Creational Design Patterns

Part 1

Creational Design Patterns

Two main goals:

1. Encapsulate knowledge about which concrete classes the system uses

2. Hide how instances of these classes are created and built

Creational Design Patterns

- System at large knows about objects through their interfaces defined by abstract classes
- Give us flexibility in:
  - *what* gets created
  - *who* creates it
  - *how* it gets created
  - *when* it gets created

Creational Design Patterns

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Creational Patterns: Singleton

- Consider a class called `Logger`
  - Logs information to a file
  - Needed by many different parts of an application

Creational Patterns: Singleton

Logger.h

```
class Logger
{
   public:
      Logger();
      virtual ~Logger();
      const Logger& log(const std::string& message) const;
      const Logger& operator<<(const  std::string&  message) const;

   private:
      mutable  std::ofstream  _output;
};
```

Creational Patterns: Singleton

Logger.cpp

```
Logger::Logger()
{
   this->_output.open("program.log");
}

Logger::~Logger()
{
   this->_output.close();
}

const Logger& Logger::log(const string& message) const
{
   this->_output   <<   message   << endl;
   return *this;
}

const Logger& Logger::operator<<(const string& message) const
{
   return this->log(message);
}
```

Creational Patterns: Singleton

main.cpp

```
void f(const Logger& log)
{
   log <<  "In function  f()";
}

int main()
{
   Logger log;
   log <<  "Starting program";

   f(log);
}
```

Creational Patterns: Singleton

Output

```
$ ./main
$ cat program.log
Starting program
In  function f()
```

Creational Patterns: Singleton

- As our application grows, we will want to have logging in more and more functions

- Potential solutions:
  - Pass around a `Logger` object to the functions that need it
  - Create a new `Logger` object in each function that needs it
  - Use a global `Logger` object that all functions can access from anywhere

Creational Patterns: Singleton

- Suppose we opt to pass around a `Logger` object
- Later, we add a `Person` class
- Each `Person` has a `Car`

Creational Patterns: Singleton

Person.h

```
class Person
{
   public:
      Person(const std::string& name);
      virtual ~Person();
      Car* car() const;

   private:
      std::string _name;
      Car* _car;
};
```

Creational Patterns: Singleton

Person.cpp

```
Person::Person(const std::string& name)
{
   this->_name = name;
   this->_car = new Car();
}

Person::~Person()
{
   delete this->_car;
}

Car* Person::car() const
{
   return this->_car;
}
```

Creational Patterns: Singleton

Car.h

```
class Car
{
   public:
      Car();

      void turnOn();
      void turnOff();
};
```

Creational Patterns: Singleton

- Now we want to add logging so that a log entry is created each time a person's `Car` is turned on or off
- Which class(es) do we need to modify?

Creational Patterns: Singleton

Person.h

```
class Person
{
   public:
      Person(const std::string& name, const Logger& log);
      virtual ~Person();
      Car* car() const;

   private:
      std::string _name;
      Car* _car;
};
```

Creational Patterns: Singleton

Person.cpp

```
Person::Person(const std::string& name, const Logger& log)
{
   this->_name = name;
   this->_car = new Car(log);
}

Person::~Person()
{
   delete this->_car;
}

Car*  Person::car()  const
{
   return this->_car;
}
```

Creational Patterns: Singleton

Car.h

```
class Car
{
   public:
      Car(const Logger& log);

      void turnOn();
      void turnOff();

   private:
      const Logger* _log;
};
```

Creational Patterns: Singleton

Car.cpp

```
Car::Car(const Logger& log) : _log(log)
{
}

void Car::turnOn()
{
   this->_log  <<  "Turning  on car";
}

void Car::turnOff()
{
   this->_log  <<  "Turning  off car";
}
```

Creational Patterns: Singleton

main.cpp

```
int main(){

   Logger log;
   Person p("Joe", log);

   log << "Starting program";

   // Side note: what design principle has been violated here?

   Car* car = p.car();
   car->turnOn();
   car->turnOff();
}
```

Creational Patterns: Singleton

- What are the problems with this solution?

- What if, instead, we created a new `Logger` object in every function that needed logging?

Creational Patterns: Singleton

Logger.cpp

```
Logger::Logger()
{
   this->_output.open("program.log");
}

Logger::~Logger()
{
   this->_output.close();
}

const Logger& Logger::log(const string& message) const
{
   this->_output   <<   message   << endl;
   return *this;
}

const Logger& Logger::operator<<(const string& message) const
{
   return this->log(message);
}
```

- Any issues with this?

Creational Patterns: Singleton

- What if, instead, we used a global variable that all functions could access?

```
const  Logger*  const  globalLogger  =  new Logger();
```

```
void f()
{
   *globalLogger << "In function f()";
}
```

```
void Car::turnOn()
{
   *globalLogger << "Turning on car";
}
```

- Problems?

Creational Patterns: Singleton

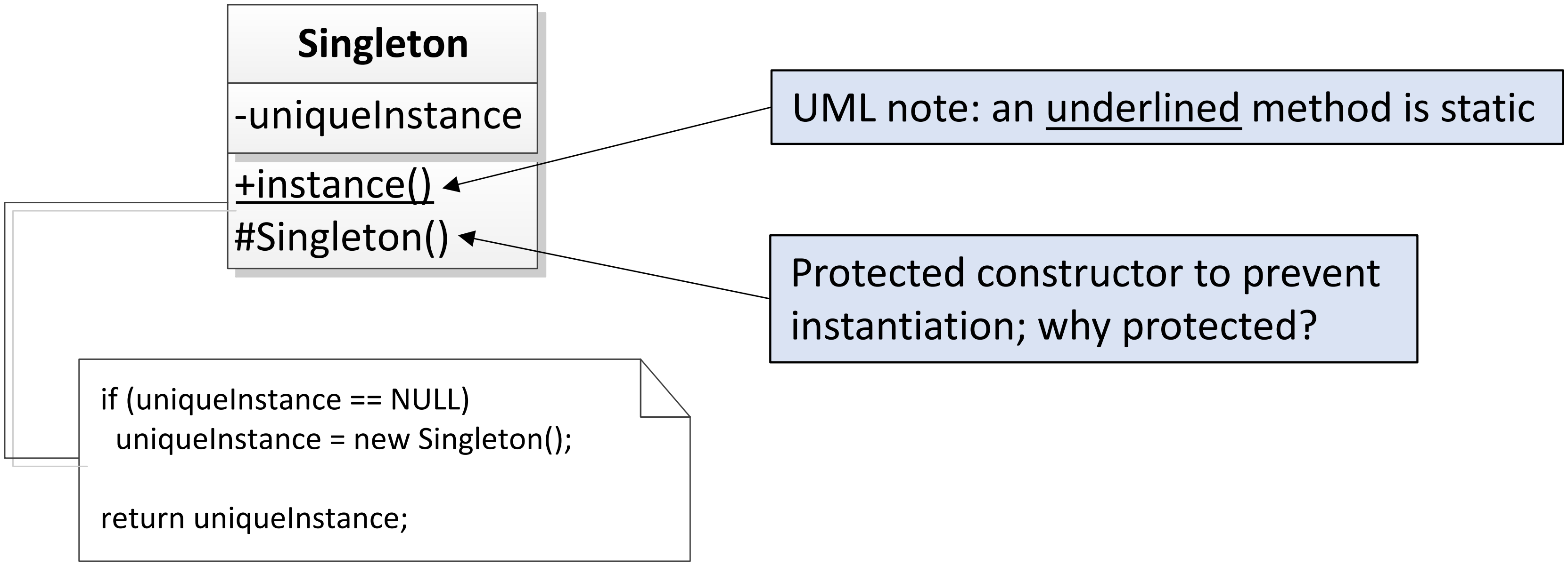
Design Pattern:
Singleton

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

Creational Patterns: Singleton

- Applicability:
  - There must be exactly one instance of a class
  - It must be accessible to clients from a well-known access point
  - The sole instance should be extensible by subclassing

Creational Patterns: Singleton



Creational Patterns: Singleton

Logger.h

```
class Logger
{
   public:
      virtual ~Logger();
      static const Logger& instance();
      const Logger& log(const std::string& message)  const;
      const Logger& operator<<(const  std::string&  message)  const;

   protected:
      Logger(); // Prevent instantiation

   private:
      // Prevent copying and assignment
      Logger(const Logger& other) { };
      Logger& operator=(const Logger& other) { };
      mutable std::ofstream _output;
      static const Logger* _instance;
};
```

Creational Patterns: Singleton

Logger.cpp

```
const Logger* Logger::_instance = NULL;
const Logger& Logger::instance()
{
   if (_instance  == NULL)
      _instance  =  new  Logger();
   return *_instance;
}
Logger::Logger()
{
   this->_output.open("program.log");
}
Logger::~Logger()
{
   this->_output.close();
}
const Logger& Logger::log(const string& message) const
{
   this->_output   <<   message   << endl;
   return *this;
}
const Logger& Logger::operator<<(const string& message) const
{
   return this->log(message);
}
```

Creational Patterns: Singleton

main.cpp

```
int main(){
   Logger::instance() << "Starting program";

   Person p("Joe");

   Car* car = p.car();

   car->turnOn();
   car->turnOff();
}
```

Creational Patterns: Singleton

- Consequences:
  - Controlled access to sole instance
  - Lazy initialization
  - Reduced name space
  - Permits refinement through subclassing
  - Permits a variable number of instances, if needed
  - Have to worry about who deletes the instance
    - `std::shared_ptr` or `boost::shared_ptr` can help with this