Builder Design Pattern

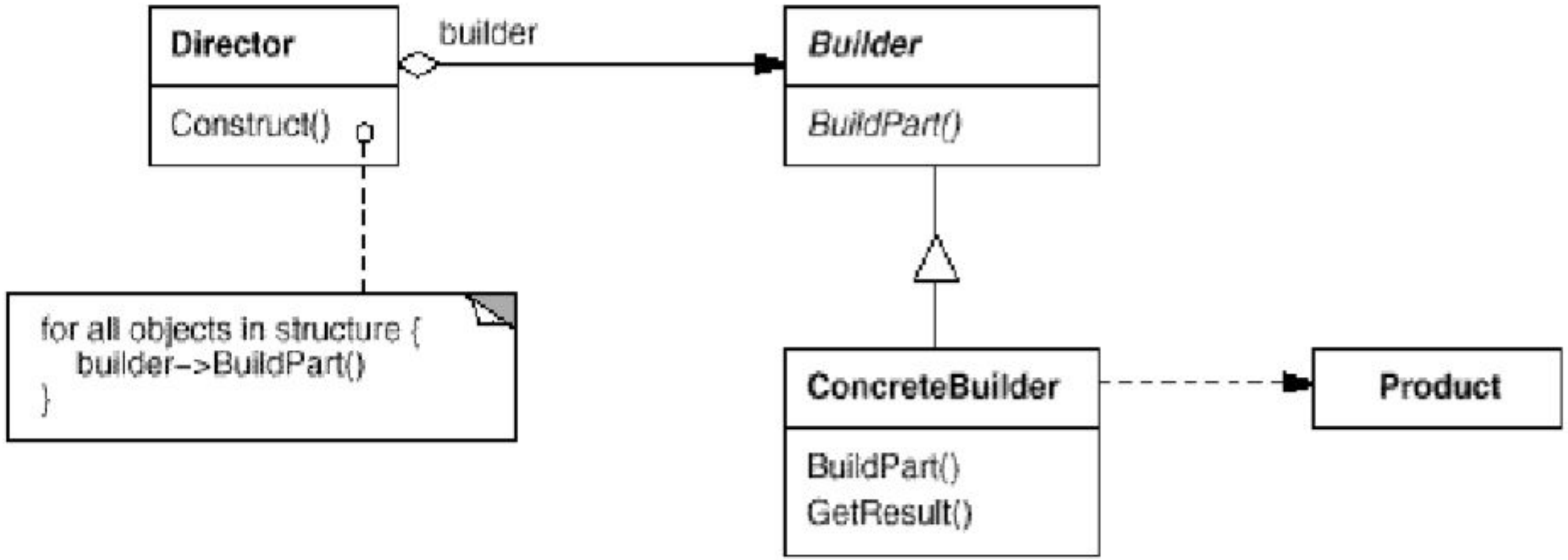
Paritosh Shirodkar
paritosh@bu.edu

Problem

- **Definition from the book:** “Separate the construction of a complex object from its representation so that the same construction process can create different representations.”

- Sometimes in our application we want to create complex objects composed of other smaller objects. We want to have the flexibility to change these participating objects while keeping the construction process the same.

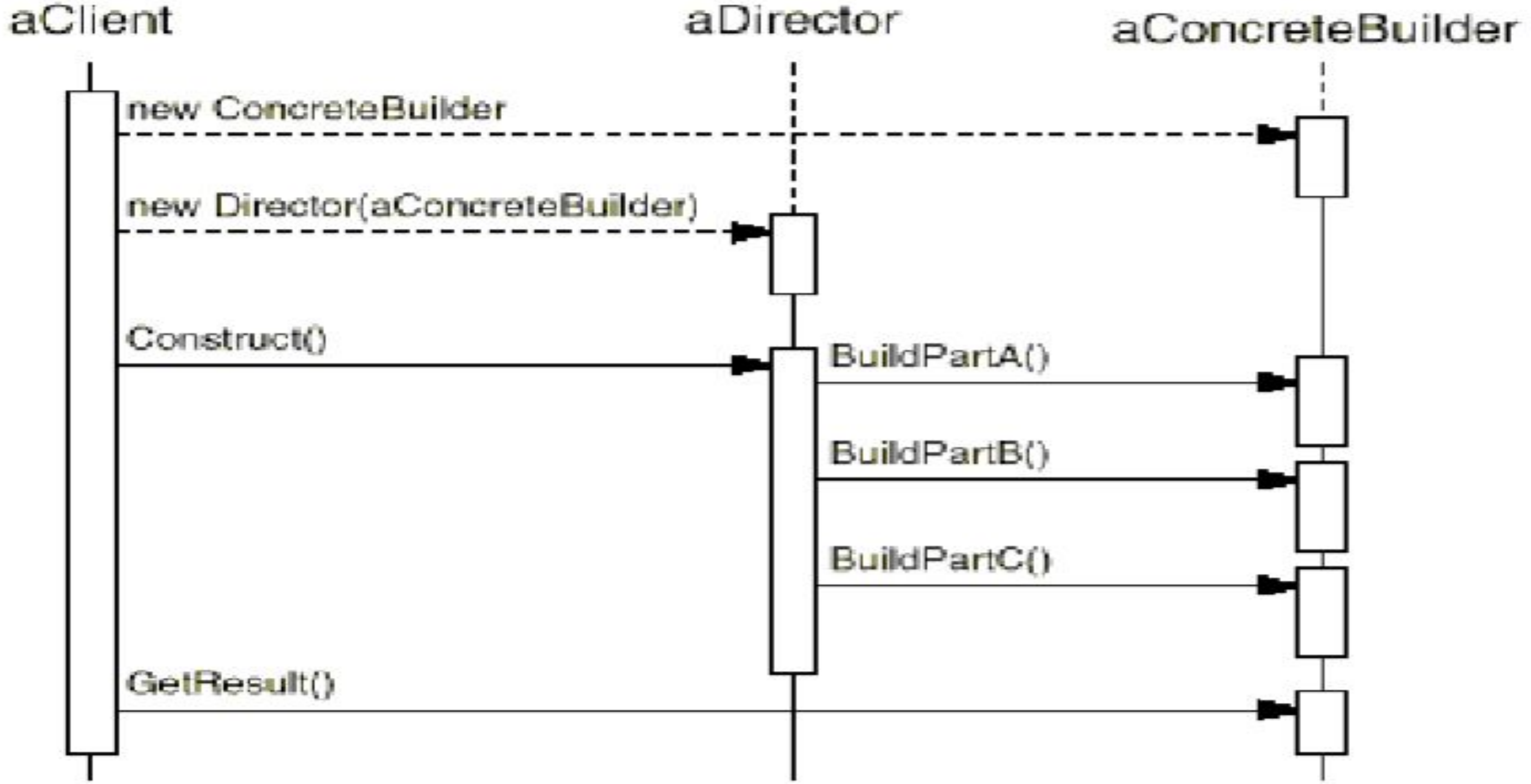
Structure of Builder Pattern



```
for all objects in structure {
    builder->BuildPart()
}
```

Participants in Builder Pattern

- **Builder**: This is an abstraction (i.e. either an abstract class or an interface) which specifies the smaller/component parts that should be used in the construction of the complex object.
- **Concrete Builder:** Implements the Builder interface and provides the concrete implementation details that are needed to produce different representations of the complex objects.
- **Director**: Responsible for creating the complex objects. It is configured with the appropriate concrete builder class corresponding to the desired representation.
- **Product:** The complex object required by the application. It consists of the appropriate classes and methods needed to create the final object.



Source: Design Patterns: Elements of Reusable Object-Oriented by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

When to use Builder Pattern?

- Use the builder pattern when you want to keep the construction process used for creating a complex objects independent of the parts used for creating these objects.

- The application needs the flexibility to obtain different representations of the complex object by following the same construction process.

Consequences of using Builder Pattern

Advantages

- Provides the **flexibility** to change the **Product's** internal representation.
- Better **code isolation** and **modularity** by keeping the construction process separate from the representation of the objects.
- Improved **abstraction,** since the client does not need to know about the internal details involved in the construction of the complex objects**.**
- Improved **code reuse** since director instances can use the different concrete builders to obtain different representation from the similar components.
- Focuses on the construction process hence finer control over it. The director creates the complex object **step by step** and only when the object is ready it fetches it from the builder.
- No need to remember the **order** in which the parameters are passed to the constructors.

Disadvantages

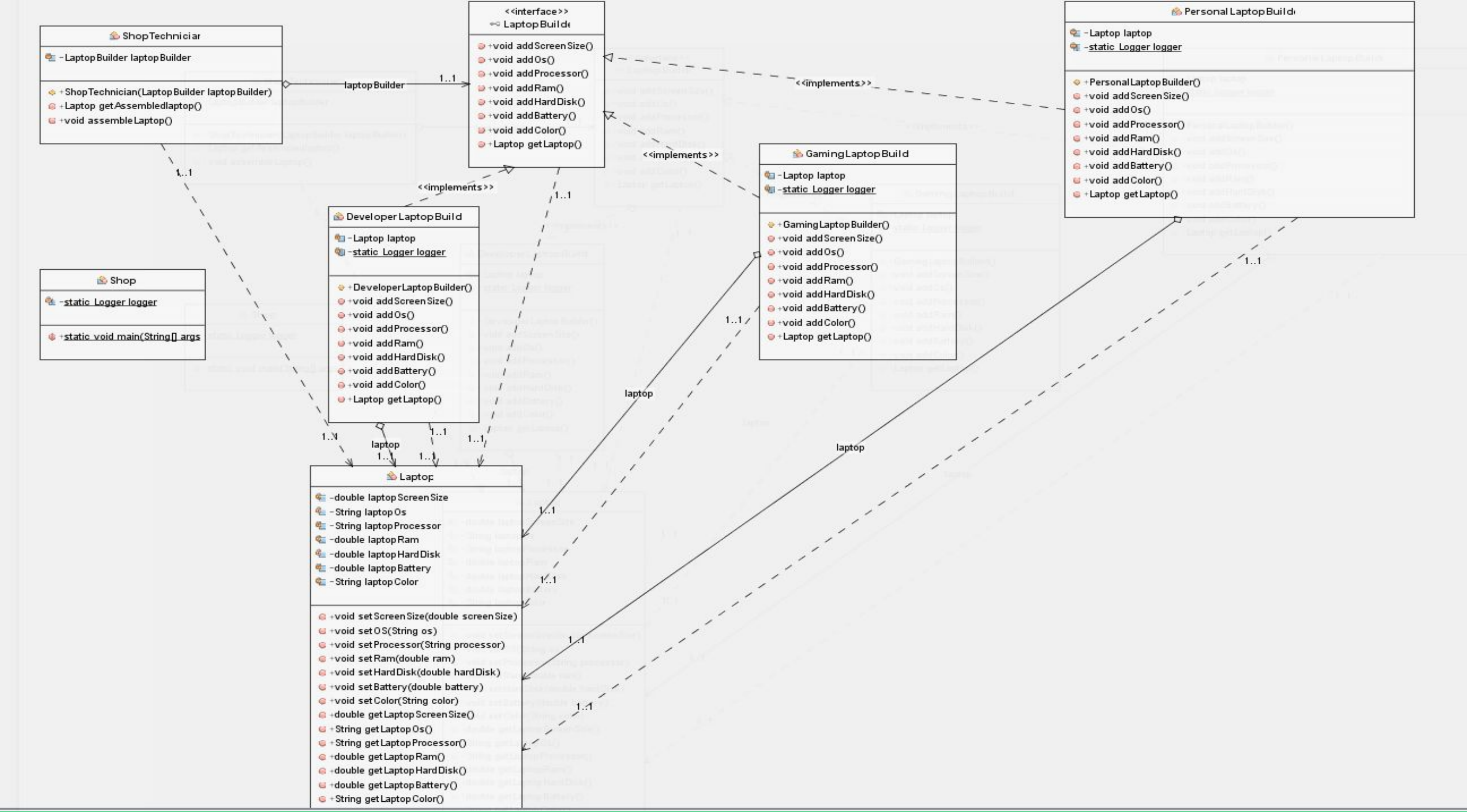
- If the concrete builders do not implement the builder abstraction correctly it may lead to improper initialization of data members in the complex objects.
- It leads to more classes depending upon the supported representations. However, this is a tradeoff if you have some mandatory and some optional parameters.

Builder Pattern Implementation

Simulate the working of a shop where the clients can request different types of laptops like - Gaming laptop, Personal laptop and Developer laptop.

The application constraints are:

- The process involved in creating the different types of laptops should be similar.
- The client should be unaware of the different parts that are used to assemble the laptop.
- Based on the type of laptop requested an appropriate shop technician instance should be created who would follow a process to pick the right parts and assemble the laptop.
- Only after all the components have been set properly and the laptop is ready the laptop should be given to the client.


ShopTechnician
-LaptopBuilder laptopBuilder
+ShopTechnician(LaptopBuilder laptopBuilder)
+Laptop getAssembledlaptop()
+void assembleLaptop()
laptopBuilder
1..1
<<interface>>
LaptopBuilde
+void addScreenSize()
+void addOs()
+void addProcessor()
+void addRam()
+void addHardDisk()
+void addBattery()
+void addColor()
+Laptop getLaptop()
<<implements>>
PersonalLaptopBuild
-Laptop laptop
-static Logger logger
+PersonalLaptopBuilder()
+void addScreenSize()
+void addOs()
+void addProcessor()
+void addRam()
+void addHardDisk()
+void addBattery()
+void addColor()
+Laptop getLaptop()
<<implements>>
GamingLaptopBuild
-Laptop laptop
-static Logger logger
+GamingLaptopBuilder()
+void addScreenSize()
+void addOs()
+void addProcessor()
+void addRam()
+void addHardDisk()
+void addBattery()
+void addColor()
+Laptop getLaptop()
<<implements>>
DeveloperLaptopBuild
-Laptop laptop
-static Logger logger
+DeveloperLaptopBuilder()
+void addScreenSize()
+void addOs()
+void addProcessor()
+void addRam()
+void addHardDisk()
+void addBattery()
+void addColor()
+Laptop getLaptop()
Shop
-static Logger logger
+static void main(String[] args
1..1
1..1
1..1
1..1
laptop
laptop
laptop
1..1
1..1
1..1
1..1
1..1
1..1
1..1
1..1
1..1
Laptop
-double laptopScreenSize
-String laptopOs
-String laptopProcessor
-double laptopRam
-double laptopHardDisk
-double laptopBattery
-String laptopColor
+void setScreenSize(double screenSize)
+void setOS(String os)
+void setProcessor(String processor)
+void setRam(double ram)
+void setHardDisk(double hardDisk)
+void setBattery(double battery)
+void setColor(String color)
+double getLaptopScreenSize()
+String getLaptopOs()
+String getLaptopProcessor()
+double getLaptopRam()
+double getLaptopHardDisk()
+double getLaptopBattery()
+String getLaptopColor()

Related Patterns

- **Abstract Factory Pattern -** The main difference between the two patterns is that the abstract factory pattern focuses on creating families/groups of similar objects together whereas builder pattern focuses on the incremental steps involved in creating complex objects.
- **Strategy Pattern -** Structurally these two patterns look similar but the main difference is that Strategy pattern is a **behavioural** pattern involving switching between different algorithms at runtime based on the objects but **Builder** pattern is a **creational** pattern which focuses more on keeping the creational steps same to get different representations of complex objects.

GitHub

https://github.com/metcs/met-cs665-assignment-project-paritoshshirodkar

References

- Design Patterns: Elements of Reusable Object-Oriented by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Gang of Four Design textbook)
- https://java-design-patterns.com/patterns/builder/