Design Patterns

Singletons, Pools and Factories

Attila Nagy

March 10, 2014

https://github.com/NagyAttila/DesignPatterns

Content

- Object Oriented Design Principles

- Singleton
- Meyers Singleton
- Phoenix Singleton

- Object Pool
- Connection Pool
- Thread Pool

- Factory Method
- Abstract Factory
- Object Factory

Object Oriented Design Principles

Open Close Principle

Classes are open for extension but closed for modification.

Object Oriented Design Principles

Open Close Principle

Classes are open for extension but closed for modification.

Liskov's Substitution Principle

Derived classes extend the base classes without changing their behavior.

Object Oriented Design Principles

Open Close Principle

Classes are open for extension but closed for modification.

Liskov's Substitution Principle

Derived classes extend the base classes without changing their behavior.

Dependency Inversion Principle

High level classes should depend on abstract classes.

Object Oriented Design Principles

Open Close Principle

Classes are open for extension but closed for modification.

Liskov's Substitution Principle

Derived classes extend the base classes without changing their behavior.

Dependency Inversion Principle

High level classes should depend on abstract classes.

Interface Segregation Principle

Fat interface avoidance.
Clients should not be forced to depend on interface that they don't use.

Object Oriented Design Principles

Open Close Principle

Classes are open for extension but closed for modification.

Liskov's Substitution Principle

Derived classes extend the base classes without changing their behavior.

Dependency Inversion Principle

High level classes should depend on abstract classes.

Interface Segregation Principle

Fat interface avoidance.
Clients should not be forced to depend on interface that they don't use.

Single Responsibility Principle

One class should be responsible for one thing.

Singleton

Purpose:

- class has at most one instance,
- that instance is accessible in a global scope.

Application Areas:

- logging,
- Thread and Connection Pools,
- Factories
- system clock,
- etc.

Singleton - How?

- private constructors
- static _pInstance_ and _Instance_

```
class Singleton {
public:
static Singleton& Instance() {
  if (nullptr == pInstance) {
    pInstance = new Singleton;
  }
  return *pInstance;
}

private:
Singleton () = default;
Singleton (const Singleton&) = delete;
Singleton operator= (const Singleton&) = delete;
~Singleton () = delete;

static Singleton* pInstance;
// More Functions and Data
};
Singleton* Singleton::pInstance = nullptr;
```

Singleton - How?

- private constructors
- static *_pInstance_* and *_Instance_*

```
class Singleton {
public:
static Singleton
  if (nullp
    pIns
  }
  ret
}

priva
Singlet
Singleto
Singleton op
~Singleton () = d

static Singleton* pInstance;
// More Functions and Data
};
Singleton* Singleton::pInstance = nullptr;
```

Destructor was never called! (memory leak)

Meyers Singleton

- local static _instance_

```
class Singleton {
public:
static Singleton& Instance() {
  static Singleton instance;
  return instance;
}

private:
Singleton () = default;
Singleton (const Singleton&) = delete;
Singleton operator= (const Singleton&) = delete;
~Singleton () = default;

// More Functions and Data
};
```

⇒ private destructor is called at process termination

Meyers Singleton

- local static _instance_

```
class Singleton {
public:
static Singleton& Instance() {
  static Singleton instance;
  return instance;
}

priva
Single
Singleton (const
Singleton operator= (const Singleton&) = delete;
~Singleton () = default;

// More Functions and Data
};
```

Dead Reference!

⇒ private destructor is called at process termination

Dead Reference Problem

How?

1. Two Singleton classes, Logger and Keyboard, in different compilation units

Dead Reference Problem

How?

1. Two Singleton classes, Logger and Keyboard, in different compilation units
2. Keyboard uses Logger

Dead Reference Problem

How?

1. Two Singleton classes, Logger and Keyboard, in different compilation units
2. Keyboard uses Logger
3. termination: Logger deallocated, Keyboard tries to use it

Dead Reference Problem

How?

1. Two Singleton classes, Logger and Keyboard, in different compilation units
2. Keyboard uses Logger
3. termination: Logger deallocated, Keyboard tries to use it
4. Logger's static object "shell" is still available

Dead Reference Problem

How?

1. Two Singleton classes, Logger and Keyboard, in different compilation units
2. Keyboard uses Logger
3. termination: Logger deallocated, Keyboard tries to use it
4. Logger's static object "shell" is still available
5. undefined behaviour (probably crash)

Dead Reference Problem

How?

1. Two Singleton classes, Logger and Keyboard, in different compilation units
2. Keyboard uses Logger
3. termination: Logger deallocated, Keyboard tries to use it
4. Logger's static object "shell" is still available
5. undefined behaviour (probably crash)

Why?

- order of deallocation of static objects is not deterministic

Dead Reference Problem

How?

1. Two Singleton classes, Logger and Keyboard, in different compilation units
2. Keyboard uses Logger
3. termination: Logger deallocated, Keyboard tries to use it
4. Logger's static object "shell" is still available
5. undefined behaviour (probably crash)

Why?

- order of deallocation of static objects is not deterministic

Solution?

- on-demand reallocation of *Singleton* after destruction

  $\Rightarrow$ Phoenix Singleton

Phoenix Singleton

Combination of the 3 approaches:

- Simple: *_pInstance_* pointer
- Meyers: static life-time
- Dead Reference: detection

Plus:

- on-demand reallocation
- destructor called using *_atexit_*
- multiple time "reborn"

```cpp
class Singleton {
public:
static Singleton& Instance() {
  if (destroyed) {
    new(pInstance) Singleton;
    atexit(KillSingleton);
  }
  if (nullptr == pInstance) {
    static Singleton instance;
    pInstance = &instance;
  }
  return *pInstance;
}
static void KillSingleton(void) {
  pInstance->~Singleton();
}
```

```cpp
private:
Singleton () {
  destroyed = false;
}
Singleton (const Singleton&) = delete;
Singleton operator= (const Singleton&) =
     delete;
~Singleton () {
  destroyed = true;
  pInstance = nullptr;
}
static bool destroyed;
static Singleton* pInstance;
};
bool Singleton::destroyed = false;
Singleton* Singleton::pInstance = nullptr;
```

Phoenix Singleton

Combination of the 3 approaches:

- Simple: *_pInstance_* pointer
- Meyers: static life-time
- Dead Reference: detection

Plus:

- on-demand reallocation
- destructor called using *_atexit_*
- multiple time "reborn"

¿Problem?

```
class
publ
static Si
  if (destroyed) {
    new(pInstance) Singleton;
    atexit(KillSingleton);
  }
  if (nullptr == pInstance) {
    static Singleton instance;
    pInstance = &instance;
  }
  return *pInstance;
}
static void KillSingleton(void) {
  pInstance->~Singleton();
}
```

```
Singleton (const Singleton&) = delete;
Singleton operator= (const Singleton&) =
     delete;
~Singleton () {
  destroyed = true;
  pInstance = nullptr;
}
static bool destroyed;
static Singleton* pInstance;
};
bool Singleton::destroyed = false;
Singleton* Singleton::pInstance = nullptr;
```

atexit Problem

Problem:

- functions registered on stack

atexit Problem

Problem:

- functions registered on stack
- last registered called first

atexit Problem

Problem:

- functions registered on stack
- last registered called first
- registering from a registered function

atexit Problem

Problem:

- functions registered on stack
- last registered called first
- registering from a registered function
- too late to be first

atexit Problem

Problem:

- functions registered on stack
- last registered called first
- registering from a registered function
- too late to be first
- some old compilers might crash

Solution?

atexit Problem

Problem:

- functions registered on stack
- last registered called first
- registering from a registered function
- too late to be first
- some old compilers might crash

Solution?

- Read the Manual!
- Use the newest compilers!

Multithreading

Problem:

- general lazy initialization problem
- race condition

```
static Singleton& Instance() {
  if (nullptr == pInstance) {
    pInstance = new Singleton;
  }
  return *pInstance;
}
```

Multithreading

Problem:

- general lazy initialization problem
- race condition

Solution:

A local static instance (Meyers approach)

```
static Singleton& Instance() {
  if (nullptr == pInstance) {
    pInstance = new Singleton;
  }
  return *pInstance;
}
```

Multithreading

Problem:

- general lazy initialization problem
- race condition

Solution:

A local static instance (Meyers approach)

B locking

```
static Singleton& Instance() {
  Lock guard(mutex);
  if (nullptr == pInstance) {
    pInstance = new Singleton;
  }
  return *pInstance;
}
```

Multithreading

Problem:

- general lazy initialization problem
- race condition

Solution:

A local static instance (Meyers approach)

B locking

```
static Singleton& Instance() {
  if (nullptr == pInstance) {
    Lock guard(mutex);
    pInstance = new Singleton;
  }
  return *pInstance;
}
```

Multithreading

Problem:

- general lazy initialization problem
- race condition

Solution:

A local static instance (Meyers approach)

B locking

C double-check locking

```
static Singleton& Instance() {
  if (nullptr == pInstance) {
    Lock guard(mutex);
    if (nullptr == pInstance) {
      pInstance = new Singleton;
    }
  }
  return *pInstance;
}
```

Notes

Why not global?

- uniqueness
- lack of laziness
- pollute global scope
- not always possible
  - dependency
  - need data for init

Notes

Why not global?

- uniqueness
- lack of laziness
- pollute global scope
- not always possible
    - dependency
    - need data for init

Possible extensions and alternative solutions:

- longevity control for dead reference problem
- registry high number of singletons
- inheritance

Notes

Why not global?

- uniqueness
- lack of laziness
- pollute global scope

- not always possible
  - dependency
  - need data for init

Possible extensions and alternative solutions:

- longevity control for dead reference problem
- registry high number of singletons
- inheritance

Final thought:

- should be used sparingly

- if needs a lot:
  - $\Rightarrow$ use registry, or
  - $\Rightarrow$ change design

Object Pool

Purpose:

- reuse objects
- eliminate object allocation / deallocation overhead

Object Pool

Purpose:

- reuse objects
- eliminate object allocation / deallocation overhead

Useful when:

- high instantiation cost
- high instantiation rate
- low number of concurrently used instances

Object Pool

Purpose:

- reuse objects
- eliminate object allocation / deallocation overhead

Useful when:

- high instantiation cost
- high instantiation rate
- low number of concurrently used instances

Pitfalls:

- reset object after use
  - $\Rightarrow$ false authentication
  - $\Rightarrow$ information leak
- object not released

Object Pool

Purpose:

- reuse objects
- eliminate object allocation / deallocation overhead

Useful when:

- high instantiation cost
- high instantiation rate
- low number of concurrently used instances

Pitfalls:

- reset object after use
  - $\Rightarrow$ false authentication
  - $\Rightarrow$ information leak
- object not released

Empty pool:

1. report error
2. increase pool size
3. blocking request

Object Pool

Purpose:

- reuse objects
- eliminate object allocation / deallocation overhead

Useful when:

- high instantiation cost
- high instantiation rate
- low number of concurrently used instances

Similar patterns

- Connection Pool
- Thread Pool

Pitfalls:

- reset object after use
  - $\Rightarrow$ false authentication
  - $\Rightarrow$ information leak
- object not released

Empty pool:

1. report error
2. increase pool size
3. blocking request

Used as

- Singleton

Connection Pool and Thread Pool

Connection Pool

- cache of DB connections
- creates new connection if empty pool (2nd approach)

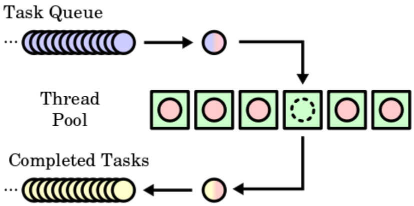
Connection Pool and Thread Pool

Connection Pool

- cache of DB connections
- creates new connection if empty pool (2nd approach)

Thread Pool

- asynchronous task processing
- more tasks than threads
- fix or dynamic number of threads



Threads

too many created $\Rightarrow$ wasting resource and time

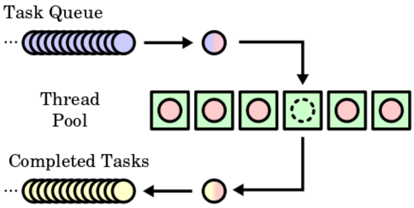
Connection Pool and Thread Pool

Connection Pool

- cache of DB connections
- creates new connection if empty pool (2nd approach)

Thread Pool

- asynchronous task processing
- more tasks than threads
- fix or dynamic number of threads



Threads

too many created $\Rightarrow$ wasting resource and time

too many destroyed $\Rightarrow$ more time recreating them

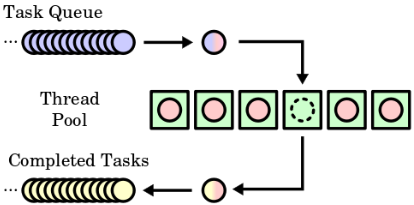
Connection Pool and Thread Pool

Connection Pool

- cache of DB connections
- creates new connection if empty pool (2nd approach)

Thread Pool

- asynchronous task processing
- more tasks than threads
- fix or dynamic number of threads



Threads

too many created $\Rightarrow$ wasting resource and time

too many destroyed $\Rightarrow$ more time recreating them

too slow creation $\Rightarrow$ long waiting times at clients

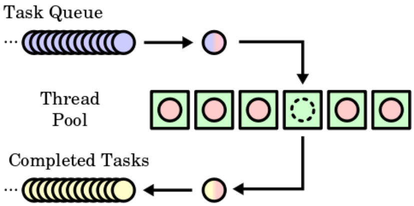
Connection Pool and Thread Pool

Connection Pool

- cache of DB connections
- creates new connection if empty pool (2nd approach)

Thread Pool

- asynchronous task processing
- more tasks than threads
- fix or dynamic number of threads



Threads

too many created ⇒ wasting resource and time

too many destroyed ⇒ more time recreating them

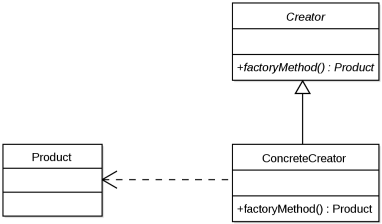
too slow creation ⇒ long waiting times at clients

too slow destroy ⇒ wasting resource and time

Factory Method

A.k.a: Virtual Constructor

- uses an abstract class for interface
- instantiation determined by subclasses



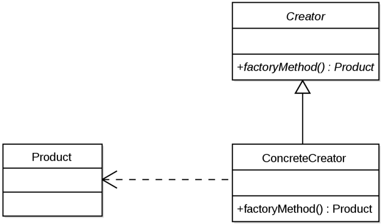
Factory Method

A.k.a: Virtual Constructor

- uses an abstract class for interface
- instantiation determined by subclasses

Features:

- decouple implementation from interface
- uses class inheritance



Factory Method

A.k.a: Virtual Constructor

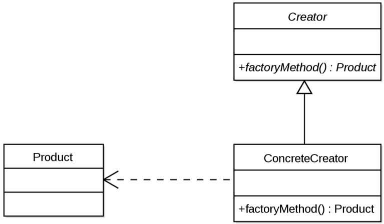
- uses an abstract class for interface
- instantiation determined by subclasses

Features:

- decouple implementation from interface
- uses class inheritance

Application areas:

- unit testing
- Abstract Factory



Abstract Factory

Features:

- create product from families of products

Abstract Factory

Features:

- create product from families of products
- client uses abstract classes (dependency inversion)

Abstract Factory

Features:

- create product from families of products
- client uses abstract classes (dependency inversion)
- client is isolated from concrete product

Abstract Factory

Features:

- create product from families of products
- client uses abstract classes (dependency inversion)
- client is isolated from concrete product
- subclasses decide what class to instantiate

Abstract Factory

Features:

- create product from families of products
- client uses abstract classes (dependency inversion)
- client is isolated from concrete product
- subclasses decide what class to instantiate
- easy change of family on client side

Abstract Factory

Features:

- create product from families of products
- client uses abstract classes (dependency inversion)
- client is isolated from concrete product
- subclasses decide what class to instantiate
- easy change of family on client side
- consistency among products

Abstract Factory

Features:

- create product from families of products
- client uses abstract classes (dependency inversion)
- client is isolated from concrete product
- subclasses decide what class to instantiate
- easy change of family on client side
- consistency among products
- uses Factory Method to create products

Abstract Factory

Features:

- create product from families of products
- client uses abstract classes (dependency inversion)
- client is isolated from concrete product
- subclasses decide what class to instantiate
- easy change of family on client side
- consistency among products
- uses Factory Method to create products
- uses object composition

Abstract Factory

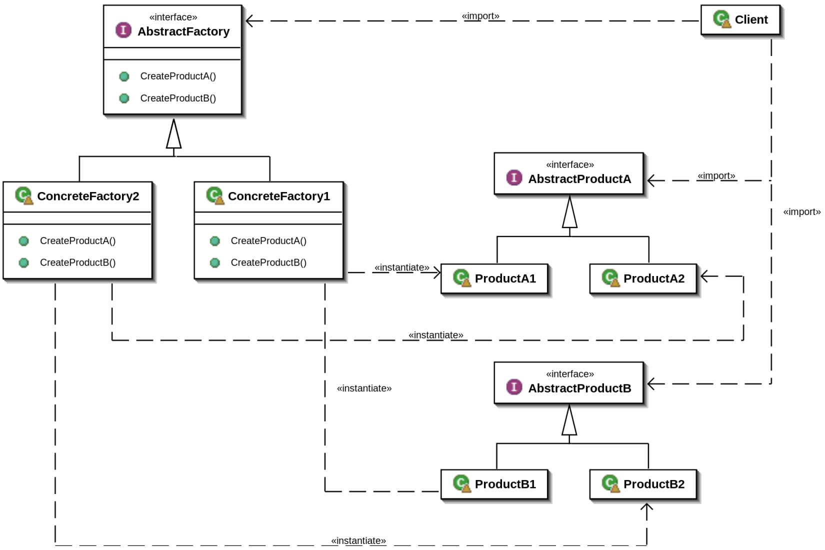
Features:

- create product from families of products
- client uses abstract classes (dependency inversion)
- client is isolated from concrete product
- subclasses decide what class to instantiate
- easy change of family on client side
- consistency among products
- uses Factory Method to create products
- uses object composition

Drawbacks:

- adding new product-family is cumbersome
  - $\Rightarrow$ abstract factory class must change
    - $\Rightarrow$ concrete factory classes must follow the change

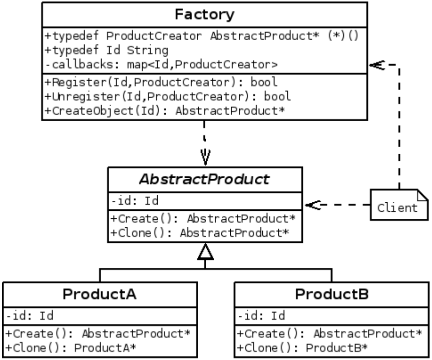
Abstract Factory



Object Factory

Features:

- each product has to register
- uses the priority inversion principle
- one family of product



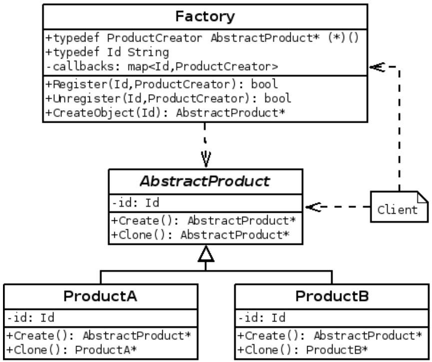
Object Factory

Features:

- each product has to register
- uses the priority inversion principle
- one family of product

Cloning:

- make use of covariant return type
- common mistake: forgetting to implement _Clone_



Resources

- Design Patterns: Elements of Reusable Object-Oriented Software
  "Gang of Four"
- Modern C++ Design: Generic Programming and Design Patterns Applied
  Andrei Alexandrescu
- Head First: Design Patterns
  Eric Freeman & Elisabeth Freeman

Questions?

Thank You!