Builder

An Object Creational Pattern

Tim Rice

CSPP51023

March 2, 2010

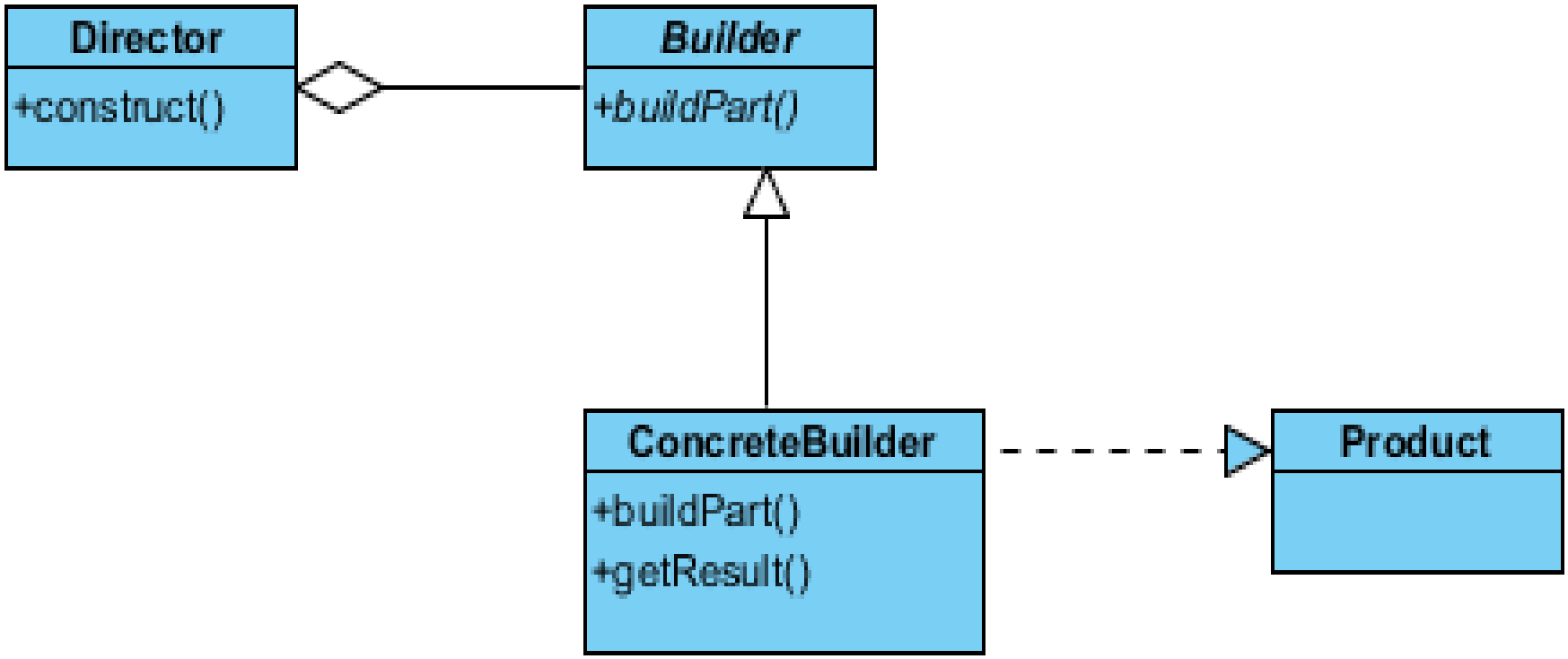
Intent / Applicability

- Separate the construction of a complex object from its representation so that the same construction process can create different representations

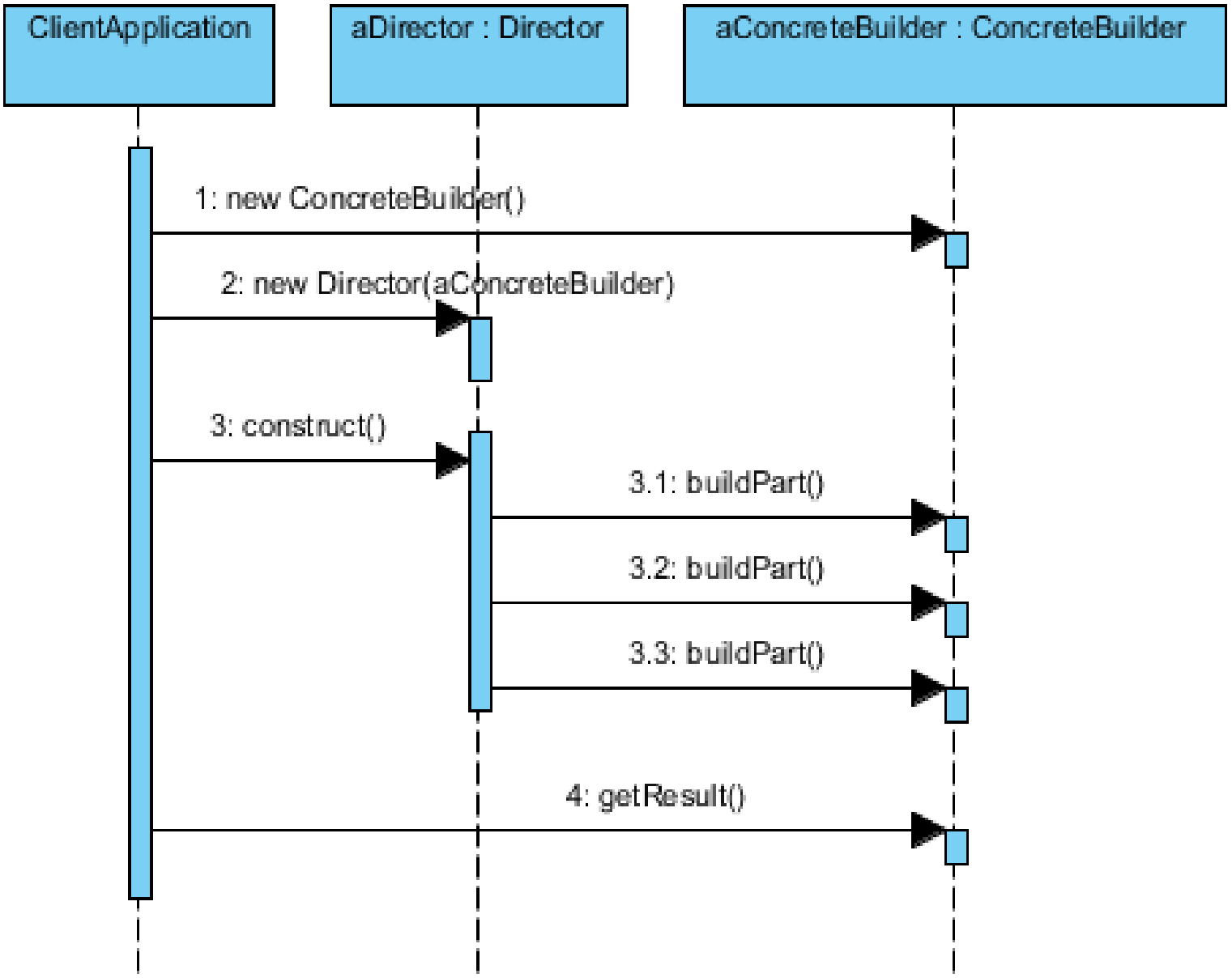
- Use the Builder pattern when:
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
  - the construction process must allow different representations for the object that is constructed

Reference: Design Patterns, Gamma, et. al., Addison Wesley, 1995, pp 97-98

UML Structure



Collaborations



Example: building different types of airplanes

- **AerospaceEngineer**: director
- **AirplaneBuilder**: abstract builder
- **Airplane**: product
- Sample concrete builders:
  - **CropDuster**
  - **FighterJet**
  - **Glider**
  - **Airliner**

Director

```
package builder;
/** "Director" */
public class AerospaceEngineer {

	private AirplaneBuilder airplaneBuilder;

	public void setAirplaneBuilder(AirplaneBuilder ab) {
		airplaneBuilder = ab;
	}

	public Airplane getAirplane() {
		return airplaneBuilder.getAirplane();
	}

	public void constructAirplane() {
		airplaneBuilder.createNewAirplane();
		airplaneBuilder.buildWings();
		airplaneBuilder.buildPowerplant();
		airplaneBuilder.buildAvionics();
		airplaneBuilder.buildSeats();
	}
}
```

AbstractBuilder

```
package builder;
/** "AbstractBuilder" */
public abstract class AirplaneBuilder {

	protected Airplane airplane;
	protected String customer;
	protected String type;

	public Airplane getAirplane() {
		return airplane;
	}

	public void createNewAirplane() {
		airplane = new Airplane(customer, type);
	}

	public abstract void buildWings();

	public abstract void buildPowerplant();

	public abstract void buildAvionics();

	public abstract void buildSeats();

}
```

Product

```
package builder;
/** "Product" */
public class Airplane {

	private String type;
	private float wingspan;
	private String powerplant;
	private int crewSeats;
	private int passengerSeats;
	private String avionics;
	private String customer;

	Airplane (String customer, String type){
		this.customer = customer;
		this.type = type;
	}

	public void setWingspan(float wingspan) {
		this.wingspan = wingspan;
	}
```

Product (continued)

```
	public void setPowerplant(String powerplant) {
		this.powerplant = powerplant;
	}

	public void setAvionics(String avionics) {
		this.avionics = avionics;
	}

	public void setNumberSeats(int crewSeats, int passengerSeats) {
		this.crewSeats = crewSeats;
		this.passengerSeats = passengerSeats;
	}

	public String getCustomer() {
		return customer;
	}

	public String getType() {
		return type;
	}
}
```

ConcreteBuilder 1

```
package builder;
/** "ConcreteBuilder" */
public class CropDuster extends AirplaneBuilder {

	CropDuster (String customer){
		super.customer = customer;
		super.type = "Crop Duster v3.4";
	}

	public void buildWings() {
		airplane.setWingspan(9f);
	}

	public void buildPowerplant() {
		airplane.setPowerplant("single piston");
	}

	public void buildAvionics() {}

	public void buildSeats() {
		airplane.setNumberSeats(1,1);
	}

}
```

ConcreteBuilder 2

```
package builder;
/** "ConcreteBuilder" */
public class FighterJet extends AirplaneBuilder {

	FighterJet (String customer){
		super.customer = customer;
		super.type = "F-35 Lightning II";
	}

	public void buildWings() {
		airplane.setWingspan(35.0f);
	}

	public void buildPowerplant() {
		airplane.setPowerplant("dual thrust vectoring");
	}

	public void buildAvionics() {
		airplane.setAvionics("military");
	}

	public void buildSeats() {
		airplane.setNumberSeats(1,0);

	}

}
```

ConcreteBuilder 3

```
package builder;
/** "ConcreteBuilder" */
public class Glider extends AirplaneBuilder {

	Glider (String customer){
		super.customer = customer;
		super.type = "Glider v9.0";
	}

	public void buildWings() {
		airplane.setWingspan(57.1f);
	}

	public void buildPowerplant() {}

	public void buildAvionics() {}

	public void buildSeats() {
		airplane.setNumberSeats(1,0);

	}

}
```

ConcreteBuilder 4

```
package builder;
/** "ConcreteBuilder" */
public class Airliner extends AirplaneBuilder {

	Airliner (String customer){
		super.customer = customer;
		super.type = "787 Dreamliner";
	}

	public void buildWings() {
		airplane.setWingspan(197f);
	}

	public void buildPowerplant() {
		airplane.setPowerplant("dual turbofan");
	}

	public void buildAvionics() {
		airplane.setAvionics("commercial");
	}

	public void buildSeats() {
		airplane.setNumberSeats(8,289);

	}

}
```

Client Application

```
package builder;
/** Application in which given types of airplanes are being constructed.
*/
public class BuilderExample {
      public static void main(String[] args) {
            // instantiate the director (hire the engineer)
            AerospaceEngineer aero = new AerospaceEngineer();

            // instantiate each concrete builder (take orders)
            AirplaneBuilder crop = new CropDuster("Farmer Joe");
            AirplaneBuilder fighter = new FighterJet("The Navy");
            AirplaneBuilder glider = new Glider("Tim Rice");
            AirplaneBuilder airliner = new Airliner("United Airlines");

            // build a CropDuster
            aero.setAirplaneBuilder(crop);
            aero.constructAirplane();
            Airplane completedCropDuster = aero.getAirplane();
            System.out.println(completedCropDuster.getType() +
                        " is completed and ready for delivery to " +
                        completedCropDuster.getCustomer());

            // the other 3 builds removed to fit the code on one slide
      }
}
```

Client Application output

```
Crop Duster v3.4 is completed and ready for
delivery to Farmer Joe

F-35 Lightning II is completed and ready for
delivery to The Navy

Glider v9.0 is completed and ready for
delivery to Tim Rice

787 Dreamliner is completed and ready for
delivery to United Airlines
```

Builder: Advantages / Disadvantages

- Advantages:
  - Allows you to vary a product's internal representation
  - Encapsulates code for construction and representation
  - Provides control over steps of construction process
- Disadvantages:
  - Requires creating a separate ConcreteBuilder for each different type of Product

Related Patterns

- Factory Method: “class creational” pattern
  - Provides interface for constructing objects, allowing subclasses decide which creation class to use
- Prototype: an “object creational” pattern
  - Separates system from object creation and representation, using a prototypical instance
- Abstract Factory: another “object creational” pattern
  - Also separates system from product creation and representation, but does not have an abstract builder – application calls the factory methods directly