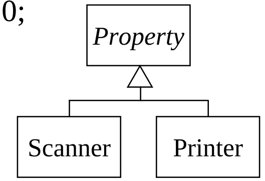
Constructors vs. Creational Design Patterns

C++ Object Oriented Programming
Pei-yih Ting
NTOU CS

Constructors

```
✧ Server  class Property {
           public:
              Property(string name, int pricePaid, Date purchaseDate);
              virtual int currentValue(Date today)=0;
               …
           };
✧ Client    void Store::takeInventory() {
              Property *p[] = {
                     new Printer("HPLJ400", 7000, Date(103,9,1));
                     new Scanner("EpsonA3", 25000, Date(100,7,1));
                            … };
               for (int i=0; i<sizeof(p)/sizeof(Property*); i++)
                  m_totalValue += p[i]->currentValue(Date(104,05,20));
             }
```



Clients are aware of all the server classes, Printer, Scanner, …

Problems with Constructors

- All ctors have the same name with different parameters.
- Ctor always creates a new instance. In some applications, reusing old objects is tidy and more efficient: e.g. object pools
- Ctor returns objects of the specified type, cannot be a subtype.
- Two objects initiated with the same types of arguments have to share the same ctor.
- The clients of the server class were developed/distributed long ago, while the class is evolving to suit the users' timely requirements. For example the classes in many frameworks.
- Using factories instead of constructors allows one to use polymorphism for object creation, not only object use.
- Using factories provides encapsulation: the codes are not tied to specific classes, and server/client class hierarchy or prototypes can be changed or refactored without changing each other.
- In class-based programming, a factory is an abstraction of a ctor.

Static Factory Method

- Joshua Bloch, Effective Java, page 5-7

-
  ```
  class MyClass {
  public:
     static MyClass* getInstance();
  private:
     MyClass();
  };
  MyClass::MyClass() {
        …
  }
  static MyClass* MyClass::getInstance() {
        return new MyClass;
  }
  ```

Advantages:

1. They have names (not just overloaded ctors)
2. Not necessarily create a new instance, maybe an instance pool
3. They can return any subtypes of the specified return types
4. Avoid difficulties when different objects needs to be created with ctor with exactly the same arguments

- This practice is **not** the Factory Method pattern in GoF.
- This practice hides the ctor (with the class name). It is possible that making the ctor public could come back to bite later on. Once it goes public it can no longer be made private without the risk of breaking existing code that has made use of the ctor. Retrofitting a ctor to a static factory method later on may be painful.
- Extremely common in Java cryptography API

Examples

- BigInteger(int, int, Random) returns an integer that is probably a prime (this is not intuitive by observing its parameter)
  Why not use a static factory method BigInteger.probablePrime()?
- Java Crypto

```
 …
SecureRandom random = SecureRandom.getInstance("ECDRBG"); // "JsafeJCE“
 …
KeyPairGenerator keyGen =
      KeyPairGenerator.getInstance("RSA", "SunRsaSign");
 …
Cipher cipher =
   Cipher.getInstance("RSA/ECB/PKCS1Padding", "SunJCE");
cipher.init(Cipher.ENCRYPT_MODE, key.getPublic(), random);
  …
```

Static Factory Method (cont’d)

Is it good to always follow this practice, or only sometimes?

- This is sort of a judgment call area, but a common problem is that developers tend to overdesign their code. While it's true that you should think ahead, you should consider *how likely* it is to require such a change. Then you have to make a judgment call. In general, I would still say that **simpler is better**. This is essentially an argument for YAGNI: en.wikipedia.org/wiki/You_ain't_gonna_need_it
- How to balance between **avoiding overdesign** on the one hand, and **good encapsulation**, that is, minimizing the accessibility of classes and members? That is, the ctor can always be made public, but once public it has to remain public until the end of time.
- It's difficult and only comes with tons of experience. It depends on each individual case, and one need to analyze it as such. It goes back to how you think something will be needed. You should see how your "**guesses**" work out and **adjust accordingly in the future**.

Ctor or Static Factory Method?

- It is an overkill unless you have a specific reason to **hide the actual type being created**, which is basically when you want to use a factory method pattern. In the majority of cases that isn't so. Consider what life would be like if you couldn't construct a String, or a Thread.
- In general, **ctors are simpler** than Factory Methods, so this is a major reason to choose ctors over Factory Method. Use creational patterns when the situation calls for it, not "by default". You should do the simplest thing that solves your problem, and most of the time it would be ctors.
- http://stackoverflow.com/questions/489623/what-is-your-threshold-to-use-factory-instead-of-a-constructor-to-create-an-object

Simple Factory

- Simple factory is **a class with a public static method** which will actually do the object creation task according to the input it gets.
- Ex.

This is one of the pattern not born from GOF, most people consider it a default factory method pattern.

```
class CourseFactory {
public:
   static AbstractCourse *createCourse(char scheduleType) {
      AbstractCourse *objCourse;
      switch (scheduleType) {
         case 'N': objCourse = new NetCourse(); break;
         case 'J': objCourse = new JavaCourse(); break;
      }
      objCourse->createCourseMaterial();   } Special configuration logics
      objCourse->createSchedule();         }
   }
};
```

- The client is further separated from the object creation (as compared with the static factory method): not knowing the specific class name.

Design Patterns

Creational

- ✧ **Abstract factory**
- ✧ **Builder**
- ✧ **Factory method**
- ✧ **Prototype**
- ✧ **Singleton**
- ✧ **Object Pool**

Structural

- ✧ Adapter
- ✧ Bridge
- ✧ Composite
- ✧ Decorator
- ✧ Façade
- ✧ Flyweight
- ✧ Proxy

Behavioral

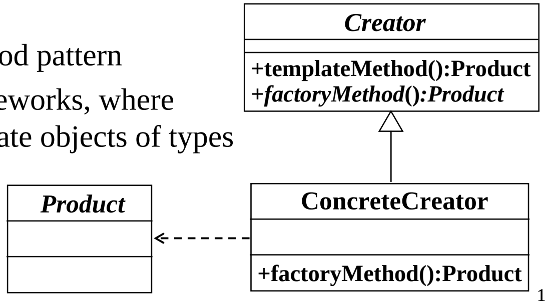
- ✧ Chain of responsibility
- ✧ Command
- ✧ Interpreter
- ✧ Iterator
- ✧ Mediator
- ✧ Memento
- ✧ Observer
- ✧ State
- ✧ Strategy
- ✧ Template Method
- ✧ Visitor

Creational Design Patterns

- ✧ These design patterns are all about **object instantiation**.
  - ✶ **class-creation patterns**: use **inheritance** effectively
  - ✶ **object-creation patterns**: use **delegation** effectively
- ✧ **Abstract Factory**: Creates objects of several families of classes
- ✧ **Builder**: Separate the construction process of a complex object from its representation. The same construction process can create objects with different representations.
- ✧ **Factory Method**: Uniformly create instances for derived classes
- ✧ **Object Pool**: Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- ✧ **Prototype**: Copy a fully initialized instance
- ✧ **Singleton**: A class of which only a single instance can exist

Factory Method Pattern (GoF p.107)

- ✧ A factory method pattern abstracts the **Creation and Configuration** of objects from the client codes that uses the objects.
- ✧ Creating objects (products) without specifying the exact class of object that will be created. The client codes of the products do not need to know which exactly subclass of products they are using.
- ✧ The essence of this pattern is to "Define an interface for creating an object, but let the classes that implement the interface decide which class to instantiate."
- ✧ Using the template-method pattern
- ✧ Used commonly in frameworks, where library code needs to create objects of types that may be subclassed by applications using the framework.

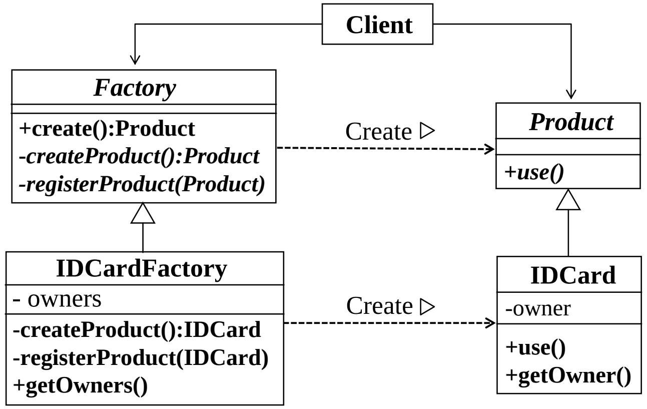


```
class MazeGame {
public:
   MazeGame();
protected:
   virtual Room *makeRoom();
   ...
};

// template method with
// fixed configuration logics
MazeGame::MazeGame() {
   Room *room1 = makeRoom();
   Room *room2 = makeRoom();
   room1.connect(room2);
   addRoom(room1);
   addRoom(room2);
}

// factory method: to be overridden
Room *MazeGame::makeRoom() {
   return new OrdinaryRoom();
```

```
class MagicMazeGame: public MazeGame {
protected:
   Room *makeRoom()
};

// concrete factory method with deferred
// instantiation
Room *MagicMazeGame::makeRoom() {
   return new MagicRoom();
}
```

A maze game may be played in two modes, one with regular rooms that are only connected with adjacent rooms, and one with magic rooms that allow players to be transported at random The regular game mode could use the ***template method***: MazeGame() ctor, which implements some common logic. It refers to the makeRoom() ***factory method*** that encapsulates the creation of rooms such that other rooms can be used in a subclass MagicMazeGame.

IDCard Example



Client

Factory

+create():Product
-createProduct():Product
-registerProduct(Product)

Create ▷

Product

+use()

IDCardFactory

- owners

-createProduct():IDCard
-registerProduct(IDCard)
+getOwners()

Create ▷

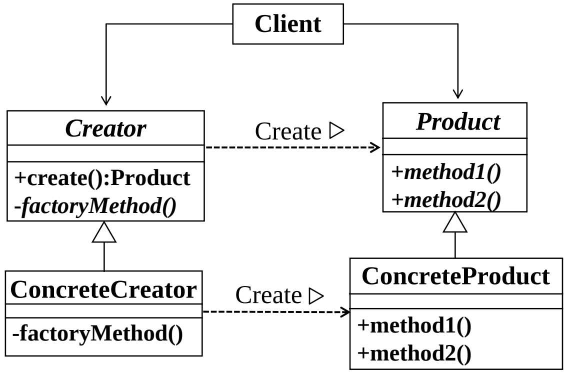
IDCard

-owner

+use()
+getOwner()

Client codes are unaware of/abstrated away from the actual Product class (i.e. IDCard)
The template method Factory::create() is final and specified the configuration logic.
The factory methods Factory::createProduct(), Factory::registerProduct() are deferred to its concrete instantiation IDCardFactory.

Factory Method



Client

Creator

+create():Product
-factoryMethod()

Create ▷

Product

+method1()
+method2()

ConcreteCreator

-factoryMethod()

Create ▷

ConcreteProduct

+method1()
+method2()

Builder

Problem:

- An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.
- E.g. Separate the algorithm for interpreting (i.e. reading and parsing) a stored persistence mechanism (e.g. RTF files) from the algorithm for building and representing one of many target products (e.g. ASCII, TeX, text widget). The focus/distinction is on creating complex aggregates.

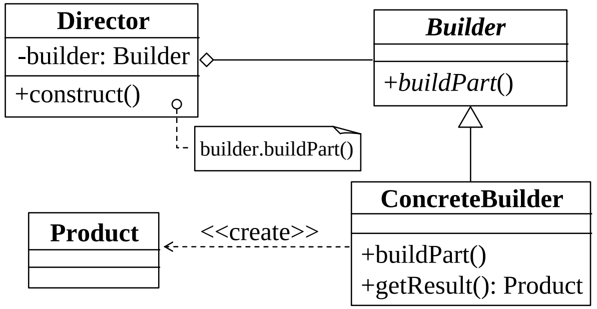
Intents:

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Parse a complex representation, create one of several targets.

Builder (cont’d)

- Structure

constructs the product step by step under the control of the "director"



Director

-builder: Builder

+construct()

builder.buildPart()

Builder

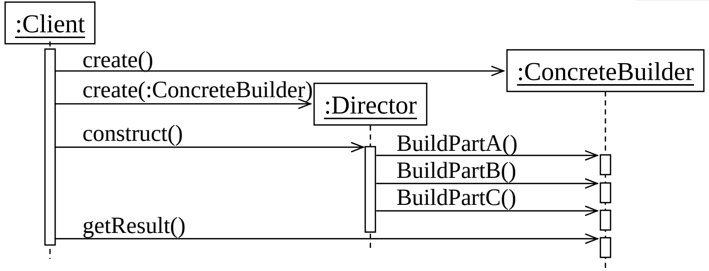
+*buildPart*()

ConcreteBuilder

+buildPart()
+getResult(): Product

Product

<<create>>



:Client

create()

:ConcreteBuilder

create(:ConcreteBuilder)

:Director

construct()

BuildPartA()
BuildPartB()
BuildPartC()

getResult()

Builder Example

- It all depends the **complexity** involved in creation and initialization of object. If they are simple then no need to use factory pattern.
- If its a bit complex (**involving lot of steps in initialization before you use it**) then better go with **Builder** pattern.
- We have a Car class. The problem is that a car has many options. The combination of each option would lead to a huge list of constructors for this class. So we will create a builder class, **CarBuilder**. We will send to the CarBuilder each car's option step by step and then construct the final car with the right options:
- **class Car** can have GPS, trip computer and various numbers of seats, can be a city car, a sports car, or a cabriolet.
- **class CarBuilder has**
  - **method getResult() outputs** a *Car* with the right options constructed
  - **method setSeats(number) inputs** the *number* of seats the car should have, tells the builder the number of seats.

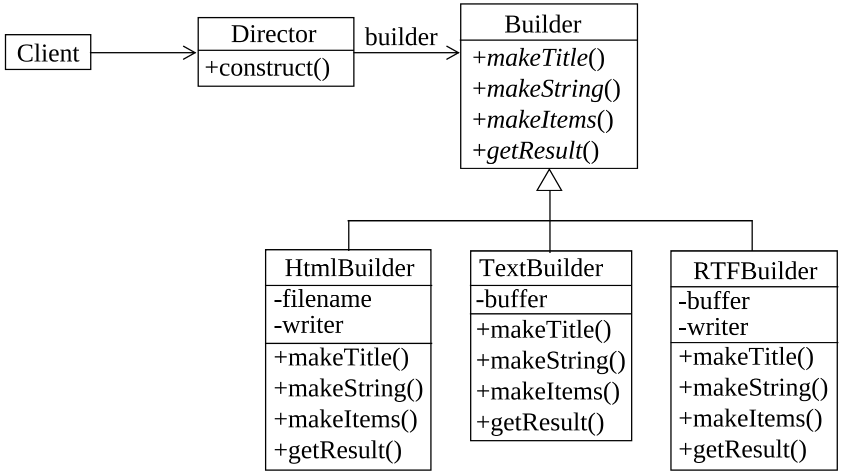
Builder Example (cont'd)

  - **method setCityCar**() makes the builder remember that the car is a city car.
  - **method setCabriolet**() makes the builder remember that the car is a cabriolet.
  - **method setSportsCar**() makes the builder remember that the car is a sports car.
  - **method setTripComputer**() tells the builder that the car has a trip computer.
  - **method unsetTripComputer**() tells the builder that the car does not have a trip computer.
  - **method setGPS**() tells the builder that the car has a global positioning system.
  - **method unsetGPS**() tells the builder that the car does not have a GPS system.
- **Object creation**

```
CarBuilder carBuilder;
carBuilder.setSeats(2);
carBuilder.setSportsCar();
carBuilder.setTripComputer();
carBuilder.unsetGPS();
Car car = carBuilder.getResult()
```

Builder Example (cont'd)

- Typesetting/Rendering html, text, rtf, pdf, documents

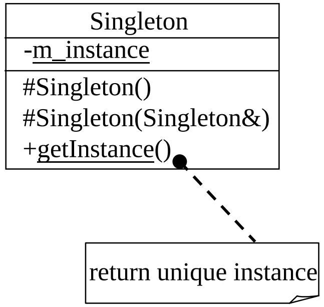


Singleton

- **Problem:** Exactly one instance of a class is allowed. Objects need a global and single point of access.
- **Solution:**
  - Define a static member variable as the sole instance
  - Define non-public constructors to limit the access
  - Define a static method that returns the instance: *getInstance()*
- Used in many patterns like State, Strategy, Factory, Prototype, …

Singleton (cont’d)

```
// Singleton.h
class Singleton {
public:
   static Singleton* getInstance() {
      return instance;
   }
protected:
   Singleton();
   Singleton(const Singleton &src);
   Singleton& operator=(const Singleton &rhs);
private:
   static Singleton* instance;
};
```



Singleton

-m_instance

#Singleton()

#Singleton(Singleton&)

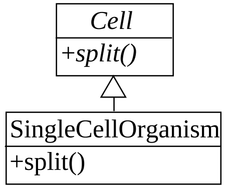
+getInstance()

return unique instance

```
// Singleton.cpp
Singleton *Singleton::instance = new Singleton;

// Client codes
Singleton *p = Singleton::getInstance();
```

Prototype

- Problem:
  - Application "hard wires" the class of object to create in each "new" expression.
  - Create a new instance is expensive.
- Solution:
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - Co-opt one instance of a class for use as a breeder of all future instances.
- Example: When a cell splits, two cells of identical genotvpe result. In other words, the cell clones itself.



Cell

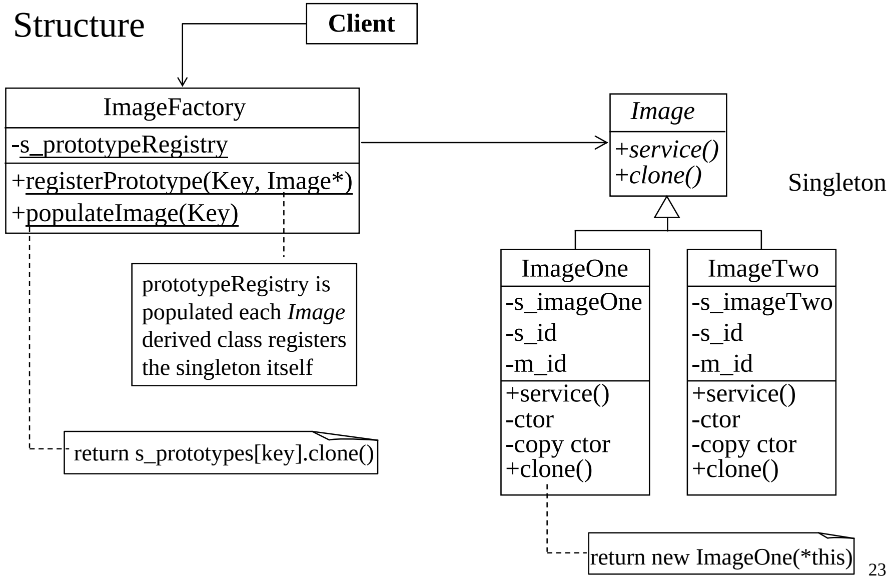
+*split()*

SingleCellOrganism

+split()

Prototype (cont’d)

- Structure



Client

ImageFactory

-s_prototypeRegistry

+registerPrototype(Key, Image*)

+populateImage(Key)

prototypeRegistry is populated each *Image* derived class registers the singleton itself

return s_prototypes[key].clone()

Image

+*service()*

+*clone()*

Singleton

ImageOne

-s_imageOne

-s_id

-m_id

+service()

-ctor

-copy ctor

+clone()

ImageTwo

-s_imageTwo

-s_id

-m_id

+service()

-ctor

-copy ctor

+clone()

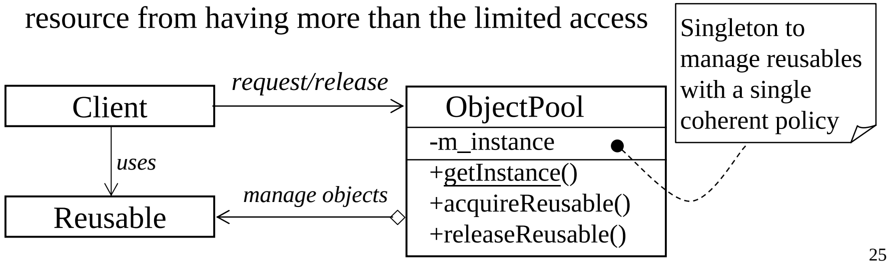
return new ImageOne(*this)

Prototype (cont’d)

- **Prototype** doesn't require sub-classing, but it does require an “initialize” operation. **Factory Method** requires sub-classing, but doesn't require “initialize”.
- Designs that make heavy use of the **Composite** and **Decorator** patterns often can benefit from **Prototype** as well.
- **Prototype** co-opts one instance of a class for use as a breeder of all future instances.
- **Prototypes** are useful when object initialization is expensive, and you anticipate few variations on the initialization parameters. In this context, **Prototype** can avoid expensive "creation from scratch", and support cheap cloning of a pre-initialized prototype.
- **Prototype** is unique among the other creational patterns in that it doesn't require a class – only an object. Object-oriented languages like Self and Omega that do away with classes completely rely on prototypes for creating new objects.

Object Pool

- **Problem:** Manages reuse of objects when a type of object is expensive to create or only limited number needed.
- **Solution:**
  - Create a reusable class to collaborate with other objects for a finite time.
  - Create a reusable pool to manage reusable objects for use by clients**.**
- **Example:** Suppose you have a database system that need to allow only a limited number of accesses to the database at one time. One typical solution is to write a counting semaphore to protect this resource from having more than the limited access



Singleton to manage reusables with a single coherent policy

Object Pool (cont'd)

```
// Reusable.h
class Reusable {
public:
    Reusable();
    void reset();
    int getValue();
    void setValue(int number);
private:
    int m_value;
};
```

```
// Reusable.cpp
#include "Reusable.h"
Reusable::Reusable():m_value(0) {}
void Reusable::reset() {
    m_value=0;
}
int Reusable::getValue() {
    return m_value;
}
void Reusable::setValue(int number) {
    m_value=number;
}
```

```
// ObjectPool.h
class Reusable;
#include <list>
using std::list;
class ObjectPool {
public:
    static ObjectPool& getInstance();
    Reusable *acquireReusable();
    void releaseReusable(Reusable* object);
private:
    list<Reusable*> m_reusables;
    static ObjectPool s_instance;
    ObjectPool();
    ObjectPool(const ObjectPool&);
    ObjectPool& operator=(const ObjectPool&);
    ~ObjectPool();
};
```

Object Pool (cont'd)

```
// ObjectPool.cpp
#include "ObjectPool.h"
#include "Reusable.h"
#include <iostream>
using std::cout;
ObjectPool ObjectPool::s_instance;
ObjectPool::ObjectPool() {}
ObjectPool::~ObjectPool() {
   list<Reusable*>::iterator iter;
   for (iter=m_reusables.begin();
     iter!=m_reusables.end(); iter++)
      delete *iter;
}
ObjectPool &ObjectPool::getInstance() {
   return s_instance;
}
Reusable *ObjectPool::acquireReusable() {
   if (m_reusables.empty()) {
      cout << "Creating new resource\n";
      return new Reusable;
   }
   else {
       cout << "Reusing existing resource\n";
       Reusable* reusable = m_reusables.front();
       m_reusables.pop_front();
       return reusable;
   }
}
void ObjectPool::releaseReusable(Reusable* object) {
   object->reset();
   m_reusables.push_back(object);
}
```

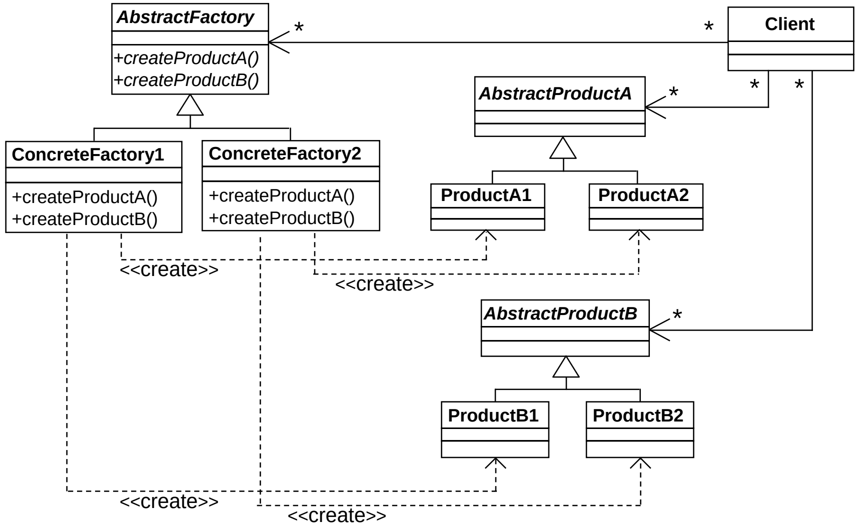
```
// client codes
int main() {
   ObjectPool &pool = ObjectPool::getInstance();
   Reusable *one, *two, *three;
   one = pool.acquireReusable(); one->setValue(10);
   cout << "one=" << one->getValue() << " [" << one << "]\n";
   two = pool.acquireReusable(); two->setValue(20);
   cout << "two=" << two->getValue() << " [" << two << "]\n";
   pool.releaseReusable(one);
   one = pool.acquireReusable();
   cout << "one=" << one->getValue() << " [" << one << "]\n";
   three = pool.acquireReusable(); three->setValue(30);
   cout << "three=" << three->getValue() << " [" << three << "]\n";
   return 0;
}
```

Abstract Factory

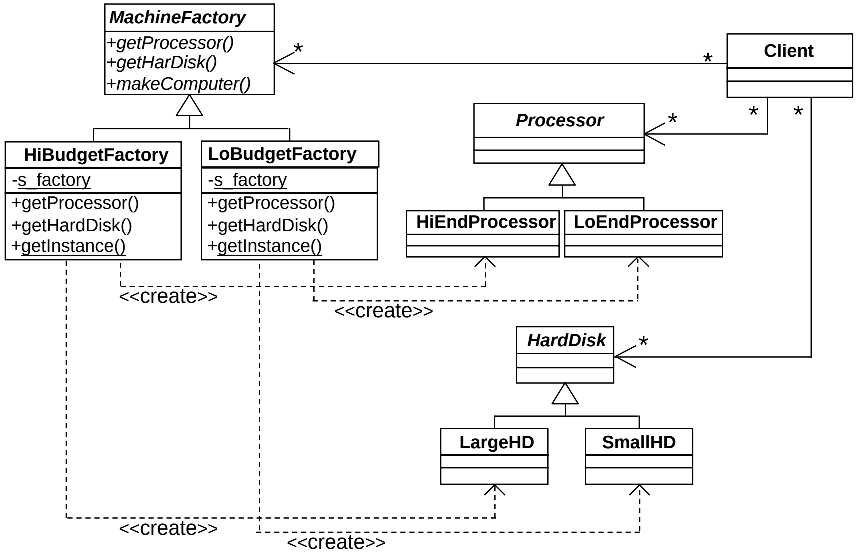
- **Problem**:
  - If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulation is not engineered in advance, and lots of #ifdef case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.
  - The **new** operator considered harmful to the platform transparency requirements.
- **Intents**:
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - A hierarchy that encapsulates many possible "platforms", and the construction of a suite of "products".
  - Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.

Abstract Factory (cont'd)

- Structure



Example: DesktopPC



Applying Creational Patterns

- Sometimes creational patterns are competitors: there are cases when either **Prototype** or **Abstract Factory** could be used properly. At other times they are complementary: **Abstract Factory** might store a set of **Prototypes** from which to clone and return product objects. **Abstract Factory**, **Builder**, and **Prototype** can use **Singleton** in their implementations.
- **Abstract Factory** classes are often implemented with **Factory Methods**, but they can be implemented using **Prototype**.
- **Factory Method**: creation through inheritance.
  Prototype: creation through delegation.
- Often, designs start out using **Factory Method** (less complicated, more customizable, subclasses proliferate) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, more complex) as the designer discovers where more flexibility is needed.

Applying Creational Patterns (cont'd)

- **Abstract Factory**, **Builder**, and **Prototype** define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system.
- **Abstract Factory** has the factory object producing objects of several classes.
- **Builder** has the factory object building a complex product incrementally using a correspondingly complex protocol.
- **Prototype** has the factory object (aka prototype) building a product by copying a prototype object.