in Builder, and empty implementations of some in *retrieveXXXInstance()* in unrelated ConcreteBuilder.

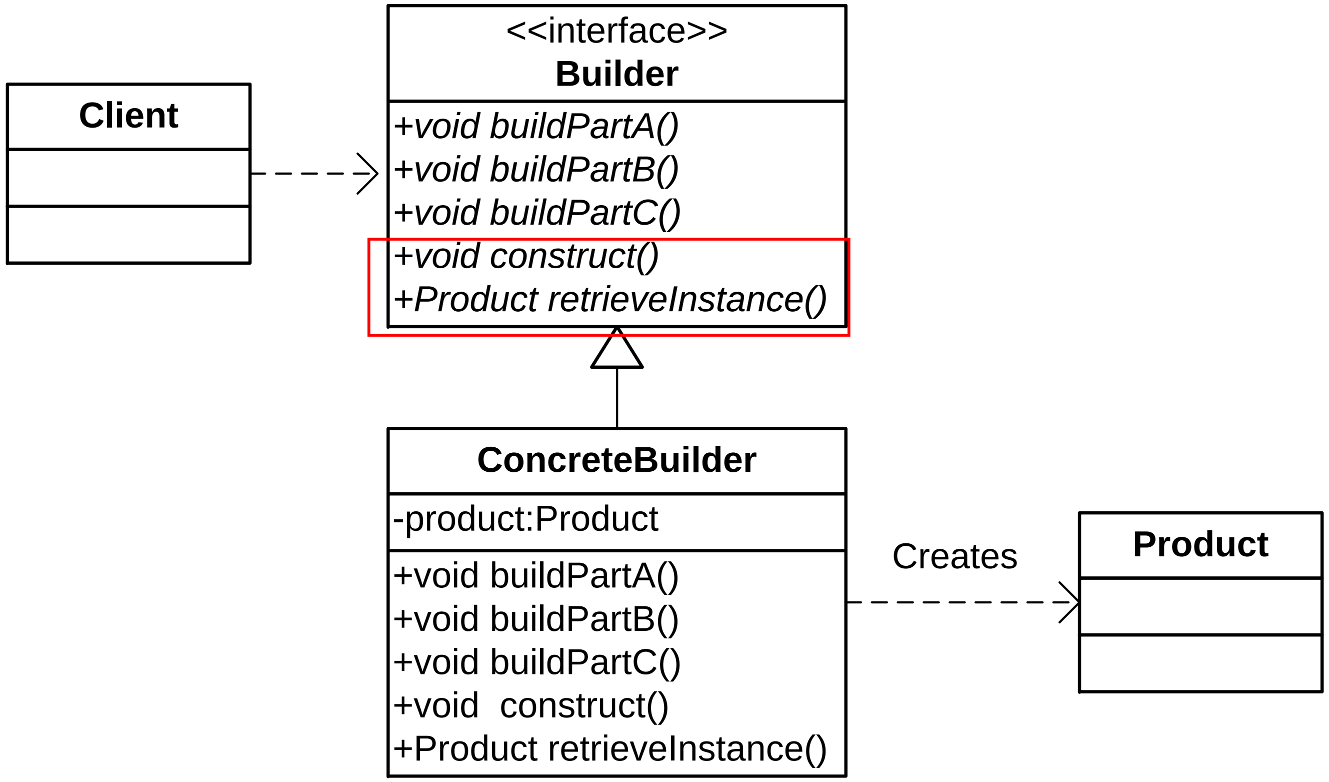

Director
-builder:Builder
+void construct()
builder
<<interface>>
Builder
+void buildPartA()
+void buildPartB()
+void buildPartC()
+Product retrieveInstance()
ConcreteBuilderX
-product:ProductX
+void buildPartA()
+void buildPartB()
+void buildPartC()
+Product retrieveInstance()
ConcreteBuilderY
-product:ProductY
+void buildPartA()
+void buildPartB()
+void buildPartC()
+Product retrieveInstance()
Creates
Creates
ProductX
ProductY
<<interface>>
Product

Variation 2: Abstract Builder is omitted



Variation 3: Director is omitted

- That is closed to factory method pattern
  - retrieveInstence() can be treated as factory method
  - construct() should be invoked before retrieveInstence()



Let’s go to next…