**Design Patterns**

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

**Uses of Design Patterns**

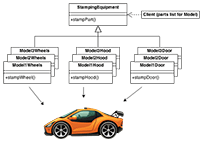
Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

[**Creational design patterns**](https://sourcemaking.com/design_patterns/creational_patterns)

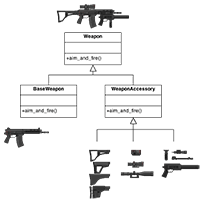
These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

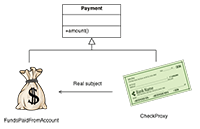
[](https://sourcemaking.com/design_patterns/abstract_factory)

* [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory)  
  Creates an instance of several families of classes
* [**Builder**](https://sourcemaking.com/design_patterns/builder)  
  Separates object construction from its representation
* [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method)  
  Creates an instance of several derived classes
* [**Object Pool**](https://sourcemaking.com/design_patterns/object_pool)  
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
* [**Prototype**](https://sourcemaking.com/design_patterns/prototype)  
  A fully initialized instance to be copied or cloned
* [**Singleton**](https://sourcemaking.com/design_patterns/singleton)  
  A class of which only a single instance can exist

[**Structural design patterns**](https://sourcemaking.com/design_patterns/structural_patterns)

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

[](https://sourcemaking.com/design_patterns/decorator)

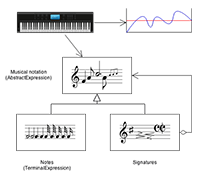
* [**Adapter**](https://sourcemaking.com/design_patterns/adapter)  
  Match interfaces of different classes
* [**Bridge**](https://sourcemaking.com/design_patterns/bridge)  
  Separates an object’s interface from its implementation
* [**Composite**](https://sourcemaking.com/design_patterns/composite)  
  A tree structure of simple and composite objects
* [**Decorator**](https://sourcemaking.com/design_patterns/decorator)  
  Add responsibilities to objects dynamically
* [**Facade**](https://sourcemaking.com/design_patterns/facade)  
  A single class that represents an entire subsystem
* [**Flyweight**](https://sourcemaking.com/design_patterns/flyweight)  
  A fine-grained instance used for efficient sharing
* **[](https://sourcemaking.com/design_patterns/proxy)**

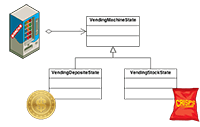
[**Private Class Data**](https://sourcemaking.com/design_patterns/private_class_data)  
Restricts accessor/mutator access

* [**Proxy**](https://sourcemaking.com/design_patterns/proxy)  
  An object representing another object

[**Behavioral design patterns**](https://sourcemaking.com/design_patterns/behavioral_patterns)

These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

[](https://sourcemaking.com/design_patterns/interpreter)

* [**Chain of responsibility**](https://sourcemaking.com/design_patterns/chain_of_responsibility)  
  A way of passing a request between a chain of objects
* [**Command**](https://sourcemaking.com/design_patterns/command)  
  Encapsulate a command request as an object
* [**Interpreter**](https://sourcemaking.com/design_patterns/interpreter)  
  A way to include language elements in a program
* [**Iterator**](https://sourcemaking.com/design_patterns/iterator)  
  Sequentially access the elements of a collection
* [**Mediator**](https://sourcemaking.com/design_patterns/mediator)  
  Defines simplified communication between classes
* [**Memento**](https://sourcemaking.com/design_patterns/memento)  
  Capture and restore an object's internal state
* [**Null Object**](https://sourcemaking.com/design_patterns/null_object)  
  Designed to act as a default value of an object
* [**Observer**](https://sourcemaking.com/design_patterns/observer)  
  A way of notifying change to a number of classes
* **[](https://sourcemaking.com/design_patterns/state)**

[**State**](https://sourcemaking.com/design_patterns/state)  
Alter an object's behavior when its state changes

* [**Strategy**](https://sourcemaking.com/design_patterns/strategy)  
  Encapsulates an algorithm inside a class
* [**Template method**](https://sourcemaking.com/design_patterns/template_method)  
  Defer the exact steps of an algorithm to a subclass
* [**Visitor**](https://sourcemaking.com/design_patterns/visitor)  
  Defines a new operation to a class without change

**Criticism**

The concept of design patterns has been criticized by some in the field of computer science.

**Targets the wrong problem**

The need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. Paul Graham writes in the essay [**Revenge of the Nerds**](http://www.paulgraham.com/icad.html).

Peter Norvig provides a similar argument. He demonstrates that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.

**Lacks formal foundations**

The study of design patterns has been excessively ad hoc, and some have argued that the concept sorely needs to be put on a more formal footing. At OOPSLA 1999, the Gang of Four were (with their full cooperation) subjected to a show trial, in which they were "charged" with numerous crimes against computer science. They were "convicted" by ⅔ of the "jurors" who attended the trial.

**Leads to inefficient solutions**

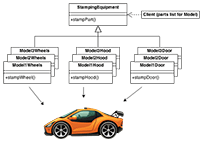
The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code. It is almost always a more efficient solution to use a well-factored implementation rather than a "just barely good enough" design pattern.

**Does not differ significantly from other abstractions**

Some authors allege that design patterns don't differ significantly from other forms of abstraction, and that the use of new terminology (borrowed from the architecture community) to describe existing phenomena in the field of programming is unnecessary. The Model-View-Controller paradigm is touted as an example of a "pattern" which predates the concept of "design patterns" by several years. It is further argued by some that the primary contribution of the Design Patterns community (and the Gang of Four book) was the use of Alexander's pattern language as a form of documentation; a practice which is often ignored in the literature.

# Creational patterns

In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

[](https://sourcemaking.com/design_patterns/abstract_factory)

* [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory)  
  Creates an instance of several families of classes
* [**Builder**](https://sourcemaking.com/design_patterns/builder)  
  Separates object construction from its representation
* [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method)  
  Creates an instance of several derived classes
* [**Object Pool**](https://sourcemaking.com/design_patterns/object_pool)  
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
* [**Prototype**](https://sourcemaking.com/design_patterns/prototype)  
  A fully initialized instance to be copied or cloned
* [**Singleton**](https://sourcemaking.com/design_patterns/singleton)  
  A class of which only a single instance can exist

## Rules of thumb

1. Sometimes creational patterns are competitors: there are cases when either [**Prototype**](https://sourcemaking.com/design_patterns/prototype) or [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory) could be used profitably. At other times they are complementary: [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory) might store a set of [**Prototypes**](https://sourcemaking.com/design_patterns/prototype) from which to clone and return product objects, [**Builder**](https://sourcemaking.com/design_patterns/builder) can use one of the other patterns to implement which components get built. [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory), [**Builder**](https://sourcemaking.com/design_patterns/builder), and [**Prototype**](https://sourcemaking.com/design_patterns/prototype) can use [**Singleton**](https://sourcemaking.com/design_patterns/singleton) in their implementation.
2. [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory), [**Builder**](https://sourcemaking.com/design_patterns/builder), and [**Prototype**](https://sourcemaking.com/design_patterns/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**](https://sourcemaking.com/design_patterns/abstract_factory) has the factory object producing objects of several classes. [**Builder**](https://sourcemaking.com/design_patterns/builder) has the factory object building a complex product incrementally using a correspondingly complex protocol. [**Prototype**](https://sourcemaking.com/design_patterns/prototype) has the factory object (aka prototype) building a product by copying a prototype object.
3. [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory) classes are often implemented with [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method)s, but they can also be implemented using [**Prototype**](https://sourcemaking.com/design_patterns/prototype).
4. [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory) can be used as an alternative to [**Facade**](https://sourcemaking.com/design_patterns/facade) to hide platform-specific classes.
5. [**Builder**](https://sourcemaking.com/design_patterns/builder) focuses on constructing a complex object step by step. [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory) emphasizes a family of product objects (either simple or complex). [**Builder**](https://sourcemaking.com/design_patterns/builder) returns the product as a final step, but as far as the [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory) is concerned, the product gets returned immediately.
6. [**Builder**](https://sourcemaking.com/design_patterns/builder) is to creation as [**Strategy**](https://sourcemaking.com/design_patterns/strategy) is to algorithm.
7. [**Builder**](https://sourcemaking.com/design_patterns/builder) often builds a [**Composite**](https://sourcemaking.com/design_patterns/composite).
8. [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method)s are usually called within [**Template method**](https://sourcemaking.com/design_patterns/template_method)s.
9. [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method): creation through inheritance. [**Prototype**](https://sourcemaking.com/design_patterns/prototype): creation through delegation.
10. Often, designs start out using [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method) (less complicated, more customizable, subclasses proliferate) and evolve toward [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory), [**Prototype**](https://sourcemaking.com/design_patterns/prototype), or [**Builder**](https://sourcemaking.com/design_patterns/builder) (more flexible, more complex) as the designer discovers where more flexibility is needed.
11. [**Prototype**](https://sourcemaking.com/design_patterns/prototype) doesn't require subclassing, but it does require an Initialize operation. [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method) requires subclassing, but doesn't require Initialize.
12. Designs that make heavy use of the [**Composite**](https://sourcemaking.com/design_patterns/composite) and [**Decorator**](https://sourcemaking.com/design_patterns/decorator) patterns often can benefit from [**Prototype**](https://sourcemaking.com/design_patterns/prototype) as well.

**Abstract Factory Design Pattern**

**Intent**

* 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".
* The new operator considered harmful.

**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.

**Discussion**

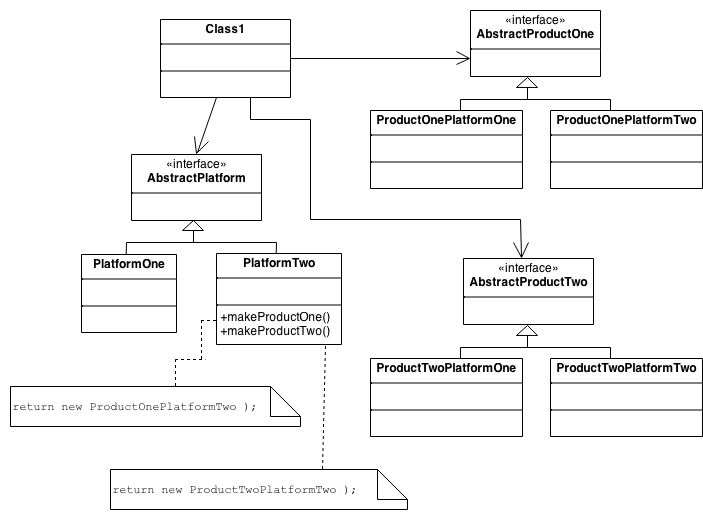
Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.

This mechanism makes exchanging product families easy because the specific class of the factory object appears only once in the application - where it is instantiated. The application can wholesale replace the entire family of products simply by instantiating a different concrete instance of the abstract factory.

Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton.

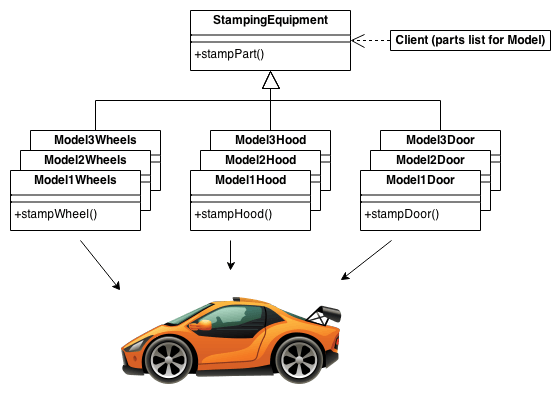
**Structure**

The Abstract Factory defines a Factory Method per product. Each Factory Method encapsulates the new operator and the concrete, platform-specific, product classes. Each "platform" is then modeled with a Factory derived class.



**Example**

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes.



**Check list**

1. Decide if "platform independence" and creation services are the current source of pain.
2. Map out a matrix of "platforms" versus "products".
3. Define a factory interface that consists of a factory method per product.
4. Define a factory derived class for each platform that encapsulates all references to the new operator.
5. The client should retire all references to new, and use the factory methods to create the product objects.

**Rules of thumb**

* Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably. At other times they are complementary: Abstract Factory might store a set of Prototypes from which to clone and return product objects, Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
* 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.
* Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype.
* Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.
* Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
* 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.

**Builder Design Pattern**

**Intent**

* 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.

**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.

**Discussion**

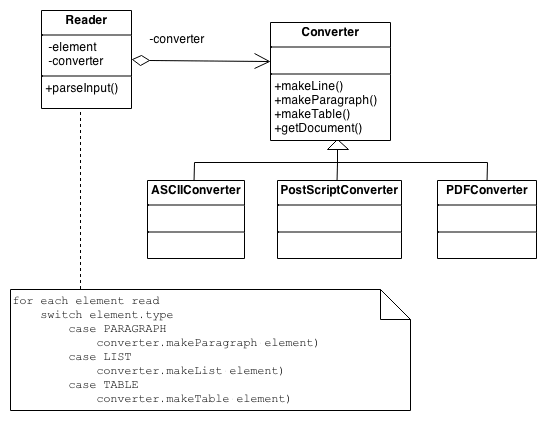
Separate the algorithm for interpreting (i.e. reading and parsing) a stored persistence mechanism (e.g. RTF files) from the algorithm for building and representing one of many target products (e.g. ASCII, TeX, text widget). The focus/distinction is on creating complex aggregates.

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".

Affords finer control over the construction process. Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the control of the "director".

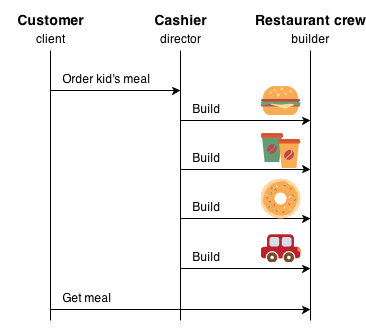
**Structure**

The Reader encapsulates the parsing of the common input. The Builder hierarchy makes possible the polymorphic creation of many peculiar representations or targets.



**Example**

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations. This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, Coke, and toy dinosaur). Note that there can be variation in the content of the children's meal, but the construction process is the same. Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants.



**Check list**

1. Decide if a common input and many possible representations (or outputs) is the problem at hand.
2. Encapsulate the parsing of the common input in a Reader class.
3. Design a standard protocol for creating all possible output representations. Capture the steps of this protocol in a Builder interface.
4. Define a Builder derived class for each target representation.
5. The client creates a Reader object and a Builder object, and registers the latter with the former.
6. The client asks the Reader to "construct".
7. The client asks the Builder to return the result.

**Rules of thumb**

* Sometimes creational patterns are complementary: Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.
* Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
* Builder often builds a Composite.
* 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.

**Factory Method Design Pattern**

**Intent**

* Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
* Defining a "virtual" constructor.
* The new operator considered harmful.

**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.

**Discussion**

Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.

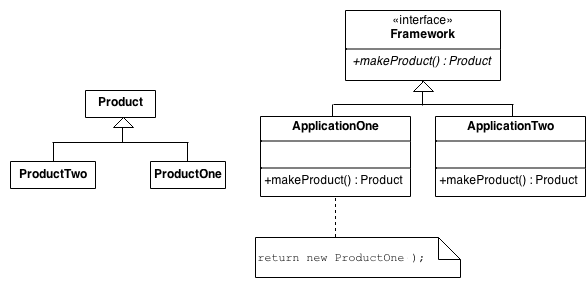
People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation).

Factory Method is similar to Abstract Factory but without the emphasis on families.

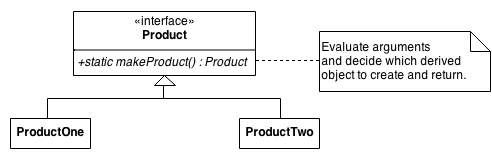
Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework.

**Structure**

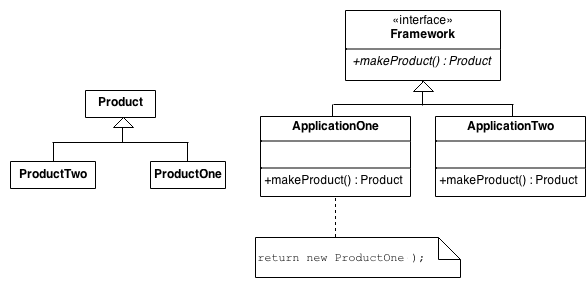
The implementation of Factory Method discussed in the Gang of Four (below) largely overlaps with that of Abstract Factory. For that reason, the presentation in this chapter focuses on the approach that has become popular since.



An increasingly popular definition of factory method is: a static method of a class that returns an object of that class' type. But unlike a constructor, the actual object it returns might be an instance of a subclass. Unlike a constructor, an existing object might be reused, instead of a new object created. Unlike a constructor, factory methods can have different and more descriptive names (e.g. Color.make\_RGB\_color(float red, float green, float blue) and Color.make\_HSB\_color(float hue, float saturation, float brightness)

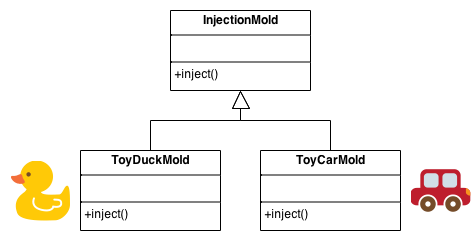


The client is totally decoupled from the implementation details of derived classes. Polymorphic creation is now possible.



**Example**

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.



**Check list**

1. If you have an inheritance hierarchy that exercises polymorphism, consider adding a polymorphic creation capability by defining a static factory method in the base class.
2. Design the arguments to the factory method. What qualities or characteristics are necessary and sufficient to identify the correct derived class to instantiate?
3. Consider designing an internal "object pool" that will allow objects to be reused instead of created from scratch.
4. Consider making all constructors private or protected.

**Rules of thumb**

* Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype.
* Factory Methods are usually called within Template Methods.
* 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.
* Prototype doesn't require subclassing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize.
* The advantage of a Factory Method is that it can return the same instance multiple times, or can return a subclass rather than an object of that exact type.
* Some Factory Method advocates recommend that as a matter of language design (or failing that, as a matter of style) absolutely all constructors should be private or protected. It's no one else's business whether a class manufactures a new object or recycles an old one.
* The new operator considered harmful. There is a difference between requesting an object and creating one. The new operator always creates an object, and fails to encapsulate object creation. A Factory Method enforces that encapsulation, and allows an object to be requested without inextricable coupling to the act of creation.

**Object Pool Design Pattern**

**Intent**

Object pooling can offer a significant performance boost; it is most effective in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instantiations in use at any one time is low.

**Problem**

Object pools (otherwise known as resource pools) are used to manage the object caching. A client with access to a Object pool can avoid creating a new Objects by simply asking the pool for one that has already been instantiated instead. Generally the pool will be a growing pool, i.e. the pool itself will create new objects if the pool is empty, or we can have a pool, which restricts the number of objects created.

It is desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the Reusable Pool class is designed to be a singleton class.

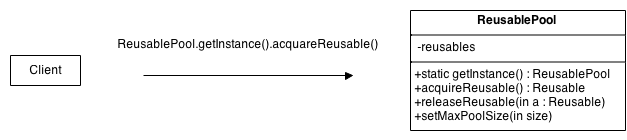
**Discussion**

The Object Pool lets others "check out" objects from its pool, when those objects are no longer needed by their processes, they are returned to the pool in order to be reused.

However, we don't want a process to have to wait for a particular object to be released, so the Object Pool also instantiates new objects as they are required, but must also implement a facility to clean up unused objects periodically.

**Structure**

The general idea for the Connection Pool pattern is that if instances of a class can be reused, you avoid creating instances of the class by reusing them.



* **Reusable** - Instances of classes in this role collaborate with other objects for a limited amount of time, then they are no longer needed for that collaboration.
* **Client** - Instances of classes in this role use Reusable objects.
* **ReusablePool** - Instances of classes in this role manage Reusable objects for use by Client objects.

Usually, it is desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the ReusablePool class is designed to be a singleton class. Its constructor(s) are private, which forces other classes to call its getInstance method to get the one instance of the ReusablePool class.

A Client object calls a ReusablePool object's acquireReusable method when it needs a Reusable object. A ReusablePool object maintains a collection of Reusable objects. It uses the collection of Reusable objects to contain a pool of Reusable objects that are not currently in use.

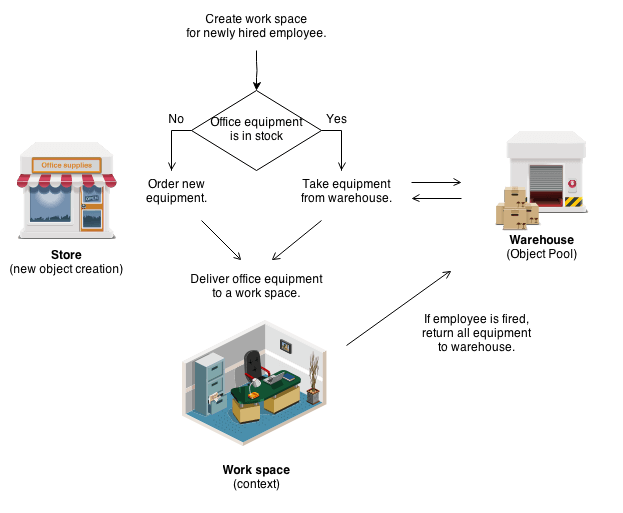
If there are any Reusable objects in the pool when the acquireReusable method is called, it removes a Reusable object from the pool and returns it. If the pool is empty, then the acquireReusable method creates a Reusable object if it can. If the acquireReusable method cannot create a new Reusable object, then it waits until a Reusable object is returned to the collection.

Client objects pass a Reusable object to a ReusablePool object's releaseReusable method when they are finished with the object. The releaseReusable method returns a Reusable object to the pool of Reusable objects that are not in use.

In many applications of the Object Pool pattern, there are reasons for limiting the total number of Reusable objects that may exist. In such cases, the ReusablePool object that creates Reusable objects is responsible for not creating more than a specified maximum number of Reusable objects. If ReusablePool objects are responsible for limiting the number of objects they will create, then the ReusablePool class will have a method for specifying the maximum number of objects to be created. That method is indicated in the above diagram as setMaxPoolSize.

**Example**

Object pool pattern is similar to an office warehouse. When a new employee is hired, office manager has to prepare a work space for him. She figures whether or not there's a spare equipment in the office warehouse. If so, she uses it. If not, she places an order to purchase new equipment from Amazon. In case if an employee is fired, his equipment is moved to warehouse, where it could be taken when new work place will be needed.



**Check list**

1. Create ObjectPool class with private array of Objects inside
2. Create acquire and release methods in ObjectPool class
3. Make sure that your ObjectPool is Singleton

**Rules of thumb**

* The Factory Method pattern can be used to encapsulate the creation logic for objects. However, it does not manage them after their creation, the object pool pattern keeps track of the objects it creates.
* Object Pools are usually implemented as Singletons.

**Prototype Design Pattern**

**Intent**

* 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.
* The new operator considered harmful.

**Problem**

Application "hard wires" the class of object to create in each "new" expression.

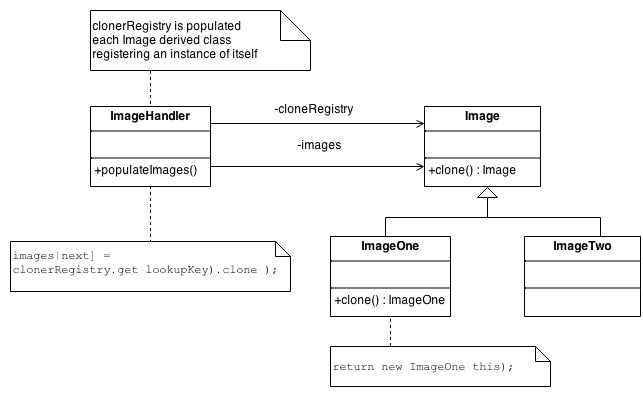
**Discussion**

Declare an abstract base class that specifies a pure virtual "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.

The client then, instead of writing code that invokes the "new" operator on a hard-wired class name, calls a "clone" operation on the abstract base class, supplying a string or enumerated data type that designates the particular concrete derived class desired.

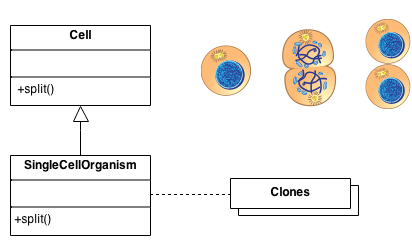
**Structure**

The Factory knows how to find the correct Prototype, and each Product knows how to spawn new instances of itself.



**Example**

The Prototype pattern specifies the kind of objects to create using a prototypical instance. Prototypes of new products are often built prior to full production, but in this example, the prototype is passive and does not participate in copying itself. The mitotic division of a cell - resulting in two identical cells - is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern. When a cell splits, two cells of identical genotype result. In other words, the cell clones itself.



**Check list**

1. Add a clone() method to the existing "product" hierarchy.
2. Design a "registry" that maintains a cache of prototypical objects. The registry could be encapsulated in a new Factory class, or in the base class of the "product" hierarchy.
3. Design a factory method that: may (or may not) accept arguments, finds the correct prototype object, calls clone() on that object, and returns the result.
4. The client replaces all references to the new operator with calls to the factory method.

**Rules of thumb**

* 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.
* Prototype doesn't require subclassing, but it does require an "initialize" operation. Factory Method requires subclassing, 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.

**Singleton Design Pattern**

**Intent**

* Ensure a class has only one instance, and provide a global point of access to it.
* Encapsulated "just-in-time initialization" or "initialization on first use".

**Problem**

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

**Discussion**

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.

The client calls the accessor function (using the class name and scope resolution operator) whenever a reference to the single instance is required.

Singleton should be considered only if all three of the following criteria are satisfied:

* Ownership of the single instance cannot be reasonably assigned
* Lazy initialization is desirable
* Global access is not otherwise provided for

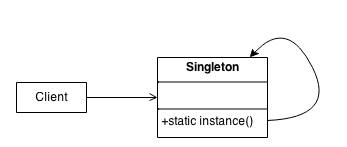
If ownership of the single instance, when and how initialization occurs, and global access are not issues, Singleton is not sufficiently interesting.

The Singleton pattern can be extended to support access to an application-specific number of instances.

The "static member function accessor" approach will not support subclassing of the Singleton class. If subclassing is desired, refer to the discussion in the book.

Deleting a Singleton class/instance is a non-trivial design problem. See "[**To Kill A Singleton**](http://sourcemaking.com/design_patterns/to_kill_a_singleton)" by John Vlissides for a discussion.

**Structure**

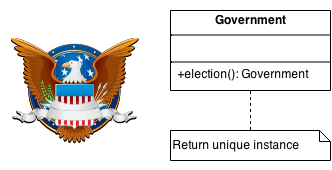


Make the class of the single instance responsible for access and "initialization on first use". The single instance is a private static attribute. The accessor function is a public static method.



**Example**

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a Singleton. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office.



**Check list**

1. Define a private static attribute in the "single instance" class.
2. Define a public static accessor function in the class.
3. Do "lazy initialization" (creation on first use) in the accessor function.
4. Define all constructors to be protected or private.
5. Clients may only use the accessor function to manipulate the Singleton.

**Rules of thumb**

* Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
* Facade objects are often Singletons because only one Facade object is required.
* State objects are often Singletons.
* The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton, and, you can change your mind and manage any number of instances.
* The Singleton design pattern is one of the most inappropriately used patterns. Singletons are intended to be used when a class must have exactly one instance, no more, no less. Designers frequently use Singletons in a misguided attempt to replace global variables. A Singleton is, for intents and purposes, a global variable. The Singleton does not do away with the global; it merely renames it.
* When is Singleton unnecessary? Short answer: most of the time. Long answer: when it's simpler to pass an object resource as a reference to the objects that need it, rather than letting objects access the resource globally. The real problem with Singletons is that they give you such a good excuse not to think carefully about the appropriate visibility of an object. Finding the right balance of exposure and protection for an object is critical for maintaining flexibility.
* Our group had a bad habit of using global data, so I did a study group on Singleton. The next thing I know Singletons appeared everywhere and none of the problems related to global data went away. The answer to the global data question is not, "Make it a Singleton." The answer is, "Why in the hell are you using global data?" Changing the name doesn't change the problem. In fact, it may make it worse because it gives you the opportunity to say, "Well I'm not doing that, I'm doing this" – even though this and that are the same thing.