Creational patterns

- Abstract Factory
- Builder
- Factory Method
- Object Pool
- Prototype
- Singleton

Abstract Factory

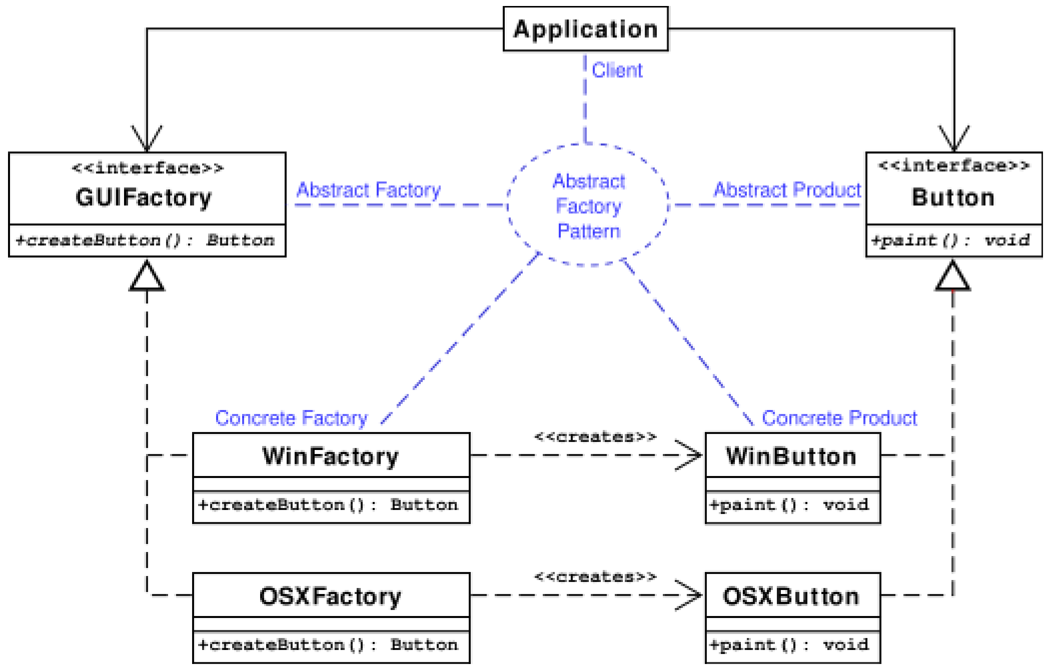
Problem

If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Encapsulation must work without if-else cases.

Solution

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.

Diagram



Application
Client
<<interface>>
GUIFactory
+createButton(): Button
Abstract Factory
Abstract Factory Pattern
Abstract Product
<<interface>>
Button
+paint(): void
Concrete Factory
Concrete Product
WinFactory
+createButton(): Button
<<creates>>
WinButton
+paint(): void
OSXFactory
+createButton(): Button
<<creates>>
OSXButton
+paint(): void

Builder

Problem

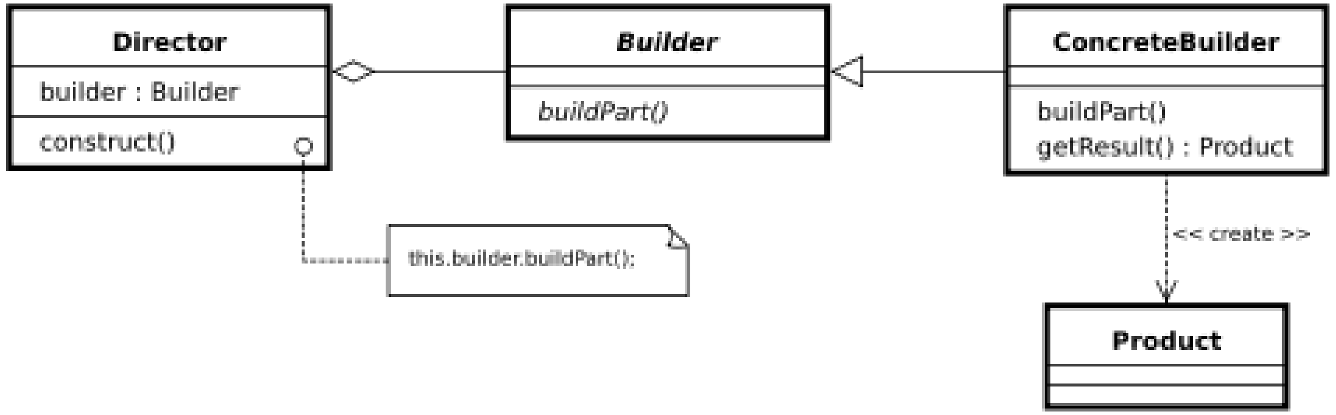
An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

Solution

Separate the construction of a complex object from its representation. Construct the product step by step under the control of the "director".

The "director" invokes "builder" services as it interprets the external format. The "builder" creates part of the complex object each time it is called and maintains all intermediate state. When the product is finished, the client retrieves the result from the "builder".

Diagram


Director
builder : Builder
construct()
Builder
buildPart()
ConcreteBuilder
buildPart()
getResult() : Product
this.builder.buildPart();
<< create >>
Product

Factory method

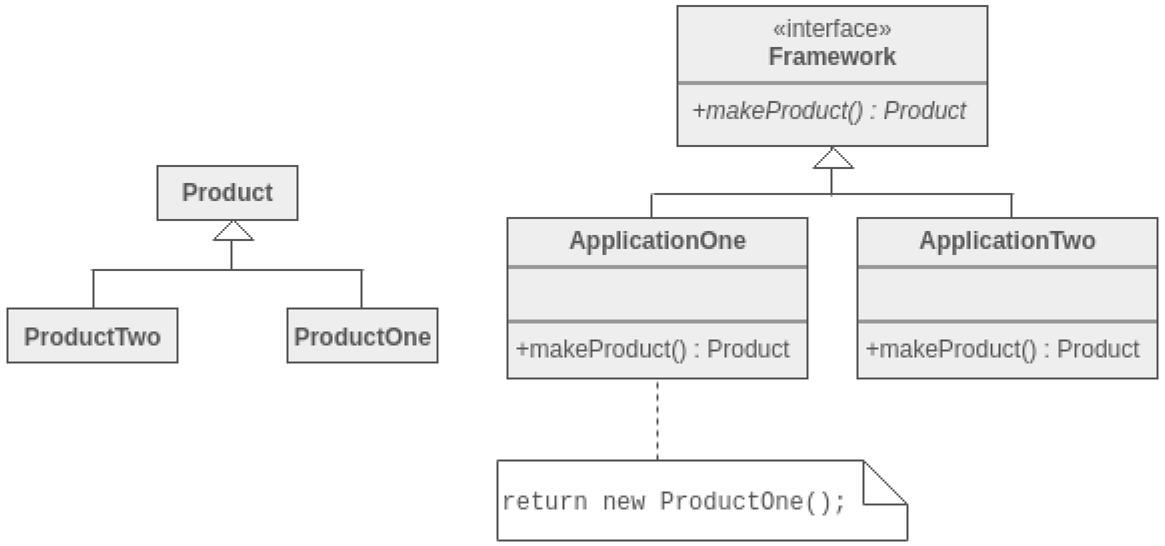
Problem

A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

Solution

Specify all standard and generic behavior in a superclass (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

Diagram


Product
ProductTwo
ProductOne
«interface»
Framework
+makeProduct() : Product
ApplicationOne
+makeProduct() : Product
ApplicationTwo
+makeProduct() : Product
return new ProductOne();

Object pool

Problem

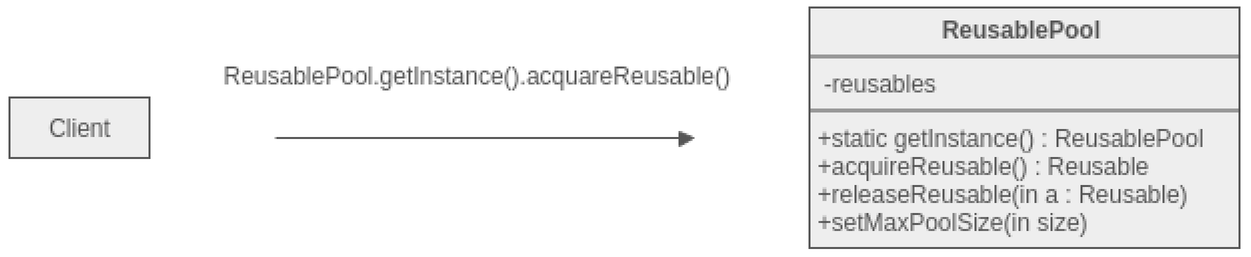
Reuse objects instead of deleting and creating new.

Solution

Ask the pool for the object that has already been instantiated instead.

Grow pool if no more objects or restrict the number of objects created.

Diagram


Client
ReusablePool.getInstance().acquareReusable()
ReusablePool
-reusables
+static getInstance() : ReusablePool
+acquireReusable() : Reusable
+releaseReusable(in a : Reusable)
+setMaxPoolSize(in size)

Prototype

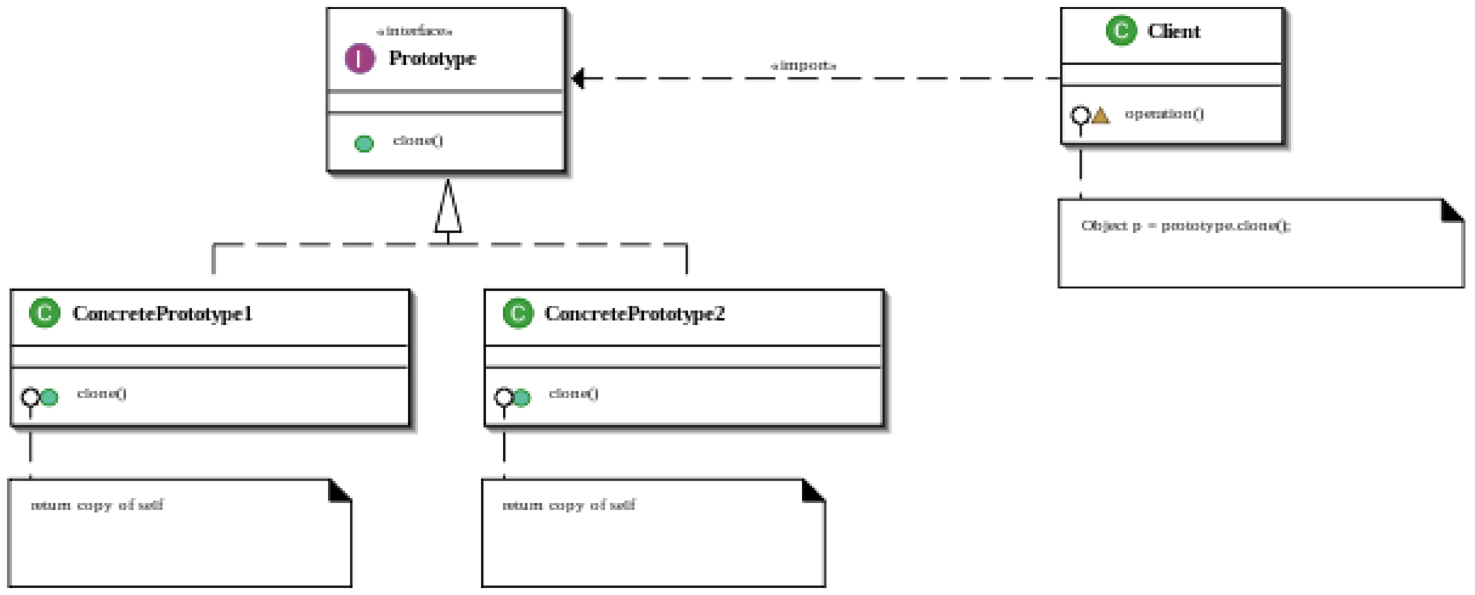
Problem

Avoid the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) if it is prohibitively expensive for a given application.

Solution

Declare an abstract base class that specifies a "clone" method, and, maintains a dictionary of all "cloneable" concrete derived classes. Any class that needs a "polymorphic constructor" capability: derives itself from the abstract base class, registers its prototypical instance, and implements the clone() operation.

Diagram


«interface»
Prototype
clone()
Client
operation()
«import»
Object p = prototype.clone();
ConcretePrototype1
clone()
ConcretePrototype2
clone()
return copy of self
return copy of self

Singleton

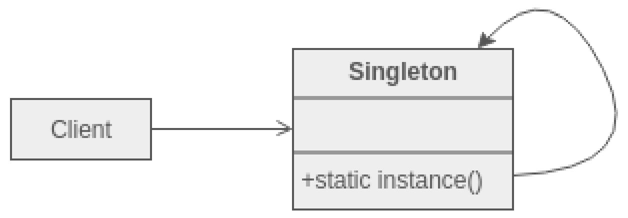
Problem

Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Solution

Make the class of the single instance object responsible for creation, initialization, access, and enforcement. Declare the instance as a private static data member. Provide a public static member function that encapsulates all initialization code, and provides access to the instance.

Diagram



Client
Singleton
+static instance()