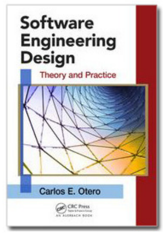
Chapter 6: Creational Design Patterns

Session III: Builder, Prototype, Singleton

Software Engineering Design: Theory and Practice
by Carlos E. Otero



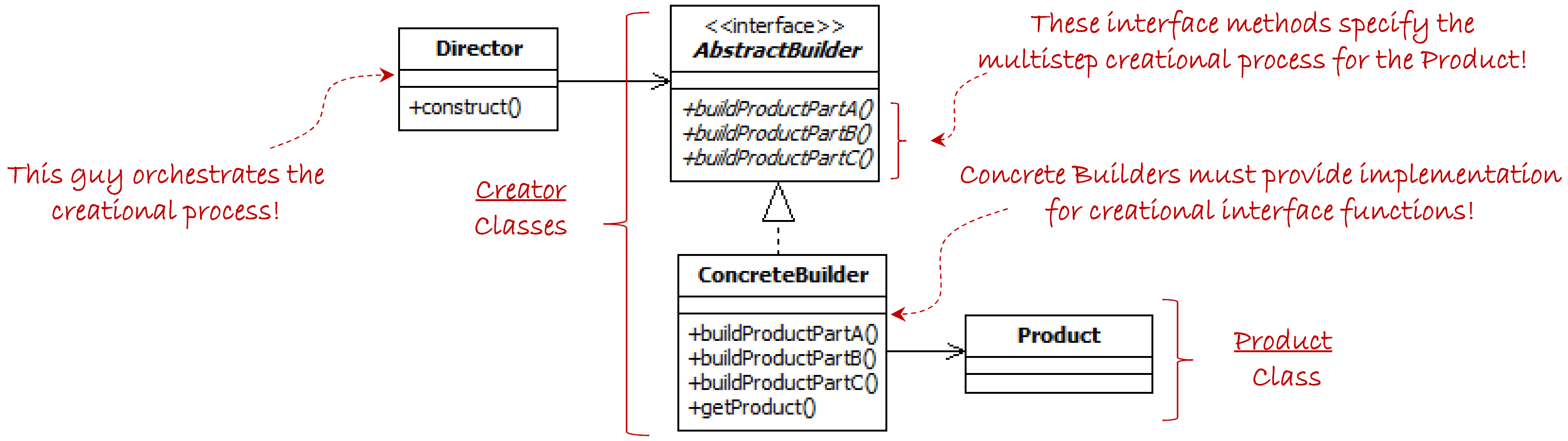
SESSION'S AGENDA

- Creational Patterns in Detailed Design
- Builder
  - ✓ Code generator example
- Prototype
  - ✓ Character cloning example
- Singleton
  - ✓ Event manager example
- What's next…

BUILDER DESIGN PATTERN

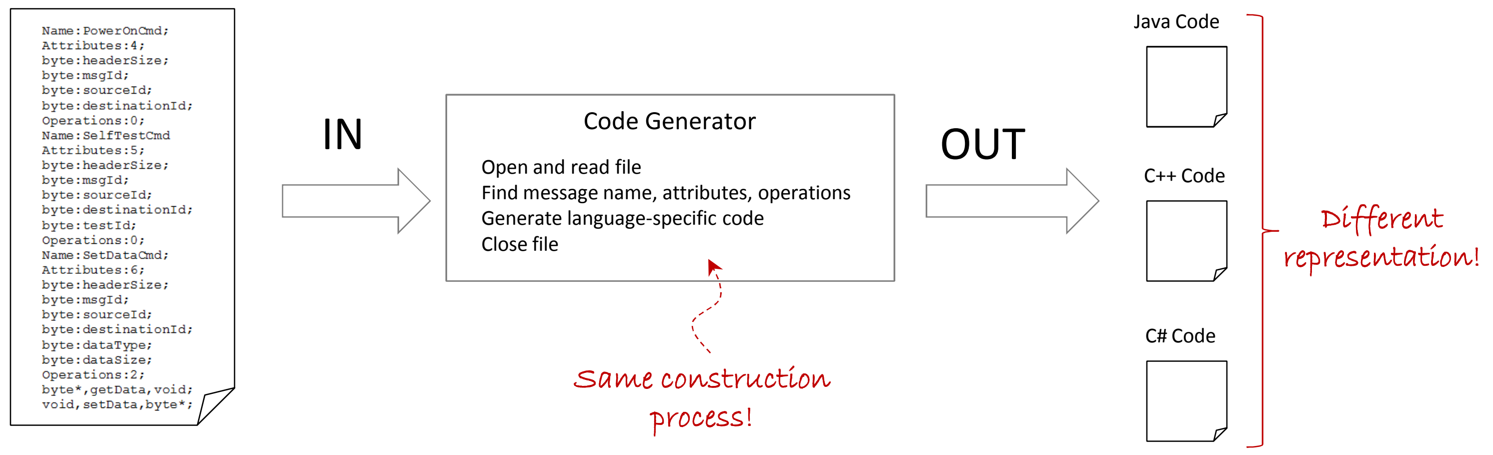
- The Builder Design pattern is an object creational pattern that encapsulates both the creational process and the representation of product objects.
  - ✓ Unlike the Abstract Factory, in which various product objects are created all at once, Builder allows clients to control the (multistep) creational process of a single product object, allowing them to dictate the creation of individual parts of the object at discrete points throughout software operations.

- According to the GoF, the intent of the Builder is to [1]
  - ✓ Separate the construction of a complex object from its representation so that the same construction process can create different representations.



Builder Design Pattern Example

- Consider a code generator:
  - ✓ Code generated from an interface specification document (ICD) used to monitor and control custom developed hardware.
  - ✓ Company sells hardware, but also includes library to monitor and control its hardware.
  - ✓ Generator reads a text document containing information about a messaging specification. It then generates C++ classes that can be used to monitor and control hardware. This is also shipped to customers so that they can use the hardware easily.
  - ✓ The ICD is quite complex, and customers are demanding support for other languages, such as Java and C#.

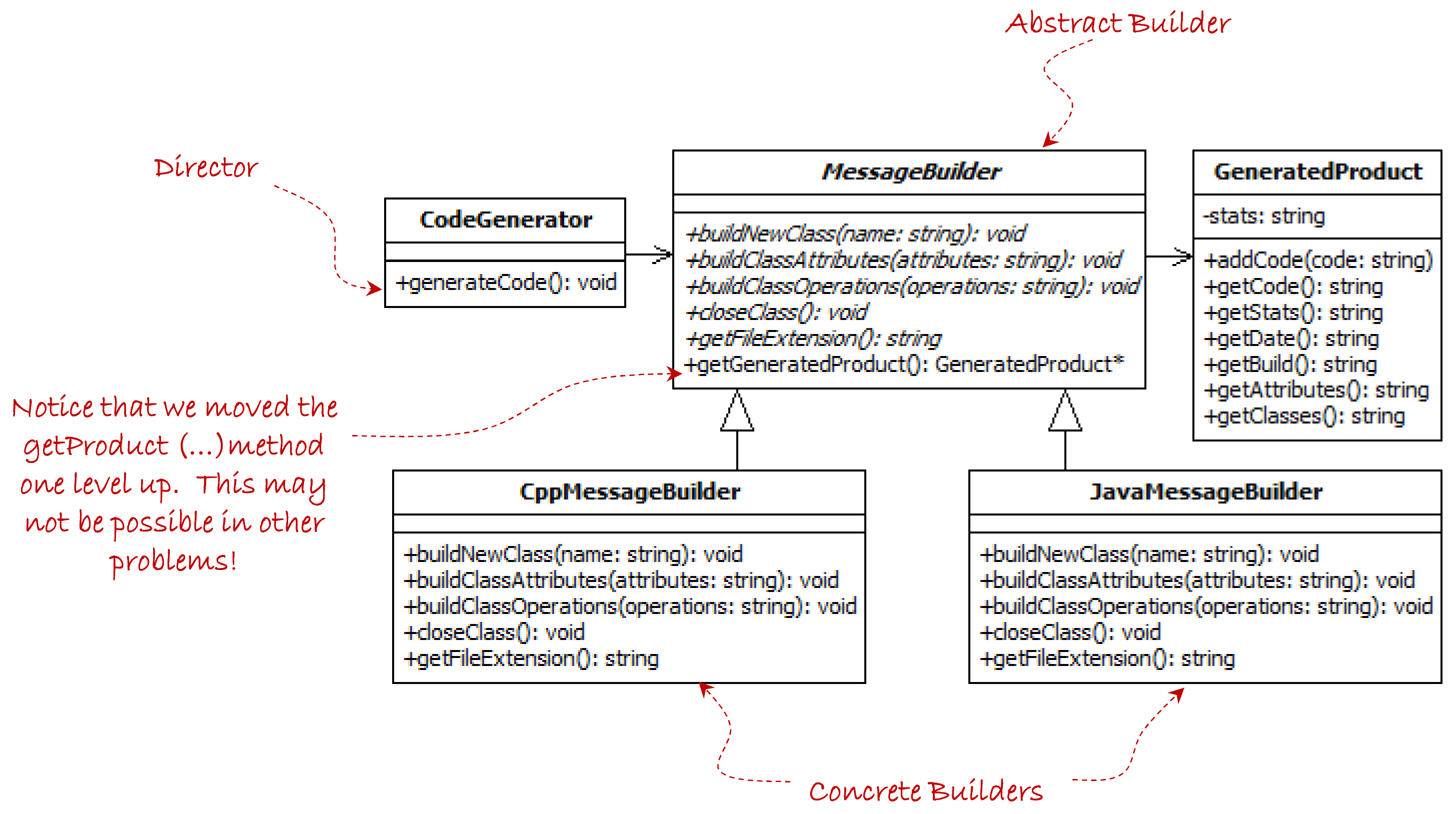


```
Name:PowerOnCmd;
Attributes:4;
byte:headerSize;
byte:msgId;
byte:sourceId;
byte:destinationId;
Operations:0;
Name:SelfTestCmd
Attributes:5;
byte:headerSize;
byte:msgId;
byte:sourceId;
byte:destinationId;
byte:testId;
Operations:0;
Name:SetDataCmd;
Attributes:6;
byte:headerSize;
byte:msgId;
byte:sourceId;
byte:destinationId;
byte:dataType;
byte:dataSize;
Operations:2;
byte*,getData,void;
void,setData,byte*;
```

Code Generator

Open and read file
Find message name, attributes, operations
Generate language-specific code
Close file

Builder Design Pattern Example


Abstract Builder
Director
CodeGenerator
+generateCode(): void
MessageBuilder
+buildNewClass(name: string): void
+buildClassAttributes(attributes: string): void
+buildClassOperations(operations: string): void
+closeClass(): void
+getFileExtension(): string
+getGeneratedProduct(): GeneratedProduct*
GeneratedProduct
-stats: string
+addCode(code: string)
+getCode(): string
+getStats(): string
+getDate(): string
+getBuild(): string
+getAttributes(): string
+getClasses(): string
Notice that we moved the getProduct (...) method one level up. This may not be possible in other problems!
CppMessageBuilder
+buildNewClass(name: string): void
+buildClassAttributes(attributes: string): void
+buildClassOperations(operations: string): void
+closeClass(): void
+getFileExtension(): string
JavaMessageBuilder
+buildNewClass(name: string): void
+buildClassAttributes(attributes: string): void
+buildClassOperations(operations: string): void
+closeClass(): void
+getFileExtension(): string
Concrete Builders

BUILDER DESIGN PATTERN EXAMPLE

The Builder interface for generating code!

The same steps used to create products in all target languages!

```
#include <string>

// Forward reference.
class GeneratedProduct;

class MessageBuilder {

public:
  // The interface method for building a new class.
  virtual void buildNewClass(string name) = 0;

  // The interface method for building class attributes.
  virtual void buildClassAttributes(string attributeList) = 0;

  // The interface method for building class operations.
  virtual void buildClassOperations(string operationList) = 0;

  // The interface method for closing a new class.
  virtual void closeClass() = 0;

  // The file extension for the target programming language.
  virtual string getFileExtension() = 0;

  // Return the generated product.
  GeneratedProduct* getGeneratedProduct() {

    return _codeProduct;
  }

private:
  // The product containing generated code and stats about code generated.
  GeneratedProduct* _codeProduct;
};
```

Builder Design Pattern Example

```
class CppMessageBuilder : public MessageBuilder {

public:

  // The interface method for building a new class.
  virtual void buildNewClass(string name) {

    // Generate code for creating a class using CPP style and the name
    // argument.

    // Once code is generated, add it to the product.
    getGeneratedProduct()->addCode(/*new C++ class code*/);
  }

  // The interface method for building class attributes.
  virtual void buildClassAttributes(string attributeList) {

    // For all items in attributeList, generate attributes using CPP style
    // and add them to the generated code.

    // Once code is generated, add it to the product.
    getGeneratedProduct()->addCode(/*C++ attributes*/);
  }

  // The interface method for building class operations.
  virtual void buildClassOperations(string operationList) {

    // For all items in operationList, generate operations using CPP style
    // and add them to the generated code.

    // Once code is generated, add it to the product.
    getGeneratedProduct()->addCode(/*C++ operations*/);
  }

  // The interface method for closing a new class.
  virtual void closeClass() {

    // Generate code to close a class in Cpp, and add it to the generated
    // code.

    // Once code is generated, add it to the product.
    getGeneratedProduct()->addCode("\n};\n\n");
};
```

C++ Builder

Step 1

```
class MessageOne
{
```

Step 2

```
class MessageOne
{
  private:
    int attributeOne;
    int attributeTwo;
```

Step 3

```
class MessageOne
{
  private:
    int attributeOne;
    int attributeTwo;

  public:
    void setValueOne(unsigned char x);
```

Step 4

```
class MessageOne
{
  private:
    int attributeOne;
    int attributeTwo;

  public:
    void setValueOne(unsigned char x);
};
```

C++ Product Created!

Builder Design Pattern Example

```
class CodeGenerator {

public:

  // Constructor.
  CodeGenerator(MessageBuilder* pBuilder) : m_pBuilder(pBuilder) {
    // Assume a valid builder pointer.  Notice that m_pBuilder is
    // initialized in the constructor's initialization list above.
  }

  // The interface method for building a new class.
  virtual void generateCode(string fileName) {

    // Open file for reading: fileName.
    while ( /* not end of file */) {

      // read next token in file.
      if( /*class name found*/) {

        // Assume that variable className holds the name.
        m_pBuilder->buildNewClass(className);

      } else if( /*attribute list found*/) {

        // Assume that attributeList contains the attributes
        m_pBuilder->buildClassAttributes(attributeList);

      } else if( /*operation list found*/) {

        // Assume that operationList contains the operations.
        m_pBuilder->buildClassOperations(operationList);

        // Close the class.
        m_pBuilder->closeClass();
      }

    } // end while( /* not end of file */)

    // Close file: fileName.

    // Create file using file extension from generated product object.
    // write(m_pBuilder->getGeneratedProduct()->getCode());
    // Close file.
    // logCodeGeneration(m_pBuilder->getGeneratedProduct());

  } // end generateCode(...)

private:
  MessageBuilder* m_pBuilder;
};
```

Product representation, i.e., generated code, depends on the Builder passed in as parameter!

In the previous slide, we presented how a C++ message builder could be created.

In similar fashion, C# or Java builders can be created.

The same creational process is used for all target languages!

The process used for identifying names, attributes, and operations in the ICD could be quite complex!

However, if we manage to separate this process from its ultimate product representation (e.g., java vs. c++ code), we can reuse it to generate these and other future products by extension and NOT modification!

BUILDER DESIGN PATTERN EXAMPLE

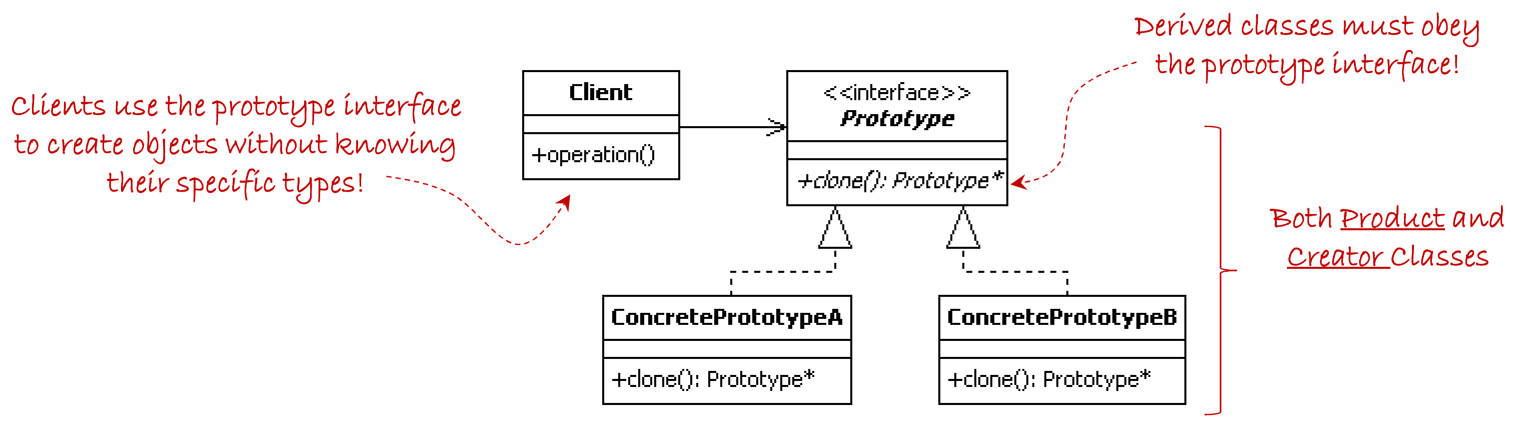
- Steps for designing using the Builder pattern:
  1. Identify and design the product class (e.g., GeneratedProduct)
  2. Identify the product's creational process and algorithm, and design a class for its execution (e.g., CodeGenerator).
  3. Using the knowledge acquired from steps 1 and 2, design the builder interface, which specifies the parts that need to be created for the whole product to exist. These are captured as abstract interface methods that need to be implemented by derived concrete builders.
  4. Identify and design classes for the different representations of the product (e.g., CppMessageBuilder, JavaMessageBuilder, etc.)

- The Builder provides the following benefits:
  - ✓ Separates an object's construction process from its representation; therefore future representations can be added easily to the software.
  - ✓ Changes to existing representations can be made without modifying the code for the creational process.
  - ✓ Provides finer control over the construction process so that objects can be created at discrete points in time, which differs from the Abstract Factory pattern.

PROTOTYPE DESIGN PATTERN

- The Prototype design pattern is a class creational pattern that allows clients to create duplicates of prototype objects at run-time without knowing the objects' specific type.

- According to the GoF, the intent of the prototype design pattern is to [1]
  - ✓ Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.


Clients use the prototype interface to create objects without knowing their specific types!
Client
+operation()
<<interface>>
Prototype
+clone(): Prototype*
Derived classes must obey the prototype interface!
ConcretePrototypeA
+clone(): Prototype*
ConcretePrototypeB
+clone(): Prototype*
Both Product and Creator Classes

PROTOTYPE DESIGN PATTERN EXAMPLE

Problem

Consider the enemy component created for a gaming system. The detailed design of the enemy component includes a wide variety of enemy specifications defined for the game, each with different profiles, weapons, etc. The game designers have identified the need to have each character provide a method for creating copies of themselves so that at any given point during the game, a character clone can be made with identical energy level, weapons, profiles, etc. This functionality is required to develop an enemy registry of different `Character` subtypes to create and add enemies at any point during the game.

Prototype Design Pattern Example

```
// The character interface.
class Character {

public:
  virtual void attack() = 0;
  // Other methods such as defend, move, etc.
};
```

```
class TerrestrialEnemyCharacter : public Character {

public:
  // Method definitions for terrestrial attack, defend, etc.

  // Duplicate this object.
  TerrestrialEnemyCharacter* duplicateTerrestrial() {

    // Use the copy constructor to create a copy of this object and
    // return it.
  return new TerrestrialEnemyCharacter (*this);
  }
};
```

Method to create copies of Terrestrial characters

```
class AerialEnemyCharacter : public Character {

public:
  // Method definitions for aerial attack, defend, etc.

  // Duplicate this object.
  AerialEnemyCharacter* duplicateAerial() {

    // Use the copy constructor to create a copy of this object and
    // return it.
    return new AerialEnemyCharacter(*this);
  }
};
```

Method to create copies of Aerial characters

Consider how client code would look like when programmed against this design…

Prototype Design Pattern Example

```
// Pre-Condition: A registry of 2 Enemy Characters has been created.
Character* createNextEnemyCharacter() {

  // Randomly pick the location of the next enemy character to be created.
  int nextEnemyLocation = rand() % MaxNumberOfEnemies;

  // Make sure that nextEnemyLocation is within proper bounds.

  // Retrieve the character at the nextEnemyLocation.
  Character* pCharacter = enemyRegistry[nextEnemyLocation];

  // The enemy character to be returned.
  Character* pNewCharacter = 0;

  // Determine if the character located at nextEnemyLocation is
  // Terrestrial.
  if( dynamic_cast<TerrestrialEnemyCharacter*>(pCharacter) != 0 ) {

    // Terrestrial Character, downcast it so that the duplicateTerrestrial
    // method can be used to duplicate the terrestrial character.
    TerrestrialEnemyCharacter* pTerrestrial =
                   dynamic_cast<TerrestrialEnemyCharacter*>(pCharacter);

    // Create the copy.  Clients are responsible for cleaning up memory
    // allocated for the copy.
    pNewCharacter = pTerrestrial->duplicateTerrestrial();
  }
  // Determine if the character located at nextEnemyLocation is Aerial.
  else if( dynamic_cast<AerialEnemyCharacter*>(pCharacter) != 0 ) {

    // Aerial Character, downcast it so that the duplicateAerial method
    // can be used to duplicate the aerial character.
    AerialEnemyCharacter* pAerial =
                   dynamic_cast<AerialEnemyCharacter*>(pCharacter);

    // Create the copy. Clients are responsible for cleaning up memory
    // allocated for the copy.
    pNewCharacter = pAerial->duplicateAerial();
  }
  else {
    // Invalid Character.
    pNewCharacter = new InvalidEnemyCharacter;
  }
  // Return the newly created enemy character.
  return pNewCharacter;
}
```

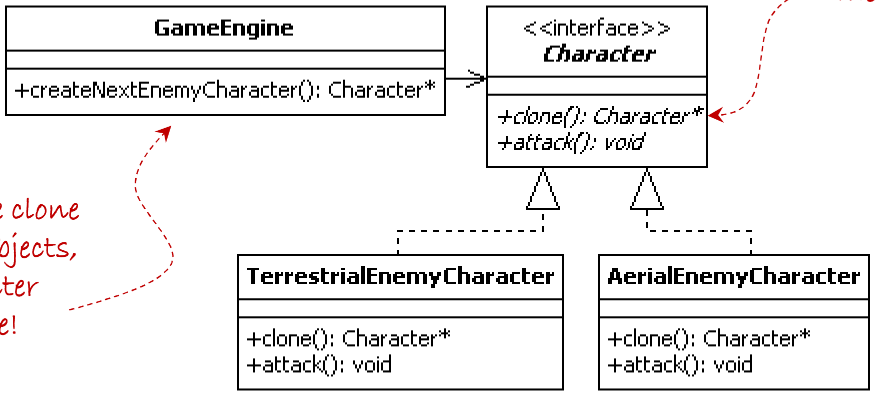
Yikes!

What happens if we introduce a new character to the game?

This violates OCP!

Let's see how we can improve things with the Prototype design pattern…

Prototype Design Pattern Example



Add the prototype clone interface method to the Character interface

The client can now rely on the clone interface to create Character objects, regardless of which Character subtype is actually active!

Concrete Characters must obey the Character interface!

Prototype Design Pattern Example

```cpp
// The character interface.
class Character {

public:
  // Interface method for initiating an attack.
  virtual void attack() = 0;

  // Interface method for duplicating objects at run-time.
  virtual Character* clone() = 0;

  // Other methods such as defend, move, etc.
};
```

```cpp
class TerrestrialEnemyCharacter : public Character {

public:
  // Method definitions for terrestrial attack, defend, etc.
  void attack() {

    // Display to the console the type of attack.
    cout<<"TerrestrialEnemyCharacter::attack()!\n";
  }

  // Implementation of the clone interface method to duplicate a
  // terrestrial enemy character.
  Character* clone(void) {

    // Use the copy constructor to create a copy of this object and
    // return it.
    return new TerrestrialEnemyCharacter (*this);
  }
};
```

```cpp
class AerialEnemyCharacter : public Character {

public:

  // Implement the attack interface method for aerial characters.
  void attack(void) {

    // Display to the console the type of attack.
    cout<<"AerialEnemyCharacter::attack()!\n";
  }

  // Implementation of the clone interface method to duplicate a
  // aerial enemy character.
  Character* clone(void) {

    // Use the copy constructor to create a copy of this object and
    // return it.
    return new AerialEnemyCharacter(*this);
  }
};
```

PROTOTYPE DESIGN PATTERN EXAMPLE

```cpp
// Pre-Condition: A registry of 2 Enemy Characters has been created.
Character* createNextEnemyCharacter() {

  // Randomly pick the location of the next enemy character to be created.
  int nextEnemyLocation = rand() % MaxNumberOfEnemies;

  // Make sure that nextEnemyLocation is within proper bounds.

  // Retrieve the character at the nextEnemyLocation.
  Character* pCharacter = enemyRegistry[nextEnemyLocation];

  // The enemy character to be returned.
  Character* pNewCharacter = 0;

  // Determine if the character located at nextEnemyLocation is
  // Terrestrial.
  if( dynamic_cast<TerrestrialEnemyCharacter*>(pCharacter) != 0 ) {

    // Terrestrial Character, downcast it so that the duplicateTerrestrial
    // method can be used to duplicate the terrestrial character.
    TerrestrialEnemyCharacter* pTerrestrial =
                   dynamic_cast<TerrestrialEnemyCharacter*>(pCharacter);

    // Create the copy.  Clients are responsible for cleaning up memory
    // allocated for the copy.
    pNewCharacter = pTerrestrial->duplicateTerrestrial();
  }
  // Determine if the character located at nextEnemyLocation is Aerial.
  else if( dynamic_cast<AerialEnemyCharacter*>(pCharacter) != 0 ) {

    // Aerial Character, downcast it so that the duplicateAerial method
    // can be used to duplicate the aerial character.
    AerialEnemyCharacter* pAerial =
                   dynamic_cast<AerialEnemyCharacter*>(pCharacter);

    // Create the copy. Clients are responsible for cleaning up memory
    // allocated for the copy.
    pNewCharacter = pAerial->duplicateAerial();
  }
  else {
    // Invalid Character.
    pNewCharacter = new InvalidEnemyCharacter;
  }
  // Return the newly created enemy character.
  return pNewCharacter;
}
```

```cpp
Character* enemyRegistry[MaxNumberOfEnemies];

// Function to clone an enemy chraacter.
Character* createNextEnemyCharacter() {

  // Randomly pick the location of the next enemy character to be created.
  int nextEnemyLocation = rand() % MaxNumberOfEnemies;

  // Make sure that nextEnemyLocation is within proper bounds.

  // Retrieve the character at the nextEnemyLocation.
  return enemyRegistry[nextEnemyLocation]->clone();
}
```

We can now replace this with that!

Prototype Design Pattern Example

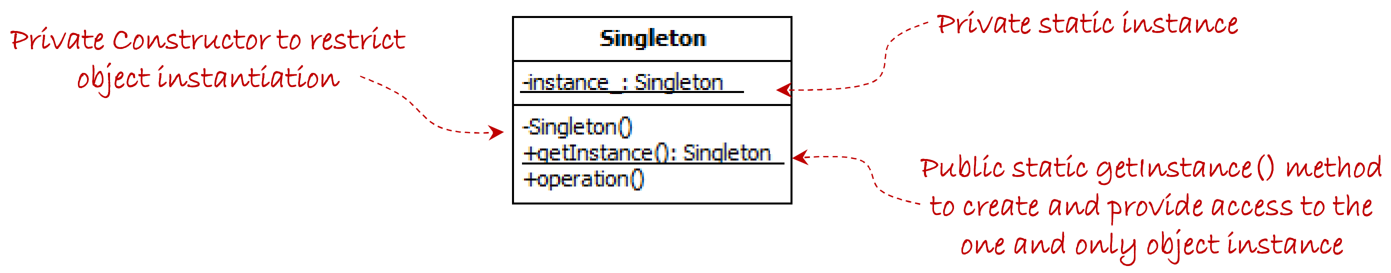
- ➢ Steps to design using the Prototype design pattern include:
  1. Identify and design the common interface that needs duplication. As part of the interface, the clone method (or equivalent) needs to be specified.
  2. Identify and design concrete products, which realize the interface created in Step 1.
  3. For each concrete product created in Step 2, implement the clone method in terms of that particular concrete product.

- ➢ Benefits of the Prototype design pattern include:
  - ✓ Clients are shielded from knowing the internal structure of objects; therefore, adding products at runtime becomes easier. This reduces the client's complexity.
  - ✓ Reduced number of classes; instead of having two classes (i.e., creator and product classes), the prototype is both, therefore eliminating the need for one class for each product.

SINGLETON DESIGN PATTERN

- The Singleton design pattern is an object creational design pattern used to prevent objects from being instantiated more than once in a running program.

- According to the GoF, the intent of the Singleton is to [1]
  - ✓ Ensure a class only has one instance, and provide a global point of access to it.


Private Constructor to restrict object instantiation
Singleton
-instance_ : Singleton
-Singleton()
+getInstance(): Singleton
+operation()
Private static instance
Public static getInstance() method to create and provide access to the one and only object instance

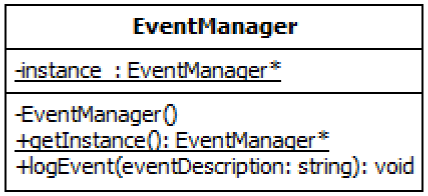
Singleton Design Pattern Example

Problem

Consider an application that requires event logging capabilities. The application consists of many different objects that generate events to keep track of their actions, status of operations, errors, or any other information of interest. A decision is made to create an event manager that can be accessed by all objects and used to manage all events in the system. Upon instantiation, the event manager creates an even list that gets updated as events are logged. At specific points during the software system's operation, these events are written to a file. To prevent conflicts, it is desirable that at any given time, there is only one instance of the event manager executing.

SINGLETON DESIGN PATTERN EXAMPLE

```cpp
#include "EventManager.h"
#include <iostream>

using namespace std;

// Initialize the instance_ static member attribute.
EventManager* EventManager::_instance = 0;

// Private constructor.
EventManager::EventManager(void)
{
    // Intentionally left blank.
}

// The global point of access to the EventManager.
EventManager* EventManager::getInstance(void) {

  // Determine if an instance of the EventManager has been created.
  if( _instance == 0 ) {

    // Create the one and only instance.
    _instance = new EventManager;
  }
  return _instance;
}

// The method that logs events.
void EventManager::logEvent(string eventDescription) {
  // Code to log event.
  cout<<eventDescription<<endl;
}
```


EventManager
-instance_: EventManager*
-EventManager()
+getInstance(): EventManager*
+logEvent(eventDescription: string): void

```cpp
// Sample usage of the singleton design pattern.
// Log events using the singleton event manager.
EventManager::getInstance()->logEvent("log some event here");

// Or store the pointer to log events later.
EventManager* pEventManager = EventManager::getInstance();
pEventManager->logEvent("log some event here");
```

Sample code to log events using the Singleton design pattern

Warning!
Singleton has been documented to cause memory leaks in multi-threaded environments!

SINGLETON DESIGN PATTERN EXAMPLE

- Steps to design using the Singleton design pattern:
  1. Set the visibility of the constructor to private
  2. Define a private static member attribute that can store a reference (i.e., a pointer) to the one instance of the singleton.
  3. Create a public static getInstance() method that can access the private constructor to instantiate one object of the singleton type and return it to clients.

- Benefits of the Singleton design pattern include:
  - ✓ It provides controlled access to single a single instance of a given type.
  - ✓ It has reduced namespace since it provides an alternatives to global variables.
  - ✓ It can be customized to permit a variable number of instances.

WHAT'S NEXT…

- In this session, we continued the discussion on creational design patterns, including:
  - ✓ Builder
  - ✓ Prototype
  - ✓ Singleton

- This finalizes the discussion on creational design patterns. In the next module, we will present structural and behavioral design patterns in detailed design. Specifically, we will cover:
  - ✓ Adapter
  - ✓ Composite
  - ✓ Façade
  - ✓ Iterator
  - ✓ Observer

References

- [1] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995.