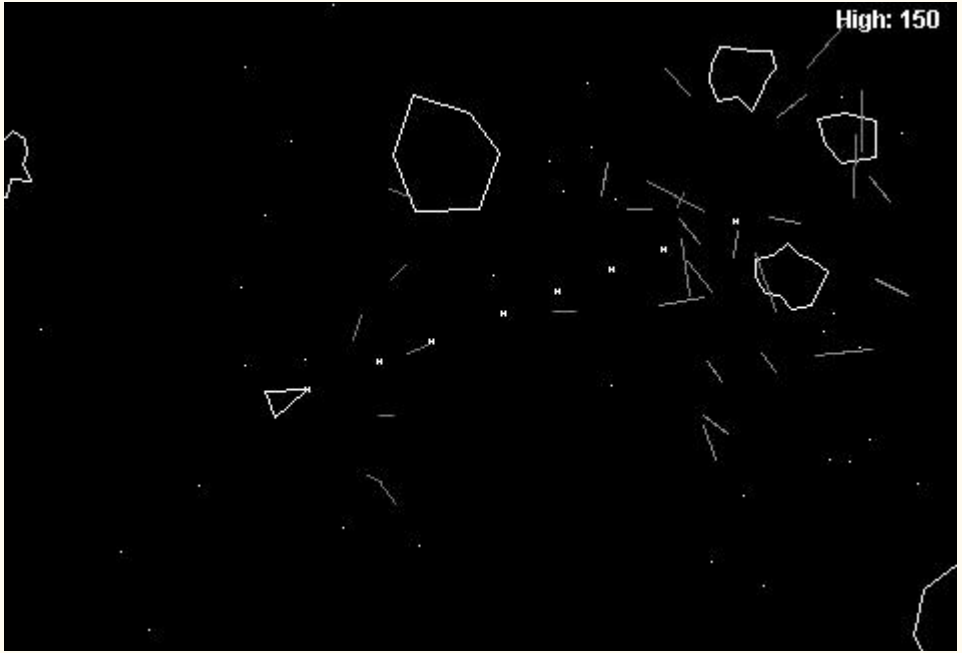
Prototype

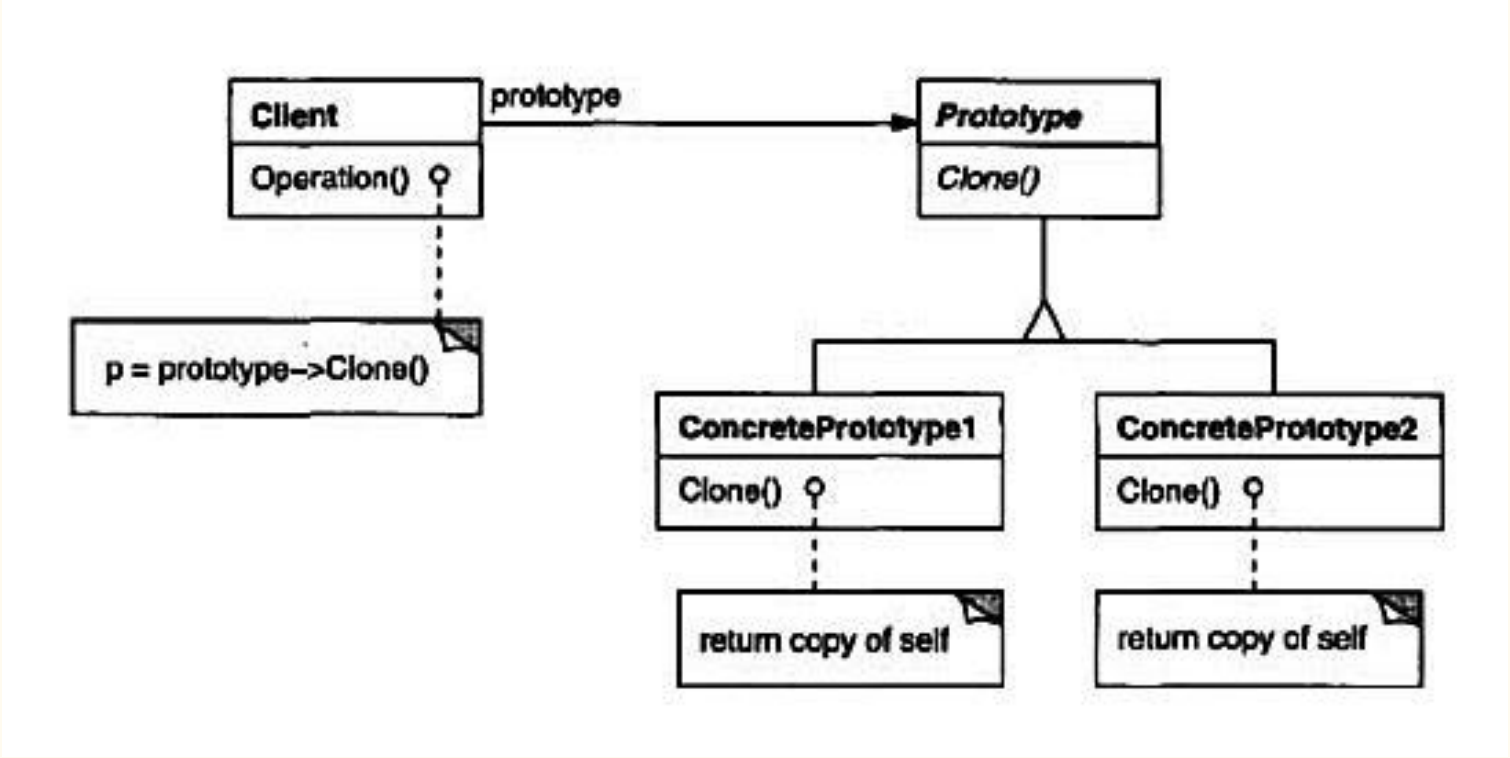
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Motivation

- You have a class you want to instantiate with the same state many times.
- Prototypes allow you to have a template object from which you can instantiate copies of at run time.
- Example: Asteroids!
  - Instantiate instances of asteroids.
  - Instantiate pellets fired by the ship.
- Better example: game framework
  - Tools that instantiate `GameObject` classes don't know how to construct the concrete classes specific to that game.
  - Create a generic factory that creates instances of an abstract `GameObject` class.
  - Users of the framework specify the concrete prototype to instantiate.



Prototype UML



Participants

- **Prototype** (`GameObject`)
  - declares an interface for cloning itself.

- **Concrete Prototype** (`Asteroid`, `Pellet`, `UpgradedPellet`)
  - implements an operation for cloning itself.

- **Client** (`Ship`, `GameObjectFactory`)
  - creates a new object by asking the prototype to clone itself.

Benefits

- By reusing the machinery to instantiate class instances we avoid building a complicated class hierarchy with a dedicated factory for each class we want to instantiate.
- If our class in question has only a small number of valid initial states, we can create a prototype for each and instantiate the appropriate one chosen at run-time.
  - Choose one of a few kinds of asteroids. If they differ only in their shape, we can certainly reuse the same class.
- By changing the object used as the prototype, you can not only change the parameters of the objects instantiated, you can also change the class of them as well.
  - Swap out the prototype for the asteroids with bigger asteroids as the game progresses so as to increase difficulty.
  - Swap out the prototype for the pellets fired by the ship when the player gets a temporary upgrade.

Consequences & Complications

- Hides the concrete product classes, reducing the number of names the client is required to know.
- If the prototypes make use of composition, we can get extremely variable objects created while still not changing the concrete class of the prototype.
- The Prototype pattern is critical for any factory that wants to instantiate classes loaded dynamically at run-time.
- Implementing a `clone` method might be difficult, especially if member variables cannot be copied or if circular or multi-references are involved.
- Clients may want to customize some of the state of the instantiated objects.
  - If the specified parameters are universal to all the classes of the prototype's hierarchy (i.e. location coordinates for a `GameObject`), they can be added as parameters to the `clone` method.
  - If the parameters vary too much from class to class (or even from client to client), the Prototype pattern might not be the best choice.