Laboratorio di Progettazione di Sistemi Software

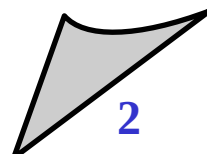
Design Patterns

Creazionali

Valentina Presutti (A-L)
Riccardo Solmi (M-Z)

Indice degli argomenti

- **Catalogo di Design Patterns creazionali:**
  - Abstract Factory
  - Factory Method
  - Singleton
  - Prototype
  - Builder

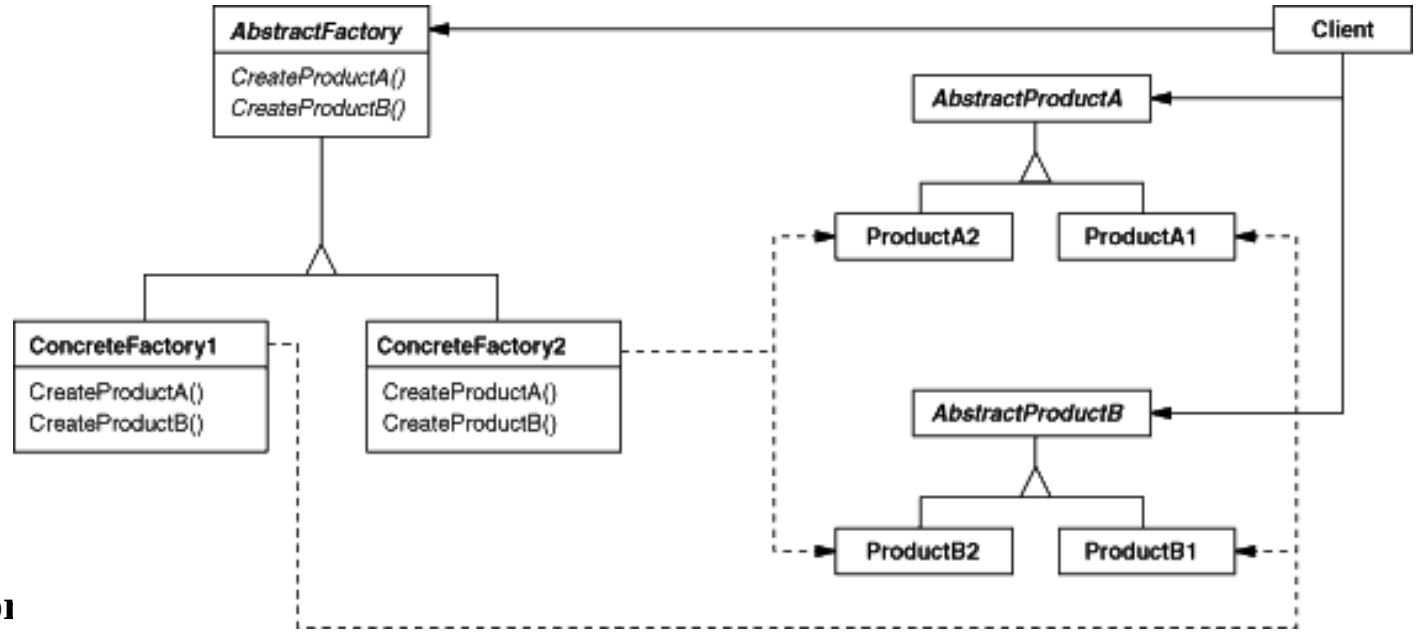


Abstract Factory

- **Intent**
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- **Applicability**
  - a system should be independent of how its products are created, composed, and represented
  - a system should be configured with one of multiple families of products.
  - a family of related product objects is designed to be used together, and you need to enforce this constraint.
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Abstract Factory /2

- **Structure**

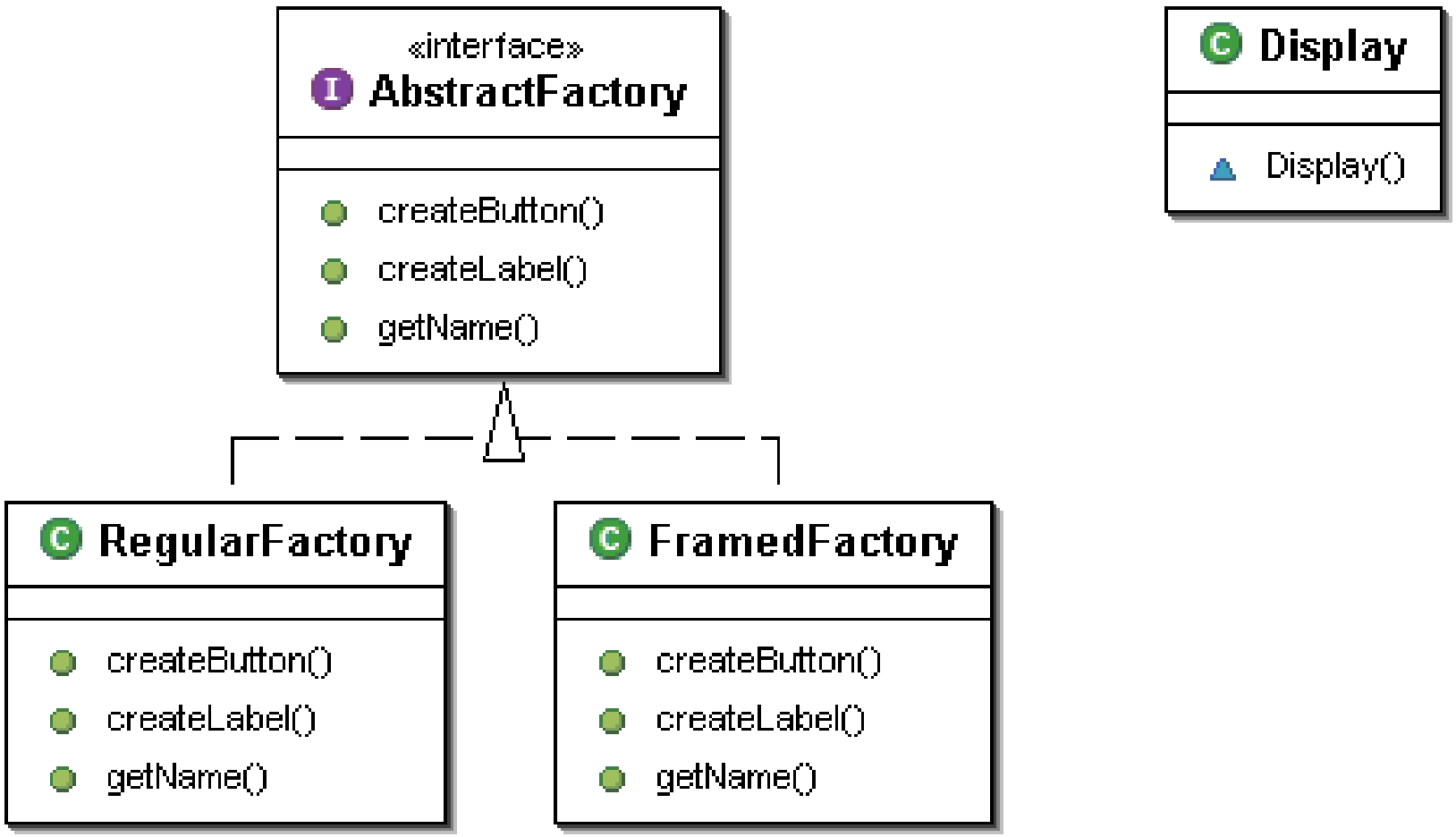


- **Participants**
  - **AbstractFactor**
  - declares an interface for operations that create abstract product objects.
  - **ConcreteFactory**
  - implements the operations to create concrete product objects.
  - **AbstractProduct**
  - declares an interface for a type of product object.
  - **ConcreteProduct**
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
  - **Client**
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes

Abstract Factory /3

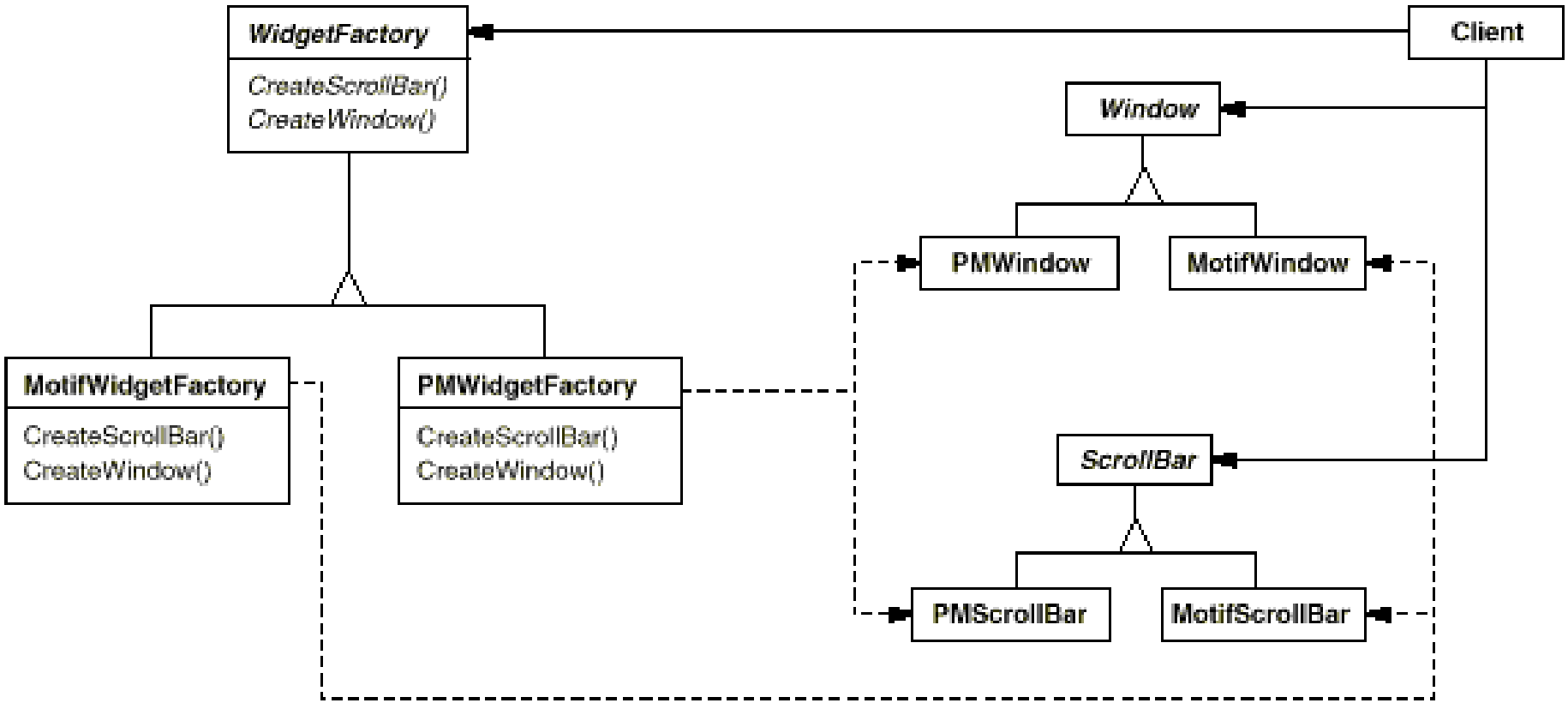
- **Collaborations**
  - Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation.
  - AbstractFactory defers creation of product objects to its ConcreteFactory subclass
- **Consequences**
  - It isolates concrete classes
  - It makes exchanging product families easy
  - It promotes consistency among products
  - Supporting new kinds of products is difficult

Abstract Factory example 1



Abstract Factory example 2

- **User interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager**



Abstract Factory questions

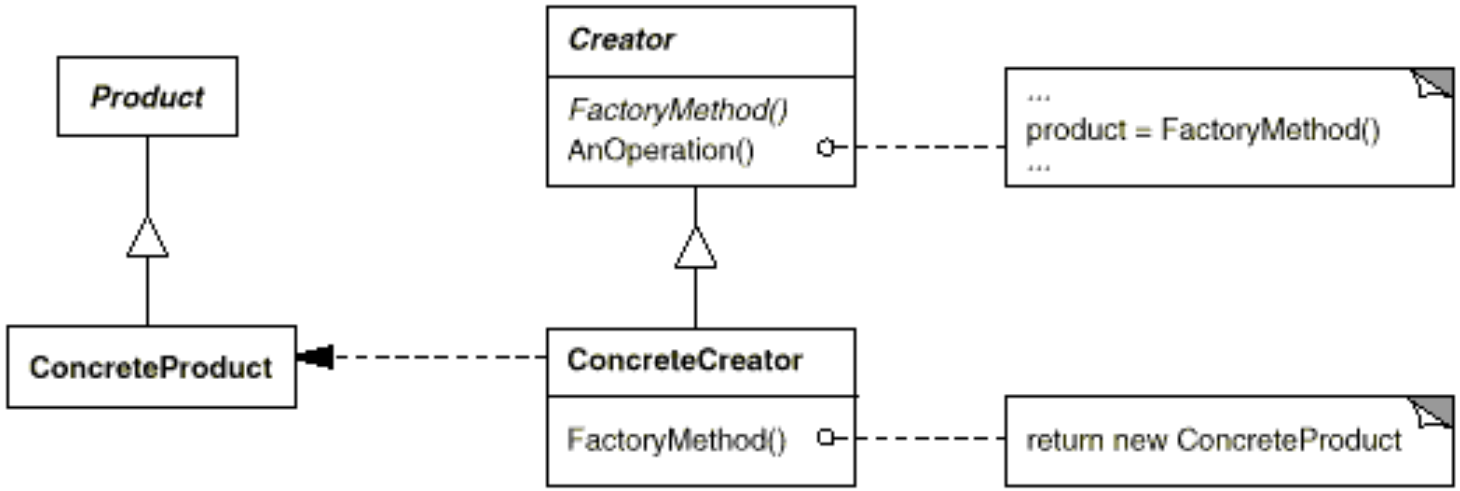
- **In the Implementation section of this pattern, the authors discuss the idea of *defining extensible factories*. Since an Abstract Factory is composed of Factory Methods, and each Factory Method has only one signature, does this mean that the Factory Method can only create an object in one way?**

Factory Method

- **Intent**
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Applicability**
  - a class can't anticipate the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Factory Method /2

- **Structure**



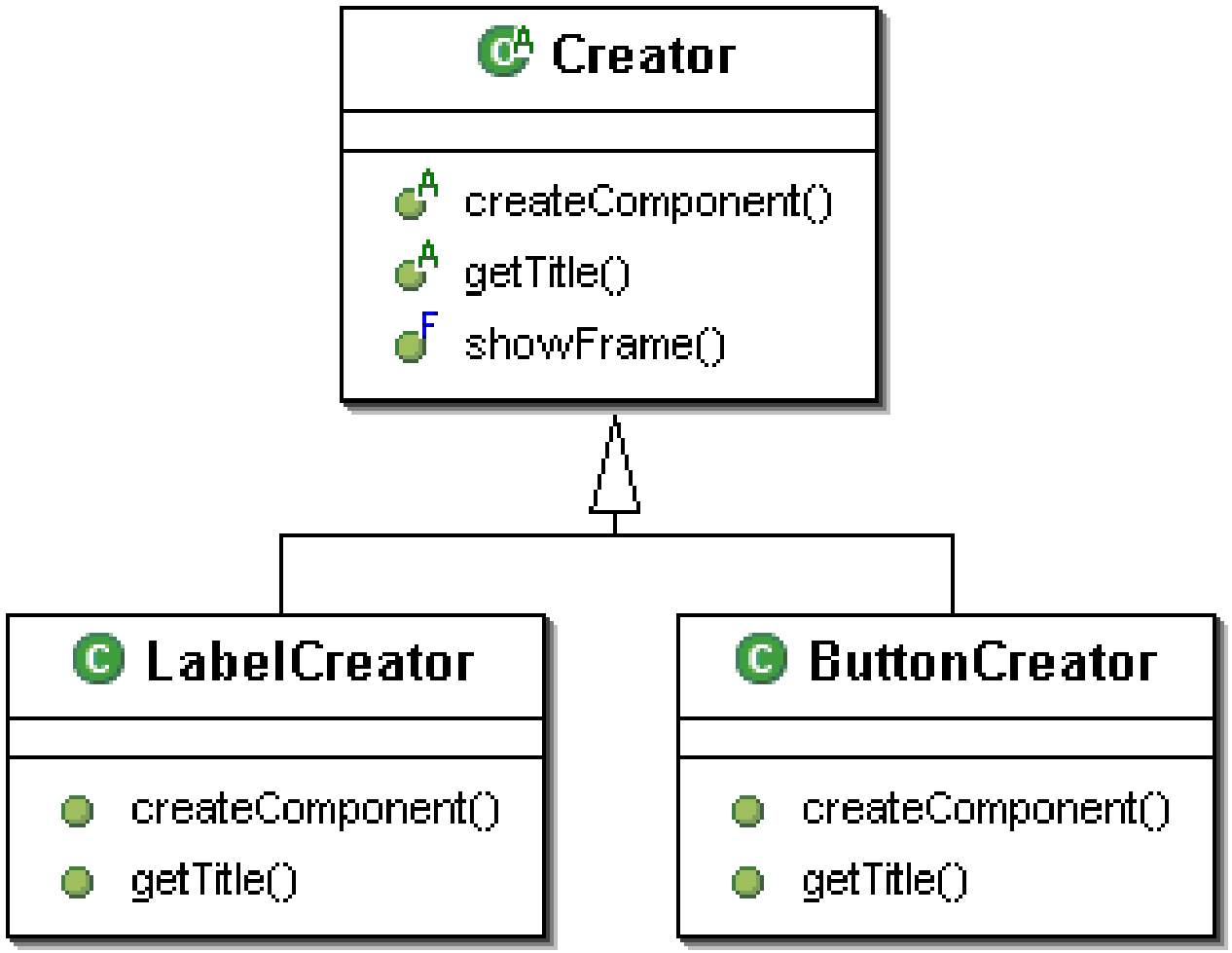
- **Participants**
  - Product (Document)
    - defines the interface of objects the factory method creates.
  - ConcreteProduct (MyDocument)
    - implements the Product interface.
  - Creator (Application)
    - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
    - may call the factory method to create a Product object.
  - ConcreteCreator (MyApplication)
    - overrides the factory method to return an instance of a ConcreteProduct

FactoryMethod /3

- **Collaborations**
  - Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.
- **Consequences**
  - The code only deals with the interface; therefore it can work with any user-defined concrete classes.
  - Provides *hooks* for subclasses
  - Connects *parallel class hierarchies*

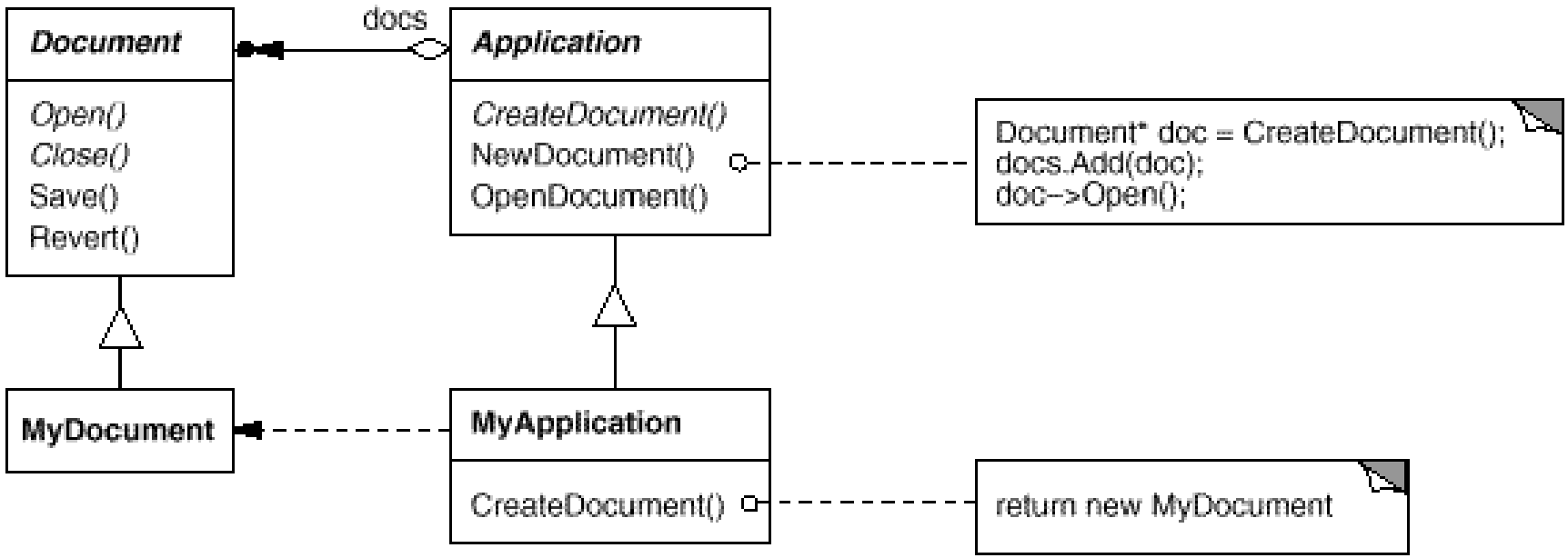
Factory Method example 1

- **JComponent**



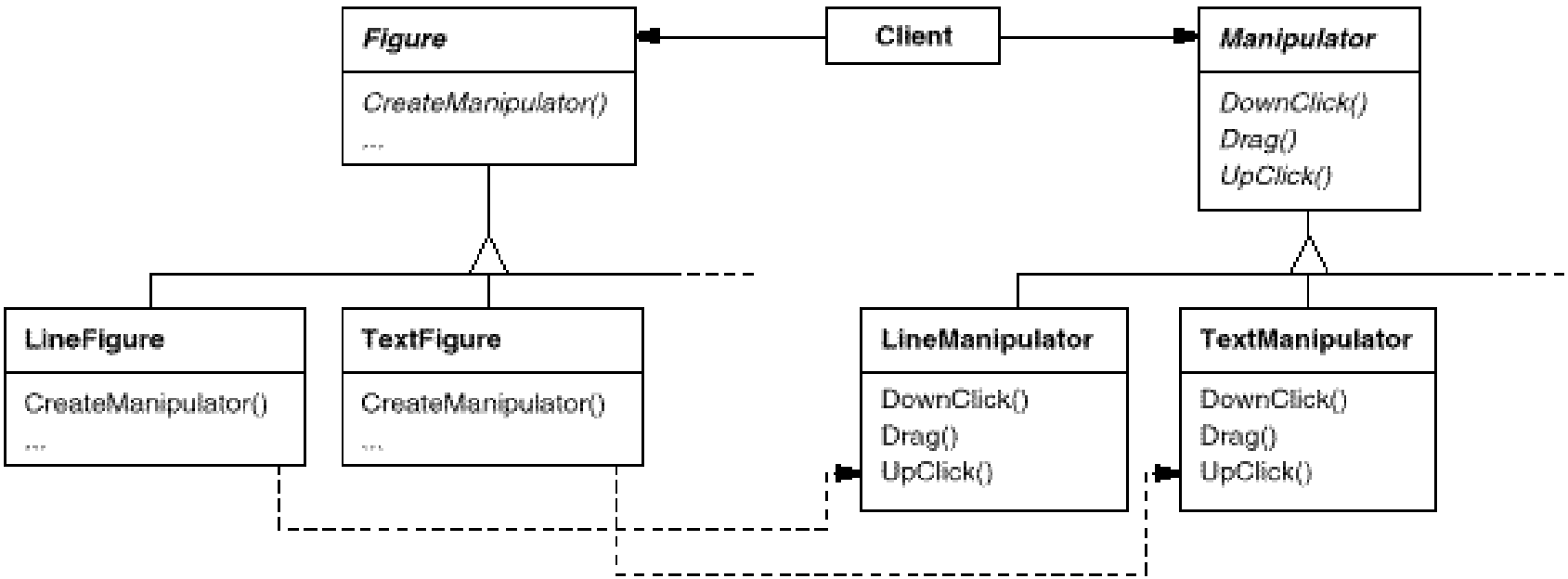
Factory Method example 2

- **The code only deals with the interface; therefore it can work with any user-defined concrete classes.**



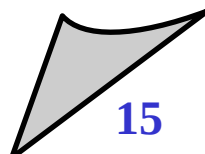
Factory Method example 3

- **Connects *parallel class hierarchies***



Factory Method questions

- **How does *Factory Method* promote loosely coupled code?**

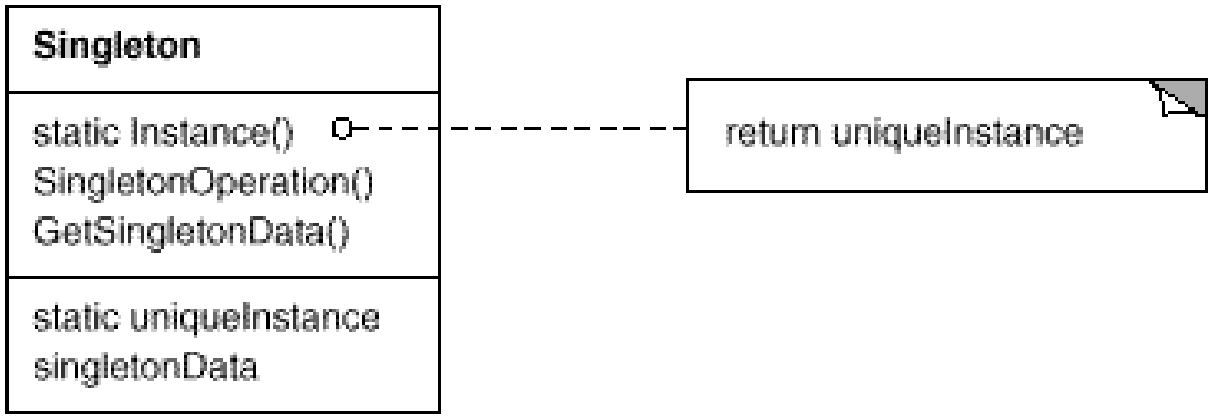


Singleton

- **Intent**
  - Ensure a class only has one instance, and provide a global point of access to it.
- **Applicability**
  - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
  - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Singleton /2

- **Structure**



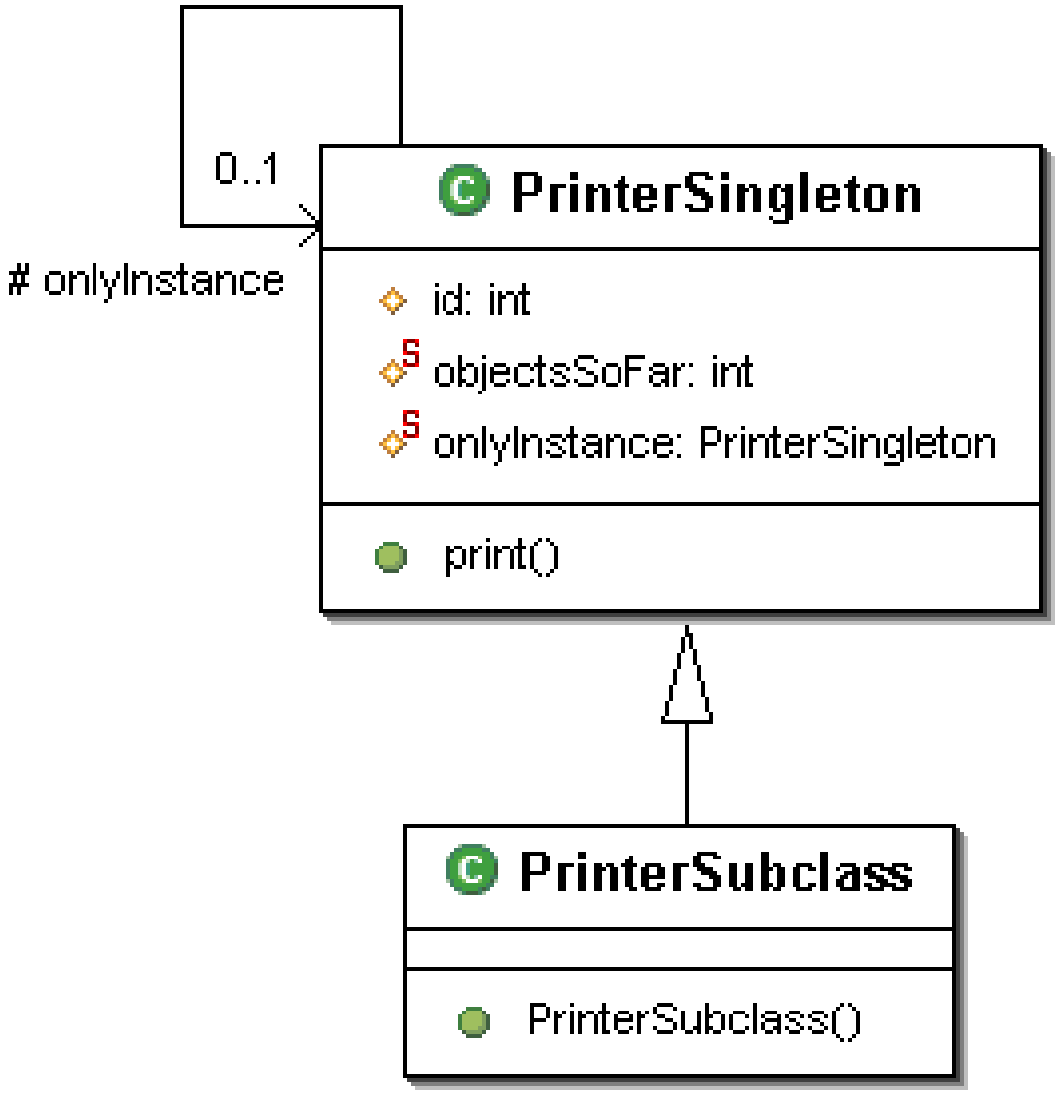
- **Participants**

- **Singleton**
- defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is, a static method in Java and a static member function in C++).
- may be responsible for creating its own unique instance.

Singleton /3

- **Collaborations**
  - Clients access a Singleton instance solely through Singleton's Instance operation.
- **Consequences**
  - Controlled access to sole instance
  - Reduced name space
  - Permits refinement of operations and representation
  - Permits a variable number of instances
  - More flexible than class operations

Singleton example 1

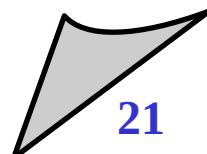


Singleton example 2

- **It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.**

Singleton questions

- **The Singleton pattern is often paired with the Abstract Factory pattern. What other creational or non-creational patterns would you use with the Singleton pattern?**

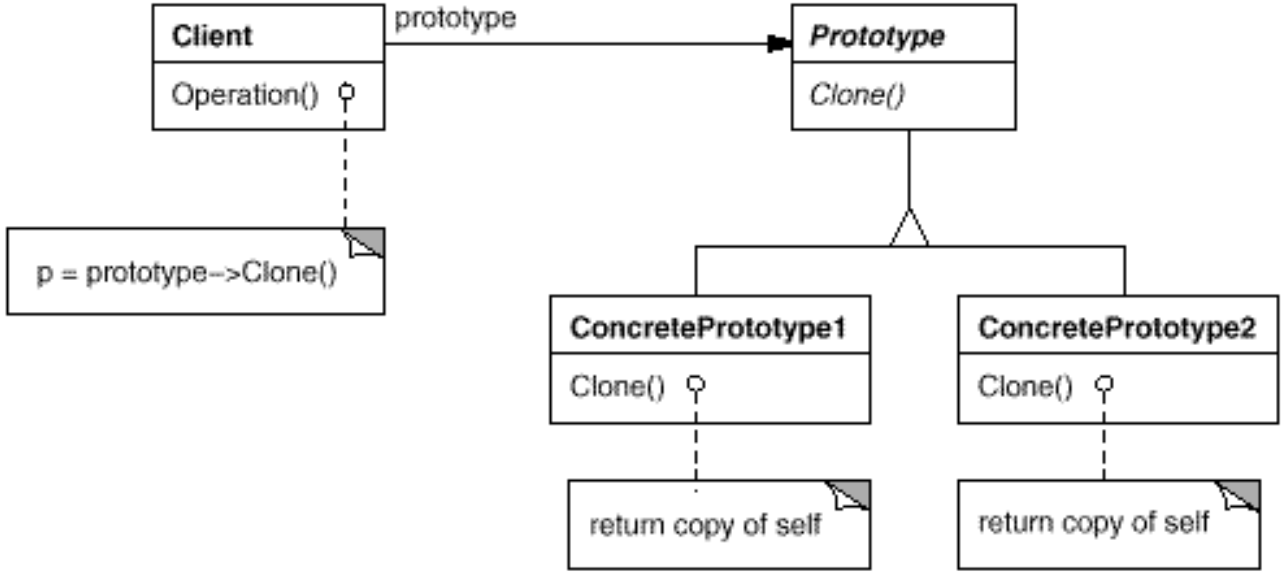


Prototype

- **Intent**
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Applicability**
  - when a system should be independent of how its products are created, composed, and represented; *and*
  - when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*
  - to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
  - when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Prototype /2

- **Structure**



- **Participants**
  - **Prototype**
  - declares an interface for cloning itself.
  - **ConcretePrototype**
  - implements an operation for cloning itself.

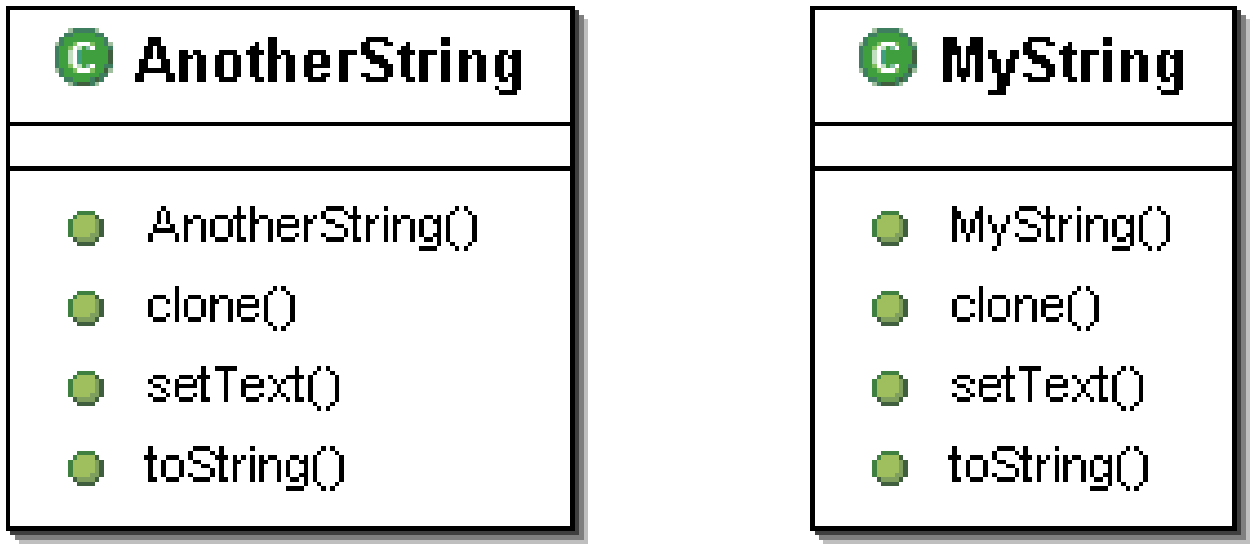
•**Client**

•creates a new object by asking a prototype to clone itself.

Prototype /3

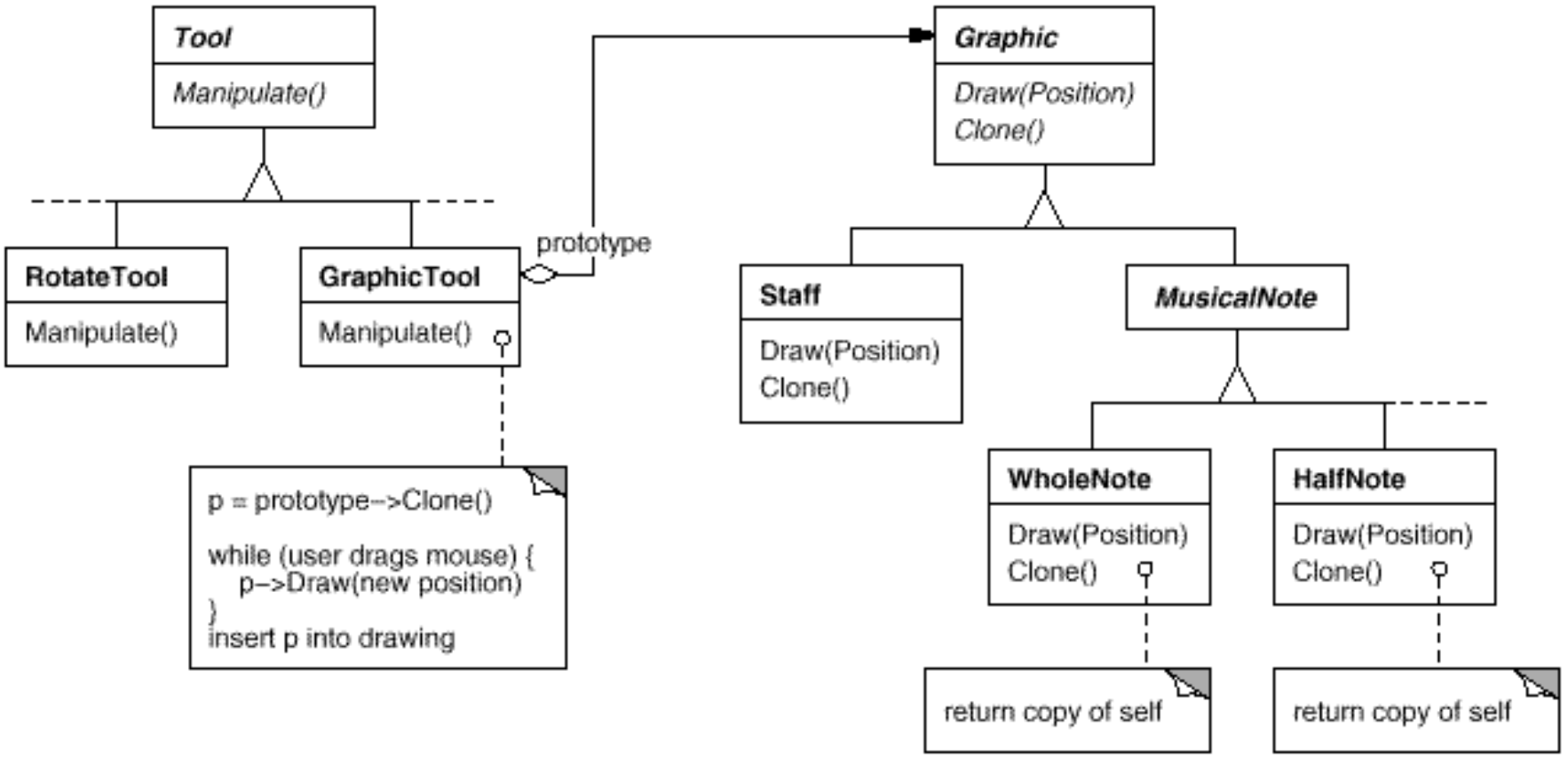
- **Collaborations**
  - A client asks a prototype to clone itself.
- **Consequences**
  - hides the concrete product classes from the client
  - Adding and removing products at run-time
  - Specifying new objects by varying values
  - Specifying new objects by varying structure
  - Reduced subclassing
  - Configuring an application with classes dynamically

Prototype example 1



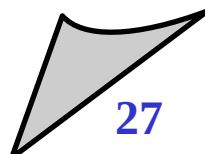
Prototype example 2

- **Build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves**



Prototype questions

- **Part 1: When should this creational pattern be used over the other creational patterns?**
- **Part 2: Explain the difference between deep vs. shallow copy**

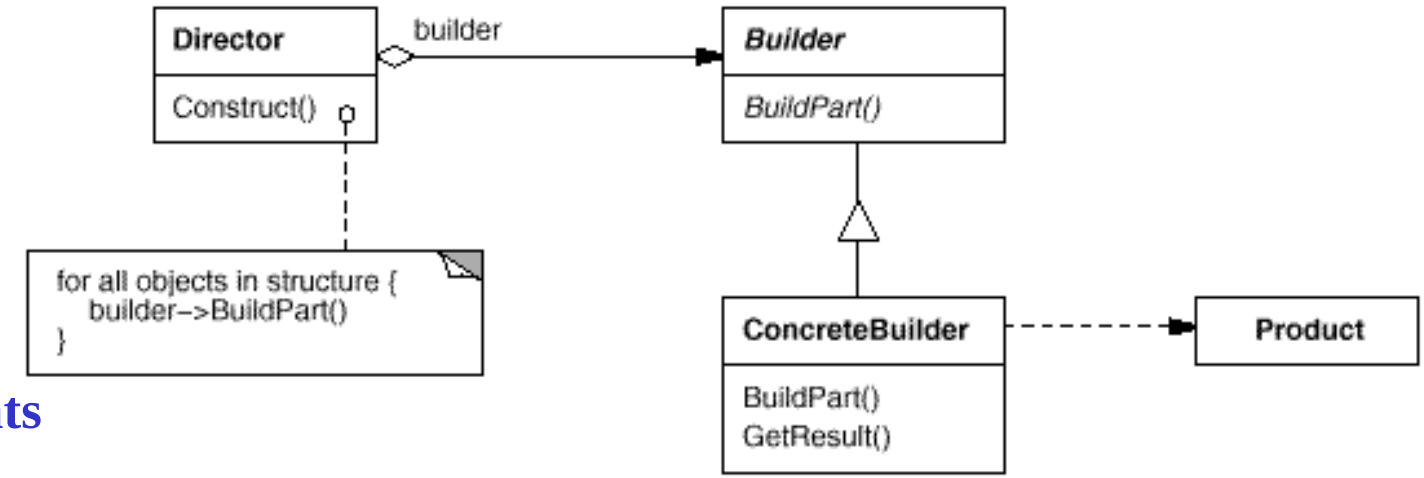


Builder

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

Builder /2

- **Structure**



- **Participants**

- **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.

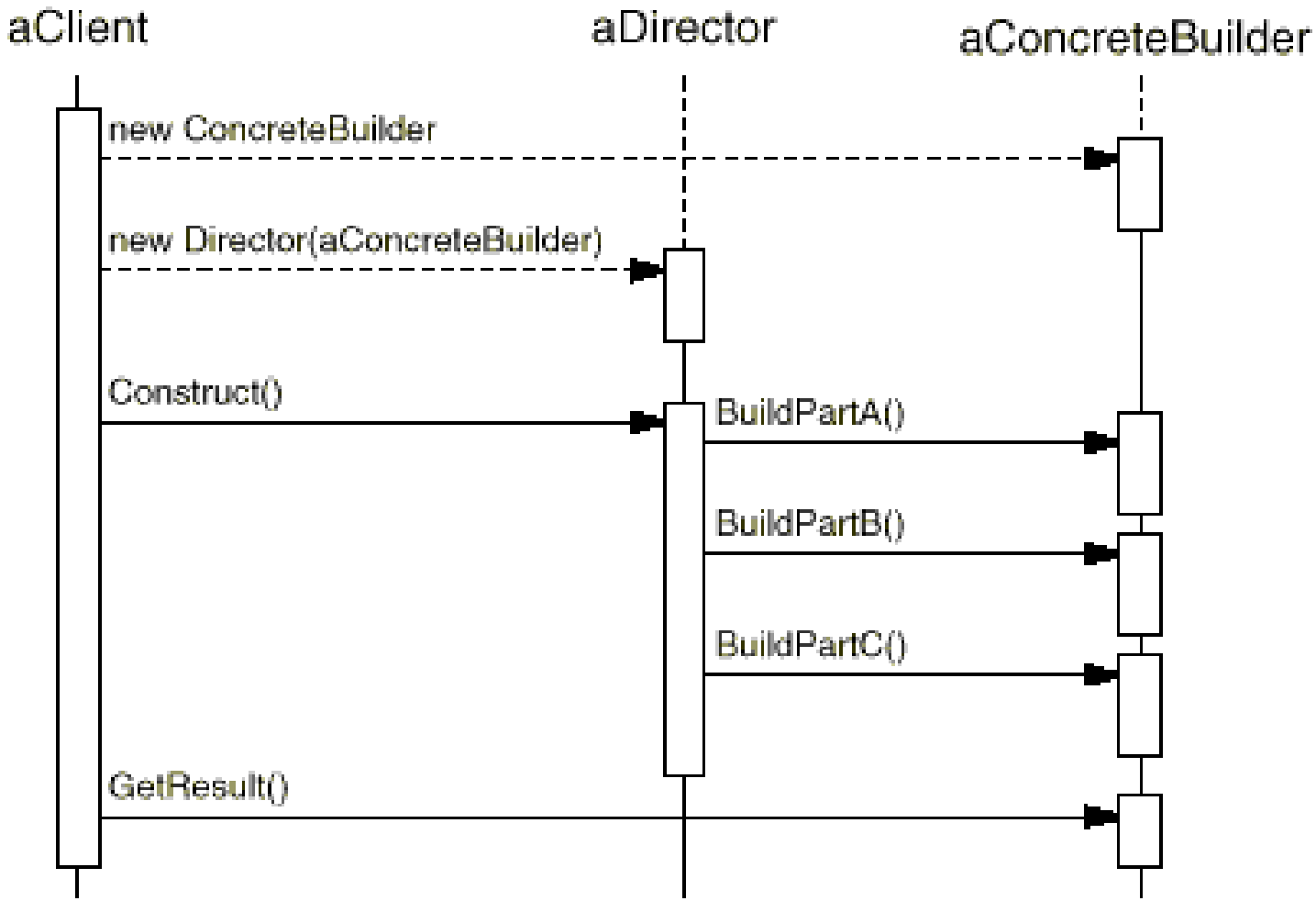
Builder /3

- **Collaborations**
  - Client creates Director object and configures it with 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.

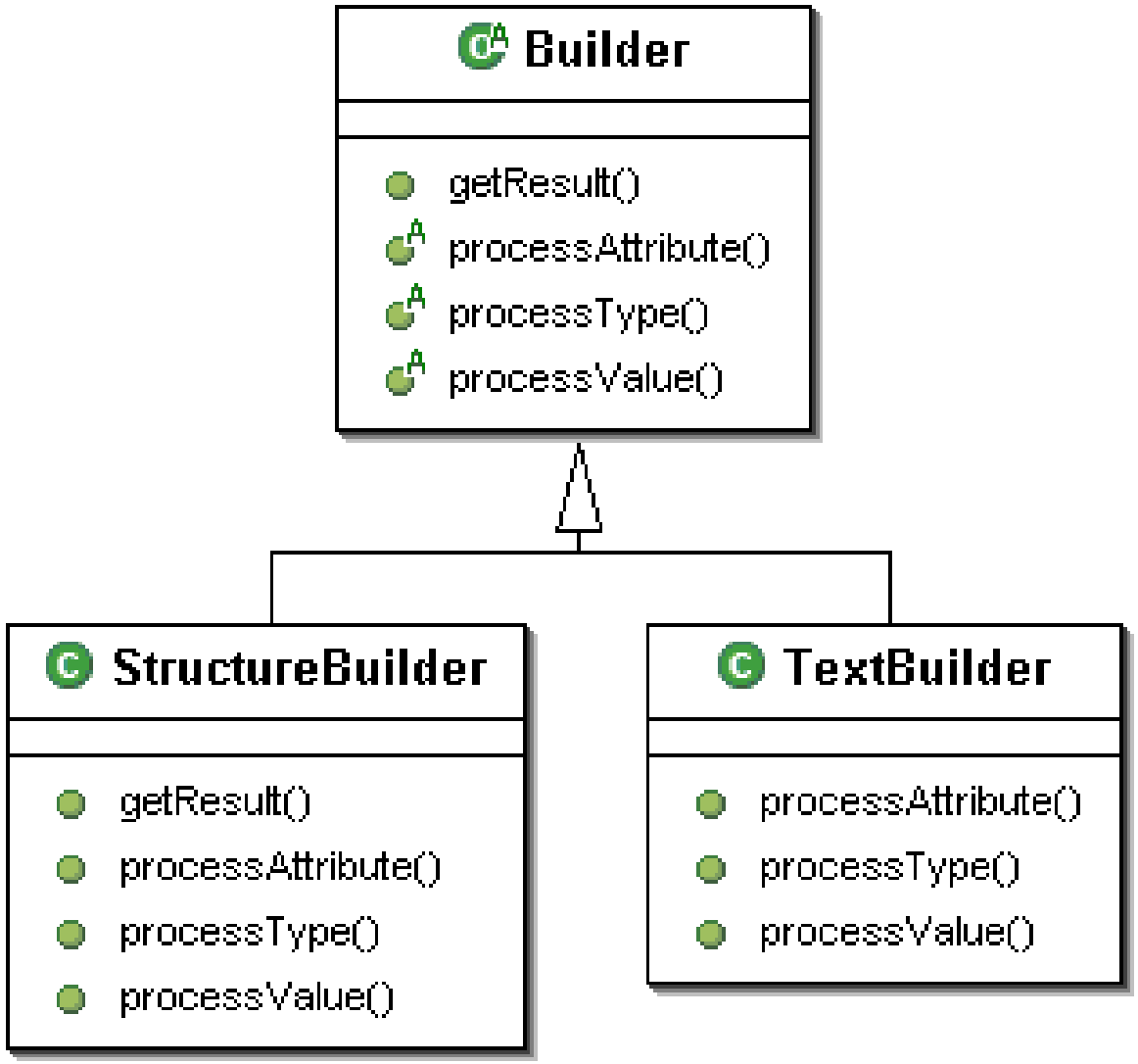
- **Consequences**
  - It lets you vary a product's internal representation
  - It isolates code for construction and representation
  - It gives you finer control over the construction process

Builder /4

- **The following interaction diagram illustrates how Builder and Director cooperate with a client.**

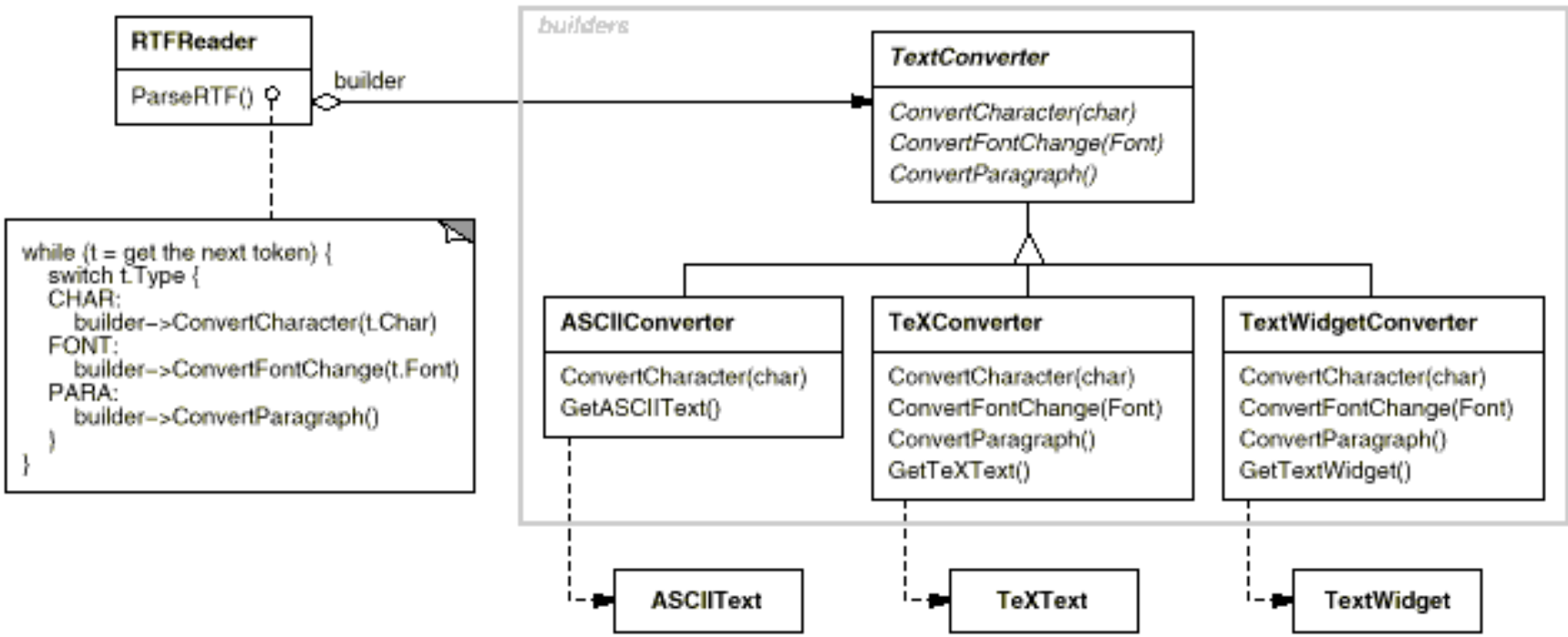


Builder example 1



Builder example 2

- **A reader for the RTF document exchange format should be able to convert RTF to many text formats. The problem is that the number of possible conversions is open-ended**



Builder questions

- **Like the Abstract Factory pattern, the Builder pattern requires that you define an interface, which will be used by clients to create complex objects in pieces. How does the Builder pattern allow one to add new methods to the Builder's interface, without having to change each and every sub-class of the Builder?**