Intent

Ensure that a class only has one instance, and provide a global point of access to it

- For some classes it is important to have exactly one instance
  - The should be only one window manager in the system
- Of course, the same can be achieved with a global variable
- However, for complex system we could run in some problems
  - the initialization order
  - the object is created many times by mistake, etc.
- A better solution is to make the class itself responsible for creating and maintaining the instance
Code

// include file
class SysParams {
    static SysParams *inst;
    // other non-static members;
    SysParams();
    SysParams(const SysParams &);
public:
    static SysParams &getInstance();
    // other non static members
};

pointer to the only instance
constructor made private
copy constructor hidden and not implemented

Lazy initialization

Code implementation

// src (cpp) file
SysParams *SysParams::inst = 0;

SysParams & SysParams::getInstance()
{
    if (inst == 0)
        inst = new SysParams();
    return *inst;
}
SysParams::SysParams() { ... }
Subclassing and registry

- Sometimes it may be useful to have different subclasses of the class, but only one instance of one of them.
- For example, we could choose one of several window managers.
- We can do that at compile/link time by using conditional compilation;
  - In this case, every subclass has its implementation of the getInstance() that returns the correct pointer, and the one to compile/link is decided through compilation switches.
- We can also do it at run-time (during instantiation), using for example an environment variable
  - In this case, it is necessary to implement the creation code in the getInstance() method of the base class.

Concurrency

- If several threads can use the singleton, we must protect the initialization through a mutex semaphore.
- Notice that every time getInstance() gets called, the mutex must be locked and unlocked, even after the object has been created.
- To reduce overhead we could use the double checked locking pattern:

```cpp
SysParams & SysParams::getInstance()
{
    lock_mutex();
    if (inst == 0)
        inst = new SysParams();
    unlock_mutex();
    return *inst;
}
```
Double checked locking

- In this case we perform a double check on the variable

```cpp
SysParams & SysParams::getInstance()
{
    if (inst == 0) {
        lock_mutex();
        if (inst == 0) inst = new SysParams();
        unlock_mutex();
        return *inst;
    }
}
```

- WARNING! This technique may not work on all architectures!
- the problem is with the re-ordering of instructions by the compiler (due to optimizations) or by the hardware (due to instruction re-ordering in the processor)

Solution (correct)

- A **memory barrier** is a processor instruction that guarantees order of instructions
  - All instructions before the barrier must be completed before any instruction after the barrier
- We will also use another helper variable to check initialization

```cpp
SysParams & SysParams::getInstance()
{
    if (val == 0) {
        lock_mutex();
        if (inst == 0) inst = new SysParams();
        unlock_mutex();
        // memory barrier
        val = 1;
        return *inst;
    }
    else {
        // memory barrier
        return *inst;
    }
}
```
When to use singletons

- A Singleton is useful to implement global variables in a safe way
  - For example, it provides a global point of access and an interface to a set of global objects (e.g. system parameters, a window manager, a configuration manager, etc.)
- It may be useful to control the order of initialisation
- The object is not created if not used
- Sometimes this pattern is overused
  -Singletons everywhere!
  - It is not worth to make it for a few primitive global variables that are local to a module

Abstract factory

- A program must be able to choose one of several families of classes
- Example,
  - a program’s GUI should run on several platforms
  - Each platform comes with its own set of GUI classes:
    - WinButton, WinScrollBar, WinWindow
    - MotifButton, MotifScrollBar, MotifWindow
    - pmButton, pmScrollBar, pmWindow
  - Inheritance:
    - Clearly, we can make all “button” classes derive from an abstract button that implements a virtual “draw” function
    - Then, we hold a pointer to button, and assign a specific button object, so that the correct draw() function is invoked each time
  - We probably need to dynamically create a lot of this objects
  - Problem: how can we simplify the creation of these objects?
Naive approach

- We keep a global variable (or object) that represents the current window manager and “look-and-feel” for all the objects
- Every time we create an object, we execute a switch/case on the global variable to see which object we must create

```java
enum {WIN, MOTIF, PM, ...} lf;
...
// need to create a button
switch(lf) {
    case WIN:    button = new WinButton(...);    break;
    case MOTIF:  button = new MotifButton(...);  break;
    case PM:     button = new PmButton(...);
    ...
}
```

Problems with the naive approach

- What happens if we need to add a new look-and-feel?
  - We must change lot of code (for every creation, we must add a new case)
- How much code must we link?
  - Assuming that each look and feel is part of a different library, all libraries must be linked together
  - Large amount of code
- This solution is not compliant with the open/closed principle
  - Every time we add a new look and feel, we must change the code of existing functions/classes
- This solution does not scale
Requirements

- Uniform treatment of every button, window, etc.
  - Once you define the interface, you can easily use inheritance
- Uniform object creation
- Easy to switch between families
- Easy to add a family

Solution: Abstract factory

Define a *factory* (i.e. a class whose sole responsibility is to create objects)

```cpp
class WidgetFactory {
    Button* makeButton(args) = 0;
    Window* makeWindow(args) = 0;
    // other widgets...
};
```

Define a concrete factory for each of the families

```cpp
class WinWidgetFactory : public WidgetFactory {
    Button* makeButton(args) {
        return new WinButton(args);
    }
    Window* makeWindow(args) {
        return new WinWindow(args);
    }
};
```
Select once which family to use:

```cpp
WidgetFactory* wf;
switch (lf) {
    case WIN:  wf = new WinWidgetFactory();
               break;
    case MOTIF: wf = new MotifWidgetFactory();
               break;
    ... 
}
```

When creating objects in the code, don’t use “new” but call:

```cpp
Button* b = wf->makeButton(args);
```

Switch families – once in the code
Add a family – one new factory, no effect on existing code
Participants

- **AbstractFactory** (WidgetFactory)
  - declares an interface for operations that create abstract product objects.

- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory)
  - implements the operations to create concrete product objects.

- **AbstractProduct** (Window, ScrollBar)
  - declares an interface for a type of product object.

- **ConcreteProduct** (MotifWindow, MotifScrollBar)
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.

- **Client**
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.
Comments

Pros:

- *It makes exchanging product families easy.* It is easy to change the concrete factory that an application uses. It can use different product configurations simply by changing the concrete factory.
- *It promotes consistency among products.* When product objects in a family are designed to work together, it's important that an application uses objects from only one family at a time. AbstractFactory makes this easy to enforce.

Cons:

- Not easy to extend the abstract factory’s interface

Other patterns:

- Usually one factory per application, a perfect example of a singleton

Known uses

- Different operating systems (could be Button, could be File)
- Different look-and-feel standards
- Different communication protocols
Separate the specification of how to construct a complex object from the representation of the object

The same construction process can create different representations

Example:

- A converter reads files from one file format (i.e. RTF)
- It should write them to one of several output formats (ascii, \LaTeX, HTML, etc.)

No limit on the number of possible output formats

- It must be easy to add a new “conversion” without modifying the code for the reader

Requirements

- Single Responsibility Principle
  - Same reader for all output formats
  - Output format chosen once in code

- Open-Closed Principle
  - Easy to add a new output format
  - Addition does not change old code

- Dynamic choice of output format
The reader: reads the input file, and invokes the converter to produce the output file

The output file is the *final product* of the construction

The converter is the *builder* that builds the final product in a complex way

---

**The solution**

We should return a different object depending on the output format:
- HTMLDocument, RTFDocument, . . .

Separate the building of the output from reading the input

Write an interface for such a builder

Use inheritance to write different concrete builders
The solution – code

- Builder interface

```cpp
class Builder {
    virtual void writeChar(char c) { }
    virtual void setFont(Font *f) { }
    virtual void newPage() { }
};
```

- Here's a concrete builder:

```cpp
class HTMLBuilder : public Builder
{
    private:
        HTMLDocument *doc;
    public:
        HTMLDocument *getDocument() {
            return doc;
        }
    void writeChar(char c) {...}
    void setFont(Font *f) {...}
    void newPage() {...}
};
```
The converter uses a builder:

```cpp
class Converter {
    void convert(Builder *b) {
        while (t = read_next_token())
            switch (o.kind) {
                case CHAR: b->writeChar(o);
                    break;
                case FONT: b->setFont(o);
                    break;
                // other kinds
            }
    }
};
```

And this is how the converter is used

```cpp
RTFBuilder *b = new RTFBuilder;
converter->convert(b);
RTFDocument *d = b->getDocument();
```

Comments

This pattern is useful whenever the creation of an object is complex and requires many different steps

- In the example, the creation of `HTMLDocument` is performed step by step as the tokens are read from the file
- Only at the end the object is ready to be used

Therefore, we separate the creation of the object from its use later on

The final object is created with one single step at the end of the creation procedure

- In this case, it is easier to check consistency of the creation parameters at once
- example: create a Square, using the interface of a Rectangle:
  - The user sets Height and Width in the builder, then tries to build the Square, and if they are different gets an exception telling what went wrong

Another example later on
**Intent**

Define an interface for creating an object, but let subclasses decide which class to instantiate

- Also known as Virtual Constructor
- The idea is to provide a virtual function to create objects of a class hierarchy
- each function will then know which class to instantiate

**Example**

Consider a framework for an office suite

- Typical classes will be Document and Application
- there will be different types of documents, and different types of applications
- for example: Excel and PowerPoint are applications, excel sheet and presentation are documents
- all applications derive from the same abstract class Application
- all documents derive from the same abstract class Document
- we have parallel hierarchies of classes
- every application must be able to create its own document object
**Participants**

- **Product** *(Document)*
  - defines the interface of the objects the factory method creates

- **ConcreteProduct** *(MyDocument)*
  - implements the Product’s interface

- **Creator** *(Application)*
  - declares the factory method

- **ConcreteCreator** *(MyApplication)*
  - overrides the factory method to return an instance of a ConcreteProduct
Implementation

- It may be useful to add parameters to the factory method, to allow the creation of multiple types of products
  - For example, suppose that you want to save a bunch of different objects on the disk (Triangle, Rectangle, Circle, etc, they are all of type shape)
  - one possibility would be to enumerate the types with an integer id, and save the id as first element in the disk record
  - when loading the objects again you may read the id first, and then pass it to a factory method which creates the correct type of object and loads it from the disk
  - further, to avoid a switch-case in the factory method, we could implement a registry (will see in a little while how to do this)
Using templates to avoid sub-classing

- Sometimes the ConcreteCreator must only implement the factory method
- to avoid writing just a class for this, we could use templates:

```cpp
class Creator {
public:
    virtual Product * createProduct() = 0;
    ...
};
template <class TheProduct>
class StandardCreator : public Creator {
public:
    virtual Product* createProduct() {
        return new TheProduct();
    }
};
StandardCreator<MyProduct> myCreator;
```

---

How to create objects

- Usually, objects are created by invoking the constructor
- however, sometimes the constructor is not as flexible as we wish
- an alternative technique is to use a static method in the class, whose purpose is to create objects of the class in a more flexible way
- this technique is called *static factory method*
  - has almost nothing to do with the GoF’s factory method

```cpp
class MyClass {
public:
    MyClass(int param); // std constructor
    static MyClass *create(int param); // static fact. method
};
```
Advantages

- The first advantage is that factory methods can have descriptive names.
- This is especially useful when there are many different ways to create an object:
  - the standard way is to implement many constructors with different argument lists,
  - however, the code readability of this technique is poor: it is difficult to understand what a certain constructor does by just looking at the list of parameters,
  - sometimes, constructors differ just in the order of the parameters!
- with static factory methods, instead:
  - It is possible to create different methods with different, more descriptive names.

Advantages

- The second important advantage is that, unlike constructors, static factory methods must not necessarily create an object:
  - This can be useful for example when you want to control how many objects are around, and eventually reuse them,
  - For example, this technique is very useful when implementing an enumeration of constant objects.
- The third advantage is the fact that they can create an object of a subtype of the original type, without the client knowing this fact:
  - Suppose for example that you implemented a BTree class,
  - The client code uses the interface of BTree to perform operations like insert/extract,
  - Then, you realize that you need different implementation of BTree in different contexts, because of performance / efficiency reasons,
  - If the BTree is created with a factory method, you can simply switch between the implementations by configuring the method differently.
Implementation

Notice that the two implementation classes need not to be exposed to the client: they can be completely hidden, and changed at any time without even informing the customer.

- The extra function `setType()` can be optionally used to let the client select the preferred implementation.
- Therefore, we have maximum separation of concerns.

Hiding the constructor?

- The static factory method looks similar to the singleton pattern (except that there is no limit to the number of instances).
- You might be tempted to make the constructor private, so the only way to construct an instance is to use the static factory method.
- However, keep in mind that, if the constructor is private, the class cannot be sub-classed.
  - The derived class cannot call the base class constructor!
- Therefore, if you want to sub-class, the constructor must be at least protected.
Other advantages

- Another advantage is the fact that you can easily specify default parameters between successive calls
- this reduces the list of parameters of complex constructors
  - This is sometimes called *telescoping constructor*

```cpp
class NutritionFacts {
public:
    NutritionFacts(int servingSize, int servings) {...}
    NutritionFacts(int servingSize, int servings, int calories) {...}
    NutritionFacts(int servingSize, int servings, int calories, int fat) {...}
    NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium) {...}
};
```

With static factory method

- see `simple_builder`
- notice how much more readable it is
- Notes:
  - The `auto_ptr` is used to guarantee that the builder object is destroyed after the last use
  - once all parameters have been set, they can be checked in the `NutritionFacts` constructor
  - this method can be extended to consistently build more complex objects step by step (see Builder Pattern)
Sometimes it is necessary to create objects by using an ID

c onsider a hierarchy of classes, with \texttt{Base} as the base class and many different derived classes

clients use the interface of \texttt{Base} to access the object methods

however, they would like to flexibly create one instance of one of the subclasses depending of an ID
  
  Could be an integer or a string, or anything else

\textbf{Therefore, we need an AbstractFactory}, with one single create method, which takes a parameter ID to decide which type of object to instantiate

 the following structure will combine:

  
  - AbstractFactory
  - Singleton
  - Static Factory Method