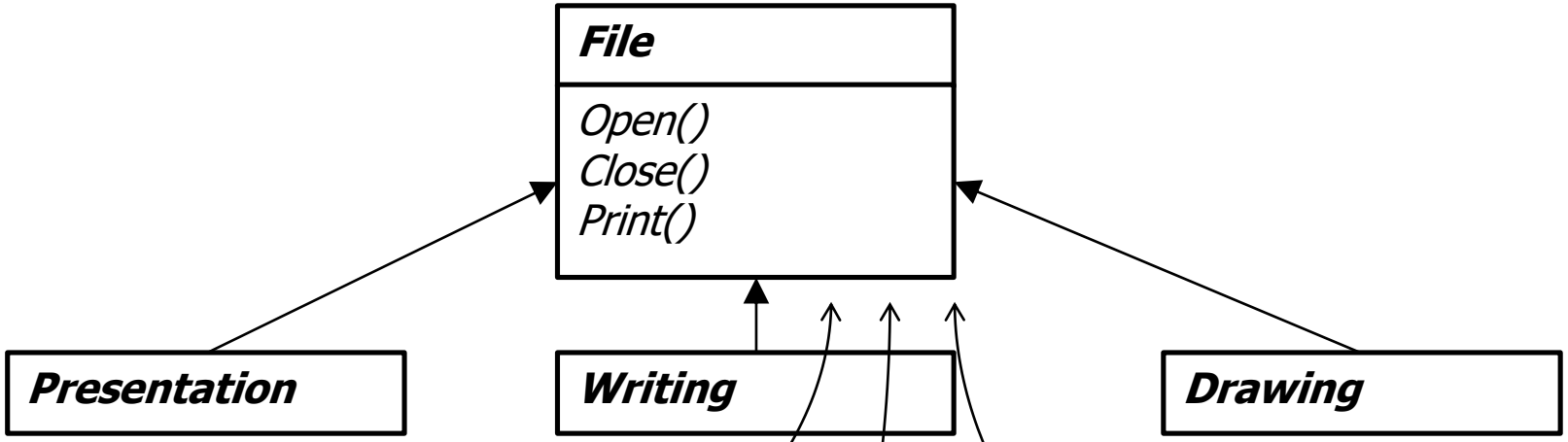
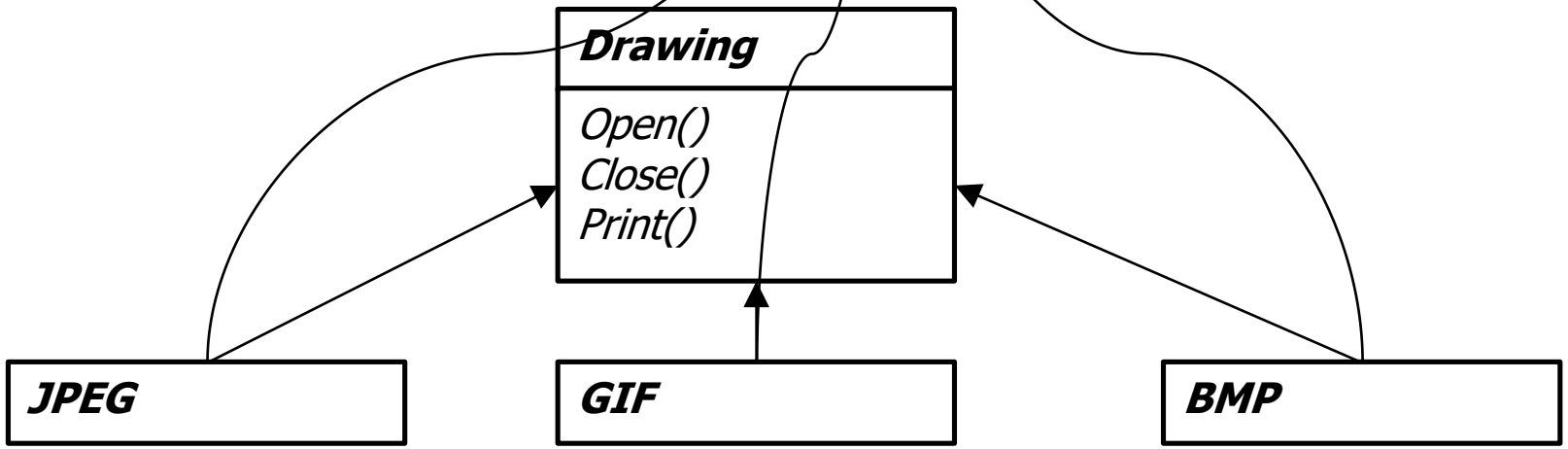
# Factory Method

# Motivace 1 – tisk

- Knihovna tiskne dokumenty do pdf




- Knihovna tiskne obrázky do pdf



# Motivace 1 – tisk

- **Jak implementovat metodu printToPDF?**

```
public class Printer {
    public File printToPDF(String path) {
        File file;

        file = new Presentation; // Writing or Drawing?

        file.Load();
        file.Print();
        file.Close();
        return file;
    }
}
```

Jak získáme tu správnou třídu k instanciování?

# Rychlé řešení

- **Rozhodujeme se na základě přípony souboru**

```
public class Printer {
    public File printToPDF(String path) {
        File file;

        if(ext(path) == "ppt") file = new Presentation(path);
        else if(ext(path) == "doc") file = new Writing(path);
        else if(ext(path) == "drw") file = new Drawing(path);

        file.Open();
        file.Print();
        file.Close();
        return file;
    }
}
```

Potřebuje metoda znát potomky třídy File?

Co když přidáme novou třídu Excel?

- **Komplikace**
  - Přidání nového typu dokumentu (např. Excel)
  - Tisk obrázků místo dokumentů

# Factory Method – řešení

```
public abstract class Printer {
     public abstract File DoMakeFile(String path);
     public void printToPDF(String path) {
        File file = this.CreateFile(path);
        file.Load();
        file.Print();
        file.Close();
     }
}
```

- **Knihovna pro tisk dokumentů**

```
public class DocumentPrinter : Printer {
     public override File DoMakeFile(String path) {
          if(ext(path) == "ppt") return new Presentation(path);
          else if(ext(path) == "doc") return new Writing(path);
          return null;
     }
}
```

- **Knihovna pro tisk obrázků**

```
public class ImagePrinter : Printer {
     public override File DoMakeFile(String path) {
          if(ext(path) == "jpeg") return new Jpeg(path);
          else if(ext(path) == "gif") return new Gif(path);
          return null;
     }
}
```

# Motivace 2 – hra

- **Metoda generuje místnosti ve hře**

```
public class Level{
    public Room GenerateRoom() {
        Room room = new Room();
        room.add(new Door());
        int monsters = random(0,5);
        for(int i = 0; i < monsters; ++i) {
            room.add(new Monster());
        }
    }
}
```

- **Chceme přidat novou třídu AdvancedMonster**

# Factory Method – řešení

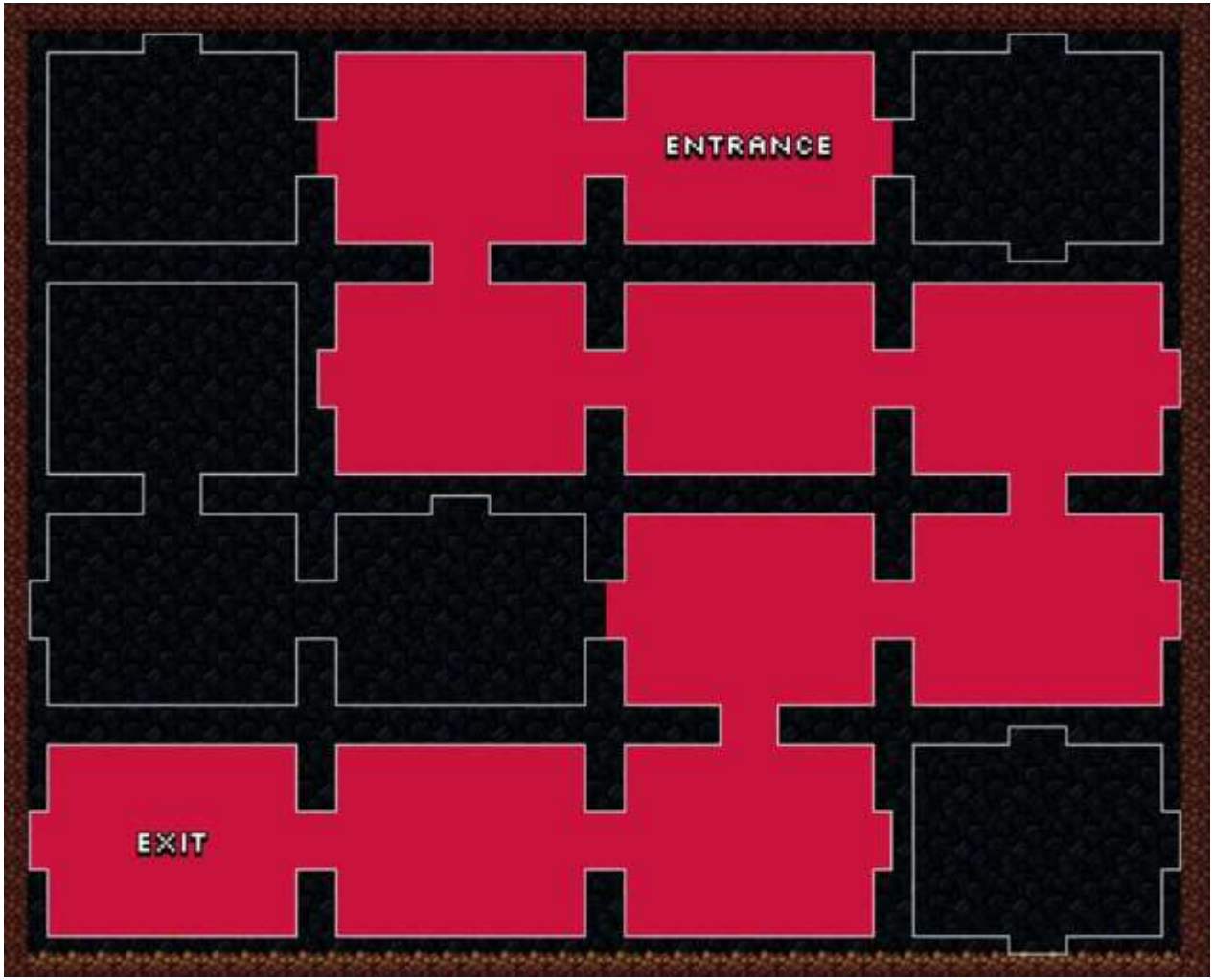
- **Delegujeme instanciování na dedikované metody**

```
public class Level {
    public virtual Room CreateRoom() {
        return new Room();
    }
    public virtual Door CreateDoor() {
        return new Door();
    }
    public virtual Monster CreateMonster() {
        return new Monster();
    }
    public Room GenerateRoom() {
        Room room = this.CreateRoom();
        room.add(this.CreateDoor());
        int monsters = random(0,5);
        for(int i = 0; i < monsters; ++i) {
            room.add(this.CreateMonster());
        }
        return room;
    }
}
```

```
public class CaveLevel : Level {
    public override Room CreateRoom() {
        return new CaveRoom();
    }
}
```

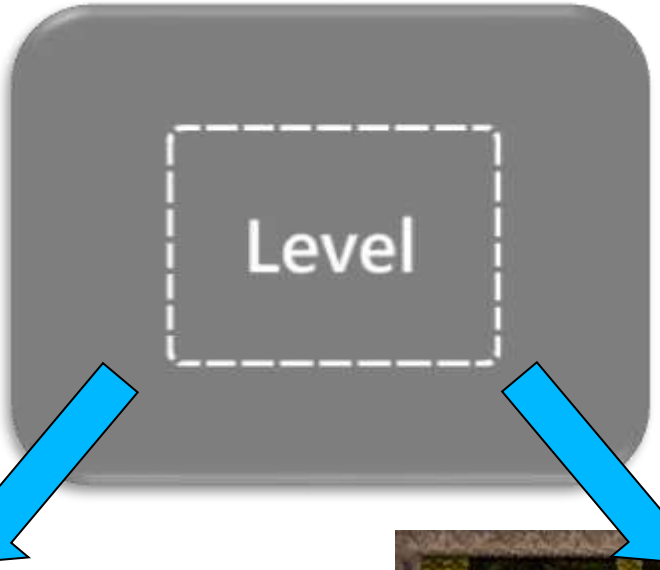
# Spelunky example

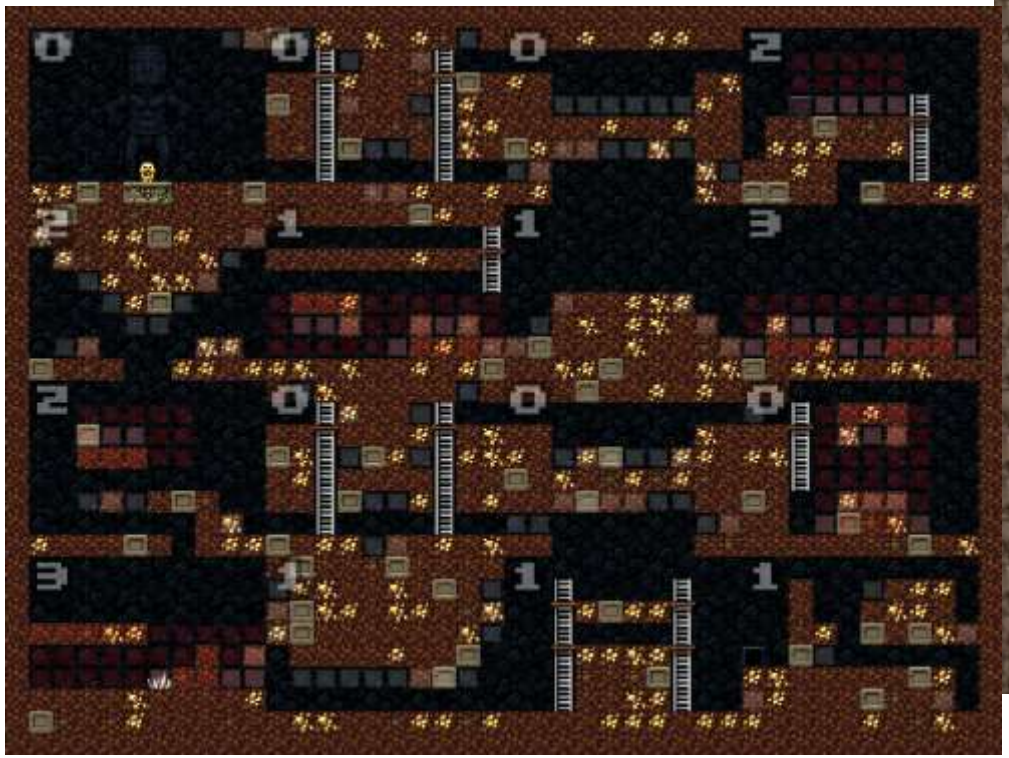
# Room Generation




Více o Spelunky generování místností zde

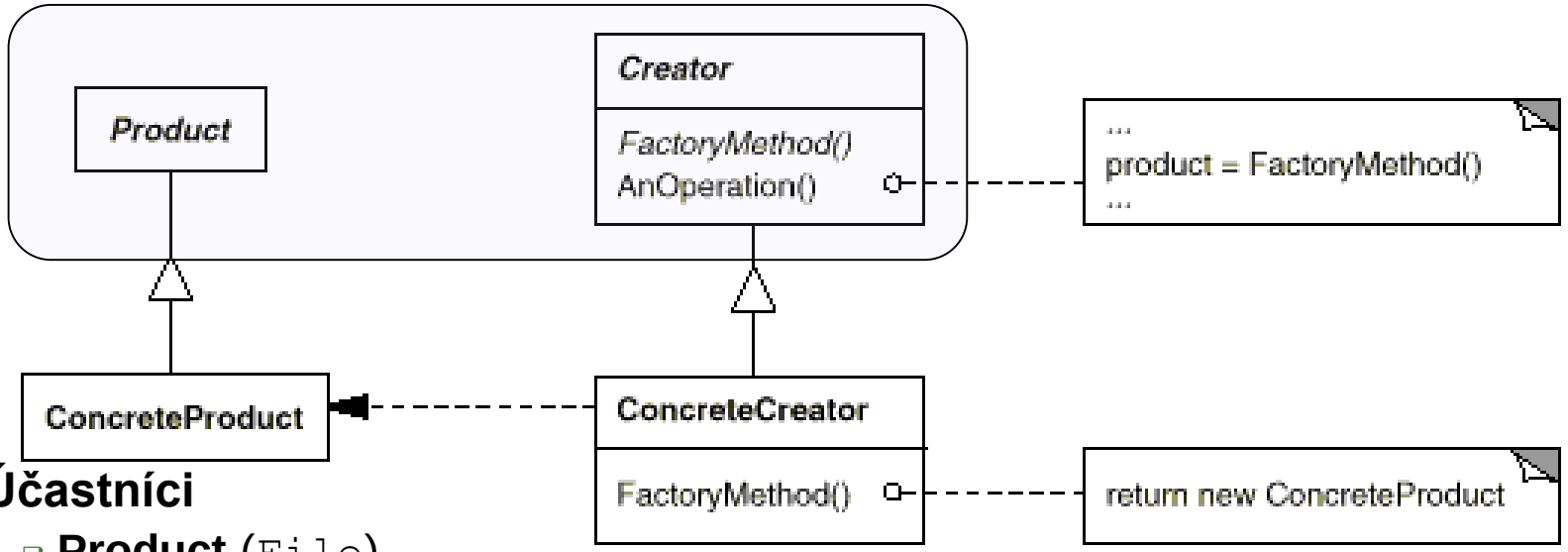
# Levels



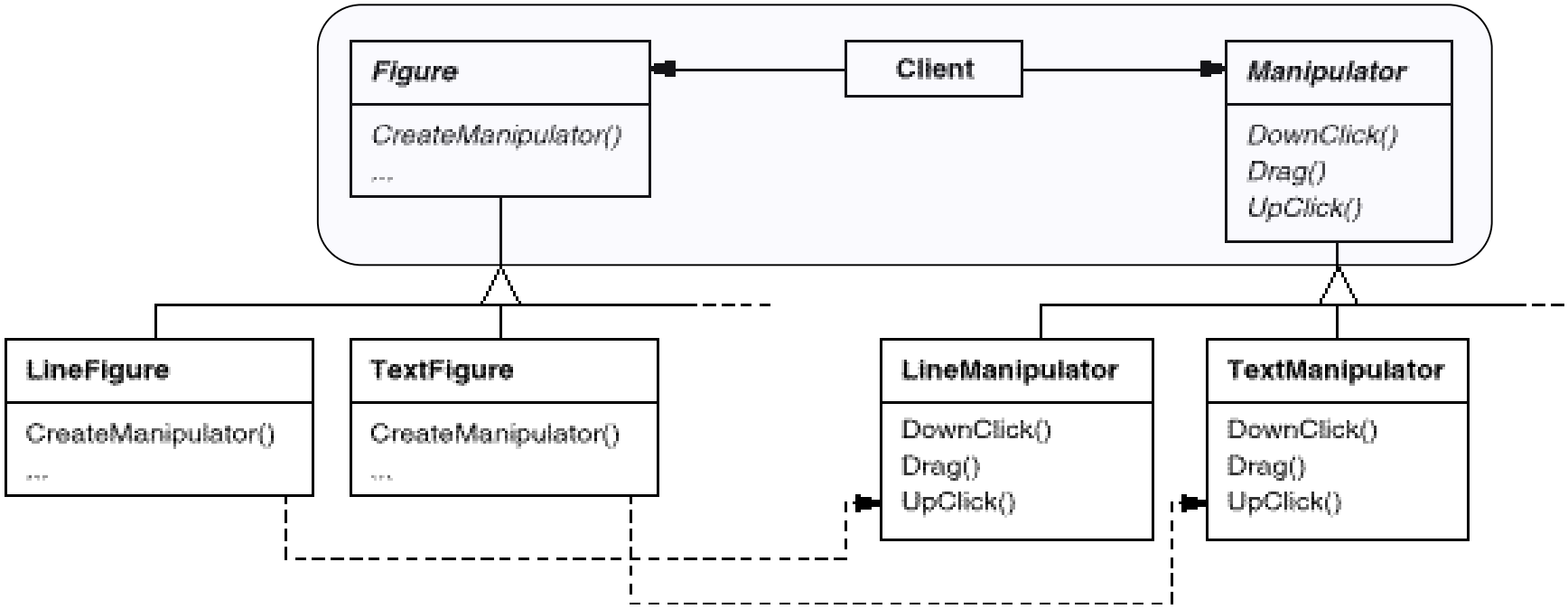
# Factory Method – struktura

- **Struktura**




- **Účastníci**
  - **Product** (`File`)
    - definuje rozhraní objektů vytvářených tovární metodou
  - **ConcreteProduct** (`Presentation`)
    - implementuje rozhraní Productu
  - **Creator** (`Printer`)
    - deklaruje tovární metodu vracející objekt typu Product
    - může definovat defaultní implementaci vracející defaultní objekt ConcreteProduct
  - **ConcreteCreator** (`DocumentPrinter`)
    - implementuje tovární metodu vracející instanci ConcreteProductu

# Factory Method – paralelní hierarchie




- **Propojení paralelních hierarchií tříd**
  - třída deleguje některé akce na jinou třídu
  - příklad: manipulovatelné grafické objekty (úsečka, obdélník, ...)

# Factory Method – implementace

- **Implementace**
  - **dvě hlavní varianty**
    - `Printer` je abstraktní třída – nutnost dědičnosti a vlastní implementace
    - `Printer` je standardní třída – defaultní implementace

# Factory Method – řešení

```
public abstract class Printer {
     public abstract File DoMakeFile(String path);
     public void printToPDF(String path) {
        File file = this.CreateFile(path);
        file.Load();
        file.Print();
        file.Close();
     }
}
```

- **Knihovna pro tisk dokumentů**

```
public class DocumentPrinter : Printer {
     public override File DoMakeFile(String path) {
          if(ext(path) == "ppt") return new Presentation(path);
          else if(ext(path) == "doc") return new Writing(path);
          return null;
     }
}
```

- **Knihovna pro tisk obrázků**

```
public class ImagePrinter : Printer {
     public override File DoMakeFile(String path) {
          if(ext(path) == "jpeg") return new Jpeg(path);
          else if(ext(path) == "gif") return new Gif(path);
          return null;
     }
}
```

# Factory Method – řešení 2.0

```
public class Printer {
     public virtual File DoMakeFile(String path) {
          return RawText(path);
     }
     public void printToPDF(String path) {
        // ...
     }
}
```

- **Knihovna pro tisk dokumentů**

```
public class DocumentPrinter : Printer {
     public override File DoMakeFile(String path) {
          if(ext(path) == "ppt") return new Presentation(path);
          else if(ext(path) == "doc") return new Writing(path);
          return super.CreateFile(path);
     }
}
```

- **Knihovna pro tisk obrázků**

```
public class ImagePrinter : Printer {
     public override File DoMakeFile(String path) {
          if(ext(path) == "jpeg") return new Jpeg(path);
          else if(ext(path) == "gif") return new Gif(path);
          return super.CreateFile(path);
     }
}
```

# Factory Method – implementace

- **Implementace**
  - **dvě hlavní varianty**
    - `Printer` je abstraktní třída – nutnost dědičnosti a vlastní implementace
    - `Printer` je standardní třída – defaultní implementace

# Factory Method – implementace

- **Implementace**
  - **dvě hlavní varianty**
    - `Printer` je abstraktní třída – nutnost dědičnosti a vlastní implementace
    - `Printer` je standardní třída – defaultní implementace
  - **parametrizované tovární metody**
    - variace – více druhů produktů
    - metoda dostane parametr identifikující druh objektu
    - všechny objekty sdílejí jedno rozhraní – `Product`

```
public class Creator {
    public virtual Product DoMakeProduct(ProdId id) {
        if(id==MINE)  return new MyProd;
        if(id==YOURS) return new YourProd;
        return null;
    }
}
```

```
public class MyCreator : Creator {
    public override Product DoMakeProduct(ProdId id) {
        if(id==MINE)   return new MyProd2;
        if(id==THEIRS) return new TheirProd;
        return super.DoMakeProduct(id);
    }
}
```

# Factory Method – výhody a nevýhody

- **Výhody**
  - Oddělení vrstev
    - Odstraní nutnost používat aplikačně-specifické třídy
  - Propojení paralelních tříd
    - Jasně určí, které třídy patří k sobě
  - Flexibilita za relativně malou cenu

- **Nevýhody**
  - Nutnost vytvořit potomka
    - Vytváření potomka Creator pouze kvůli instanci ConcreteProduct
    - Nevadí pokud klient musí vždy vytvořit potomka

# Factory Method

- **Použití generické třídy**
  - někdy zbytečné vytvářet další potomky Creatora
  - parametrizovaná Produktem
  - není zapotřebí vytvářet potomky

```
public interface IProduct {/*...*/}
public class MyProduct : IProduct {
    public int getSize() { return 10; }
}
public class Creator<T> where T :
        IProduct, new() {
    public IProduct? create() {
        try {
            return new T();
        }
        catch (Exception ex) {
            return null;
        }
    }
}
public class MyShop {
    private Creator<MyProduct> myCreator;
    private MyProduct myProduct;
    public MyShop() {
        myCreator = new
                Creator<MyProduct>();
        myProduct =(MyProduct)
                myCreator.create();
    }
}
```

# Factory Method – známé použití

- **Java API – iterace přes kolekce**

```
Set<Double> myHashSet = new HashSet<>();
Iterator<Double> mySetIterator = myHashSet.iterator();
```

```
List<Integer> myArrayList = new ArrayList<>();
Iterator<Integer> myIterator = myArrayList.iterator();
```

# Factory Method – Static Factory Method

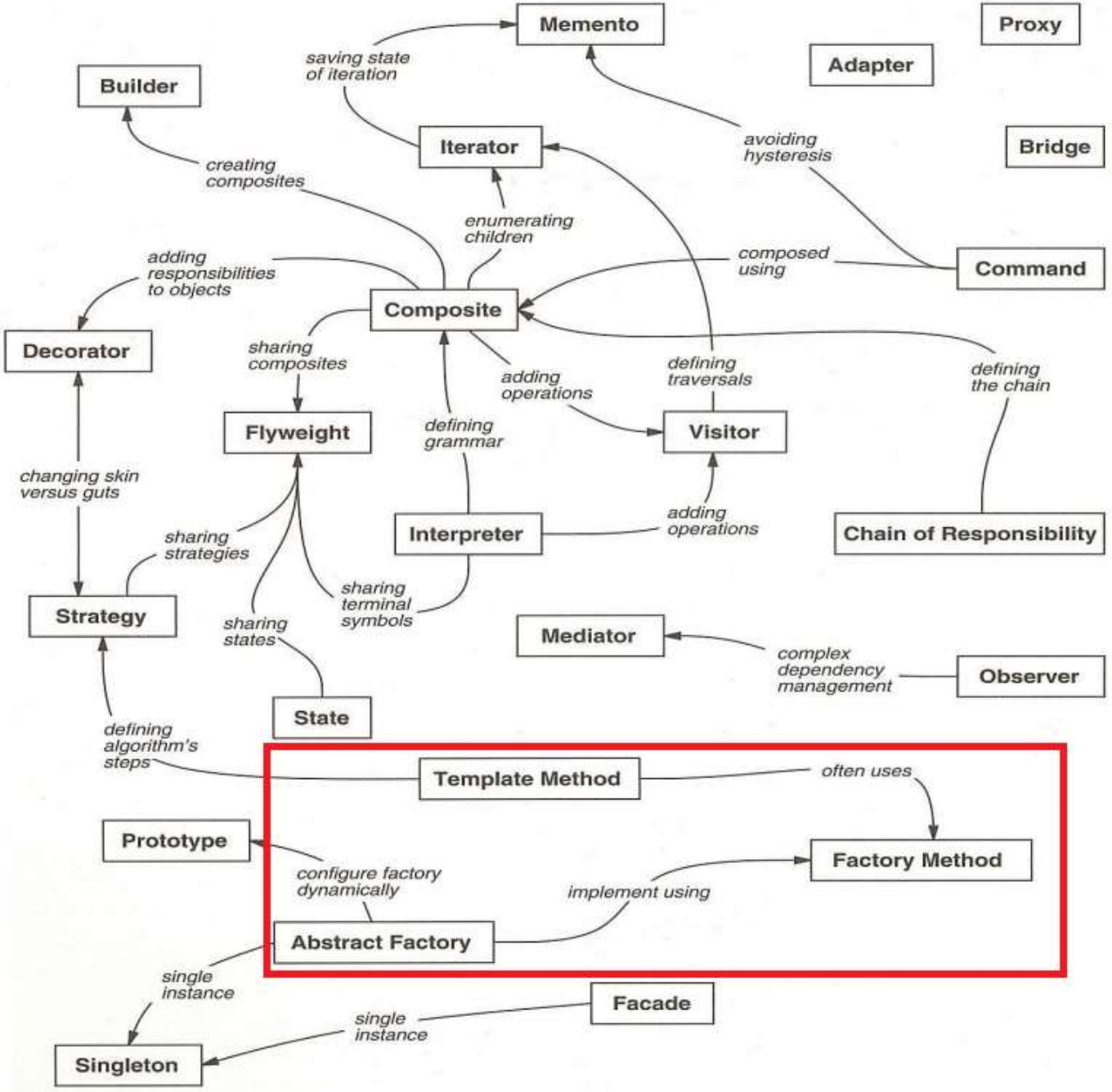
- **Když už není možné přetížit konstruktory**

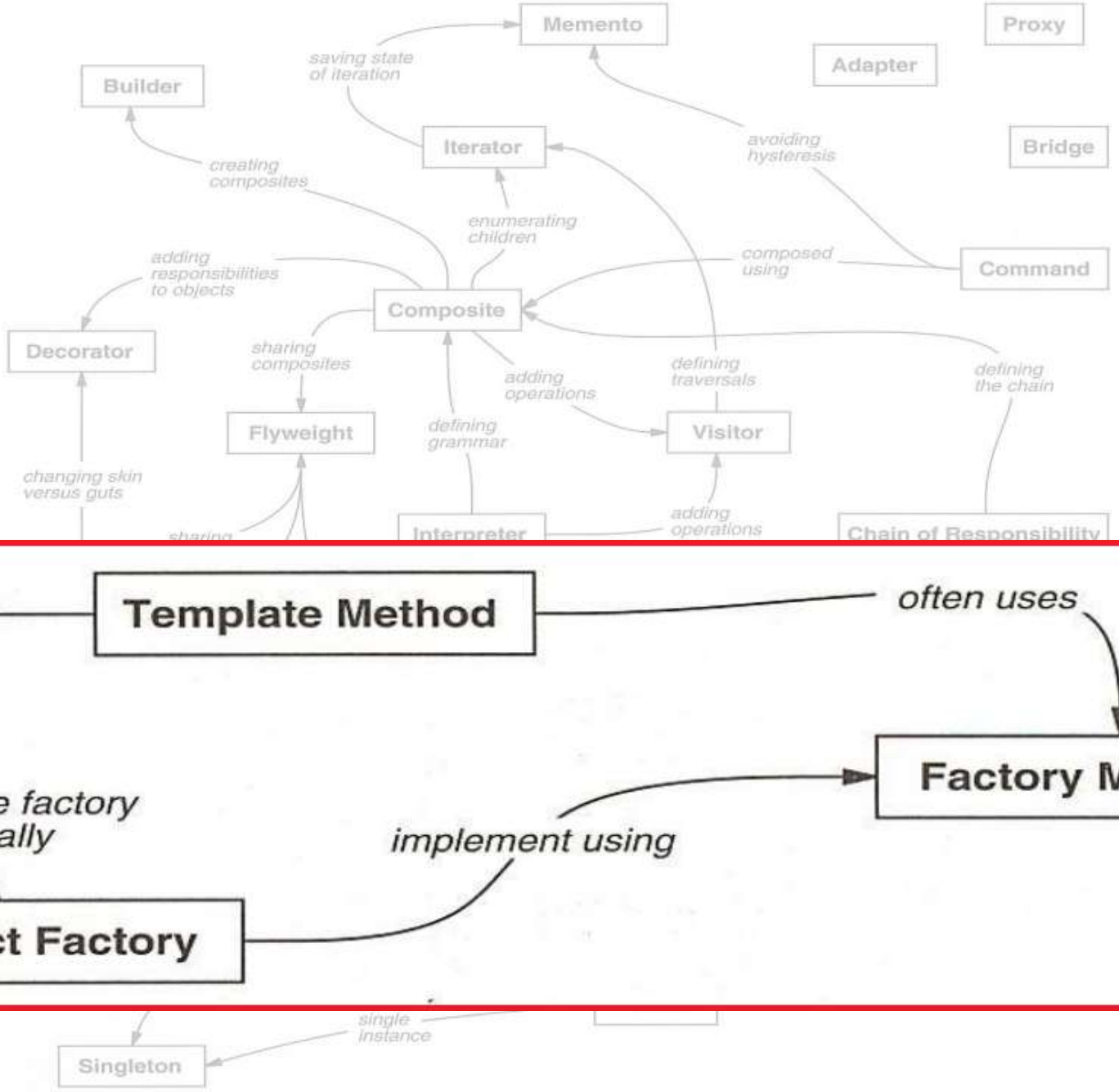
```
public class Complex {
    public double real;
    public double imaginary;

    public static Complex FromCartesian(double real, double imaginary) {
        return new Complex(real, imaginary);
    }

    public static Complex FromPolar(double modulus, double angle) {
        return new Complex(modulus * Cos(angle), modulus * Sin(angle));
    }

    private Complex(double real, double imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }
}
```

- **Výhody**
  - Není nutné vždy vytvářet novou instanci
- **Nevýhody**
  - Bez public nebo protected konstruktorů nelze vytvářet potomky Complex

# Factory Method - související vzory

- **Abstract Factory**
  - často implementovaná pomocí továrních metod
- **Template Method**
  - často volány tovární metody


Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
Strategy
sharing terminal symbols
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton


Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
Interpreter
adding operations
Chain of Responsibility
Template Method
often uses
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Singleton