Creational Patterns

Prototype:

efficiently create complex objects

Motivation

- Implementing portfolios in a financial software
  - Each portfolio consists of a diverse mix of stocks and bonds, with specific strategies, risk profiles, and performance metrics.
    - Balanced portfolio, international portfolio, retirement portfolio
    - A wide class hierarchy
  - Creating and managing new portfolio instances *from scratch* every time would be ***prone to errors*** and *slow*.
  - I need to create it fast

Clone preconfigured objects instead

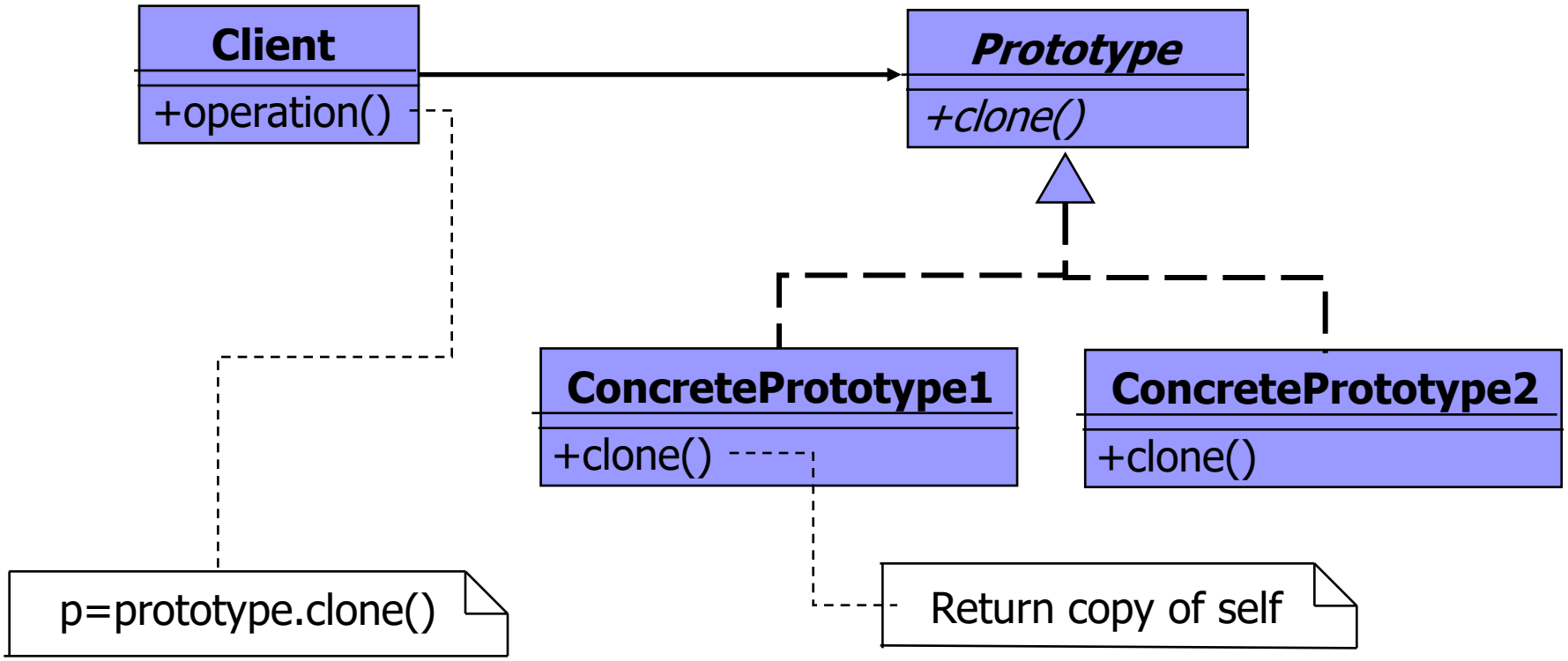
- Need to create new objects many times in a complex class hierarchy

Prototype

- **Intent:** Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype. (clone)

- **Applicability**
  - Creation is *expensive* and need many different copies of complex objects,
  - Instances can have one of *only a few* different combinations of states or a few states
  - When the classes to instantiate are specified at run-time
  - to avoid building a class hierarchy of factories

Prototype - Structure



Back to our example

- Define prototypes for different types of portfolios
  - we can quickly clone these preconfigured, fully initialized objects.

- Fill in the repository with Portfolio prototypes.

```
Portfolio p=new Portfolio(“balanced”);
p.addStocks(new Stock(“abc”)); //add other stocks and bonds
p.addStrategy(new BalancedSt());//some other configurations
prototypes.add( p);

Portfolio p1=new Portfolio(“retirement); //fill in the details for retirement
prototypes.add( p1);
```

Back to our example

- Define prototypes for different types of portfolios
  - we can quickly clone these **preconfigured**, **fully initialized** objects.

- Fill in the repository with Portfolio prototypes.

```
….//pre-configure portfolio objects for retirement
prototypes.add( p1);
```

- Now create!

```
Portfolio balance1=prototypes.get(“balanced”).clone()
Portfolio retire1=prototypes.get(“retirement”).clone();
```

- avoid **repeating complex setup logic**

Exercise

- Character creation part of a role-playing game
  - There are hero characters
  - There are monsters
  - There are villains
  - Client wants a random character through some events (e.g. button click)

Attempt1:

```
class Scene{
public Character getRandomChar(){
    int n=rando
    switch(n){
       case 0: retu          ero();
        case1: r             ster();
       case 2: retu  new      lain();
    }
}
}
class Hero implements Character{}
class Monster implements Character{}
class Villiain implements Character{}
```

Exercise –Attempt2 - Solution

- Character creation part of a role-playing game
  - There are hero characters
  - There are monsters
  - There are villains
  - Client wants a random character through some events (e.g. button click)

```
interface Character{
  Character clone(); …}
class Monster implements Character{
   Character clone(){…// copy of this}}
class Hero implements Character{
   Character clone(){…// copy of this}}
```

```
class CharacterManager{
  private List<Character> cs;
  public Character get(){
       Character c=cs.get (random());
       result= c.clone();
       //adjust the clone like its location
       return result;
   }
public void add(Character prototype){
     cs.add(prototype);
  }
  //… //initialize cs with prototypes
```

Exercise –Attempt2 - Solution

- Character creation part of a role-playing game
  - There are hero characters
  - There are monsters
  - There are villains
  - Client wants a random character through some events (e.g. button click)

```
interface Character{
  Character clone(); …}
class Monster implements Character{
   Character clone(){…// copy of this}}
class Hero implements Character{
   Character clone(){…// copy of this}}
```

```
class CharacterManager{
  private List<Character> cs;
  public Character get(){
       Character c=cs.get (random());
       result= c.clone();
       //adjust the clone like its location
       return result;
   }
public void add(Character prototype){
     cs.add(prototype);
  }
  //… //initialize cs with prototypes
 private int random(){
     return (int)Math.random()*cs.size();
}
```

Prototype pattern!

Exercise –Attempt3 – solution 2

```
class  Character{
   public Character clone(){…}
 …}


init(){   //populating the prototype
     pool
  Character c=new Character();
   //set it up to make it a hero
   c.setImage(hero.jpg);
   add(c);
  //new prototypical Monster
   c=new Character();
  // set up to monster
   add(c);
  …
}}
```

```
class CharacterManager{ /
  private List<Character> cs;
  public Character get(){
       Character c=cs.get (random());
       result= c.clone();
       //adjust the clone like its location
       return result;
   }
public void add(Character prototype){
     cs.add(prototype);
  }
```

- **Prototype supports dynamic behavior.**
- Reduced subclassing

Prototype: reduce hierarchy

Applicability: “to avoid building a class hierarchy of factories”

- Consider the Abstract Factory pattern
  - Each variant needs a Concrete Factory
- Can I have 1 configurable Concrete Factory class instead?
  - Each **instance** of this class will be configured with a different family/variant

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: Factories (C++)

```
class GameElementFactory {// The Abstract Factory
 public:
   virtual Player*  makePlayer()=0;
   virtual Obstacle* makeObstacle()=0;
} ;
```

// Concrete factory: **one configurable factory that clones prototypes**.

```
class ElementFactory: public GameElementFactory {
 public:
   ElementFactory(Player* p, Obstacle* o):
          playerPrototype(p), obstaclePrototype(o){}
   virtual Player* makePlayer() {
            return playerPrototype->clone(); }
   virtual Obstacle* makeObstacle() {
            return obstaclePrototype->clone(); }
 private:
     Player* playerPrototype; Obstacle* obstaclePrototype;
};
```

Prototype: another applicability

Applicability: “When the classes to instantiate are specified at run-time”

- you are **passed** an object and use that object as a **template** to create a new object.
- you **don’t know** the implementation details of the object,
  - How is it built? What is the internal structure?
  - cannot create a new instance of the object
- Instead, ask the object ***itself*** to give you a copy of itself.
- *You only need to know the Interface*

Prototype

Allows programs to perform operations like ….

- initialize objects to a state that has been established through use of the system.
  - often preferable to initializing the object to some generic set of values.
  - Some business systems produce an initial model from an existing business object. The copy can then be modified to its desired new state
  - E.g. Reporting object: Consider a report object that contains processed information to be passed to the GUI. The original report contains the data in ascending order. Now, using this pattern one can create a similar report but with data sorted in descending order. (chavan @stackoverflow)

Prototype to Save time…

- Where **creation** of an object is **expensive**, but copying is cheap, the prototype pattern will be more **efficient**.

```
Image loadUserImage() {
    //loads from disk. will be slow
    return new JPEGImage("path/to/user/image.jpg");
}
```

- If this method is going to be called repeatedly….

```
Image loadUserImage() {
      return userImagePrototype.clone();
    //copy in memory. will be fast
}
```

Time saving

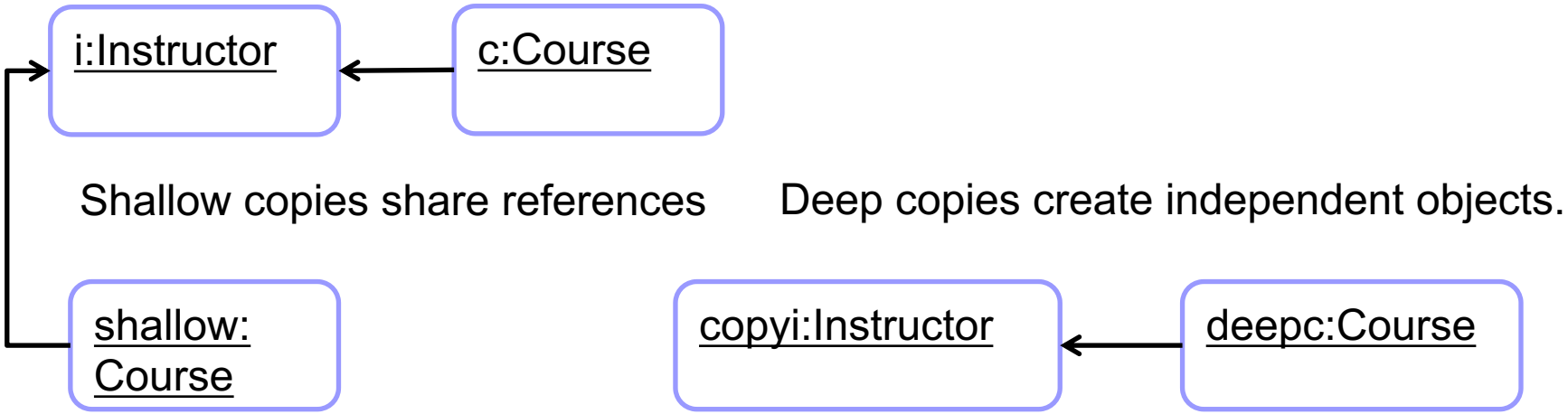
- Time savings: when creating an object requires a costly access to auxiliary information
  - requesting configuration data from a file, a database, or over a network.

- Example: Building lots of pages from a template that is stored on a web server.
  - It is cheaper to read the template once and clone it to get the starting point for each new page,
  - rather than querying the web server separately for each page.

http://stackoverflow.com/questions/11706513/java-prototype-design-pattern-object-creation

Implementation issue-1

- Shallow or Deep Copy



- cloning prototypes with complex structures usually requires a deep copy, as the clone and the original must be independent.
- Be careful with **circular references** in the complex object

	Shallow Copy	Deep Copy
What it does:	**Copies all of the object's fields directly.** If a field is a reference to another object (like Instructor), only the memory address is copied, not the Instructor object itself.	Creates a copy of the Course object and recursively creates a new copy of the Instructor object it references.
Result:	The original Course and its clone end up sharing the exact same Instructor object.	The original Course and its clone are completely independent. They have their own, separate Instructor objects.
Danger or Benefit	If you change the Instructor's name through the original Course object, the name also changes for the shallow copy! This can lead to unexpected bugs.	This is the safer and more common approach. Changes to one Course or its Instructor will not affect the other. Cloning circular reference needs care.
When to use:	when your objects contain only primitive types (like int, String) or when you intentionally want to share the referenced objects.	when you need a true, independent duplicate of a complex object.

Implementation -2

- Clone method does not accept parameters
  - Create the clone
  - Then adjust it, like initialization
- Using a prototype manager with a registry
  - Client could expand the registry at runtime
  - There is no registry in the original pattern description.
    - Not all prototype use case require them.

sometimes generics/templates do the job

This is not prototype, but a “template factory”

```
class Spawner {
    public: virtual Monster* spawnMonster() = 0;
};
template <class T> class SpawnerFor : public Spawner {
  public: virtual Monster* spawnMonster() {
            return new T();
  }
};
```

How is this different from the Prototype?

- prototype is about the cloning of an existing object's state.

This is not Factory Method pattern either.

Code: https://gameprogrammingpatterns.com/prototype.html

Prototype-Consequences

- Adding and removing products at runtime
  - Simply register a prototype to the client
- Specifying new objects by varying values and structure
  - Prototypes for parts of a complex object
  - Each part could be a different structure
  - Some parts are same structure but different value
- Hides concrete product classes from clients
  - Reduce number of names the clients know
- Reduced subclassing
  - Cloning instead of inheritance
- Implementing clone() may be difficult
  - Circular references, choosing deep or shallow copy

Related Patterns

- **Abstract Factory** and Prototype may work together
  - They are also competing patterns.
- Designs that make heavy use of the **Composite** and **Decorator** patterns often can benefit from Prototype
- Flyweight may seem similar, but Flyweights are shared whereas Prototypes are not

Factory Method vs Prototype

- Both create customized objects without knowing their class or any details of how to create them
  - client can create any of the derived class objects without knowing anything about their own structure.
- Prototype: self duplication, but FM a fresh creation via subtype of Creator