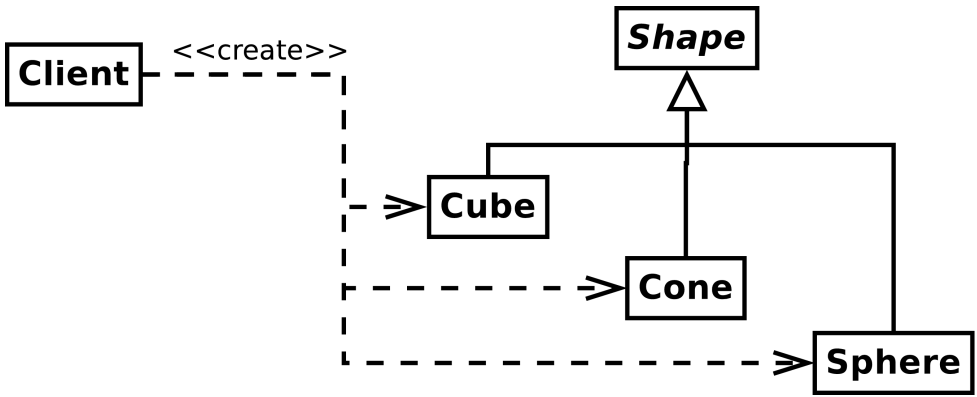
Prototype (prototyp)

Cel:

Określenie rodzaju tworzonych obiektów poprzez wskazanie ich prototypu. Nowe instancje tworzymy kopiując prototyp.

Przykład:



- Edytor 3D – klient tworzy obiekty różnych kształtów

```
abstract class Shape {
    public void transform(AffineTransformation af)
    { /* ... */ }
}

class Cube extends Shape { /* ... */ }
class Cone extends Shape { /* ... */ }
class Sphere extends Shape { /* ... */ }

class Application {
    public Application() {
        toolbox.add(new Icon("cube"));
        toolbox.add(new Icon("cone"));
        toolbox.add(new Icon("sphere"));
    }
    public void action(Icon ic) {
        if(ic.name.equals("cube")) scene.add(new Cube());
        if(ic.name.equals("cone")) scene.add(new Cone());
        if(ic.name.equals("sphere")) scene.add(new Sphere());
     }
}
```

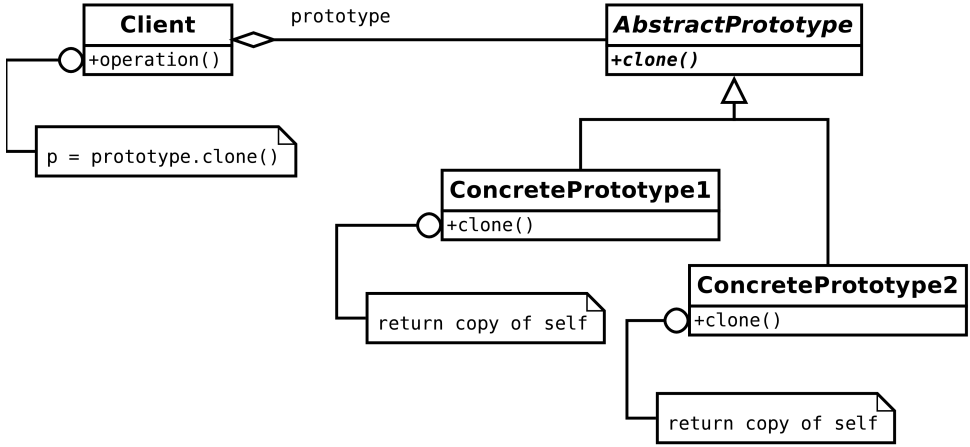
Rozwiązanie:

```
abstract class Shape {
    public void transform(AffineTransformation af)
    { /* ... */ }
}

class Cube extends Shape { /* ... */ }
class Cone extends Shape { /* ... */ }
class Sphere extends Shape { /* ... */ }

class Application {
    public Application() {
        toolbox.add(new Icon(new Cube()));
        toolbox.add(new Icon(new Cone()));
        toolbox.add(new Icon(new Sphere()));
    }
    public void action(Icon ic) {
        Shape nowy = ic.prototype.clone();
        scene.add(nowy);
    }
}
```

Struktura:



Składniki:

- Prototype – deklaruje interfejs klonowania
- ConcretePrototype – implementuje operację klonowania samego siebie
- Client – tworzy nowe obiekty prosząc prototyp o sklonowanie siebie

Zależności:

Klient prosi prototyp o zwrócenie swojej kopii.

Zastosowanie:

- Gdy system ma być uniezależniony od tego, w jaki sposób obiekty są tworzone, składane i reprezentowane, i:
  - gdy klasy tworzonych obiektów są podawane w trakcie działania programu (np. przez dynamiczne ładowanie)
  - aby zapobiec tworzeniu całej hierarchii klas fabryk równolegle do hierarchii klas produktów
  - gdy instancje klasy mogą przyjmować jedynie jeden z niewielu stanów – może wówczas lepiej będzie zrobić sobie odpowiednią liczbę prototypów i klonować je, zamiast tworzyć klasy ręczne wszędzie tam, gdzie ich potrzebujemy?

Konsekwencje:

Ukrywa przed klientem konkretną klasę produktu (zatem: mniej nazw, o których musi on wiedzieć), wymiana klas bez angażowania klienta. A poza tym:

1. dodawanie i usuwanie produktów w trakcie wykonania programu – dołączenie nowej klasy produktu polega tylko na zarejestrowaniu jego prototypu
2. definiowanie nowych obiektów przez zmianę wartości – nowe zachowania możemy definiować przez kompozycję (np. ustawiając wartości zmiennych), a nie definiowanie nowych klas. Rejestrujemy stworzone i odpowiednio sparametryzowane instancje klas jako prototypy, w ten sposób nawet użytkownik może „definiować” nowe „klasy”

3. definiowanie nowych obiektów przez zmianę struktury – obiekty mogą składać się z części, np. układ cyfrowy w odpowiednim edytorze; możemy pozwolić użytkownikowi zdefiniować jego układ jako prototyp na równi z naszymi

```
toolbox.add(new Cube());
toolbox.add(new Cone());
toolbox.add(new Shape(new Vertex(0.0, 1.0, 0.0),
                      new Vertex(1.0, 1.0, 1.0),...));
toolbox.add(new Shape(new Face(...),new Face(...),...));
toolbox.add(new Shape(new File("Pyramid.shp")));
```

4. mniej dziedziczenia – Factory Method zazwyczaj produkuje hierarchię kreatorów równoległą do hierarchii produktów, prototyp w ogóle nie wymaga kreatorów; doceniane zwłaszcza w C++, bo np. Java ma klasę Class, która sama może działać jak prototyp i służyć jako kreator/prototyp:

```
Class prototyp = Cube.class;
Shape sh;
try {
     sh = (Shape)prototype.newInstance();
} catch(Exception ex) { sh = null; }
scene.add(sh);
```

5. Najpoważniejszym minusem jest konieczność implementacji metody Clone w każdej podklasie prototypu, co może być trudne – np. gdy te klasy już istnieją i/lub zawierają obiekty które nie wspomagają klonowania lub zawierają cykliczne referencje.

Implementacja:

1. menadżer prototypów – ilość prototypów w systemie nie jest ustalona i może zmieniać się dynamicznie – klient nie będzie wówczas musiał sam się nimi zajmować, a jedynie pobierał je z rejestru, gdzie są składowane; menadżer prototypów pozwala na rejestrowanie prototypu pod określoną nazwą-kluczem, wyrejestrowywanie, przeglądanie, zamianę, etc.

```
class Manager {
     private HashMap<String, Prototype> map =
                              new HashMap<String, Prototype>();
     public void register(String name, Prototype p) {
          // usuwa poprzedniego o tej samej nazwie
          map.put(name, p);
     }
     public void unregister(String name, Prototype p) {
          map.remove(name);
     }
     public Prototype createNew(String name) {
          return map.get(name).clone();
     }
}

Manager manager = new Manager();
manager.register("cube", new Cube());
manager.register("cone", new Cone());
// ...
scene.add(manager.createNew("cube"));
```

2. implementacja operacji clone – może być trudnym zadaniem, zwłaszcza przy złożonych obiektach o cyklicznych referencjach; problemu kopii płytkiej/ głębokiej: niekiedy płytka wystarcza, ale klonując prototypy o skomplikowanej strukturze oczekujemy kopii głębokiej

```
class Prototype implements Cloneable {
     private int a;
     private double b;
     // ...
     public Object clone() throws CloneNotSupportedException
     {
          return super.clone();
     }
}
// ...
Prototype prototyp = ...
try {
     Prototype nowy = (Prototype)prototyp.clone();
} catch(CloneNotSupportedException ex) {
     // ...
}
```

```
class Prototype implements Cloneable {
     private int a;
     private double b;
     // ...
     public Prototype clone() {
          Prototype p = null;
          try {
               p = (Prototype)super.clone();
          }catch(CloneNotSupportedException e) { }
          return p;
     }
}
// ...
Prototype prototyp = ...
Prototype nowy = prototyp.clone();
```

uwaga na kopię płytką!

```
class Prototype implements Cloneable {
     private Object a;
     private double[] b;
     // ...
     public Prototype clone() { /* ... */ }
     public String toString() {
          return super.toString() + "(" + a + "," + b + ")";
     }
}
// ...
Prototype prototyp = ...
Prototype nowy = prototyp.clone();
System.out.println(prototyp);
System.out.println(nowy);
```

```
class Prototype implements Serializable{
     private Object a;
     private double[] b;
     public Prototype clone() {
          Prototype p = null;
          try {
               ByteArrayOutputStream bufor =
                                   new ByteArrayOutputStream();
               ObjectOutputStream out =
                                  new ObjectOutputStream(bufor);
               out.writeObject(this);
               out.close();
               ObjectInputStream in = new ObjectInputStream(
                                      new ByteArrayInputStream(
                                         bufor.toByteArray()));
               p = (Prototype)in.readObject();
               in.close();
          } catch(Exception ex) { }
          return p;
}    }
// ...
Prototype prototyp = ...
Prototype nowy = prototyp.clone();
```

3. inicjalizacja klonów – czasem wystarczy nam stworzenie klonu z wartościami skopiowanym z prototypu, ale niekiedy klient oczekuje własnej inicjalizacji jego składowych; nie możemy przekazać tych wartości jako parametru w Clone, bo ich liczba może być różna w różnych klasach; jeśli już mamy jakieś operacje do ustawiania stanu, klient musi je wywołać tuż po sklonowaniu prototypu, jeśli nie, możemy stworzyć operację Initialize, która będzie przyjmować odpowiednie argumenty i ustawiać początkowy stan obiektu; *uwaga: być może wykonana wcześniej kopia głęboka będzie znikać podczas reinicjalizacji*

```
Shape nowy = toolbox.get("cube").clone();
nowy.place(0.0, 2.0, -1.0);
```

Powiązania:

- Jeśli korzystamy ze złożonych wzorców strukturalnych, np. Composite lub Decorator, wzorzec Prototype pozwala nam uprościć proces ich konstrukcji.
- Wzorce Prototype i Abstract Factory są konkurencyjne względem siebie – często tę samą rzecz możemy osiągnąć przy użyciu dowolnego z nich. Niekiedy mogą być też użyte razem – np. Abstract Factory może składować prototypy produktów do utworzenia

```
class Factory {
    private AbstractŚciana prototype_s;
    public Factory(AbstractŚciana s, AbstractSzafka sz)
    { prototype_s = s; ... }
    public AbstractŚciana createŚciana()
    { return prototype_s.clone(); }
     //...
}
//...Client...
Factory factory = new Factory(new Ściana2D(),
                              new Szafka2D());
model.add(factory.createŚciana());
model.add(factory.createŚciana());
model.add(factory.createSzafka());
//...
```