Creational Patterns

Creational Patterns

- Factory Method (FM)
- Abstract Factory (AF)
- Singleton (SI)
- Prototype (PR)
- Builder (BU)

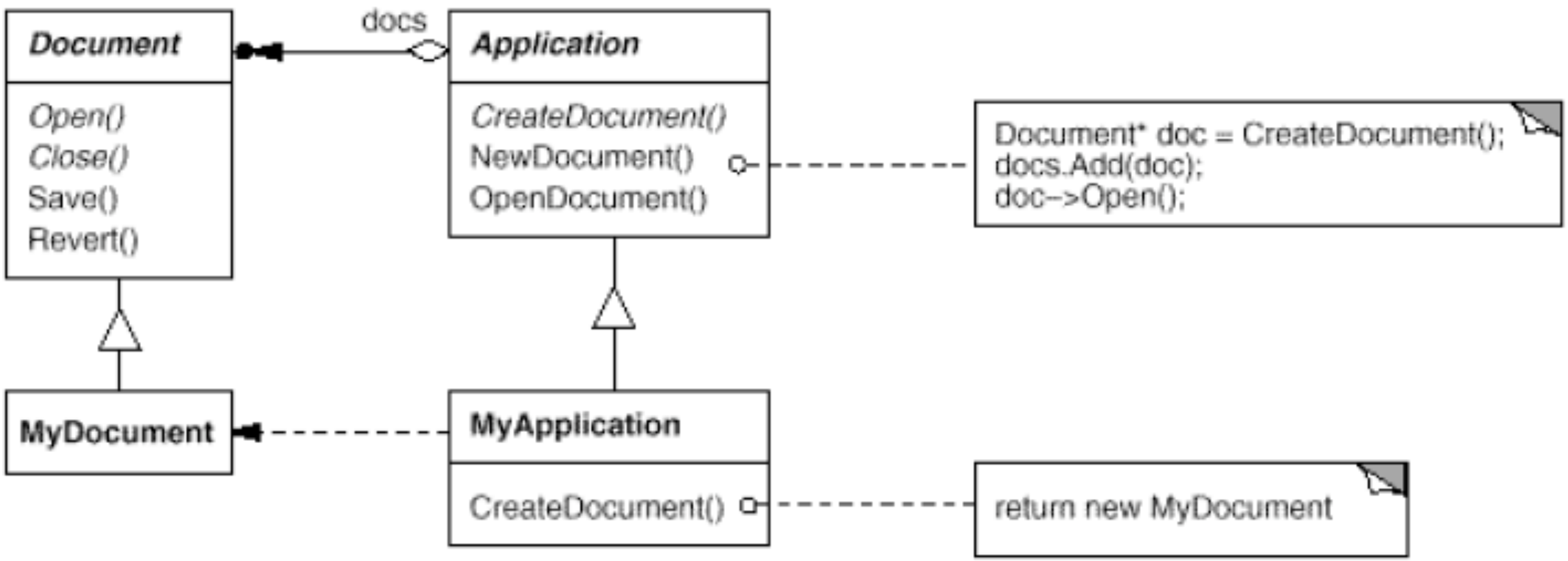
Factory Method (FM)

- 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.
- Also Known As
  - Virtual Constructor

FM Motivation (1)

- Consider a framework for applications that can present multiple documents to the user.
- To create a drawing application, for example, we define the classes DrawingApplication and DrawingDocument.
- The Application class is responsible for managing Documents
- the Application class can't predict the subclass of Document to instantiate
- Application subclasses redefine an abstract CreateDocument operation on Application to return the appropriate Document subclass.

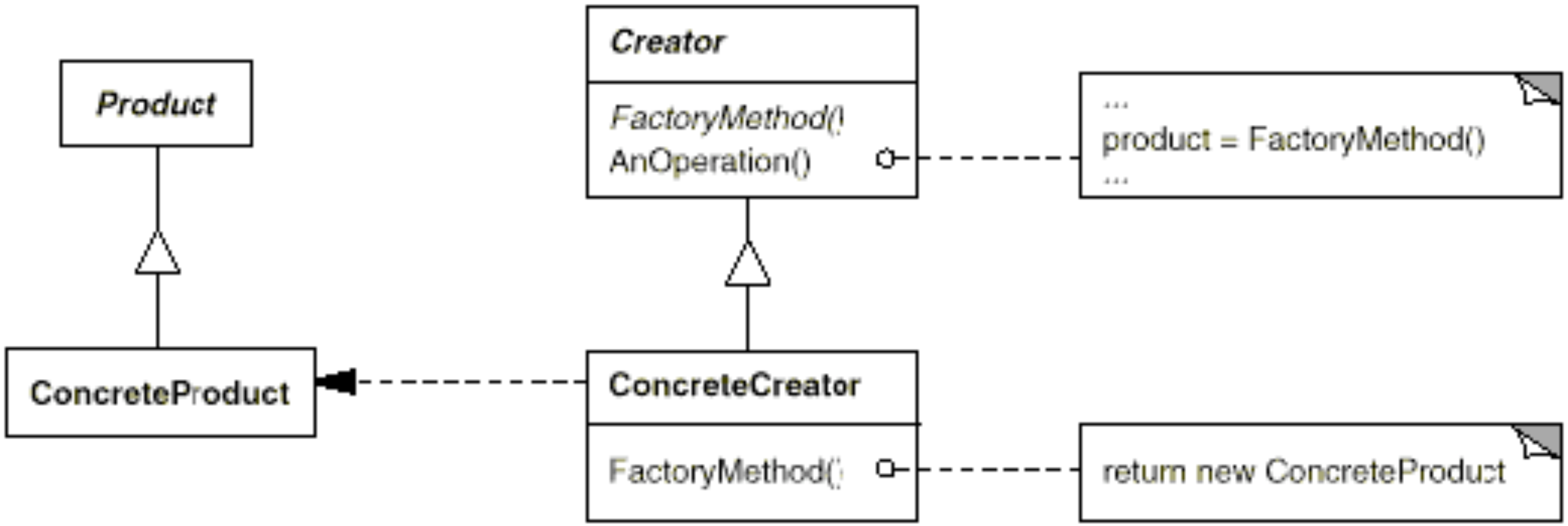
FM Motivation (2)



FM Applicability

- Use a FM when:
  - 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.

FM Structure



FM Participants

- Product (Document)
  - the interface of objects the factory method creates.
- ConcreteProduct (MyDocument)
  - implements the Product interface.
- Creator (Application)
  - declares the factory method (returns a Product).
  - may define a default implementation of the FM
  - may call the factory method to create a Product
- ConcreteCreator (MyApplication)
  - overrides the factory method to return an instance of a ConcreteProduct.

FM Collaboration

- Creator relies on its subclasses to define the factory method
so that it returns an instance of the appropriate ConcreteProduct.

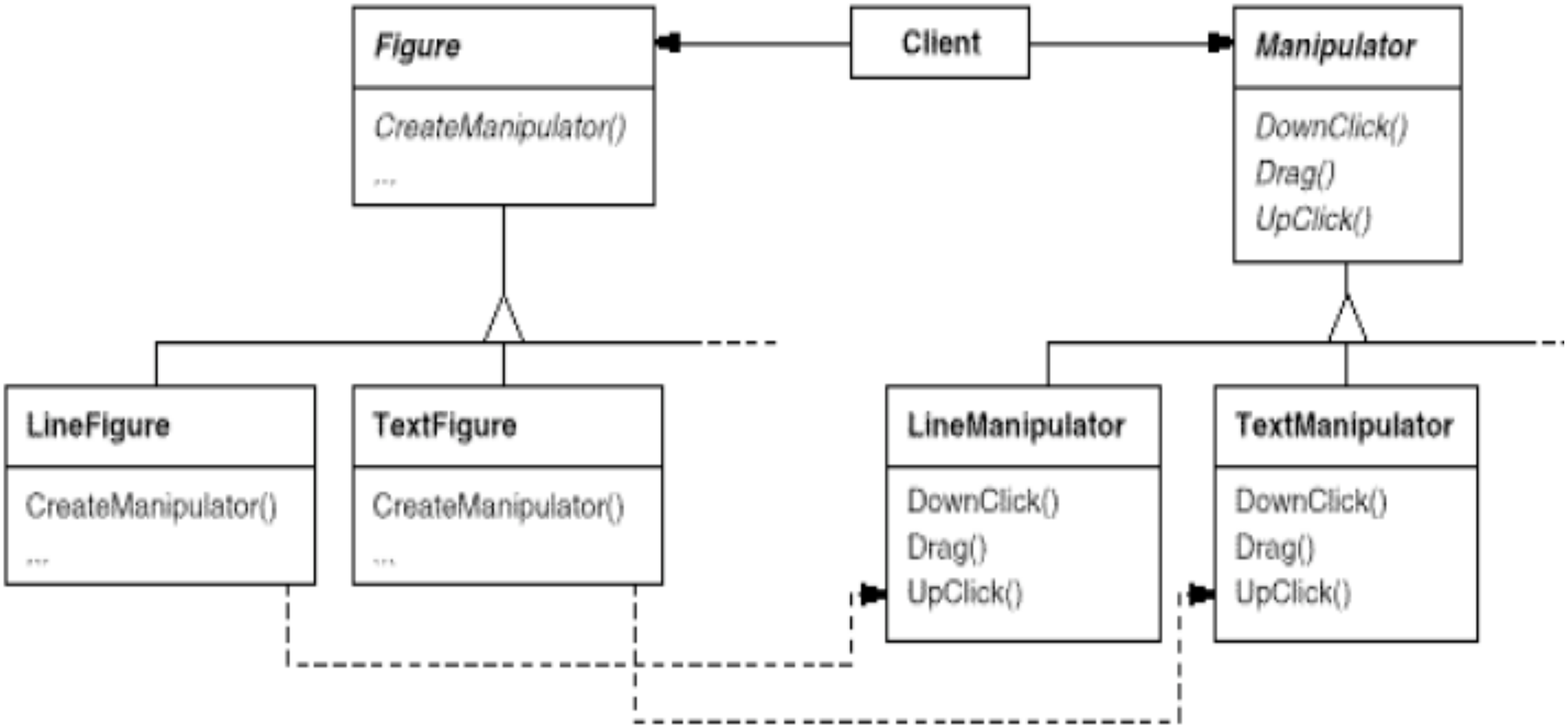
FM Consequences(1)

- FM eliminate the need to bind application-specific classes into your code.
  - The code only deals with the Product interface
- Clients might have to subclass Creator just to create a particular ConcreteProduct object.
  - is fine when the client has to subclass the Creator class anyway!
  - …otherwise is a drawback
    (the hierarchy can explode)

FM Consequences(2)

- FM
  - Provides hooks for subclasses.
    - Creating objects inside a class with a FM is always more flexible than creating an object directly.
    - FM gives subclasses a hook for providing an extended version of an object.
  - Connects parallel class hierarchies.
    - Parallel class hierarchies result when a class delegates some of its responsibilities to a separate class.

FM Consequences(2)



FM Implementation(1)

- Two possibilities:
  - Creator is an abstract class and does not provide an implementation for the FMs it provides
  - Creator is a concrete class and provides a default implementation for the FMs it provides
- Parameterized factory methods.
  - lets the factory method create *multiple* kinds of products.
  - The factory method takes a parameter that identifies the kind of object to create.
  - May require downcasting

FM Implementation(2)

- Naming Conventions
  - It's good practice to use naming conventions that make it clear you're using factory methods.
- Using templates to avoid subclassing.

FM Implementation(2)

- Naming Conventions

```
class Creator {
public:
    virtual Product* CreateProduct() = 0;
};

template <class TheProduct>
class StandardCreator: public Creator {
public:
    virtual Product* CreateProduct();
};

template <class TheProduct>
Product* StandardCreator<TheProduct>::CreateProduct () {
    return new TheProduct;
}
```

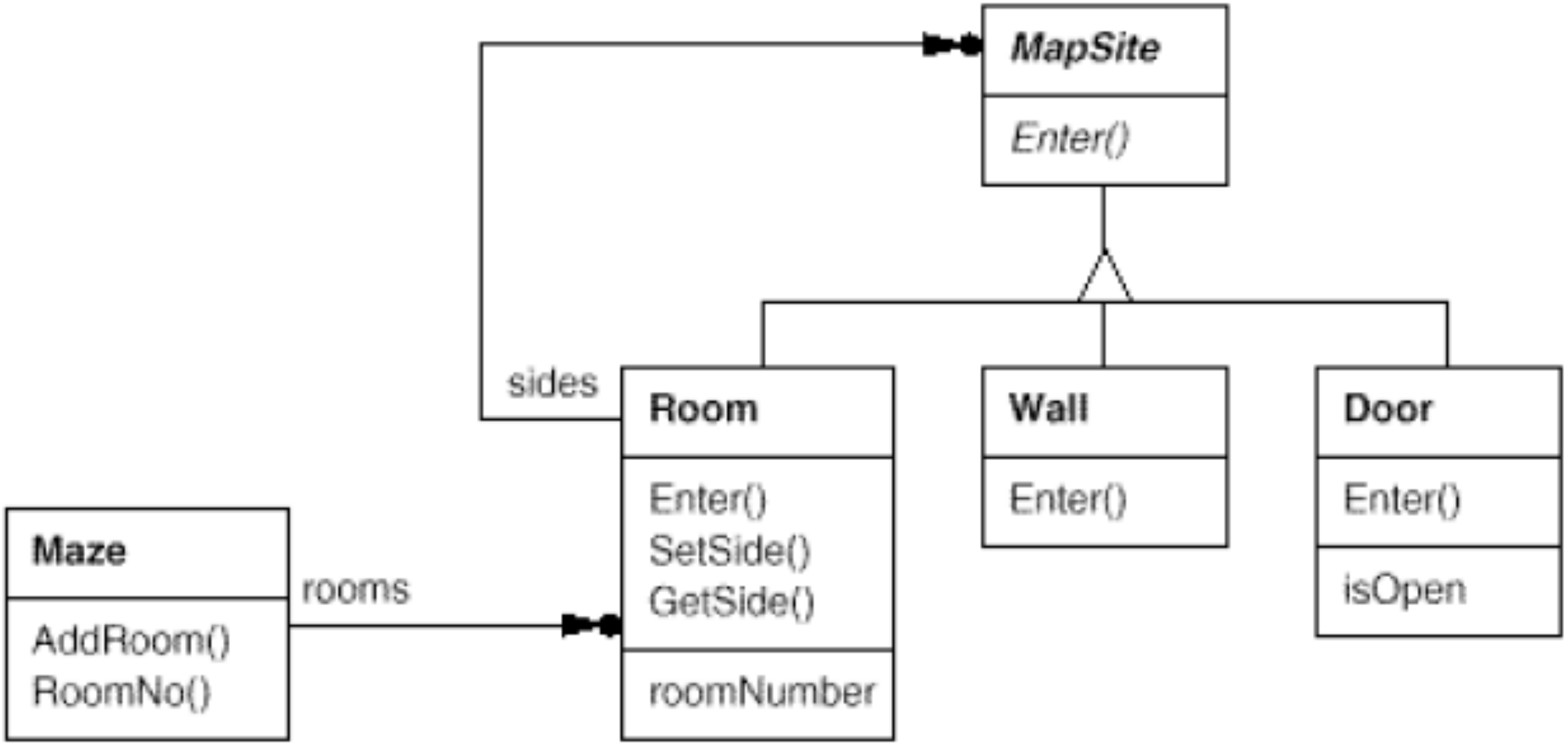
FM Implementation(2)

- Naming Conventions
  - It's good practice to use naming conventions that make it clear you're using factory methods.
- Using templates to avoid subclassing.

```
class MyProduct : public Product {
public:
    MyProduct();
    // ...
};

StandardCreator<MyProduct> myCreator;
```

Building a Maze for a game



Sample Code

```
class MazeGame {
public:
    Maze* CreateMaze();

// factory methods:

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze();

    Room* r1 = MakeRoom(1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, MakeWall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, MakeWall());
    r1->SetSide(West, MakeWall());

    r2->SetSide(North, MakeWall());
    r2->SetSide(East, MakeWall());
    r2->SetSide(South, MakeWall());
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* MakeWall() const
        { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};

class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();

    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

FM Known uses and Related Patterns

- Known Uses
  - Can be used in Abstract Factory
  - ….many softwares
- Related Patterns
  - Abstract Factory
  - Template Methods
  - Prototypes

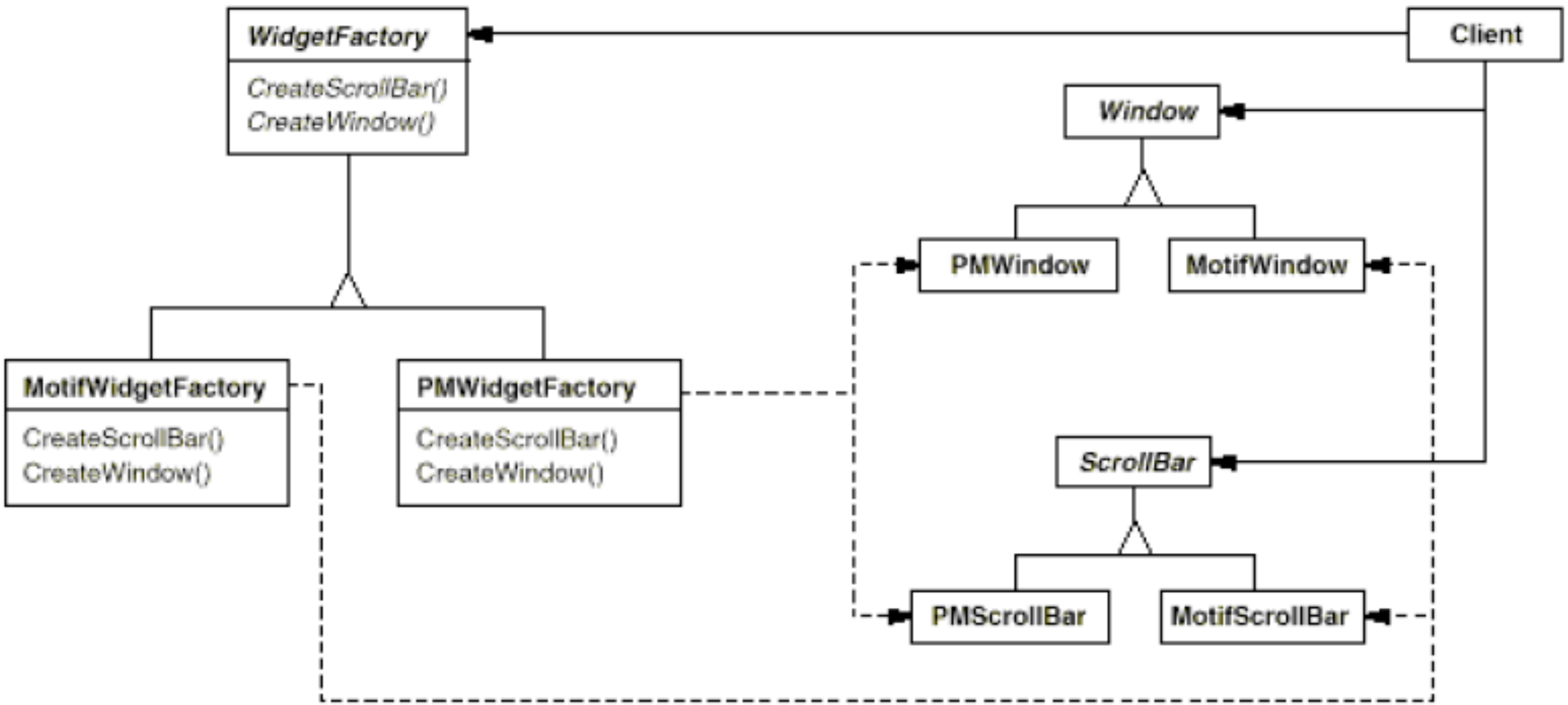
Abstract Factory (AF)

- Intent
  - Provide an interface for creating families of related or dependent objects without specifying theirconcrete classes.
- Also Known As
  - Kit

AF Motivation (1)

- Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager.
- Different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons.
- To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel.

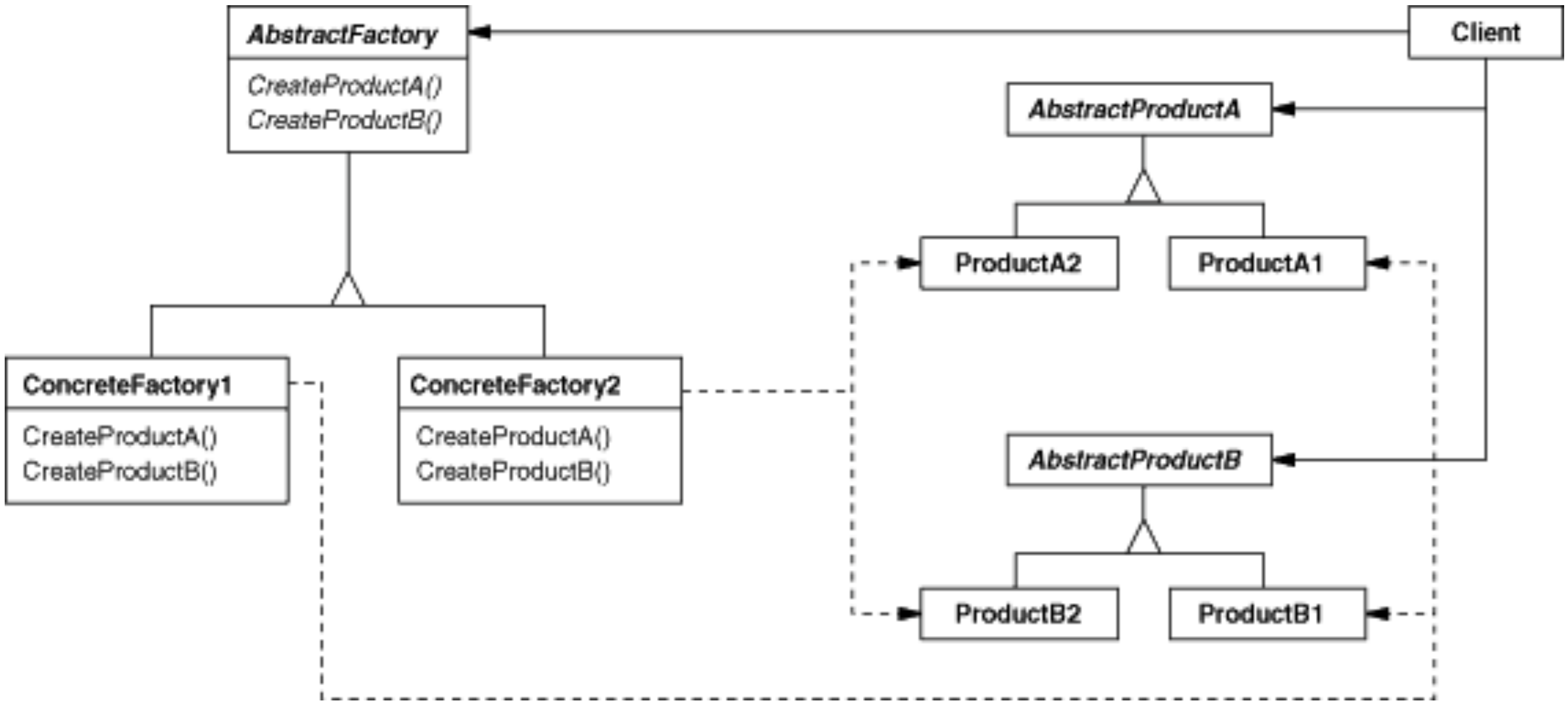
AF Motivation (2)



AF Applicability

- Use the AF when:
  - 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.

AF Structure



AF Participants

- AbstractFactory (WidgetFactory)
  - declares an interface for creating abstract products.
- ConcreteFactory (MotifWidgetFactory, …)
  - implements the operations creating concrete products
- AbstractProduct (Window, ScrollBar)
  - declares an interface for a type of product object.
- ConcreteProduct (MotifWindow, …)
  - defines a product object to be created by the AF
  - implements the AbstractProduct interface.
- Client
  - uses only interfaces declared by AF and Abs. Prod.

AF Collaborations

- A single instance of a ConcreteFactory class is created at run-time.
  - This concrete factory creates product objects having a particular implementation.
  - To create different product objects, clients should use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

AF Consequences (1)

- It isolates concrete classes.
  - The Abstract Factory pattern helps you control the classes of objects
  - Clients manipulate instances through their abstract interfaces.
  - Product class names do not appear in client code.
- It makes exchanging product families easy.
  - The class of a concrete factory appears only once in an application.
    - This makes it easy to change the concrete factory an application uses.
    - It can use different product configurations simply by changing the concrete factory.

AF Consequences (2)

- It promotes consistency among products.
  - an application use objects from only one family at a time.
- Supporting new kinds of products is difficult.
  - Extending abstract factories to produce new kinds of Products isn't easy.
  - Supporting new kinds of products requires extending the factory interface
    - involves changing the AF class and all of its subclasses
    - This can be (partially) solved (see implementation)

AF implementation (1)

- Factories as singletons.
  - An application typically needs only one instance of a ConcreteFactory (Singleton).
- Creating the products.
  - AF only declares an interface for creating products.
  - It's up to ConcreteFactory subclasses to actually create them.
  - Implement AF by using Factory Method
  - Implement AF by using Prototype

AF implementation (2)

- Defining extensible factories.
  - AF usually defines a different operation for each kind of product
  - A more flexible design is to add a parameter to operations that create objects.
  - easier to use in a dynamically typed language like Smalltalk than in C++.
  - (see FM implementation)

AF Sample Code

```
class MazeFactory {
public:
    MazeFactory();

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
    r1->SetSide(West, factory.MakeWall());

    r2->SetSide(North, factory.MakeWall());
    r2->SetSide(East, factory.MakeWall());
    r2->SetSide(South, factory.MakeWall());
    r2->SetSide(West, aDoor);

    return aMaze;
}
```

```
class EnchantedMazeFactory : public MazeFactory {
public:
    EnchantedMazeFactory();

    virtual Room* MakeRoom(int n)  const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2)  const
        { return new DoorNeedingSpell(r1, r2); }

protected:
    Spell* CastSpell() const;
};
```

```
Wall* BombedMazeFactory::MakeWall () const {
    return new BombedWall;
}

Room* BombedMazeFactory::MakeRoom(int n) const {
    return new RoomWithABomb(n);
}
```

```
MazeGame game;
BombedMazeFactory factory;

game.CreateMaze(factory);
```

AF Known Uses and Related Patterns

- Known Uses
  - …many applications
- Related Patters
  - Factory Mathod
  - Singleton
  - Prototype

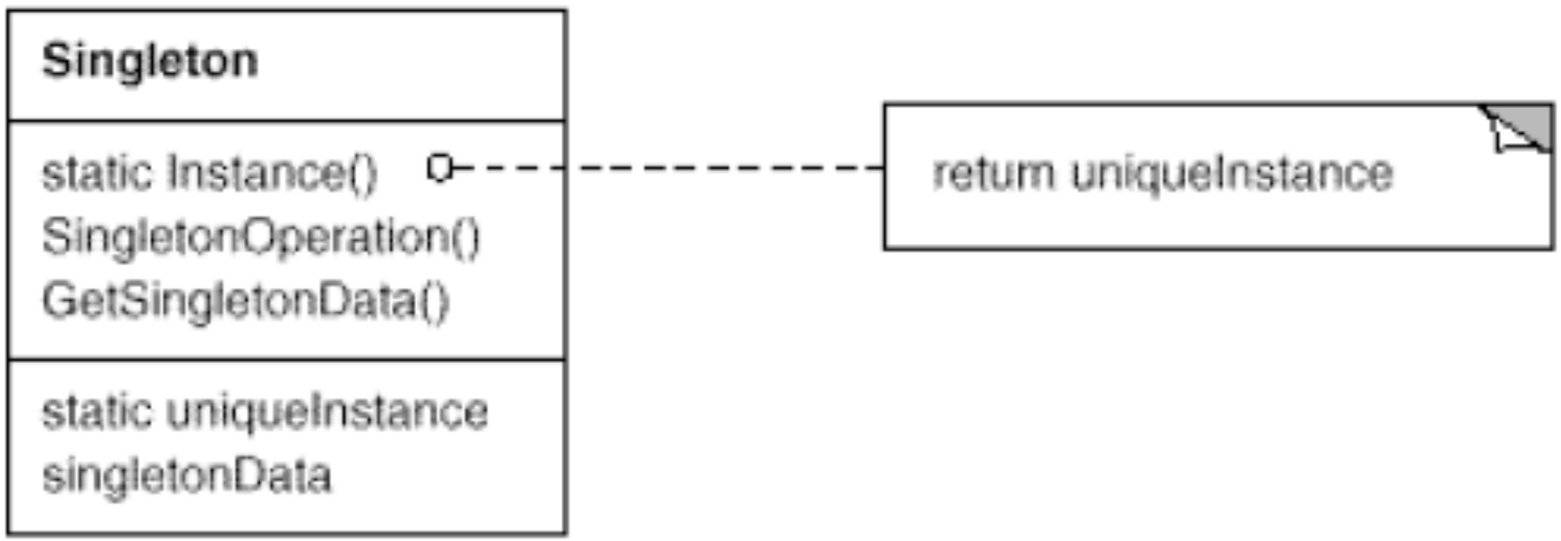
Singleton (SI)

- Intent
  - Ensure a class only has one instance, and provide a global point of access to it.
- Motivation
  - If is needed to have exactly one instance of a class.
  - A global variable makes an object accessible,
    - but it doesn't keep you from instantiating multiple objects.
  - Make the class itself responsible of its sole instance.
    - The class can ensure that no other instance can be created
      - by intercepting requests to create new objects,
    - …and it can provide a way to access the instance.

SI Applicability

- Use the Singleton pattern when
  - there must be exactly one instance of a class,
  - …and it must be accessible to clients from a wellknown 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.

SI Structure



SI Participants and Collaborations

- Participants
  - Singleton
    - defines an Instance operation that lets clients access its unique instance.
    - Instance is a class operation (e.g. static member)
  - may be responsible for creating its own unique instance.
- Collaborations
  - Clients access a Singleton instance solely through Singleton's Instance operation.

SI Consequences

- Controlled access to sole instance.
  - strict control over how and when the client access.
- Reduced name space.
  - No name space-pollution by global variables
- Permits refinement of operations and representation.
  - The Singleton class may be subclassed
  - Use the instance of the class you need at run-time.
- Permits a variable number of instances.
- More flexible than class operations.
  - static member functions in C++ are never virtual

SI Implementation

- Ensuring a unique instance.
  - Hide the operation that creates the instance behind a class operation
    (private constructor + static member)
- Subclassing the singleton class
  - The variable that refers to the singleton instance must get initialized with an instance of the subclass.
  - May be flexible to use a registry of singletons.

```cpp
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};


Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

```
class Singleton {
public:
    static void Register(const char* name, Singleton*);
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};
```

```
Singleton* Singleton::Instance () {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // user or environment supplies this at startup

        _instance = Lookup(singletonName);
        // Lookup returns 0 if there's no such singleton
    }
    return _instance;
}
```

```
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}

static MySingleton theSingleton;
```

SI Sample Code

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // existing interface goes here
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};


MazeFactory* MazeFactory::_instance = 0;
```

SI Sample Code

```
MazeFactory* MazeFactory::Instance () {

    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

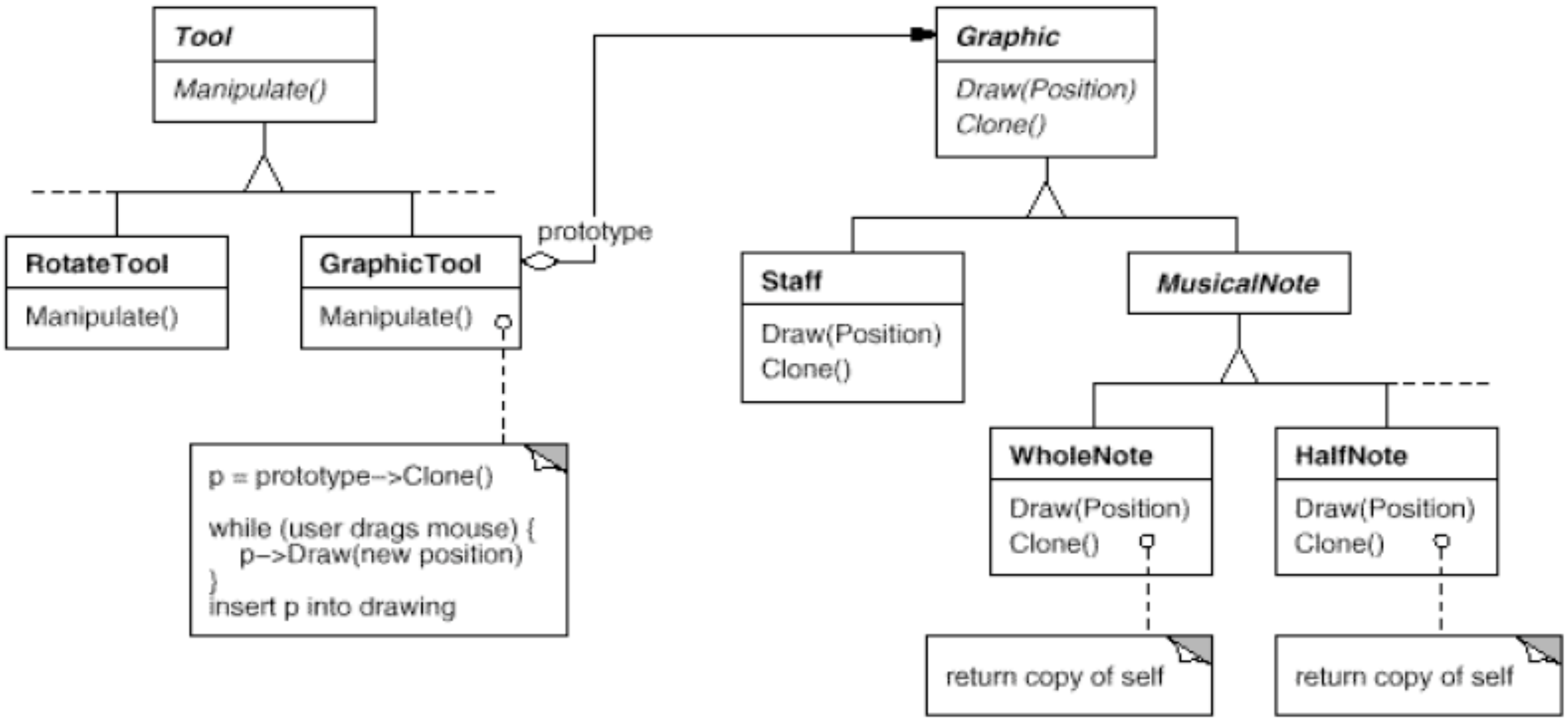
SI Known Uses and Related Patterns

- Known Uses
  - …many applications
- Related Patterns
  - Abstract Factory
  - Builder
  - Prototype

Prototype (PR)

- Intent
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying (cloning) this prototype.
- Motivation
  - Build an editor for music scores
    - by customizing a general framework
    - Scores are created by adding new objects that represent notes, rests, and staves from a palette
    - Sublcassing from an abstract Graphic class
      - produce lots of subclasses that differ only in the kind of music object they instantiate.

PR Motivation (continued)

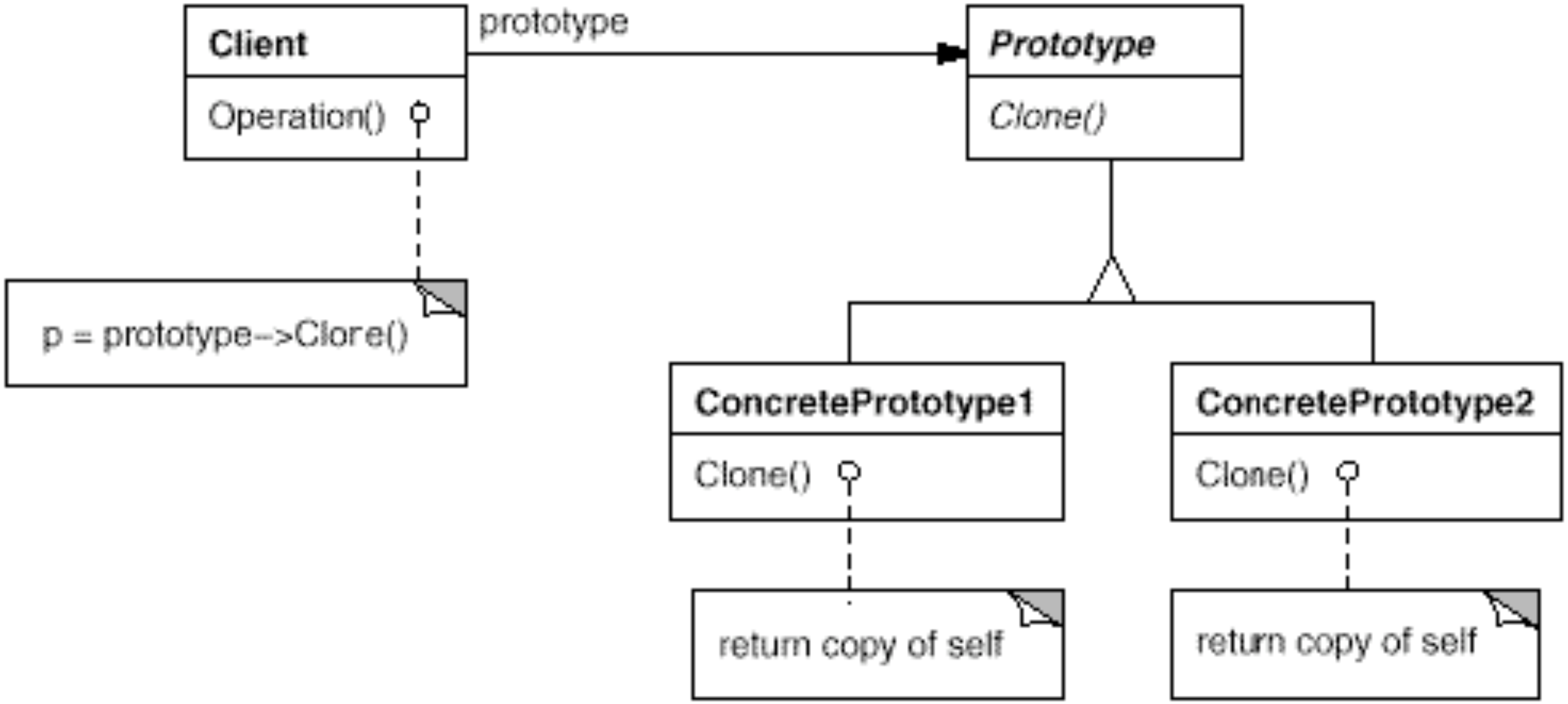


…make GraphicTool create a new Graphic by cloning an instance of a Graphic subclass (the prototype!).

PR Applicability

- Use the PR when
  - a system should be independent of how its products are created, composed, and represented;
  - the classes to instantiate are specified at run-time
  - Parallels AF hierarchy may be very big
  - instances have one of only a few different combinations of state.

PR Structure



PR Participants and Collaborations

- Participants
  - Prototype (Graphic)
    - declares an interface for cloning itself.
  - ConcretePrototype (Staff, WholeNote, HalfNote)
    - implements an operation for cloning itself.
  - Client (GraphicTool)
    - creates a new object by asking a prototype to clone itself.
- Collaborations
  - A client asks a prototype to clone itself.

PR Consequences (1)

- PR similar to AF and BU it:
  - hides the concrete product classes from the client
  - lets a client work with application-specific classes without modification.
- Adding and removing products at run-time.
  - a bit more flexible than other creational patterns, because a client can install and remove prototypes at run-time.
- Specifying new objects by varying values.
  - PR lets users define new "classes" without programming.

PR Consequences (2)

- Reduced subclassing.
  - FM often produces a hierarchy of Creator classes that parallels the product class hierarchy.
  - PR lets you clone a prototype instead of asking a factory method to make a new object.
    - NO Creator class hierarchy at all.
- Configuring an application with classes dynamically.
  - The Prototype pattern is the key to exploiting facilities like Java reflection in a language like C++.

PR Implementation

- Particularly useful with static languages (C++)
  - where classes are not objects, and little or no type information is available at run-time.
- Implementation issues:
  - Using a prototype manager.
    - When the number of prototypes in a system isn't fixed keep a registry of available prototypes.
  - Implementing the Clone operation.
    - It's particularly tricky when object structures contain circular references.
  - Initializing clones
    - If clients want to initialize the internal state
      - Introduce an Initialize(…) operation.

PR Sample Code

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;

private:

    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

PR Sample Code

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d
) {
    _prototypeMaze = m;
    _prototypeWall = w;
    _prototypeRoom = r;
    _prototypeDoor = d;
}
Wall* MazePrototypeFactory::MakeWall () const {
    return _prototypeWall->Clone();
}

Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {
    Door* door = _prototypeDoor->Clone();
    door->Initialize(r1, r2);
    return door;
}
```

PR Sample Code

```
class BombedWall : public Wall {
public:
    BombedWall();
    BombedWall(const BombedWall&);

    virtual Wall* Clone() const;
    bool HasBomb();
private:
    bool _bomb;
};

BombedWall::BombedWall (const BombedWall& other) : Wall(other) {
    _bomb = other._bomb;
}

Wall* BombedWall::Clone () const {
    return new BombedWall(*this);
}
```

PR Sample Code

```
MazeGame game;
MazePrototypeFactory simpleMazeFactory(
    new Maze, new Wall, new Room, new Door
);

Maze* maze = game.CreateMaze(simpleMazeFactory);


MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
);
```

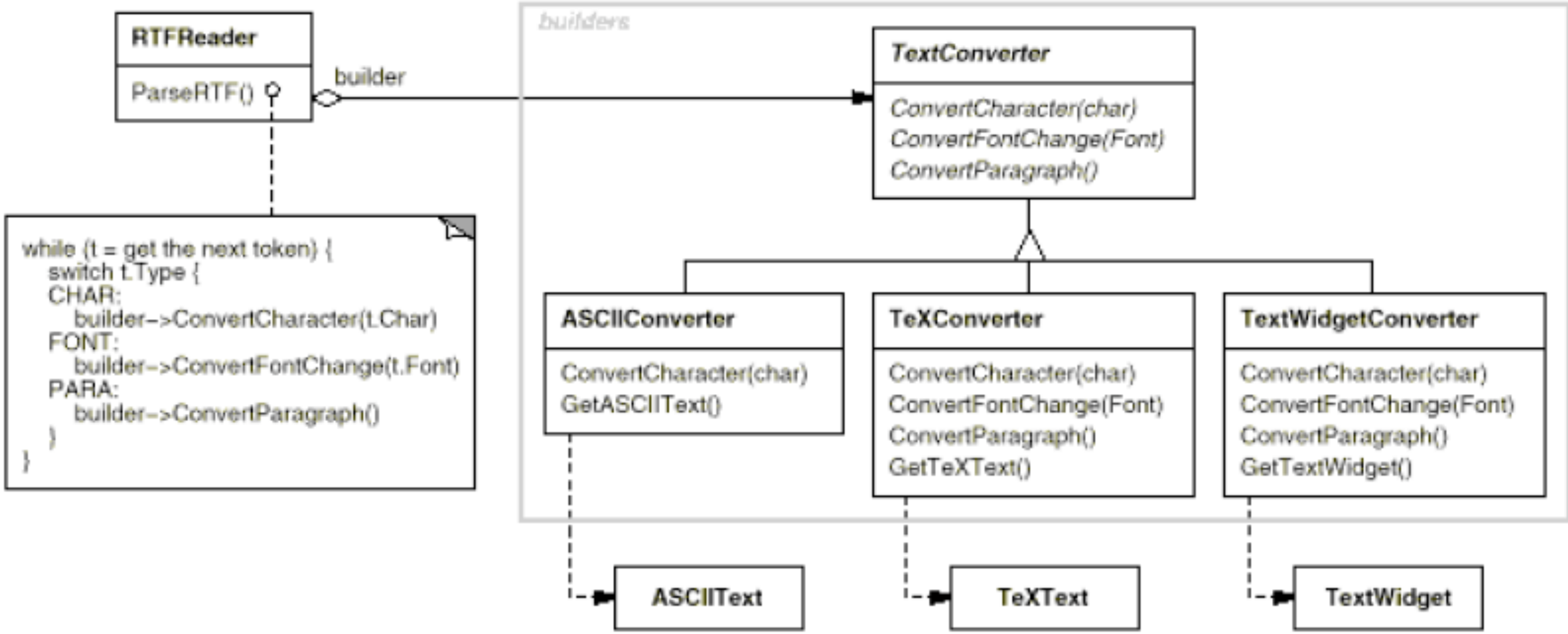
PR Known Uses and Related Patterns

- Known Uses
  - …many applications
- Related Patterns
  - Abstract Factory
  - Composite
  - Decorator

Builder (BU)

- Intent
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Motivation
  - A reader for the RTF (Rich Text Format) format
    - should be able to convert RTF to many text formats.
      - into plain ASCII text
      - or into a text widget that can be edited interactively.
    - It should be easy to add a new conversion without modifying the reader.

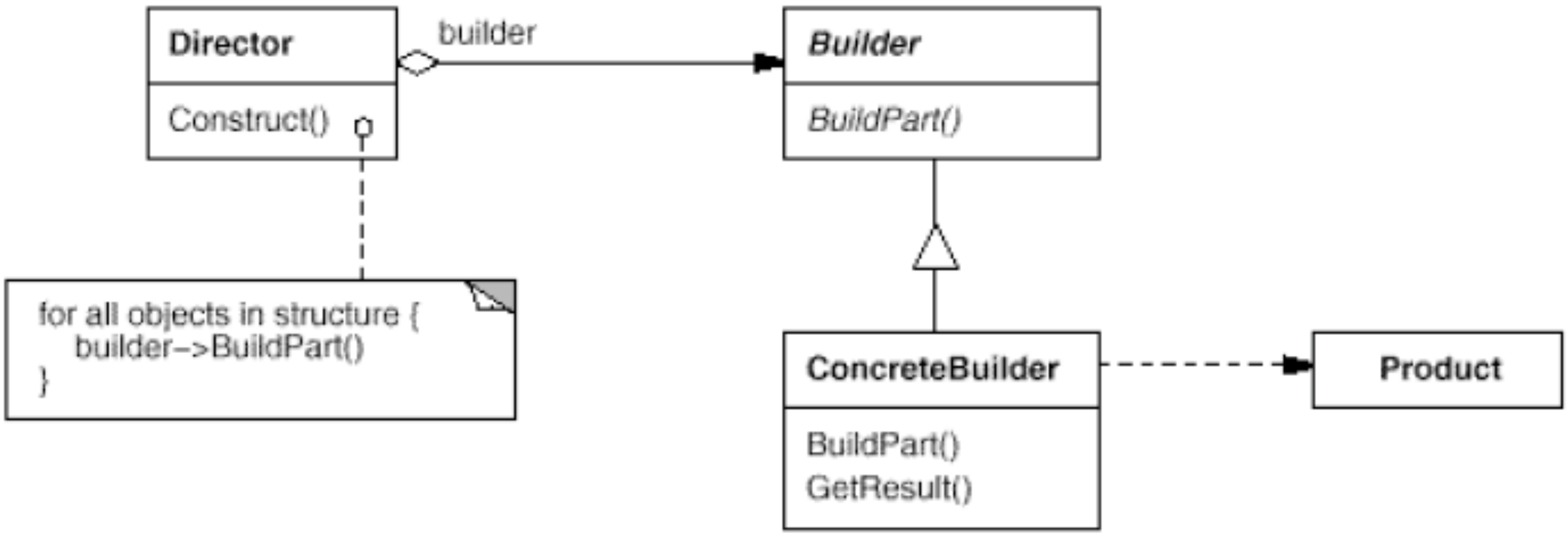
BU Motivation (continued)



BU Applicabiliy

- 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's constructed.

BU Structure



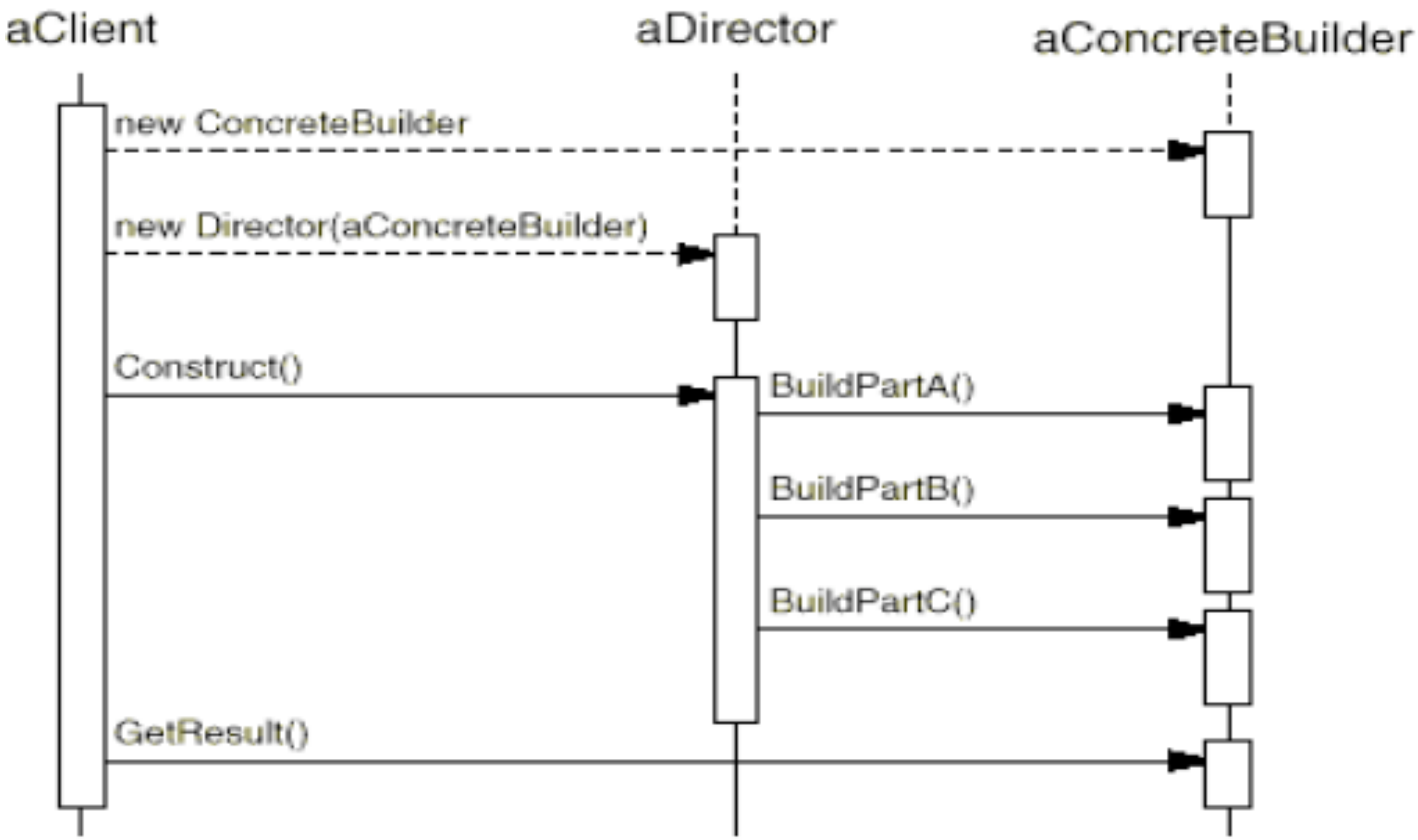
BU Participants

- Builder (TextConverter)
  - abstract interface for creating parts of a Product
- ConcreteBuilder (ASCIIConverter, …)
  - constructs and assembles parts of the product by implementing the Builder interface.
  - provides an interface for retrieving the product (e.g., GetASCIIText, GetTextWidget).
- Director (RTFReader)
  - constructs an object using the Builder interface.
- Product (ASCIIText, TeXText, TextWidget)
  - represents the complex object under construction.
  - includes classes that define the constituent parts

BU 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.

BU Collaborations



BU 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.

BU Consequences (2)

- It isolates code for construction and representation.
  - BU improves modularity by encapsulating the way a complex object is constructed and represented.
  - Clients needn't know anything about the product's internal structure
- It gives you finer control over the construction process.
  - The product is built step by step under the director's control.
  - Only when the product is finished does the director retrieve it from the builder.

BU Implementation (1)

- 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 it's interested in creating.

BU Implementation (2)

- Assembly and construction interface.
  - Builders construct their products step-by-step
  - The BU interface allows the construction of products for all kinds of concrete builders.
- Why no abstract class for products?
  - The products differ so greatly in their representation
    (e.g. ASCIIText and TextWidget differs)
- Empty methods as default in Builder.
  - Virtual member functions have empty methods letting clients override only the operations they're interested in.

BU Sample Code

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }

    virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};
```

BU Sample Code

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

BU Known Uses and Related Patterns

- Known Uses
  - …many applications
- Related Patterns
  - Abstract Factory
  - Composite

Discussion (1)

- Two common ways to parameterize a system by the classes of objects it creates:
  - To subclass the class that creates the objects
    (Factory Method)
  - to parameterize a system relying on object composition
    (Abstract Factory, Builder, Prototype)

Discussion (2)

- The main drawback of FM is that it can require creating a new subclass just to change the class of the product.
  - Such changes can cascade.
- The “composition-based” pattern
  - Involve creating a new "factory object" whose responsibility is to create product objects.
    - **Abstract Factory** has the factory object producing objects of several classes.
    - **Builder** has the factory object building a complex product incrementally using a complex protocol.
    - **Prototype** has the factory object (the prototype itself) building a product by copying a prototype object.