SHRI RAMDEOBABA COLLEGE OF ENGINEERING AND MANAGEMENT, NAGPUR - 440013

DESIGN PATTERNS
(CST324)
V SEMESTER SECTION A

COURSE COORDINATOR: RINA DAMDOO

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Builder Design Patterns

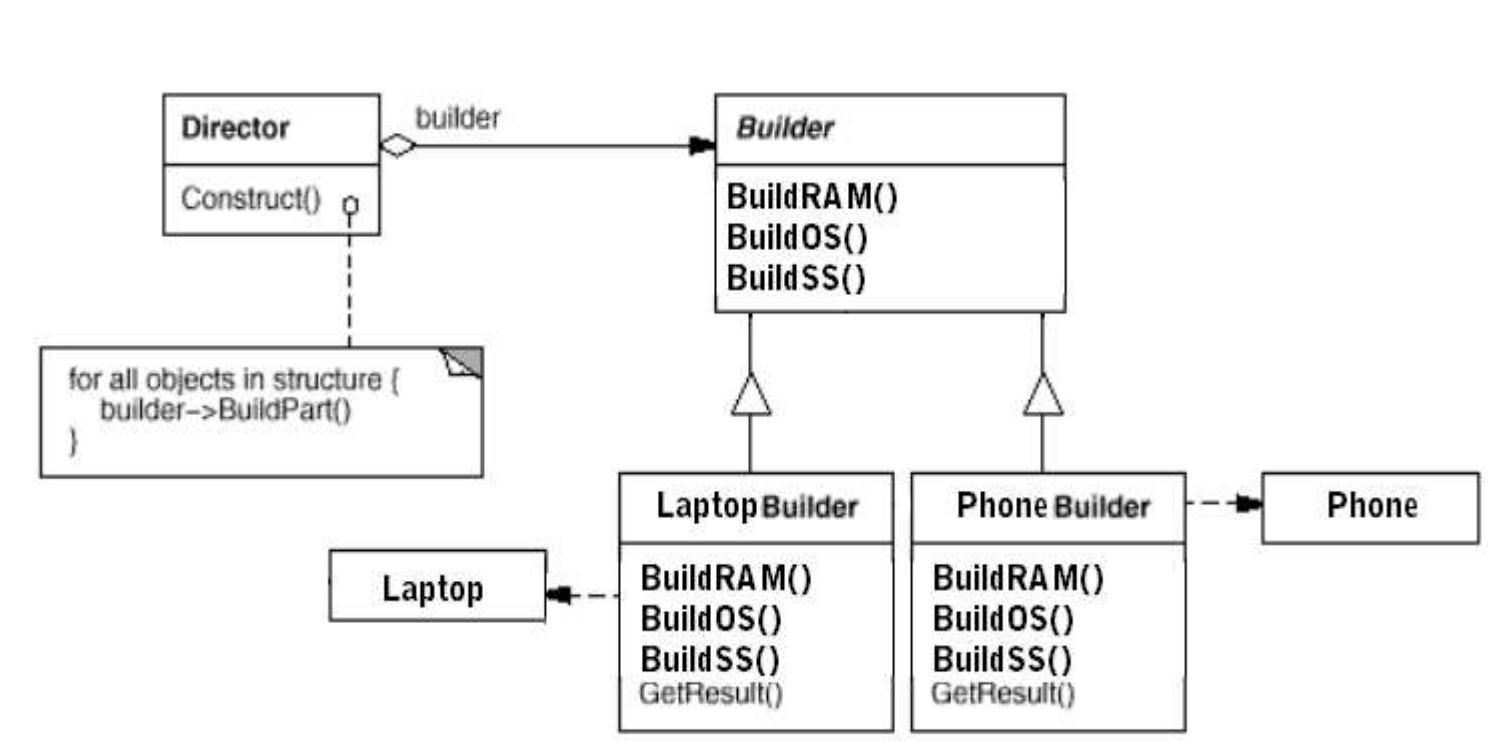
Intent

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

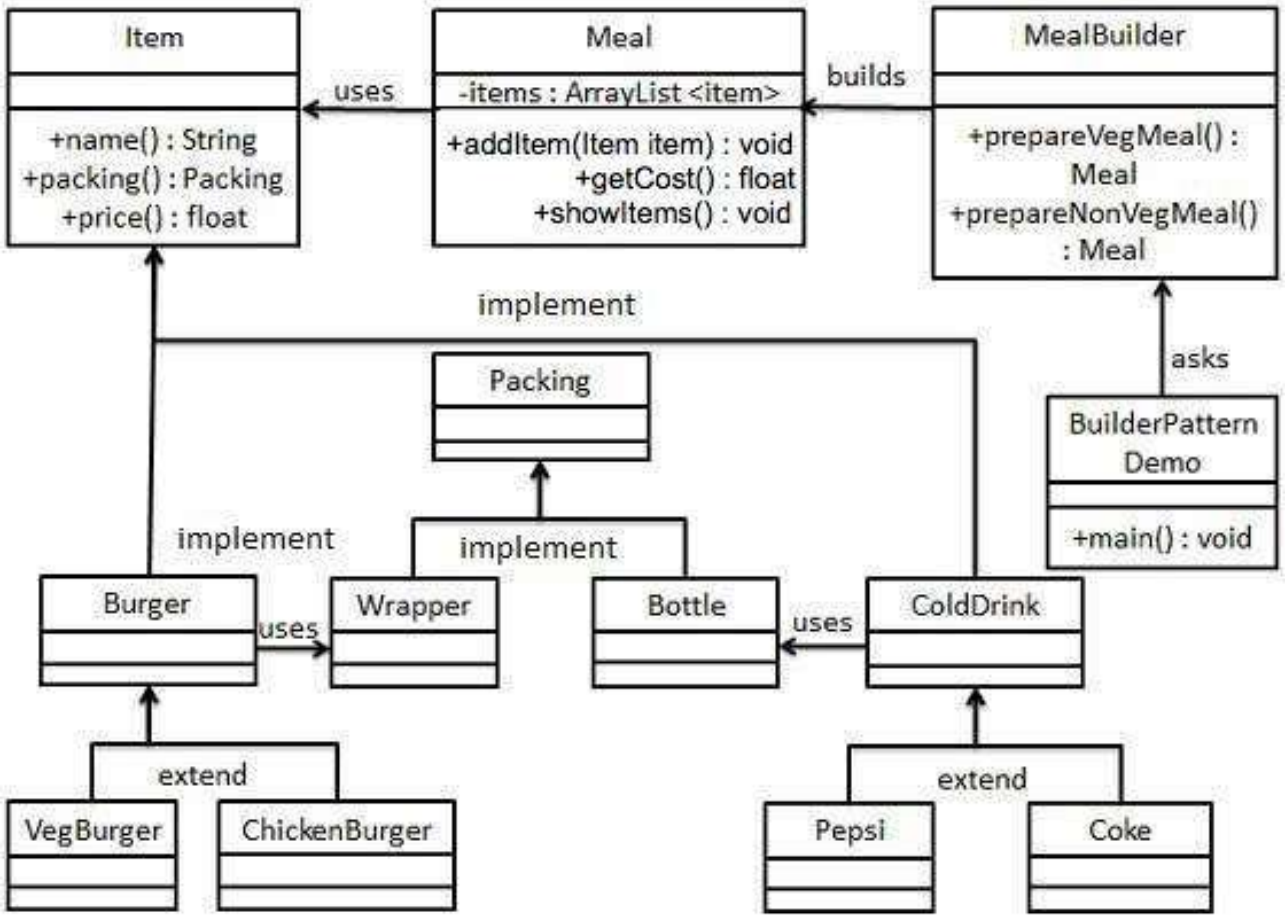
BUILDER DESIGN PATTERNS

Problem Statement :

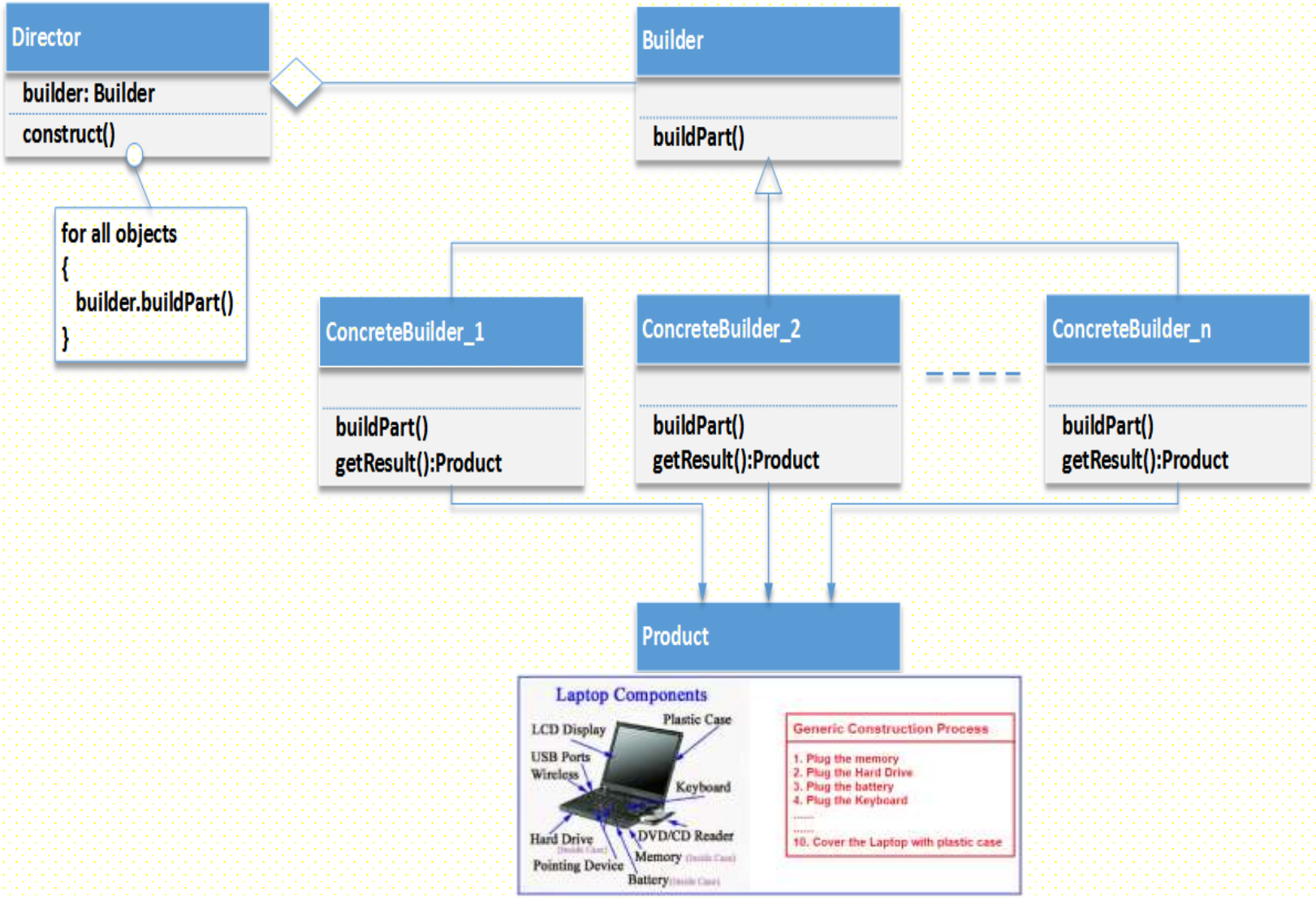
Implement a service to provide two types of complex products Phone and Laptop using Builder Design Patterns.



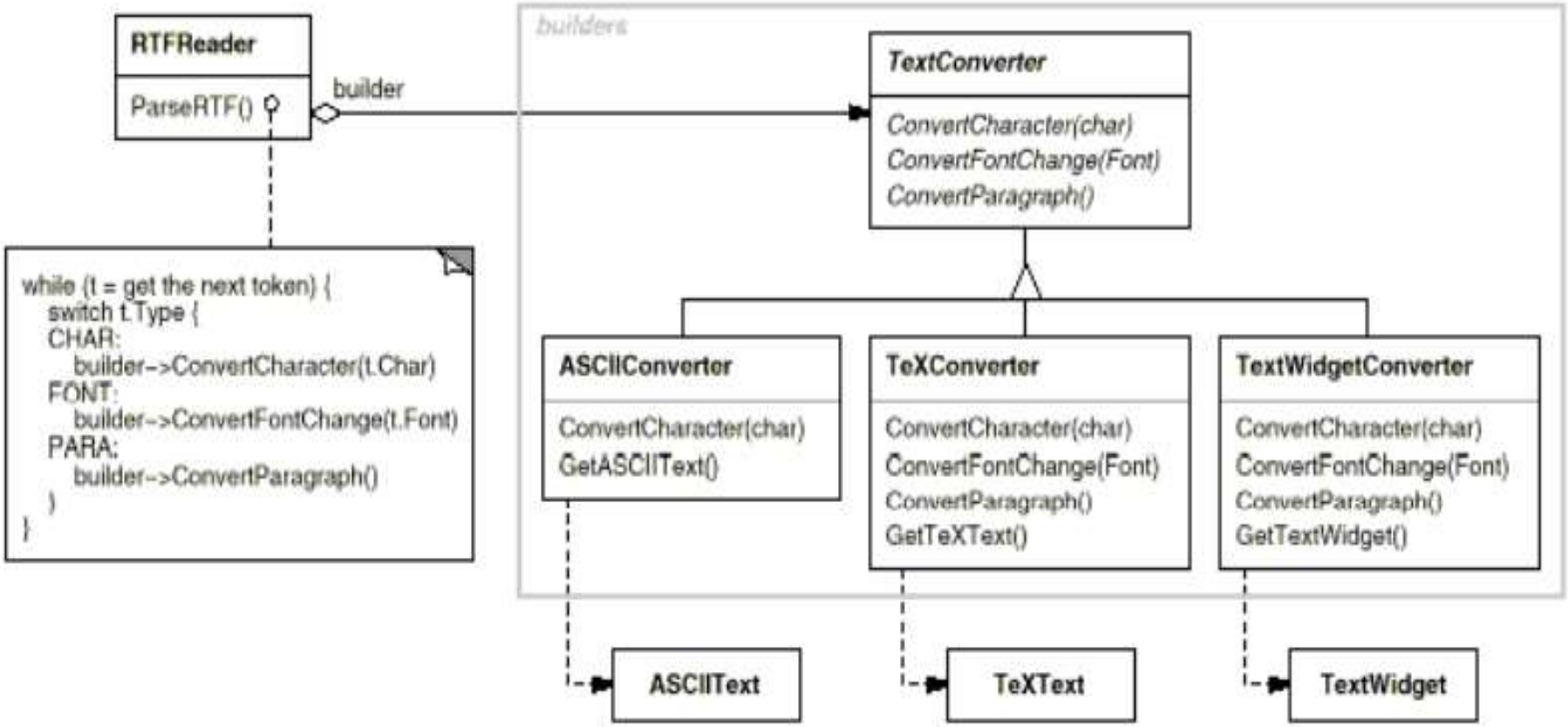
BUILDER EXAMPLE



BUILDER EXAMPLE



MOTIVATION



Text Converter

Applicability

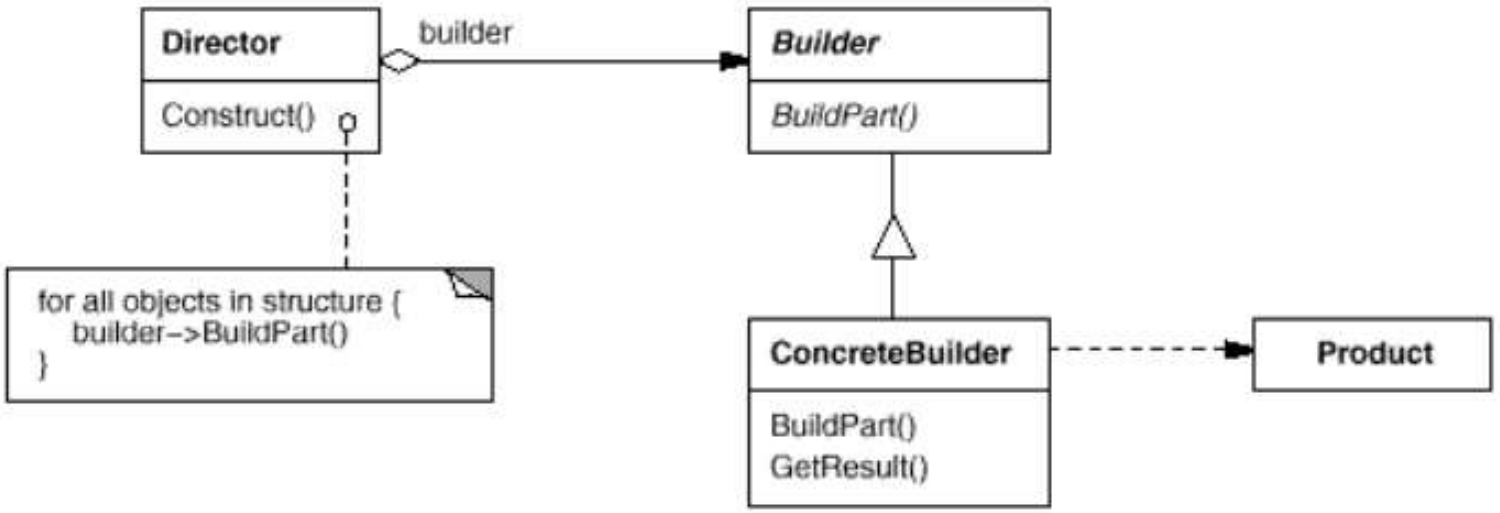
Use the Builder pattern when:

- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.

- the construction process must allow different representations for the object that is constructed.

BUILDER DESIGN PATTERNS

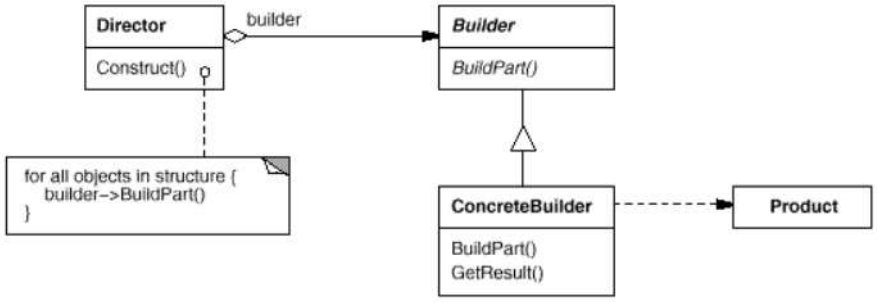
Structure



PARTICIPANTS

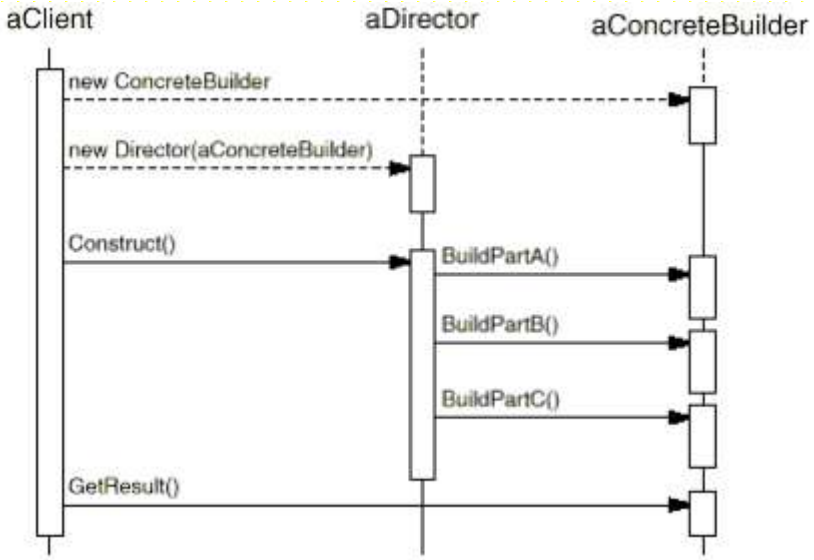
Participant	Responsibility
Builder	• specifies an abstract interface for creating parts of a Product object.
ConcreteBuilder	• constructs and assembles parts of the product by implementing the Builder interface. • defines and keeps track of the representation it creates. • provides an interface for retrieving the product
Director	• constructs an object using the Builder interface.
Product	• represents the complex object under construction. **ConcreteBuilder** builds the product's internal representation and defines the process by which it's assembled. • includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

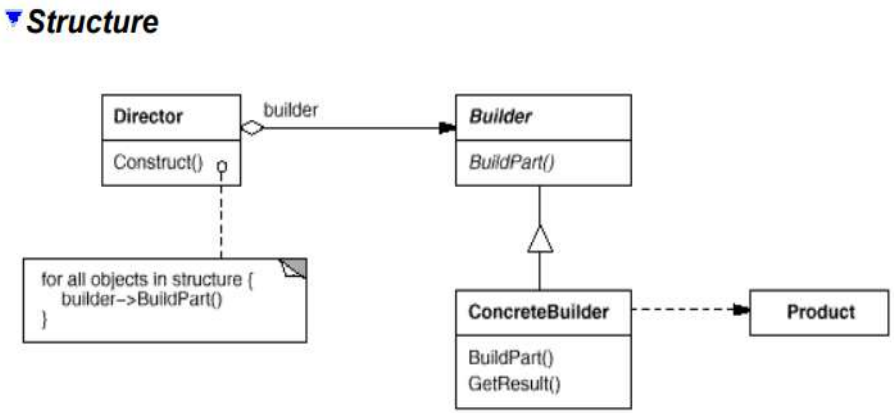
Structure



COLLABORATIONS

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

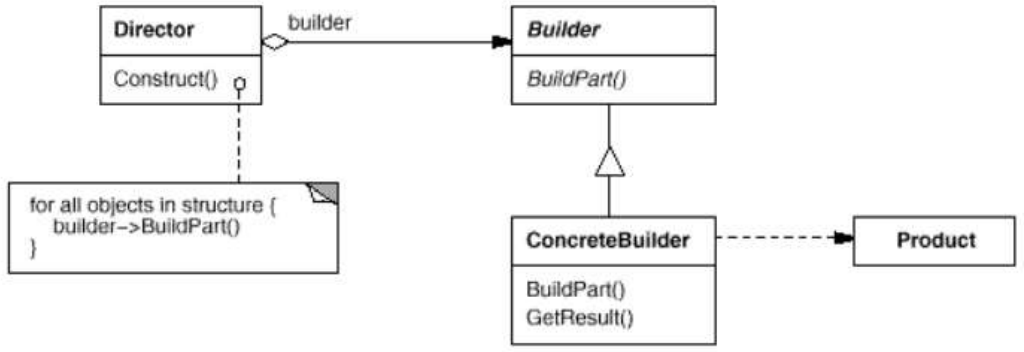




CONSEQUENCES

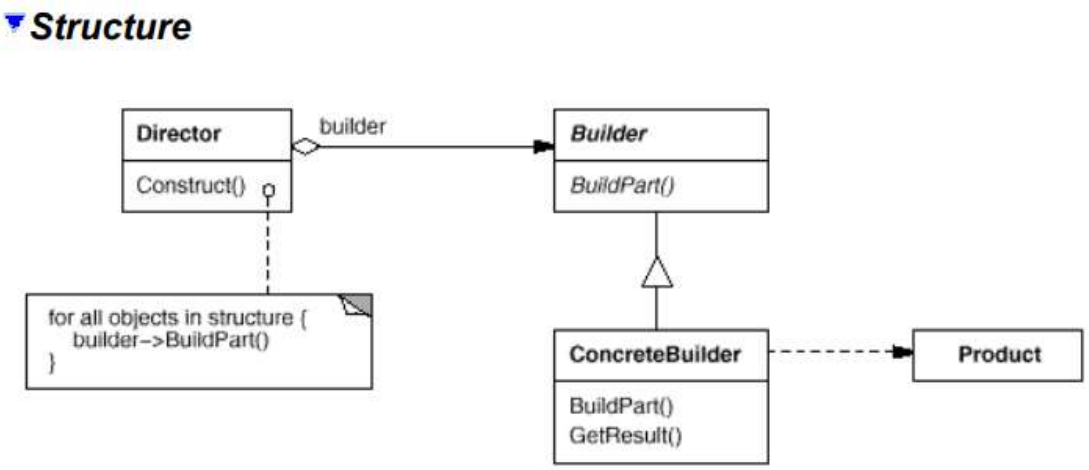
1. **It lets you vary a product's internal representation.** The Builder object provides the director with an abstract interface for constructing the product. The interface lets the builder hide the representation and internal structure of the product. It also hides how the product gets assembled.

Structure



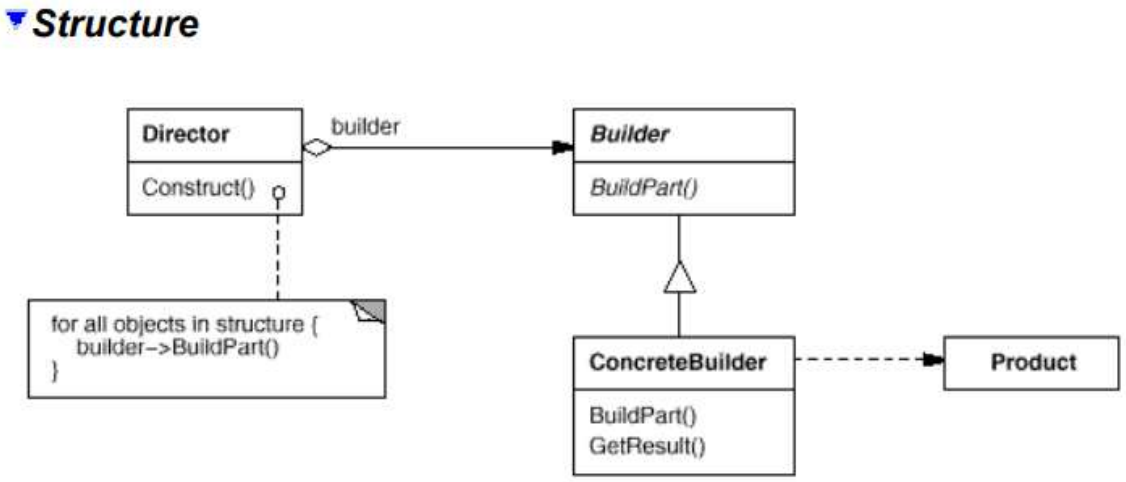
CONSEQUENCES

2. **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. Each **ConcreteBuilder** contains all the code to create and assemble a particular kind of product.



CONSEQUENCES

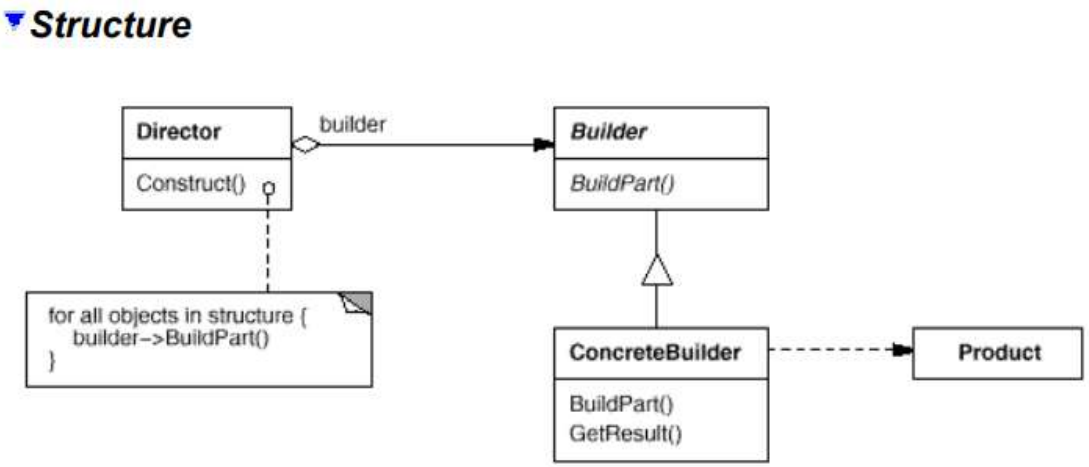
3. **It gives you finer control over the construction process.** Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the director's control. Only when the product is finished the director retrieves it from the builder. This gives you finer control over the construction process and consequently the internal structure of the resulting product.



IMPLEMENTATION

Typically there's an abstract Builder class that defines an operation for each component that a director may ask it to create. The operations do nothing by default. A ConcreteBuilder class overrides operations for components. Here are other implementation issues to consider:

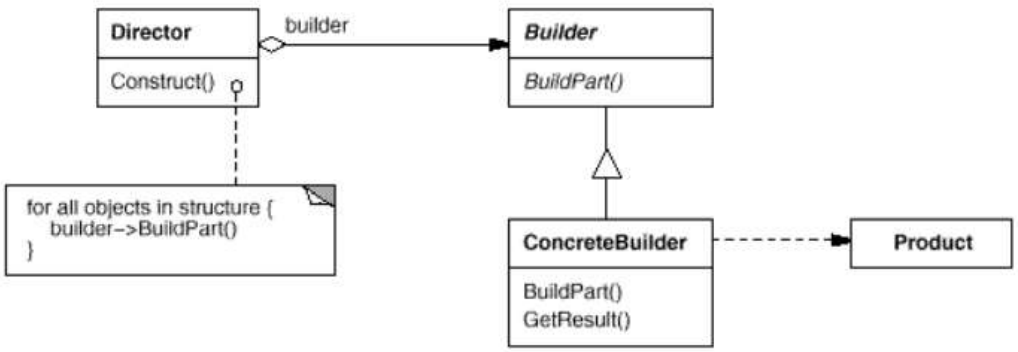
1. Assembly and construction interface. Builders construct their products in step-by-step fashion. Therefore the Builder class interface must be general enough to allow the construction of products for all kinds of concrete builders. A key design issue concerns the **model for the construction and assembly process**. A model where the results of construction requests are simply appended to the product is usually sufficient. But sometimes you might need access to parts of the product constructed earlier.



IMPLEMENTATION

2. **Why no abstract class for products?** In the common case, the products produced by the concrete builders differ so greatly in their representation that there is little to gain from giving different products a common parent class.

Structure



KNOWN USES & RELATED PATTERNS

- **ClassBuilder** is a builder that Classes use to create subclasses for themselves. In this case a Class is both the Director and the Product.
- **ByteCodeStream** is a builder that creates a compiled method as a byte array.

- **Abstract Factory** is similar to Builder. The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract Factory's emphasis is on families of product objects. Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately.
- A **Composite** is what the builder often builds.

THE BUILDER PATTERN IS USEFUL.....

- when a class has many fields and it can be difficult to set them all through a no-args constructor or setter methods.
- The builder pattern allows for a more organized and readable way of creating objects by providing methods for setting each field individually.
- The builder pattern can enforce certain constraints on the fields, such as making certain fields required or setting default values for certain fields. This can make the code more maintainable and less error-prone.
- In most cases, one uses a builder pattern because one wants to keep the object — once created — *immutable*. That is, create it, and never change it again
- to check that the fields are consistent (e.g. if Start_Date is set to something, then End_Date must be set as well), which is messier to do with setters.

A builder pattern ensures we create a valid object to work with.

Let's use a Person class as an example. If we opt for the no-args constructor approach it means users of those objects must check the validity of the object; has the name field been set? Has the DOB field been set? The list goes on.
So instead we use a PersonBuilder that won't allow a Person object to be constructed without all the required attributes.