Creational Design Patterns

Part 5

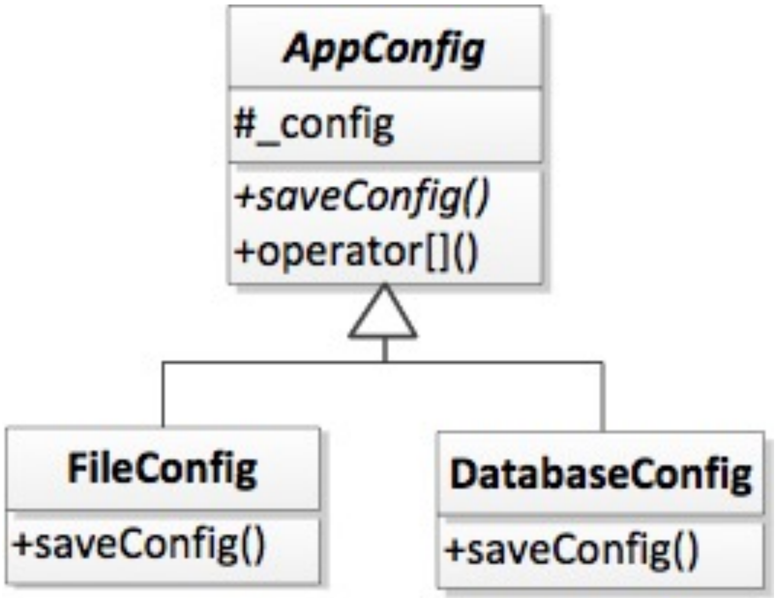
Creational Design Patterns

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Creational Patterns: Prototype

- Suppose we have a set of classes to load our application configuration from a database, a file, etc.
  - Our configuration is large and takes a while to load
  - Sometimes, we must duplicate our configuration objects
    - e.g. We might want to make changes to one configuration object and save it to a different configuration file without changing the original object

Creational Patterns: Prototype



Creational Patterns: Prototype

AppConfig.h

```
class AppConfig
{
   public:
      virtual void saveConfig() = 0;
      const std::string& operator[](const std::string& key)
      {
         return this->_config[key];
      }

   protected:
      std::map<std::string, std::string> _config;
};
```

Creational Patterns: Prototype

DatabaseConfig.cpp

```
DatabaseConfig::DatabaseConfig(const string& hostname, int port, const string& username,
                               const string& password)
{
   // Simulate load of large configuration data from remote database server
   sleep(3 + (rand() % 3));

   // Simulate adding configuration from the file
   this->_config["config_source"] = hostname;

   // ...
}

void DatabaseConfig::saveConfig()
{
   // ...
}
```

Creational Patterns: Prototype

FileConfig.cpp

```
FileConfig::FileConfig(const string& filename)
{
   // Simulate load of large configuration file on remote network share
   sleep(2 + (rand() % 2));

   // Simulate adding configuration from the file
   this->_config["config_source"] = filename;

   // ...
}

void FileConfig::saveConfig()
{
   // ...
}
```

Creational Patterns: Prototype

- Our data takes a long time to load
  - Maybe the configuration data is large
  - Maybe we're accessing a remote file on a network share or data in a database
- Need to clone it from time to time
- Why can't we simply use the copy constructor?

Creational Patterns: Prototype

```
void f(AppConfig* cfg)
{
   // Clone cfg using copy constructor? Nope … AppConfig is an abstract class, so we can’t
   // use a constructor with it …
   AppConfig cfg2(*cfg);
}

int main()
{
   AppConfig* cfg = new FileConfig("app.conf");
   f(cfg);
}
```

Creational Patterns: Prototype

- Copy constructors won’t work

- Instead, we’ll just create a new object and reload the configuration each time we need a “clone”...

Creational Patterns: Prototype

main.cpp

```
AppConfig* loadConfig()
{
  boost::timer::auto_cpu_timer t;

  cout << "Loading config..." << endl;
  return new FileConfig("/mnt/fileserver/app.conf");
}

int main()
{
  AppConfig* cfg1 = loadConfig();
  AppConfig* cfg2 = loadConfig();
}
```

Creational Patterns: Prototype

```
Output
Loading config...
 3.000832s wall
Loading config...
 3.000379s wall
```

- We take an expensive performance hit each time we reload the configuration
- Can we avoid this somehow?

Creational Patterns: Prototype

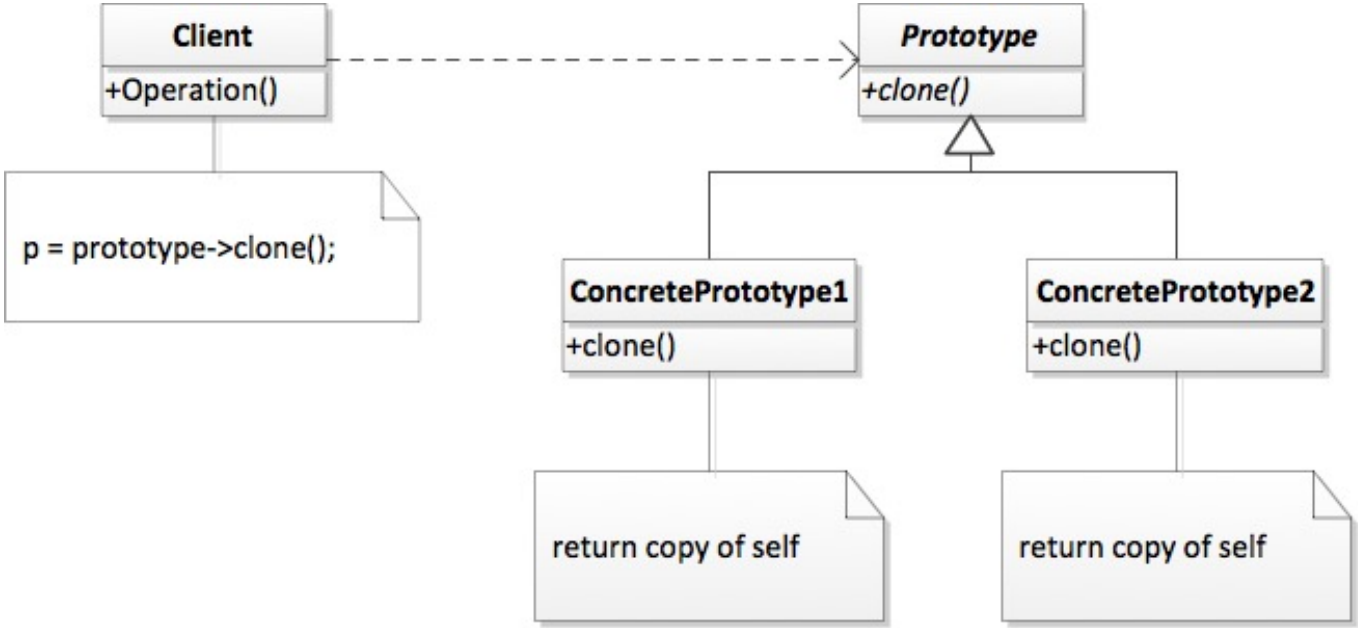
Design Pattern:
Prototype

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

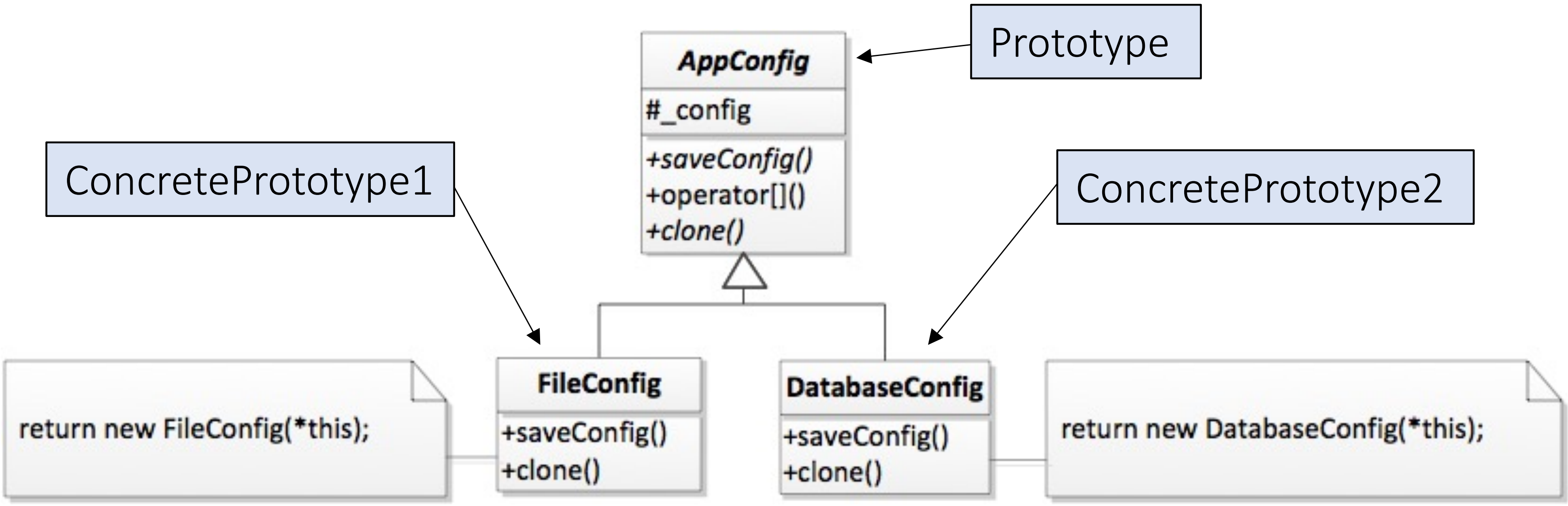
Creational Patterns: Prototype

- Applicability:
  - When the classes to instantiate are specified at run-time, for example, by dynamic loading; or
  - When instances are expensive to create, but easy to copy; or
  - When instances of a class can have one of only a few different combinations of state; in such a case, it may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state

Creational Patterns: Prototype



Creational Patterns: Prototype



Creational Patterns: Prototype

AppConfig.h

```
class AppConfig
{
   public:
      virtual~AppConfig()
      {
      }

      virtual AppConfig* clone() const = 0;
      virtual void saveConfig() = 0;

      const std::string& operator[](const std::string& key)
      {
         return this->_config[key];
      }
   protected:
      std::map<std::string, std::string> _config;
};
```

Creational Patterns: Prototype

DatabaseConfig.cpp

```
AppConfig* DatabaseConfig::clone() const
{
   return new DatabaseConfig(*this);
}
```

Creational Patterns: Prototype

FileConfig.cpp

```
AppConfig* FileConfig::clone() const
{
   return new FileConfig(*this);
}
```

Creational Patterns: Prototype

main.cpp

```
AppConfig* loadConfig()
{
   boost::timer::auto_cpu_timer t;

   cout << "Loading config..." << endl;
   return new FileConfig("/mnt/fileserver/app.conf");
}

int main()
{
   AppConfig* cfg1 = loadConfig();

   boost::timer::auto_cpu_timer t;
   cout << "Cloning config..." << endl;
   AppConfig* cfg2 = cfg1->clone();
}
```

Creational Patterns: Prototype

- Before:

```
Output

Loading config...
 3.000832s wall
Loading config...
 3.000379s wall
```

- After:

```
Output

Loading config...
 3.001179s wall
Cloning config...
 0.000008s wall
```

Creational Patterns: Prototype

- Another example:
  - When creating a game level, we could pass prototypes to use when creating and populating the level

```
GameLevel myLevel(FireMonster, IceSky, GlassWalls, ...)
```

Creational Patterns: Prototype

- Prototype vs Abstract Factory
  - Abstract Factory

```
GameLevel myLevel(FireObjectFactory)
```

    - Creates a family of related products; enforces constraint that they belong together
    - Likely need a factory subclass for each type of level (Fire, Ice, Electric, etc.)
  - Prototype

```
GameLevel myLevel(FireMonster, IceSky, GlassWalls, ...)
```

    - Prototypes allow more flexible mixes of objects
    - May reduce need to have extensive factory hierarchy, especially if there are many different combinations

Creational Patterns: Prototype

- Can use Abstract Factory and Prototype together:

```
Monster* m = new FireMonster();
Wall* w = new IceWall();
Sky* s = new ElectricSky();

ObjectFactory* f = new ObjectFactory(m, w, s);

// ...

// Creates the monster by cloning the
// prototype passed in
Monster* monster = f->createMonster();
```

Creational Patterns: Prototype

- For further flexibility, we could modify our factory to return a random monster from a pool of prototypes:

```
class ObjectFactory
{
   public:
      void addMonsterPrototype(Monster* prototype)
      {
         this->_monsterPrototypes.push_back(prototype);
      }
      Monster* createMonster()
      {
         int idx = random() % this->_monsterPrototypes.size();
         return this->_monsterPrototypes[idx].clone();
      }
   protected:
      std::vector<Monster*> _monsterPrototypes;
};
```

Creational Patterns: Prototype

- Consequences:
  - Hides the concrete product classes from the client – we don’t have to know which concrete type we’re cloning
  - Specify new objects by varying values
  - Configuring an application with classes dynamically
  - Add/remove varieties at run time from a pool of prototypes
  - May reduce need for subclassing
    - Dragons, salamanders, etc. may not have to be subclasses – just generic FireMonsters cloned and then given different characteristics

Creational Patterns: Prototype

- Consequences:
  - May even remove need for Factory subclasses
    - Fire object factory = generic ObjectFactory given several FireMonsters as prototypes
    - Ice object factory = generic ObjectFactory given several IceMonsters as prototypes

Creational Design Patterns

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype