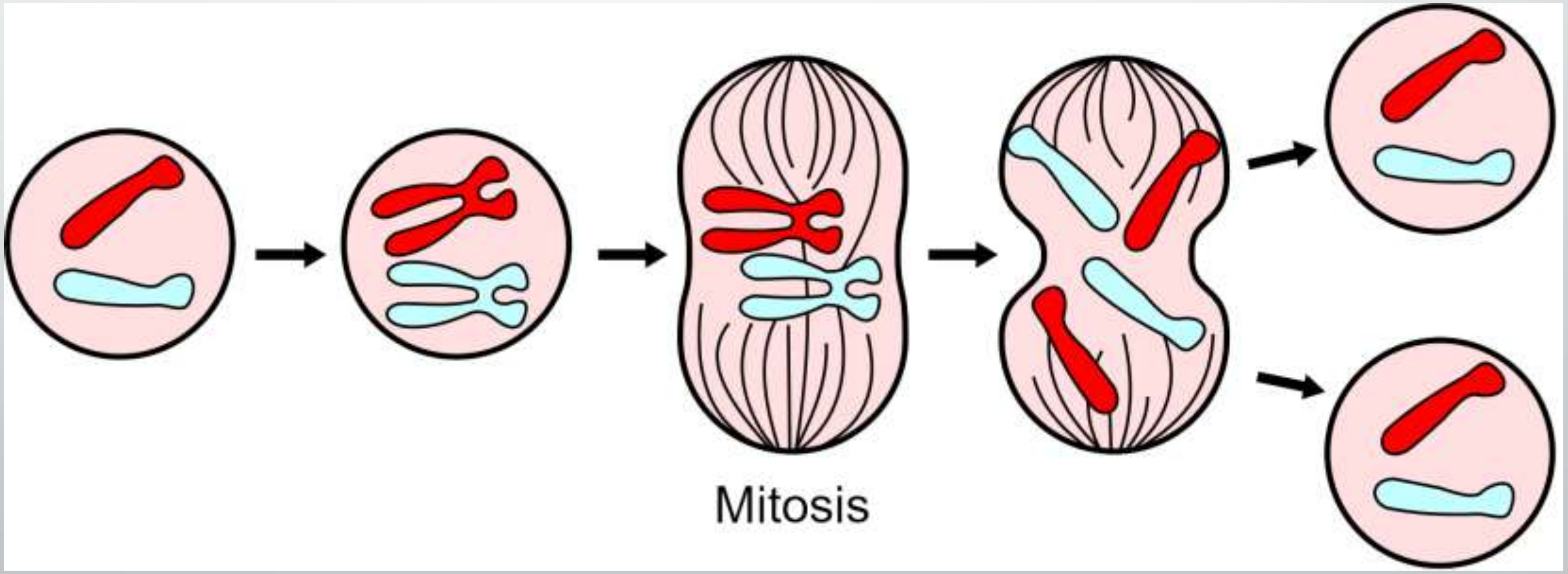
Design pattern: Prototype

Example from *real life*

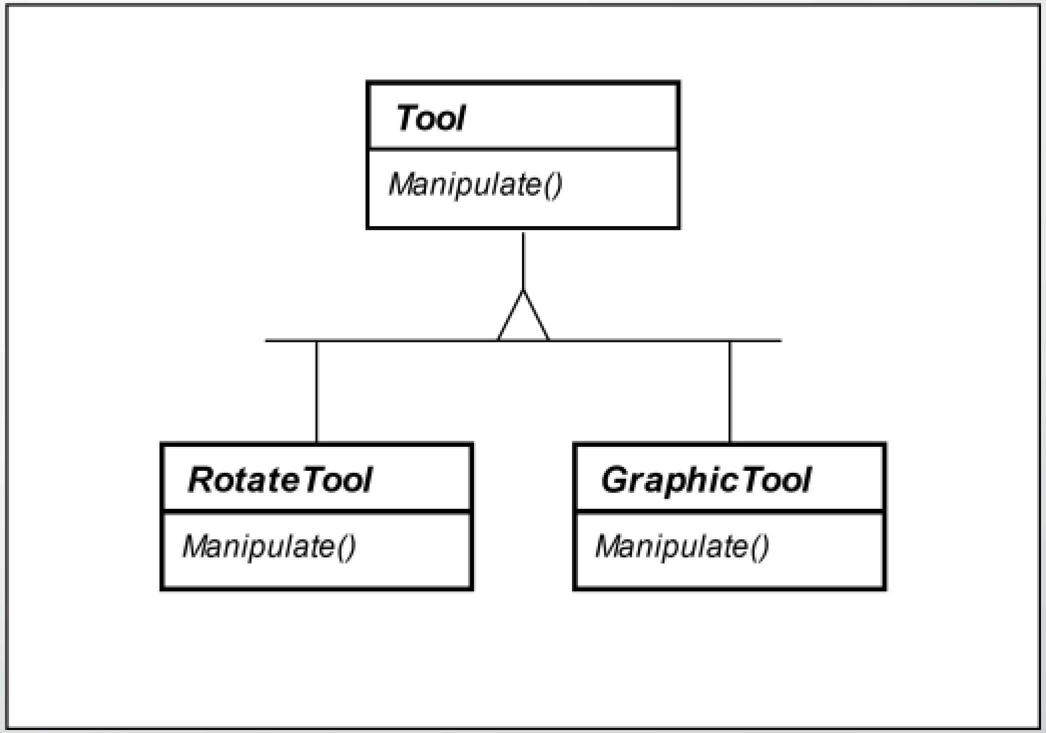
- Asexual reproducing (mitosis)

- Original cell => **prototype**
- *Takes part in creation*

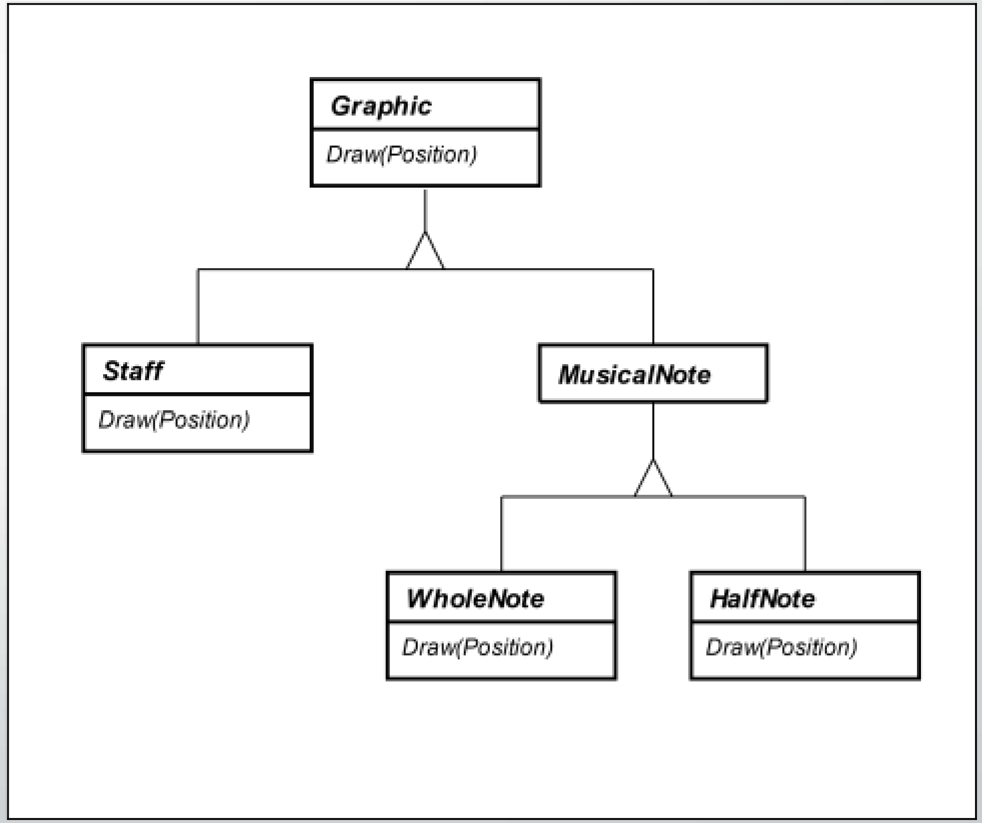


Example from *IT world*

- Graphical editor
- We want to draw notes, pauses etc.
- What should we do?

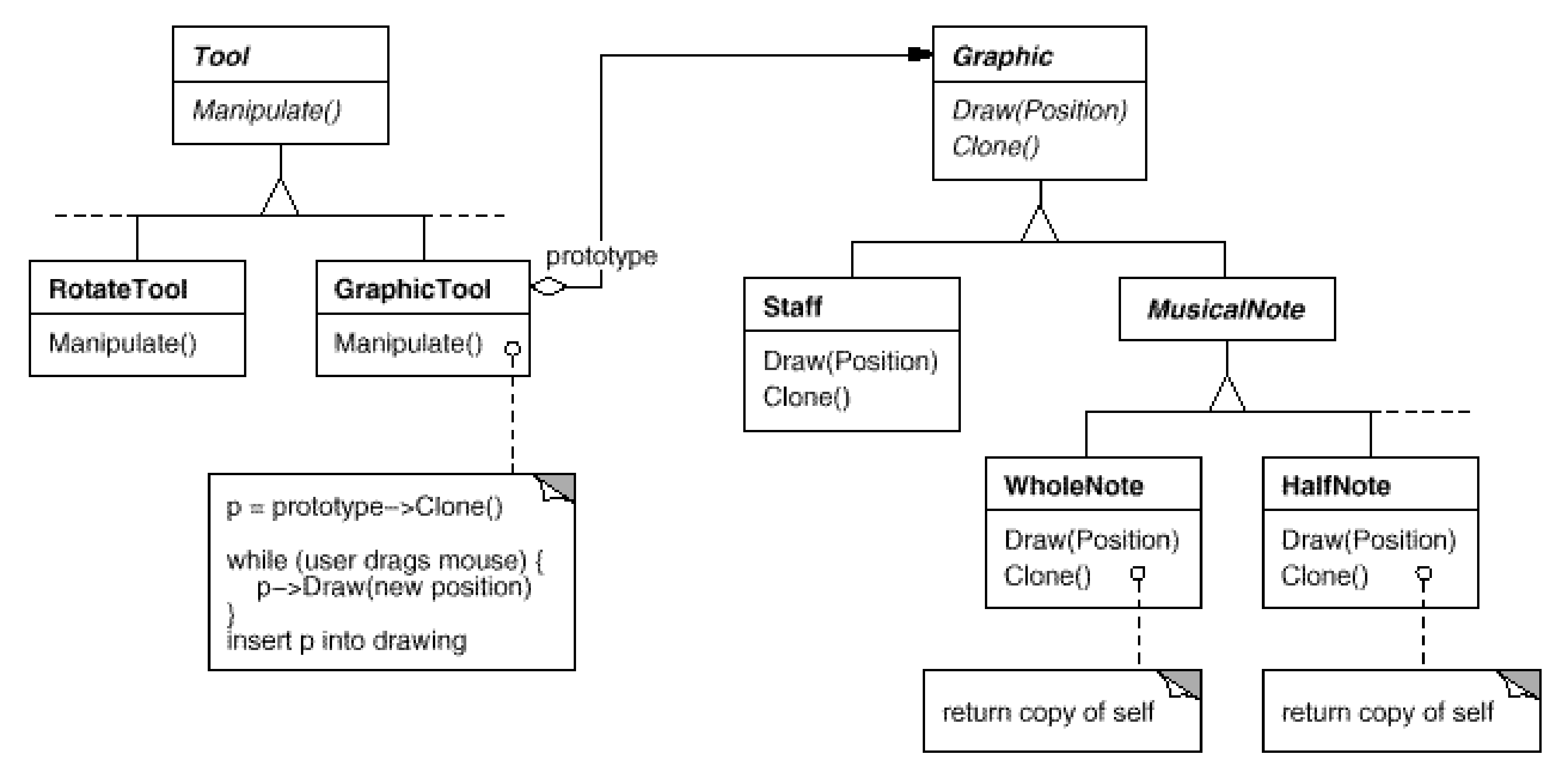


- **Summary**
  - Abstract class *Graphic* used as *ancestor for all objects that will be added to document*
  - Descendants of class Graphic are specific for our implementation



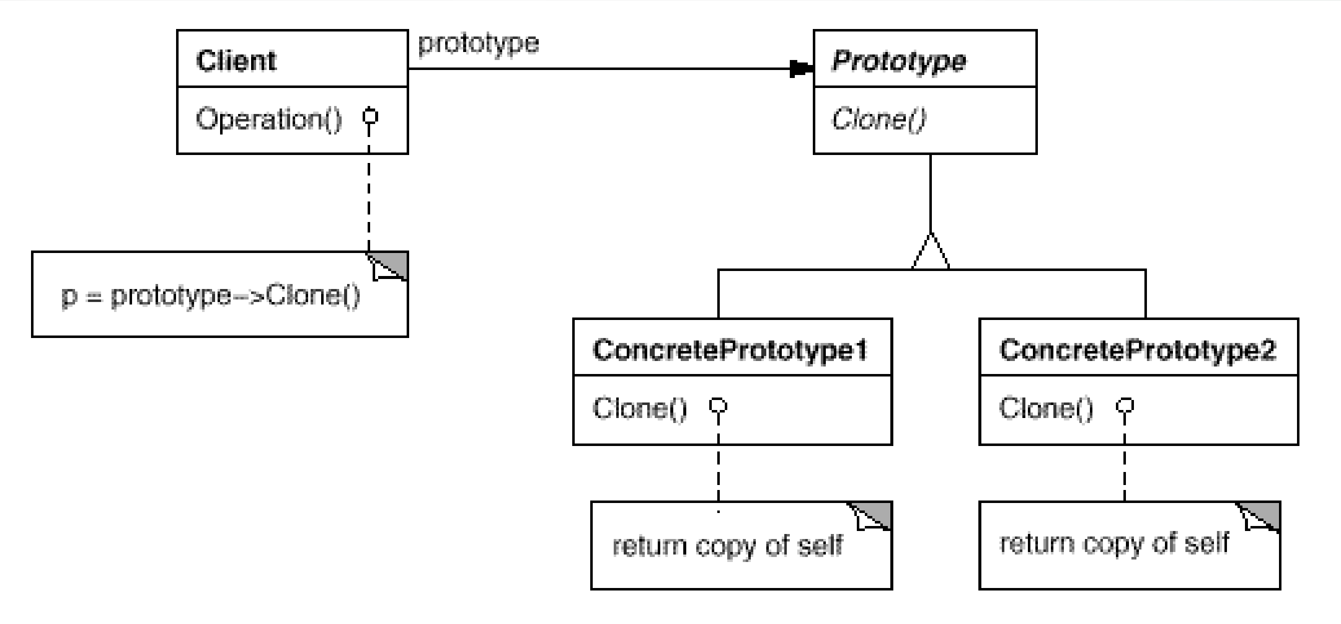
- **Problem**
  - *GraphicTool* does not know about our specific classes, it doesn't know how to create them
- **Solution**
  - Create new class similar to class *GraphicTool* for each of our specific classes
- **New problem**
  - Potencialy many classes that differ only in class they initialize

- **Other possible solution**
  - We create new instance of Graphic through cloning instance of subclass of Graphic (Staff, notes)
  - This type of *Graphic* is called **prototype**



```
p = prototype->Clone()

while (user drags mouse) {
    p->Draw(new position)
}
insert p into drawing
```

More general example

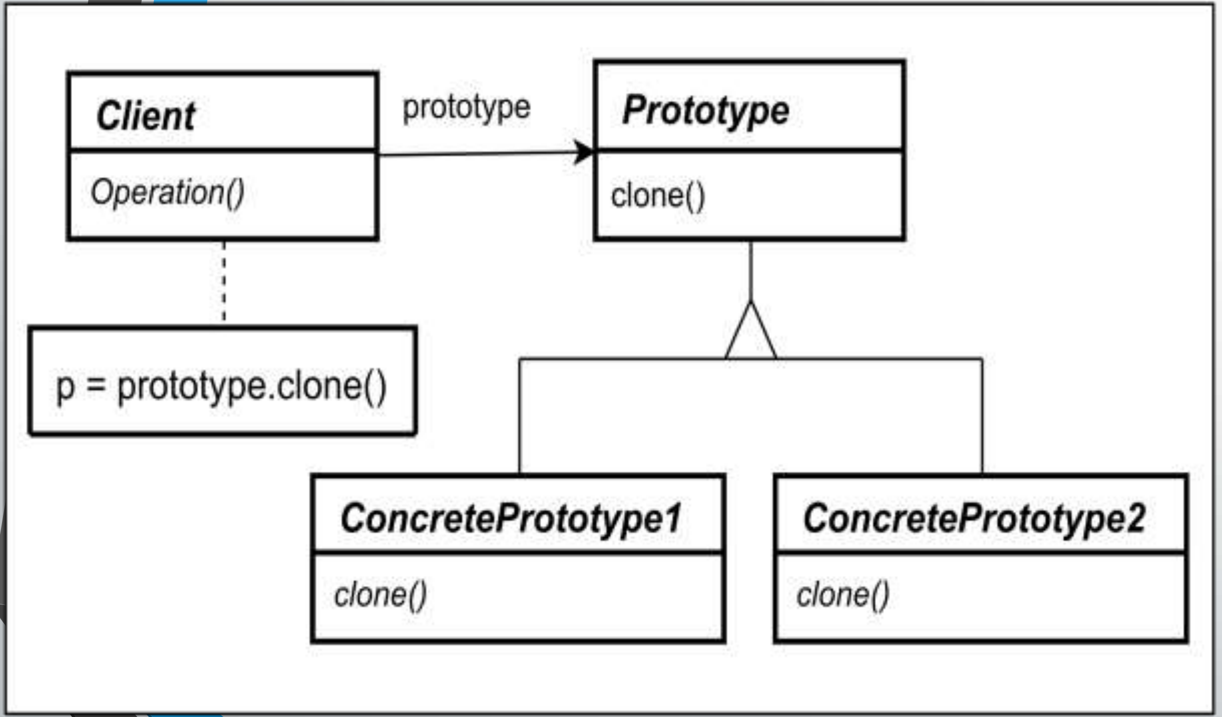


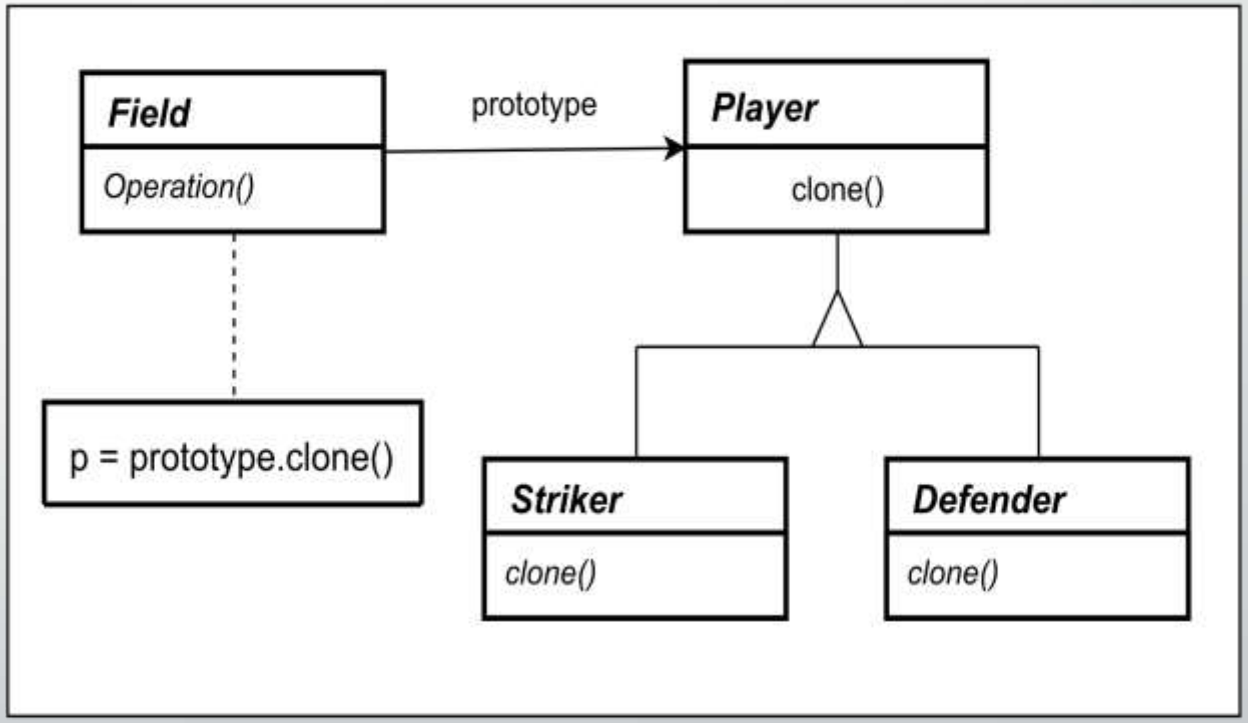
- **Prototype**
  - type *Graphic*
  - interface for cloning
- **ConcretePrototype**
  - *Staff, WholeNote, HalfNote*
  - Contains implementation for cloning
- **Client**
  - *GraphicTool*
  - It asks *prototyp* for creation of new objects (in reality cloning)

Characteristics of Prototype

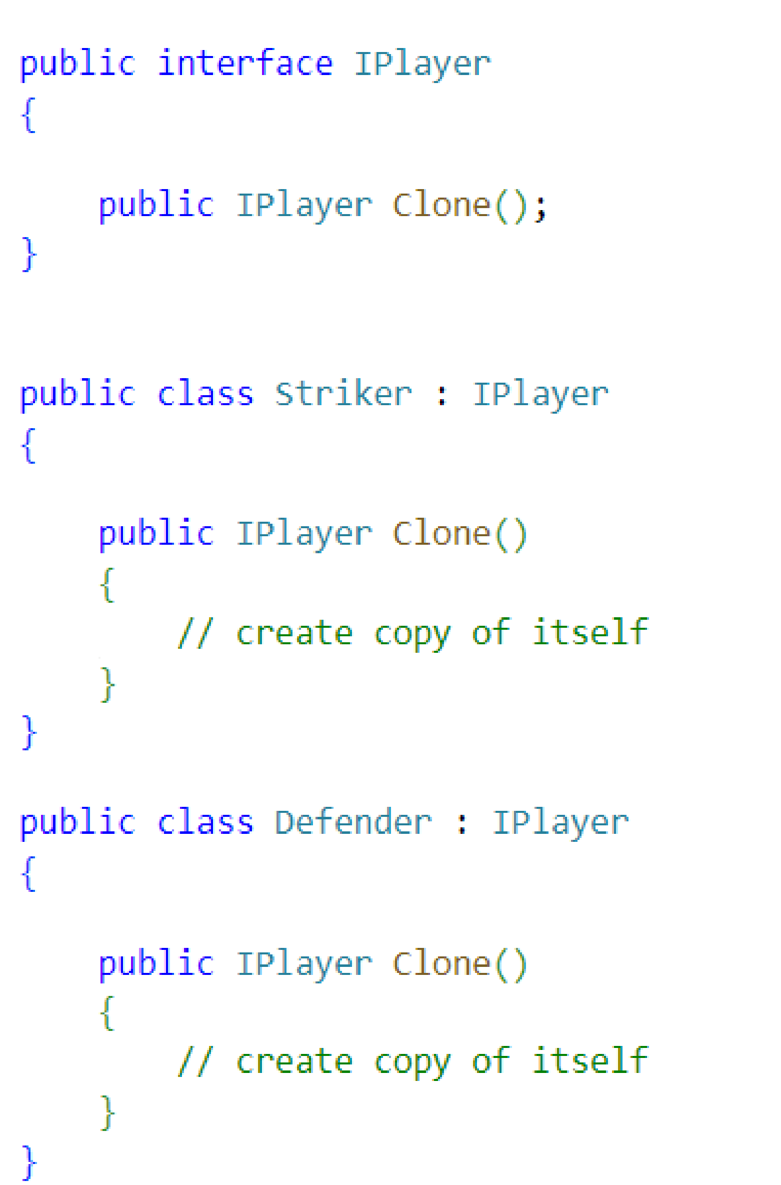
- **Creational** design pattern
- It simplifies implementation of client
  - It does not have to know about all classes
  - It is not dependent in types of objects which it interacts with (it knows only about interface)
- The main point
  - Creating copies (**clone**) via method *clone()*
  - *prototype->clone()*

Another example from *real world*

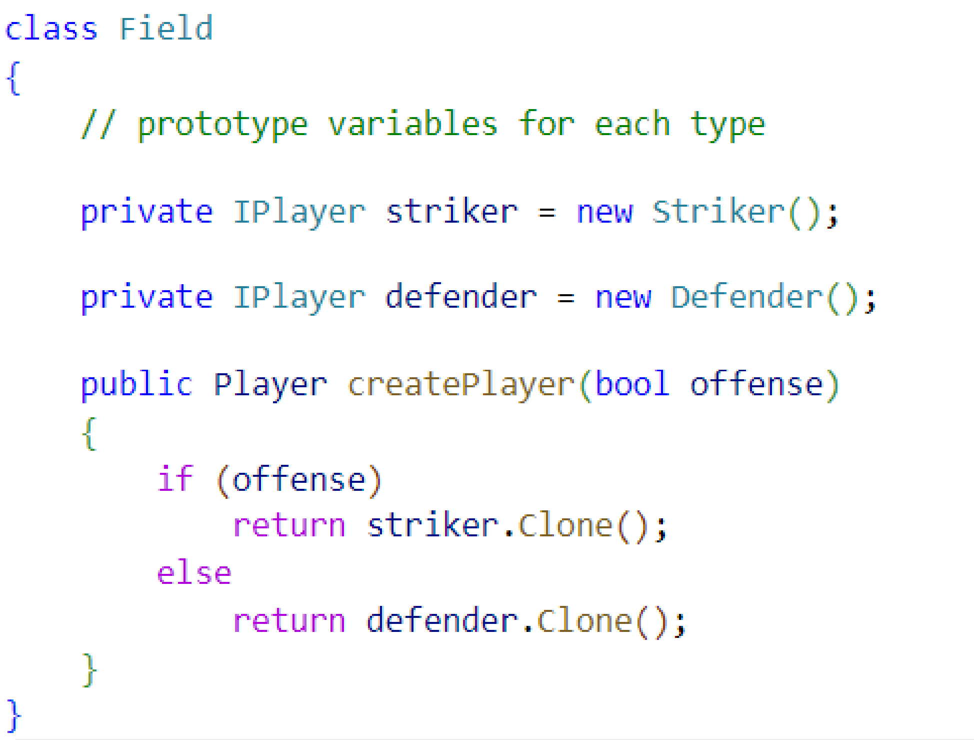




Specific example (C# + its conventions)



```
public interface IPlayer
{

    public IPlayer Clone();
}


public class Striker : IPlayer
{

    public IPlayer Clone()
    {
        // create copy of itself
    }
}

public class Defender : IPlayer
{

    public IPlayer Clone()
    {
        // create copy of itself
    }
}
```



```
class Field
{
    // prototype variables for each type

    private IPlayer striker = new Striker();

    private IPlayer defender = new Defender();

    public Player createPlayer(bool offense)
    {
        if (offense)
            return striker.Clone();
        else
            return defender.Clone();
    }
}
```

More characteristics of Prototype

- *Clone()* doesn't have to do **deep copy**
  - Sometimes **shallow copy** is enough, other time combination of shallow and deep copy os required
    - In many languages you can create a shallow copy of reference by assigning variable to a new variable (the reference gets copied)
- PrototypeManager
  - Used, if number of prototypes *is not fixed*
  - They can be registered in catalogue using an indicator (often used as memory optimization)
  - Client clones only prototype from catalogue
- Possible cons
  - Clone() can be a problem to implement for classes with cyclic references or attributes that cannot be copied

Other design patterns and their relations with Prototype

- **Factory Method**
  - Very similar, but requires subclassing (*Prototype* requires *initialization*)
- **Abstract Factory Method**
  - It can use collection of prototypes whole clone it will return as resulting objects
- **Singleton**
  - *Prototype* can use Singleton in its implementation
- **Composite, Decorator**
  - *Prototype* can be used for saving already created composites

Use case for *Prototype* (1)

- **Construction of object (or its part) is non-trivial**
  - example: We need to load and process file during its creation

- We don't want to construct object again and unneccessarily slow down program
  - We are interested in time effectivity

- We use design pattern *Prototype*
  - We create first object using constructor and we clone other from this first object (we assume we don't need to load and process file again)

Use case for *Prototype* (2)

- **Classes are being created dynamically during runtime**
  - example: we procedurally generate different assets for game
- We want to create instances of such class
- We use design pattern *Prototype*
  - For each type, we create 1 instance (prototype) we will clone more instances from
- Other design patterns don't enable this kind of creation

Use case for *Prototype* (3)

- **We create many kinds of some class** (example: many different parametrizations)
    - example: we are creating a game where many units are very similar and differ only in parameters and assets used
- *Possible solutions:*
    - For each kind of instance –> new class that inherits from main
        - cons:
            - Unusable, if classes are generated during runtime
            - Impractical, if the amount of classes can change
    - For each kind of instance –> set of parameters of main class (we have no descendants)
        - pros:we don't have many classes, one is enough
        - cons: complicated to use
    - We use *Prototype*
        - For each kind we create *prototype*
        - Flexible solution and we only have one class

Use case for *Prototype* (4)

- **We want many instances of the same type –> creation of object requires too much code**
  - example: when using design patterns such as Builder or Decorator
- *Possible solutions:*
  - Abstract Factory / Factory Method
    - We have method that runs constructing code
  - Prototype
    - Why should we use this instead of factory?
      - Object is constructed once and further we can simply clone
      - Factory generally cannot create new types during runtime
  - Builder
    - More flexible and readable solution, but quite "wordy" if it needs to be repeated

Thank you for your attention