Creational Patterns

Factory Method

Abstract Factory

Object creation

- We create objects with **new**
  - `obj= new ConcreteClassName()`
- Technically there's nothing wrong with **new**.
- The real culprit is CHANGE and how change impacts our use of **new**.

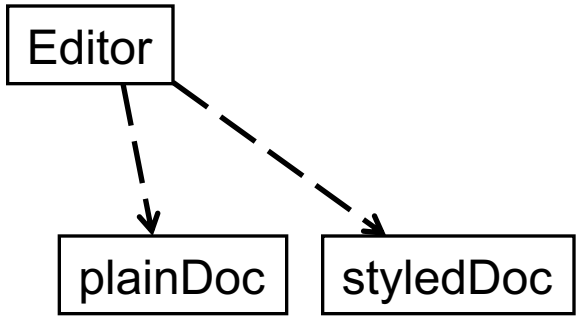
Recall: implement to interface

- By coding to an interface, you know you can insulate yourself from a lot of changes that might happen to a system due to concrete class changes.
- If your code is written to an interface, then it will work with any new classes implementing that interface through polymorphism.

Coupling with “new”

- TextEditor depends on concrete classes because of “new”

```
class TextEditor{
    private Document doc;
    …
        doc=new plainDoc();
        //do some adjustments
   …
    public void openDoc(){…..doc.xxx ……}
}
```

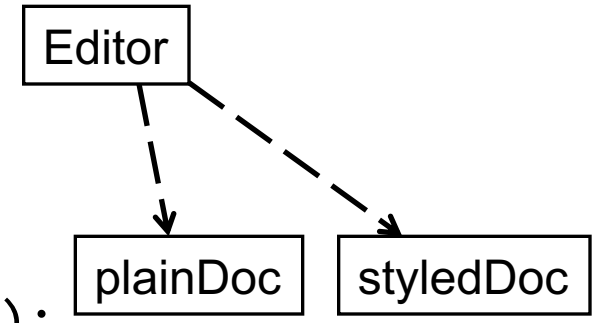


- What if I want to work with StyledDoc later ?

Coupling with “new”

- TextEditor depends on concrete classes because of “new”

```
class TextEditor{
 private Document doc;
 …
 public void loadDocument(Type t) {
   if(t.equals(plain)) doc=new plainDoc();
   else if ((t.equals(styled)) doc=new styledDoc();
    //…do some adjustments on the doc
 }
     …//do some generic document manipulation
}
```

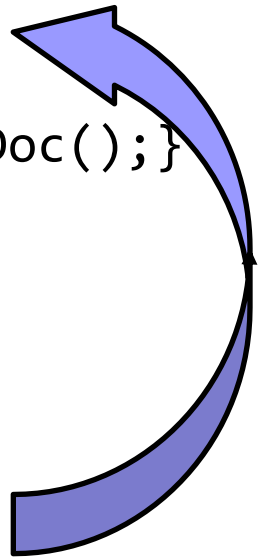


- What if I want to add new kinds of documents into my design?
- needs to be recompiled each time a dep. changes add new classes, change this code remove existing classes, change this code
- What is varying? **Encapsulate what varies**

Encapsulate what varies

- dealing with *which* concrete class is instantiated is complicating your method and preventing it from being closed for modification.
- But now that we know what is varying and what isn't, let's encapsulate it.

```
class SimpleFactory{ ……. //make this singleton
  public Document createDoc(Type t) {
    if(t.equals(plain))  return new plainDoc();
    else if ((t.equals(styled)) return new styledDoc();}
}
class TextEditor{
 private Document doc;
 public void loadDocument(Type t) {
    doc=SimpleFactory.getInstance().createDoc(t);
        //delegate creation logic
    //do some adjustments on the doc
 }…
```



- take the creation code and move it out into another object that is only going to be concerned with creating **products**
  - **Single responsibility**
  - Your Editor class has other jobs than creating objects saveDoc, reopenDoc,newDoc…

How is this an Improvement?

- ***It looks like we are just pushing the problem off to another object.***
- *Yes.*
- But…when any other method needs the object creation, they can delegate to this creator
- Isolating the change of product
  - encapsulating the document creating in one class → there is only one place to make modifications

JAVA

```
class SimpleFactory{
 public Document createDoc(Type t){
     if(t.equals(plain))
           return new plainDoc();
     else if ((t.equals(styled))
           return new styledDoc();
  } /…
}
class TextEditor{
 private Document doc;…
 public void loadDocument(Type t){
   doc=SimpleFactory.getInstance().
           createDocument(t);
   //…do some adjustments on the doc
 }
 //other methods
}
```

- This is a **SimpleFactory**. It is **not** a design pattern
  - more like an idiom
- Factory method is the design pattern
  - Yet to be defined

C++

```
class SimpleFactory{
 public:
  Document* createDoc(Type t){
     if(t.equals(plain))
           return new plainDoc();
     else if ((t.equals(styled))
           return new styledDoc();
  }
}
class TextEditor{
 private: Document* doc;…
 public: void loadDocument(Type t){
    doc=SimpleFactory.getInstance()->
           createDocument(t);
    //…do some adjustments on the doc
}
 //other methods
}
```

- This is a **SimpleFactory**. It is **not** a design pattern
  - more like an idiom
- Factory method is the design pattern
  - Yet to be defined

*Should have used smart pointers, but kept like this for parallelism with the Java code

Question

```
class SimpleFactory{ ……. //make this singleton
  public Document createDoc(Type t) {
    if(t.equals(plain))  return new plainDoc();
    else if ((t.equals(styled)) return new styledDoc();}
}
```

- Why not a static create method?
- Then *you can't subclass and change the behavior of the create method.*

- *NO SIMPLE FACTORY IN HWs and EXAMs (except flyweight factory)*

Some adjustments…

```
class TextEditor{
 private Document doc;
 public void loadDocument(Type t){
   doc= createDocument(t);
   //..do some adjustments on the doc
 }
 public Document createDocument(Type t){
     return SimpleFactory.getInstance(). createDocument(t);
 }
 …
 public void openDoc(){…} //and other file management ops
 public void export(){…} //and other print and export ops
 //…formatting ops delegating to doc using its interface
}
```

Extending the Editor – OCP?

- Now, I want to have Text Editors for other kinds of environments
  - Such as an editor for PDF documents
- I want to get all the TextEditor functionality for free.
  - the editor functionalities care about the Document interface
    - doc.open() ; doc.save() ; doc.find() ….
  - File management, recovery, print .. functionalities

Editor Framework

- I want to create a framework that ties the editor and the document creation together, *yet* still allows things to remain flexible.

- How to localize all the document-loading activities to the TextEditor class, and yet give **the other editors to have their own style of documents?**

- Remember, the TextEditor already has a system (algorithm) in the `loadDocument()` method, and you want to ensure that it is consistent across all editors

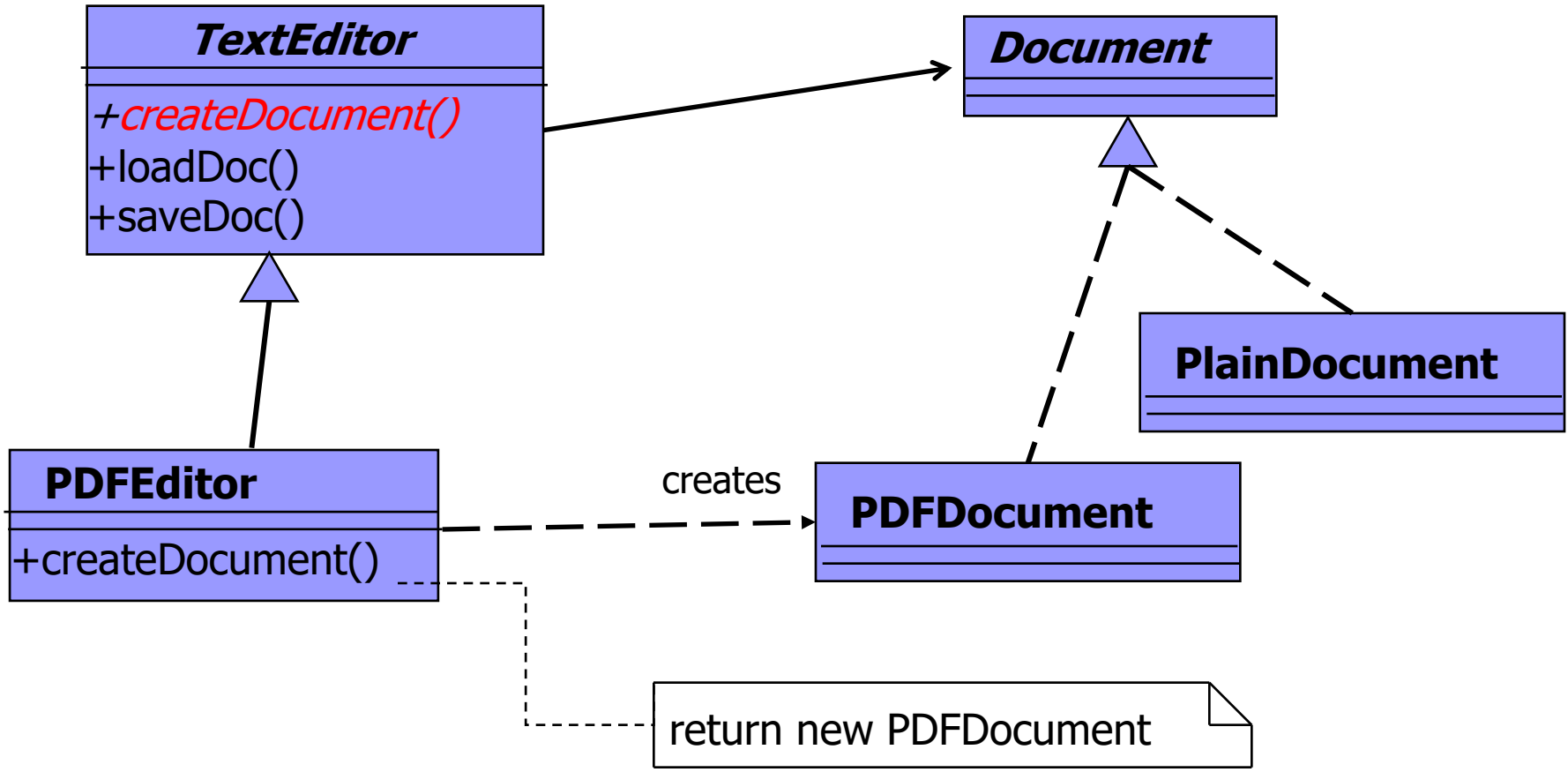
Editor Framework

```
class TextEditor{
 private Document doc;
 public void loadDocument(Type t){
   doc= createDocument(t);
   //..do adjustments on the doc
 }
 public abstract Document createDocument(Type t);
}
```

TextEditor has an **algorithm** here and we want to ensure that it's consistent across all editors

- All the variations need to do is subclass TextEditor and supply a createDocument() method that creates their style of Document
  - Subclass of Editor creates a Subclass of Document they want to work with.

Class Diagram of the Framework



PDFEditor is consistent with TextEditor

Collaborations

- The framework provides one service "`create document`".
- The framework invokes the `createDocument()` factory method to create a document that it can prepare using a well defined, consistent process.
- A "user" of the framework will subclass this class and provide an implementation of the `createDocument()` method.
- Any dependencies on concrete "product" classes are encapsulated in the subclass.
  - Product here is the Document

Factory Method

- Intent: Define an interface, but let the subclasses decide which class to instantiate

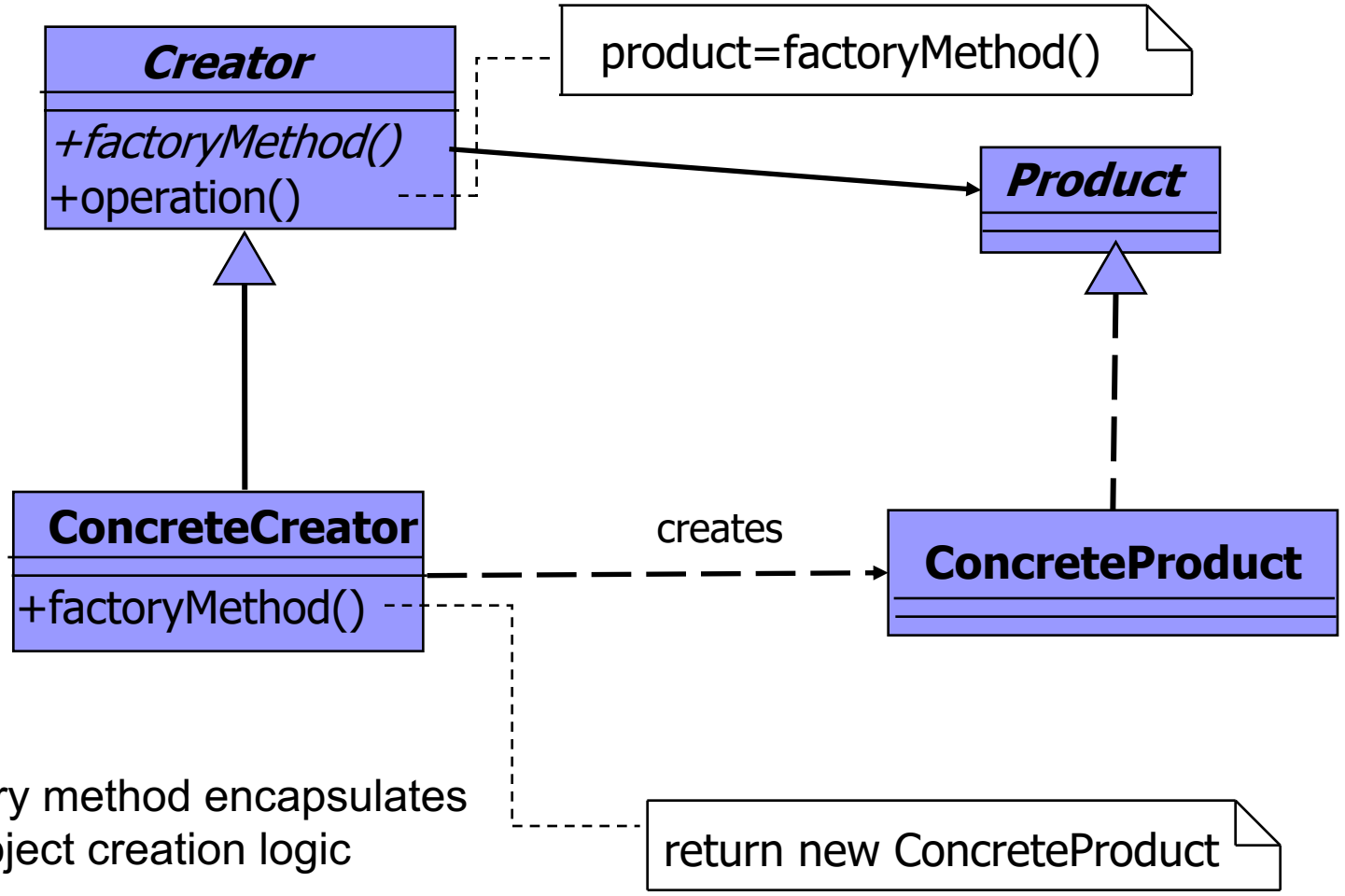
- the Factory Method pattern is used with **application frameworks** or highly reusable and modular components
  - e.g., Plug-in based systems
  - e.g., Logger frameworks

Factory Method

- Intent: Define an interface, but let the subclasses decide which class to instantiate
- A.k.a. Virtual Constructor
- Applicability:
  - A class can't anticipate the kind of objects to create.
    - Do not know beforehand which specific subclasses need to be instantiated
  - A class delegates its work to a helper class, and you want localize the information about which class the work is delegated to

Factory Method - Structure

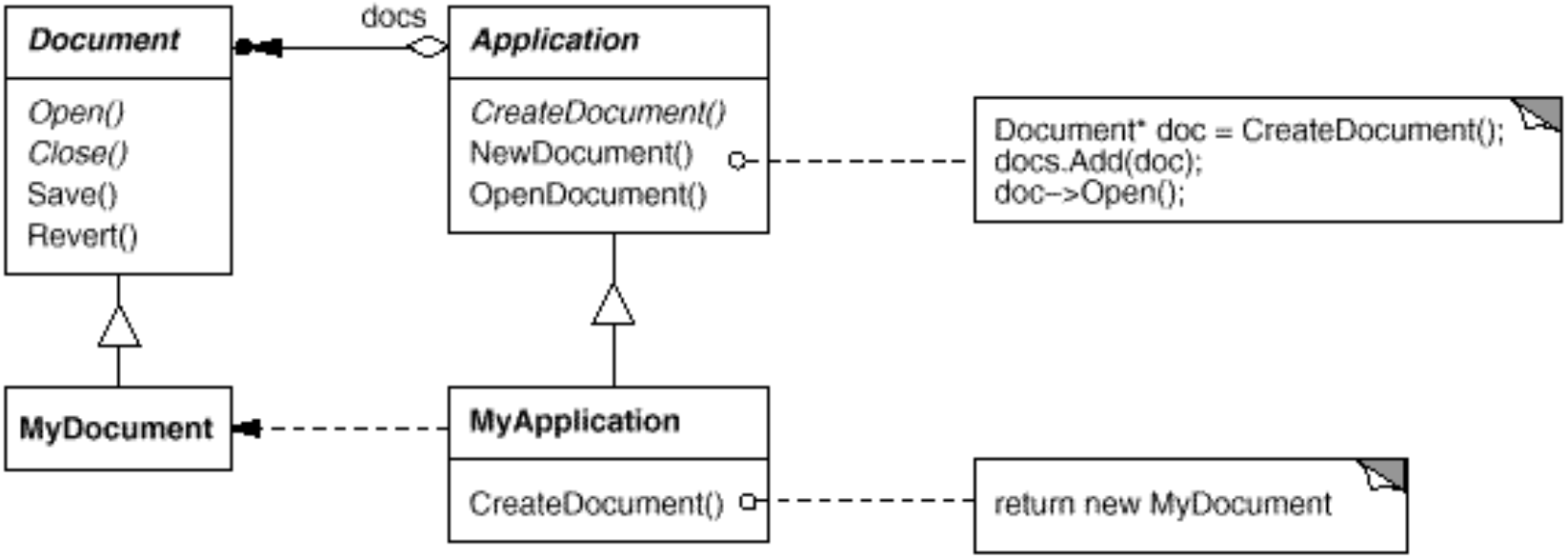
Participants?



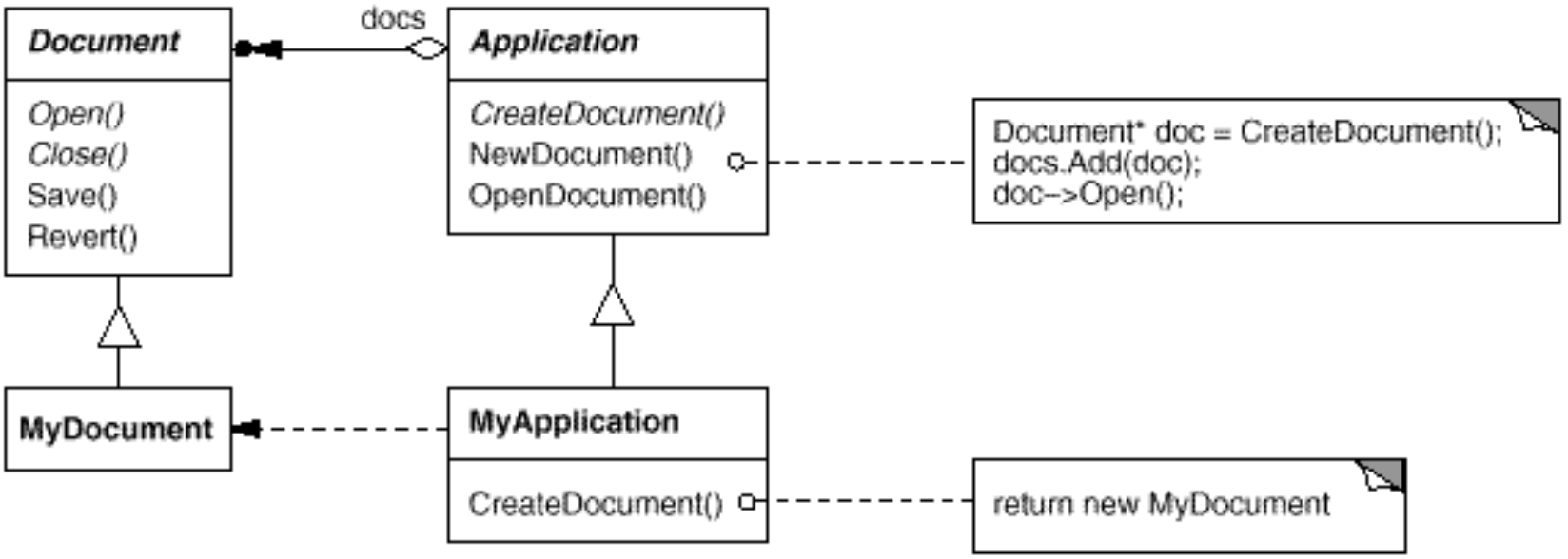
Factory method encapsulates the object creation logic

GoF example: Scenario

- We implement a customizable framework that opens, saves, closes a document with menus, shortcuts.
- App knows **only when** a new document should be created but **does not know what kind** of Document to create.
- App **cannot predict what subclasses** of Document will be



GoF example



- Which class plays which role?
- Anyone can customize this framework. All the functionalities in the Application will come for free.
  - How do you customize it?

Notes

1. Product creation is not the primary responsibility of the Creator.
   - Creator class already has some core business logic related to products.
     - e.g., open, close documents
   - The factory method helps to decouple this logic from the concrete product classes.

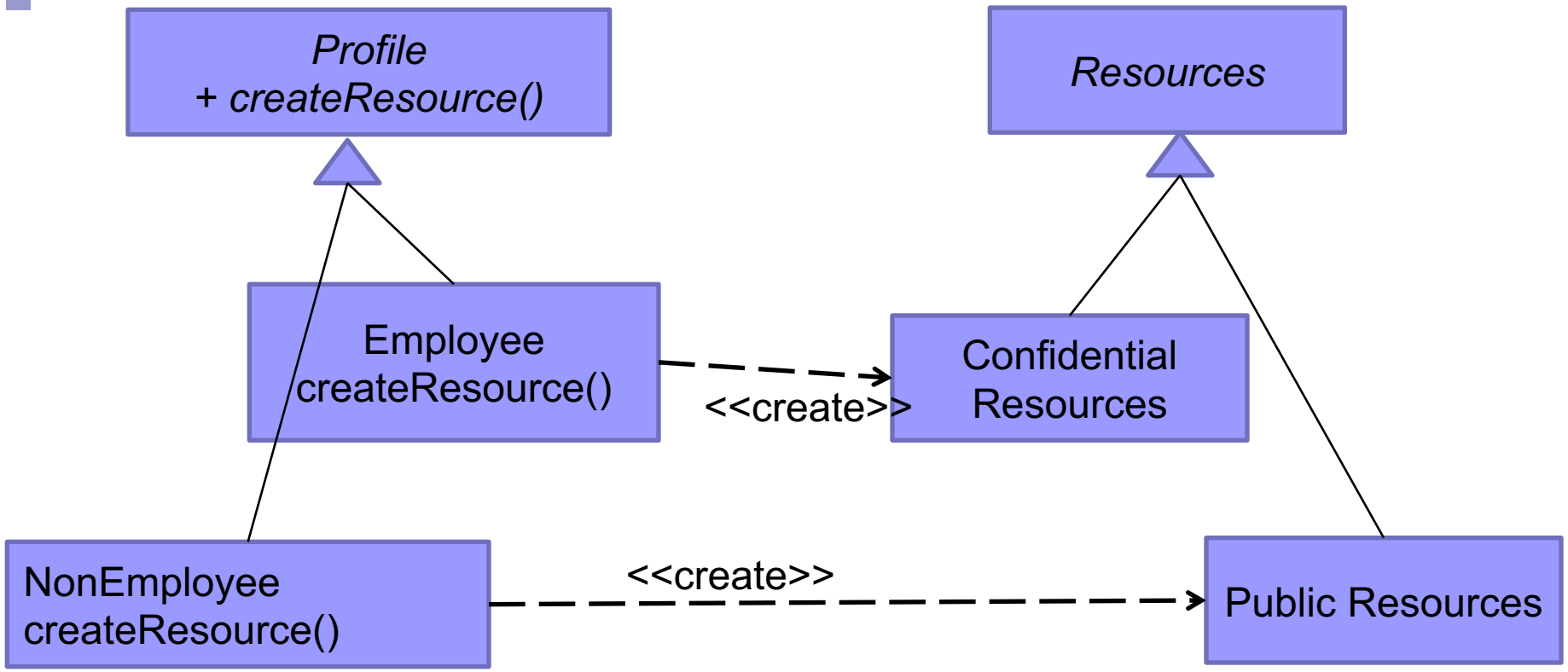
2. It is easier to extend the product construction code independently from the rest of the code.
   - The Factory Method separates product construction code from the code that actually uses the product.

https://refactoring.guru/design-patterns/factory-method

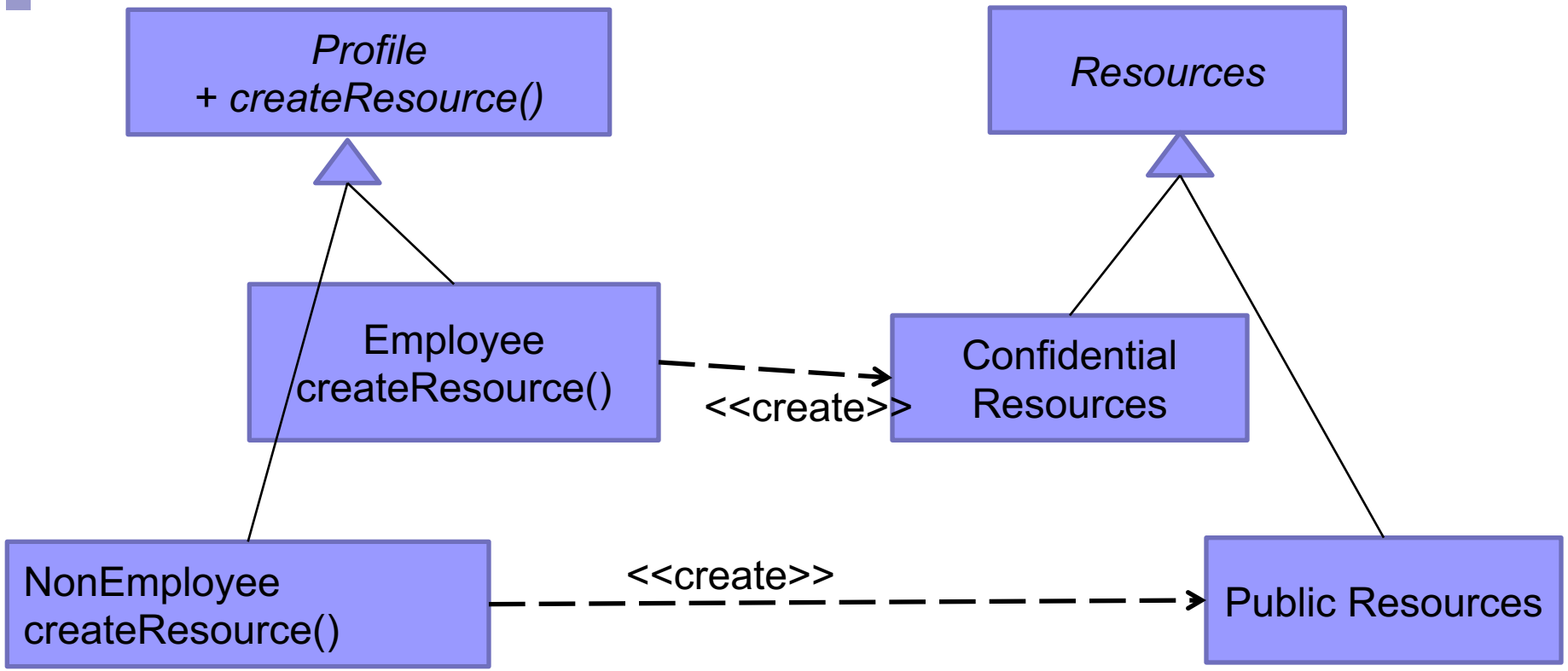
Example: Profiles and Resources

- My application has profile management to store information about its users.
- I want to provide individual or multiple users create and access to resources (e.g. files) based on their profiles
  - Other than this, my app treats resources uniformly
- Q:Ensure information control, by providing *Employees* create *Confidential Resources*, while restricting the generic *public* to only *Public Resources.*

- MyApp should only know it is dealing with Resources.
  - Decouple from of Public and Confidential resources.
- When creating a resource, MyApp wants to defer the resource type to user profile type.
- MyApp does not know which specific subclasses need to be instantiated when getResource() called.



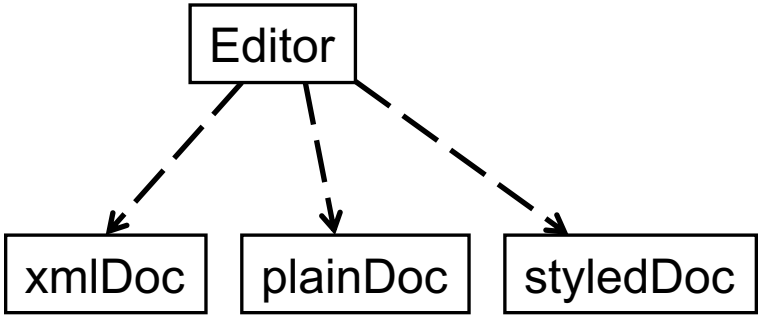
- MyApp should only know it is dealing with Resources.
  - Decouple from of Public and Confidential resources.
- When creating a resource, MyApp wants to defer the resource type to user type.
- MyApp does not know which specific subclasses need to be instantiated when getResource() called.

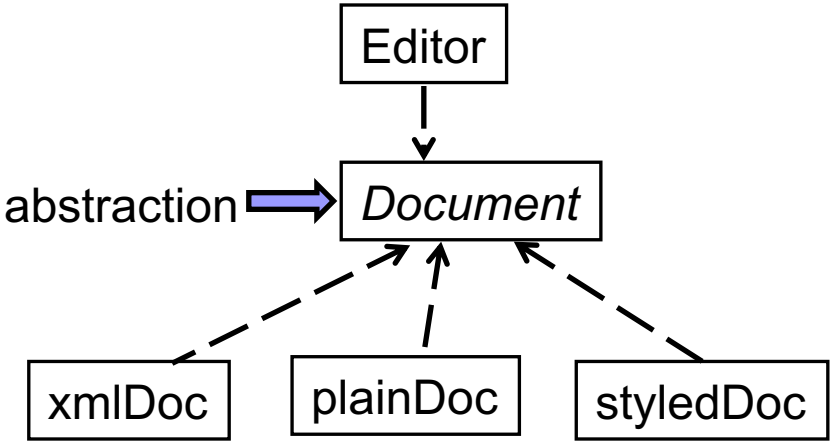


- Factory Method pattern creates parallel hierarchies

Dependency Inversion

1. Depend upon abstraction.
2. Do not depend upon concrete classes.





- Factory method enables dependency inversion.
  - Editor uses `createDocument()`, depends only to Document interface
  - Subclasses implement (i.e.) depend on the Document interface

Factory Method-Consequences

- Eliminates the need to bind application specific classes into your code
  - Code only deals with the Product
- Hides complex creation logic
  - Reading from DB, building a composite, creating an object with several decorations..etc
- Provide hooks for subclasses
  - This is the method to override for versions of the product
- Connects parallel class hierarchies
  - Parallel class hierarchies often require objects from one hierarchy to create appropriate objects from another
  - hides the secret of which classes belong together

Implementation

- Two major varieties
  - creator class is abstract
    - *requires* subclass to implement
  - creator class is concrete, and provides a default implementation
    - *optionally allows* subclass to re-implement

```
abstract class DocumentEditor {
    public void open() {
        Document doc = createDocument(); //FM
        doc.load();
    }
    protected abstract Document createDocume
}
```

```
class DocumentEditor {
    public void open() {
        Document doc = createDocument(); //FM
        doc.load();
    }
    protected Document createDocument(){
        return new PlainDocument();}
}
```

Implementation issues -1

- Parameterized factory methods
  - takes a class id (or anything) as a parameter to a generic create() method.
    - Conditional to select a type and then return new object of that type
    - When some product types are anticipated
    - Subclasses override this FM for not anticipated ones
    - May use reflection but risky –instantiation using class name
    - If too generic, then NOT type safe – needs casting

```
public Object create(String type) {
    if (type.equals("pdf"))   return new PDFDocument();
    else if (type.equals("word")) return new WordDocument();
    else  return null;
}
```

Implementation issues -2

- Can use templates ***instead*** of inheritance

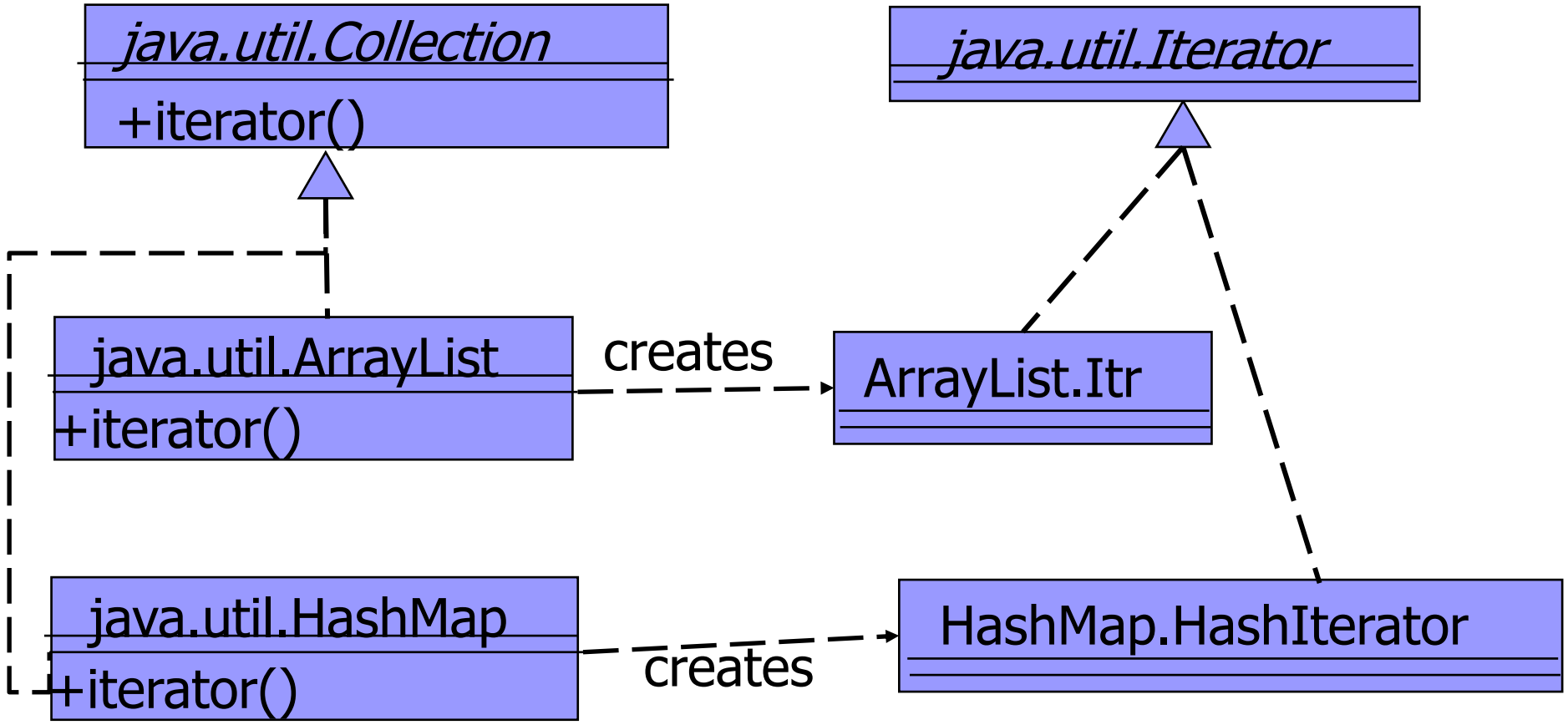
```
class Creator {
    public:   virtual Product* createProduct() = 0;
};
template <class TheProduct>
class StandardCreator: public Creator {
    public:
        virtual Product* createProduct();
};
template <class TheProduct>
Product* StandardCreator<TheProduct>::createProduct () {
        return new TheProduct;
}
/*Client*/ StandardCreator<MyProduct> myCreator;
Product* p=myCreator.createProduct();
```

Implementation issues -3

- DO NOT call factory method in the Creator's constructor.
  - Factory methods in C++ are always virtual functions and are often pure virtual
  - The factory method in the ConcreteCreator won't be available yet.
  - do **Lazy Initialization**
    - Constructor initializes the Product to 0 (null)
    - Product will be created when getProduct() is called

```
Product* Creator::getProduct () {
 if (_product == 0){
     _product = createProduct()
  } return _product;
}
```

Known uses: Iterators in Java



Similarly, **begin()** and **end()** are factory methods in **C++ STL containers**

Related patterns

- All the creational patterns use factory method
  - Singleton::getInstance() is a factory method
- Iterators are created using factory method
  - ArrayList.iterator() returns and iterator that is specialized to traversing an array list
- Template method (later)

Family of Products

	Pawn	King	Knight
LOTR			
Wood			

- We need a way to create individual chess piece objects so that they match others in the same family.
- App looks not so good when players see non-matching chess pieces

Family of Products

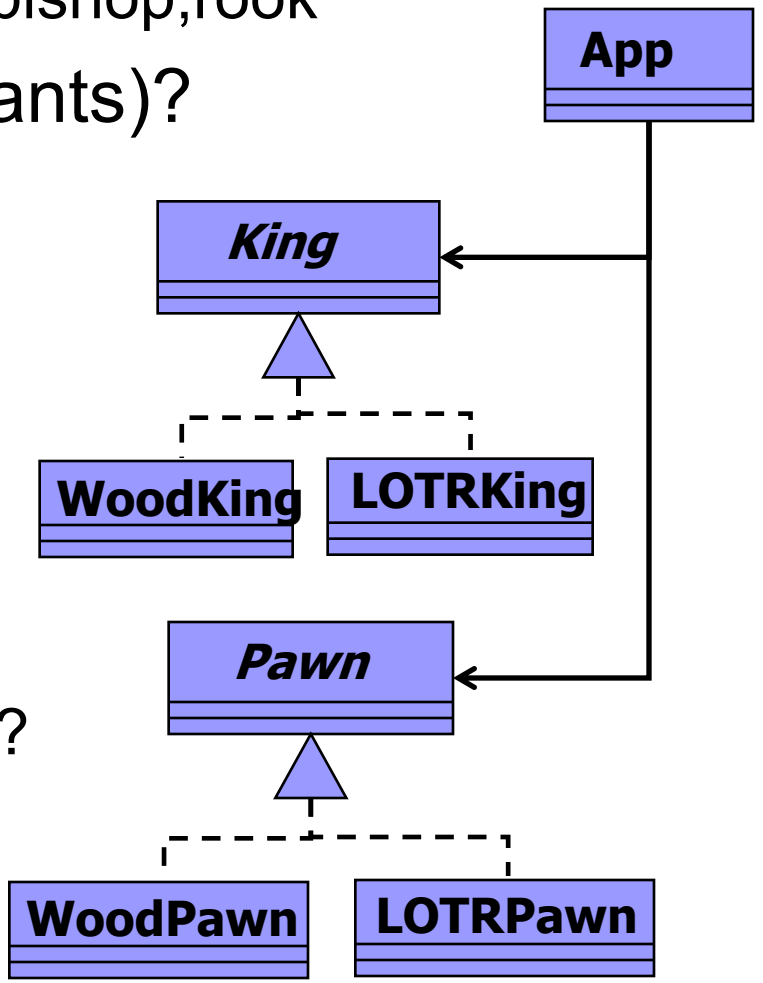
- App only cares there are Kings, pawns etc for the game play
- Objects need to be created as a set to be compatible.
- The application should be configured with one of multiple families of products

wood king, wood pawn, wood knight

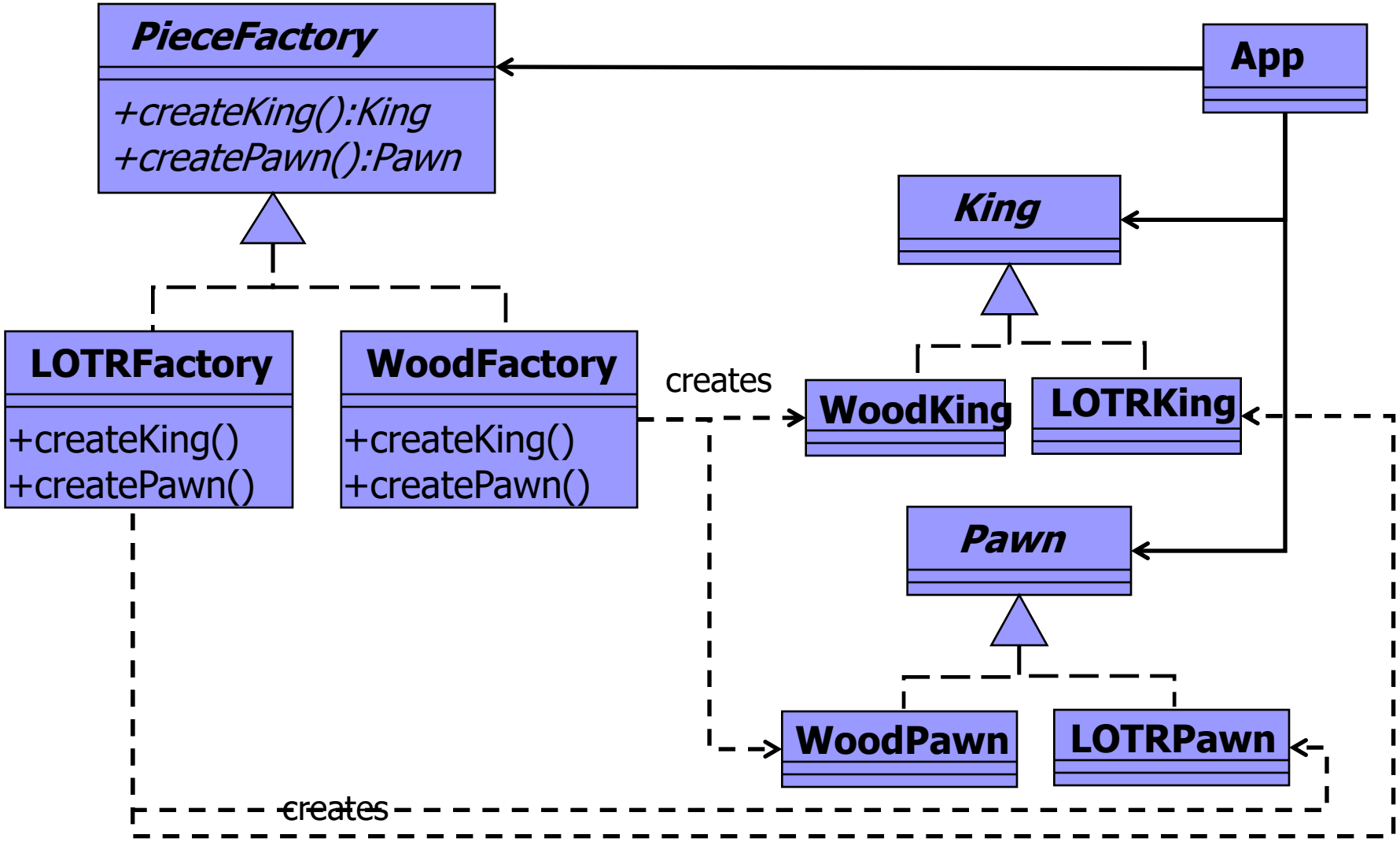
Lord of the Rings chess set
Lotr king, lotr pawn, lotr knight

Families and Interfaces

- What are the Abstract Products
  - Queen, King, pawn, knight, bishop,rook
- What are the families (variants)?
- How could I make the App only depend on interfaces?
- Have a class for Creation
  - What would be the methods?
  - Where would you put the variations of these methods?



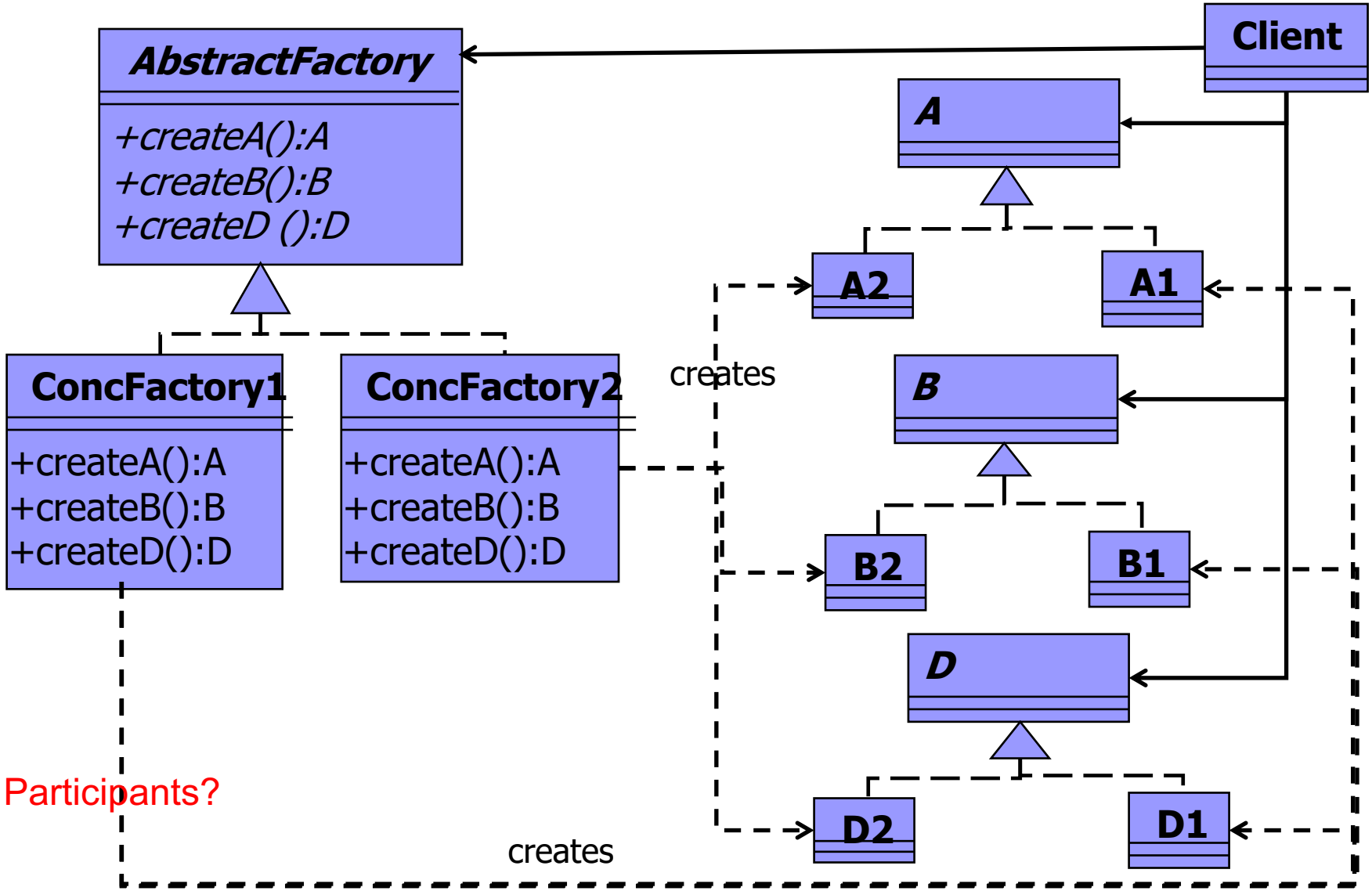
Chess Pieces Example



Abstract Factory

- **Intent**: Provide an interface for creating families of related or dependent objects without specifying their concrete classes
  - Create family of compatible products
- Applicability
  - System needs to be configured with one of multiple family (variant) of products
  - Enforce the constraint that a family of related products is designed to be used together
  - Want to provide a library but reveal just their interfaces
  - System needs to be independent of how its product are created and represented

Abstract Factory-Structure



Enforcing Family

- The Abstract Factory provides an interface for creating objects from each class of the product family.
- As long as your code creates objects via this interface, you don't have to worry about creating the wrong variant of a product which doesn't match the products already created by your app.
- Pass another Concrete Factory, the program will create objects of another family
  - Pluggable and configurable

https://refactoring.guru/design-patterns/abstract-factory

Abstract Factory-Consequences

- Promotes consistency among products
  - Application uses a product family.
  - All products from a factory are compatible with each other
- Makes exchanging product families easy
  - Change the concrete factory, whole product family changes at once
- Isolates clients from the implementation classes
  - Product names do not appear in clients, only interfaces are known to clients
- Extensible in terms of new variants
- Supporting new kinds of products is difficult
  - Fixed set of products in the interface, need to change the AbstractFactory interface

Exercise

- A game where players interact with obstacles.
  - For young children, kitties run around and face puzzles to solve
  - For teenagers, fighters wander around and battle with monsters
- A simple simulation
  - Player (kitty or fighter) interacts with an obstacle (puzzle or monster)

Player (kitty or fighter) interacts with an obstacle (puzzle or monster)

- Product families:
- Factories:

Exercise: Factories (Java)

```
// The Abstract Factory
public interface GameElementFactory {
    public Player makePlayer();
    public Obstacle makeObstacle();
}

// Concrete factories:
class KittiesPuzzlesFactory implements GameElementFactory {
     public Player makePlayer() { return new Kitty(); }
     public Obstacle makeObstacle() { return new Puzzle(); }
}

class MonsterFighterFactory implements GameElementFactory {
     public Player makePlayer() { return new Fighter(); }
     public Obstacle makeObstacle() { return new Monster();
   }
}
```

Exercise: Factories (C++)

```
// The Abstract Factory
class GameElementFactory {
   public:
   virtual Player*  makePlayer()=0;
   virtual Obstacle* makeObstacle()=0;
} ;
// Concrete factories:
class KittiesPuzzlesFactory: public GameElementFactory {
   public:
   virtual Player* makePlayer() { return new Kitty(); }
   virtual Obstacle* makeObstacle() { return new Puzzle();
   }
};
class MonsterFighterFactory:public GameElementFactory {
   public:
   virtual Player* makePlayer() { return new Fighter(); }
   virtual Obstacle* makeObstacle() { return new Monster();
   }
}
```

Exercise: Client code (Java)

```
class Game {
  private Player p;
  private List<Obstacle> obstacles=new ArrayList<Obstacle>();

  public Game( GameElementFactory factory, int num) {
    p = factory.makePlayer();
    for(int i=0;i<num;i++)
       obstacles.add(factory.makeObstacle());
  }
  public void play() {
     while(true) {
        p.wander();
        Obstacle ob=p.collided();
        if(ob!=null) p.interactWith(ob);
  }
}
```

This is a very oversimplified game loop, but focus on the factories

Exercise: Client code (C++)

```
class Game {
 private:
  Player* p;
  std::list<Obstacle*> obstacles;
 public:
  Game( GameElementFactory factory, int num) {
    p = factory.makePlayer();
    for(int i=0;i<num;i++)
       obstacles.push_back(factory.makeObstacle());
  }
  public void play() {
     while(true) {
        p.wander();
        Obstacle ob=p.collided();
        if(ob!=null) p.interactWith(ob);
  }
}
```

This is a very oversimplified game loop, but focus on the factories

Exercise: setting up

How to setup the Game? //just testing

```
public static void main(String args[]){
 GameElementFactory kpuz = new KittiesPuzzlesFactory();
 GameElementFactory mons = new MonsterFighterFactory();

    Game g1 = new Game(kpuz);
    Game g2 = new Game(mons);
}
```

- "Exchanging product families is easy" –consequence#2
- Usually, lookup environment or configuration to select a concrete factory.

Dependency Injection

- A dependency is an object that can be used (a service).
- An injection is the passing of a dependency to a dependent object (a client) that would use it.

```
public Game( GameElementFactory factory) {
    p = factory.makePlayer();
    for(int i=0;i<num;i++)
       obstacles.add(factory.makeObstacle());
  }
```

- Helps testing, refactoring, extending

Abstract Factory Helps testing

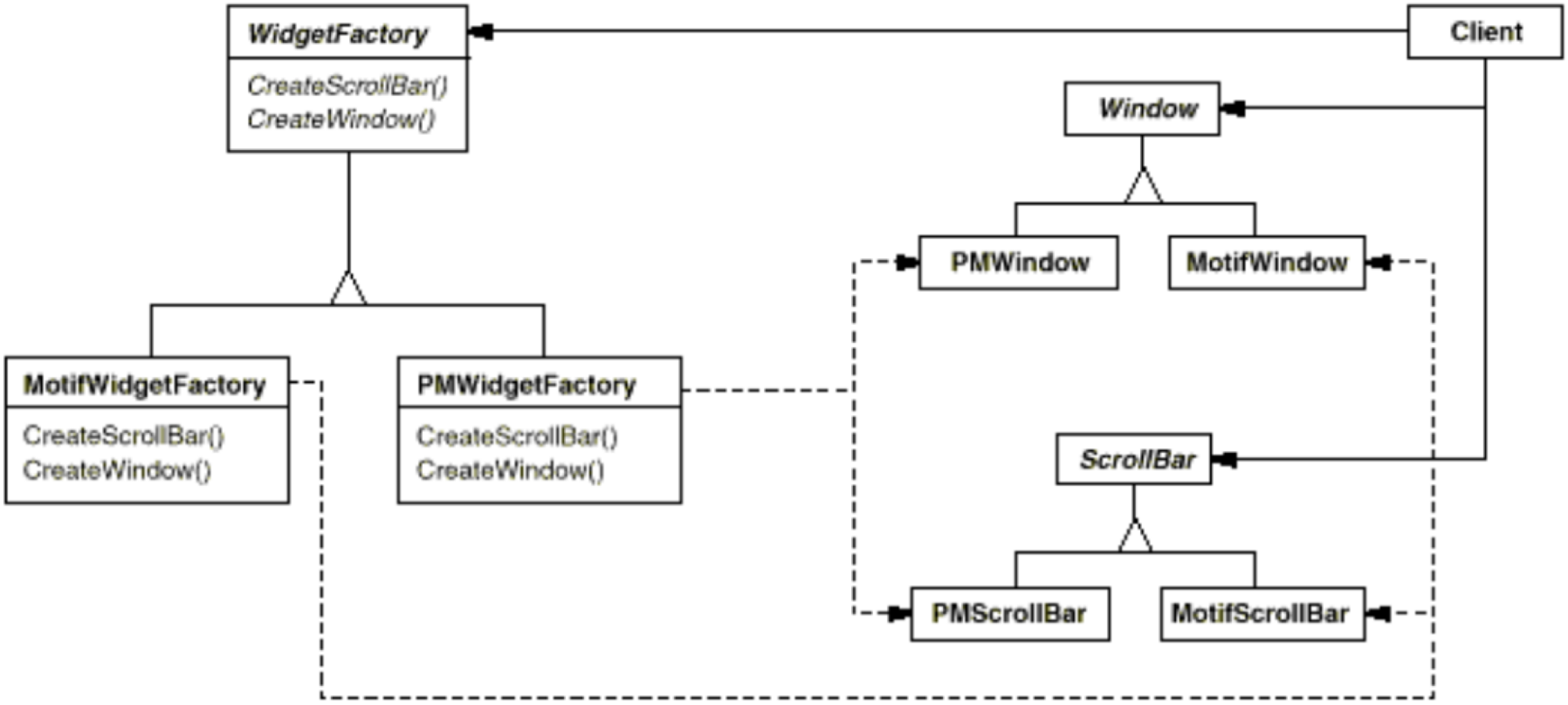
- Abstract Factory can simplify testing by helping unit isolation
- Implement a TestConcreteFactory and TestConcreteProduct
  - They can simulate the expected resource behavior.
  - Mocks and Stubs

What about platform independence?

Applicability: when to use?

- The application should be configured with one of multiple families of products.
- Objects need to be created as a set to be compatible with each other.
- You want to provide a collection of classes and you want to reveal just their contracts and their relationships, not their implementations.
- The client should be independent of how the products are created.

Platform independence



- "user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager." GoF
- Same in platform independence: WinFactory, MacFactory

Platform independence

- When an application launches, it checks the type of the operating system.
  - System.getProperty("os.name"); #ifdef _WIN64 #elif __MACH__
- The app uses this information to create a factory object matching the operating system.
  - WinFactory or MacFactory
- The rest of the code uses this factory to create UI elements whenever needed.
- Result: prevent from creating the wrong kind of UI elements.
- No need to modify the code each time we add a new variation of UI elements to the app.

Implementation issues-1

"Supporting new kinds of products is difficult"

– can we define extensible factories?

- Have a parameter to make method to select what product to create
  - Implementation is similar to parameterized Factory Method
  - Only when all objects have the same abstract base class or when the product objects can be safely coerced to the correct type by the client that requested them.
  - Not type safe, due to casting
  - I do not recommend

Implementation issues-2

- Abstract Factory
  - Make it java interface, i.e. pure virtual methods
    - Frequently used
  - Have default implementation for the create methods
    - If the variants (families) share some concrete product type, then less overriding

Known uses

- Graphic libraries like OpenGL for platform independent rendering
  - OpenGLFactory creating OpenGLShader, OpenGLContext, OpenGLBuffer as products
  - Example concrete products WindowsOpenGLContext,LinuxOpenGLContext, MacOSOpenGLContext are
- Libraries for database connections e.g. .Net
  - System.Data.Common.DbProviderFactory implemented by
  - System.Data.OleDb.OleDbFactory
  - System.Data.OracleClient.OracleClientFactory
  - System.Data.SqlClient.SqlClientFactory
    - https://learn.microsoft.com/en-us/dotnet/api/system.data.common.dbproviderfactory?view=net-8.0

Abstract Factory OR Factory Method?

- **Factory Method** is for **single product creation**, while **Abstract Factory** is for **families of related products**
- Use abstract factory when you have families of products you need to create
  - to make sure your users create products that belong together
- Use factory method when you want parallel hierarchies, or you don't know all the concrete classes ahead

Read AF vs FM in the Head First Design patterns book

Abstract Factory OR Factory Method?

- Factory method: A class pattern
  - Inheritance:
    - Abstract type has a factory method
    - Subclasses (concrete class) overrides the factory method
- Abstract factory: An object pattern
  - Delegation
    - Abstract type for creating a family of products
    - the responsibility of object instantiation to another object

AF-Related Patterns

- AF consists of **Factory Methods**
  - ConcreteFactory may use **Prototype**s instead
- Factories could be **Singletons**
  - Might be an overkill
- AF together with **Bridge** is useful when some abstractions of Bridge can only work with some specific implementors.
  - AF encapsulates and hides this relation from clients
  - java.awt.Toolkit creates native peers for GUI components
- **Builder** (next pattern)

Thread safety

- We discussed thread safe singleton
- All the rest of the creational patterns need thread safety as well
  - depending on how instances are shared or reused like Flyweight
  - FM, AF, Builder, Prototype
  - Use locking or monitors for thread safety

Creational Patterns

- Common reason for using a creational pattern
  - need to change the class that's instantiated to fit the situation
    - a constructor in a single class is an inadequate method to return instances of possibly different classes.
    - Almost always the objects returned are instances of some common superclass.
- Class creational patterns use *inheritance* to vary the object being created.
  - only Factory Method is class creational
- Object creational patterns generally delegate the actual construction to a different object that is responsible for deciding which class is required and invoking the necessary constructor.