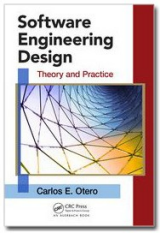
CHAPTER 6: CREATIONAL DESIGN PATTERNS

SESSION I: OVERVIEW OF DESIGN PATTERNS, ABSTRACT FACTORY

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



SESSION'S AGENDA

- Patterns in Detailed Design
  - ✓ Again, Architectural vs. Design Patterns.
- Classification of Design Patterns
  - ✓ Purpose
  - ✓ Scope
- Documenting Design Patterns
- Creational Design Patterns
  - ✓ Abstract Factory
  - ✓ Computer Store Example
- What's next…

PATTERNS IN DETAILED DESIGN

- In the previous sessions, the concept of patterns was introduced with an emphasis on software architecture.
  - ✓ During detailed design, a wide variety of design patterns exist for providing solutions to recurring problems; these are documented by the GoF.

- Remember, in 1994, Gamma, Helm, Johnson, and Vlissides—better known as the Gang of Four (GoF)—published their influential work that focused on a finer-grained set of object-oriented detailed design solutions that could be used in different problems "a million times over, without ever doing it the same way twice."
  - ✓ Influenced by Alexander's work on architectural patterns, they called these **Design Patterns**.
  - ✓ Their work resulted in the creation of a catalogue of 23 (detailed design) patterns.
  - ✓ Each pattern was described in detail, using a specific pattern specification format.

- Design patterns are recurring solutions to object-oriented design problems in a particular context.
  - ✓ They are different than architectural patterns!

PATTERNS IN DETAILED DESIGN

- Architectural vs. Design Patterns
  - ✓ Architectural patterns take place during the architecture activity of the software design phase; therefore, they serve best to identify the major components and interfaces of the system.
    - Design Patterns take place during detailed design; therefore, they serve best to identify the inner structure of components identified during the architecture activity.
  - ✓ Architectural patterns are too abstract to be translated directly to working code. Although they provide the general structure of the system, they do not fill the gaps required to create working code directly from the model.
    - Design Patterns provided the details necessary for creating working code.
  - ✓ Architectural patterns have a direct effect on the architecture of software and are associated with specific system types (e.g., interactive systems)
    - Design Patterns have no direct effect on the architecture of systems and are independent of the type of systems. That is, a specific design pattern, e.g., the observer, can be used within every component specified by all architectural patterns.

CLASSIFICATION OF DESIGN PATTERNS

- Design patterns can be classified based on:
  - ✓ Purpose
  - ✓ Scope

- The purpose of a design pattern identifies the essence of the pattern; therefore, it serves as fundamental differentiation criterion between design patterns. The three types of purposes used for classification are:
  - ✓ Creational
    - Patterns that deal with creation of objects.
  - ✓ Structural
    - Patterns that deal with creation of structures form existing ones.
  - ✓ Behavioral
    - Patterns that deal with how classes interact, the variation of behavior, and the assignment of responsibility between objects.

- The scope criterion captures whether a design pattern primarily applies to classes (during design time) or objects (during run-time).
  - ✓ Although we will use the scope criterion when discussing specific design patterns, scope is not used much in practice. The dominant criterion for classifying (and talking about) pattern is the *purpose* criterion (i.e., creational, structural, and behavioral).

DOCUMENTING DESIGN PATTERNS

Note:
The GoF identified 13 categories for documenting design patterns. Together, these categories provide detailed information of existing design patterns and provide direction for documenting future patterns.

Important:
In this course, we're not concerned with presenting this extensive documentation for each pattern, so you won't see this in future presentations of design patterns!

Category	Description
Name and Classification	The unique pattern name that reflects the essence of the patterns and its classification.
Intent	Describes the purpose of the pattern in such way that it is clear what types of design problems the pattern solves, what the pattern does, its rationale and intent.
Also Known As	A list of alternate well-known names for the pattern.
Motivation	En example scenario that serves as motivation for the application of the pattern.
Applicability	Describes the situations, or design problems, that lend themselves for the application of the design pattern. Provides examples of poor designs that can benefit from the pattern and ways for identifying these situations.
Structure	Provides a structural (e.g., UML class diagram) view of the design pattern.
Participants	List the classes and objects required in the design pattern and their responsibilities.
Collaborations	Provides information about how the participants work together to carry out their responsibilities.
Consequences	Describes the effects of the design pattern, good or bad, on the software solution.
Implementation	Provides information and techniques for successfully implementing the design pattern.
Sample Code	Provides sample code that demonstrates how to implement the design pattern in different programming languages.
Known Uses	Provides examples of real systems that employ the design pattern.
Related Patterns	Provides information about other design patterns that are related, or that can be used in combination with the design pattern.

CREATIONAL DESIGN PATTERNS

- Creational design patterns abstract and control the way objects are created in software applications.
  - ✓ They do so by specifying a common creational interface.

- By controlling the creational process with a common interface, enforcing creational policies become easier, therefore giving systems the ability to create objects that share a common interface but vary widely in structure and behavior.

- Examples of creational patterns include:
  - ✓ The Abstract Factory
  - ✓ The Factory Method
  - ✓ The Builder
  - ✓ The Prototype
  - ✓ The Singleton

THE ABSTRACT FACTORY

- The Abstract Factory is an object-creational design pattern intended to manage and encapsulate the creation of a set of objects that conceptually belong together and that represent a specific family of products.

- According to the GoF [1], the intent of the Abstract Factory is to
  - ✓ Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

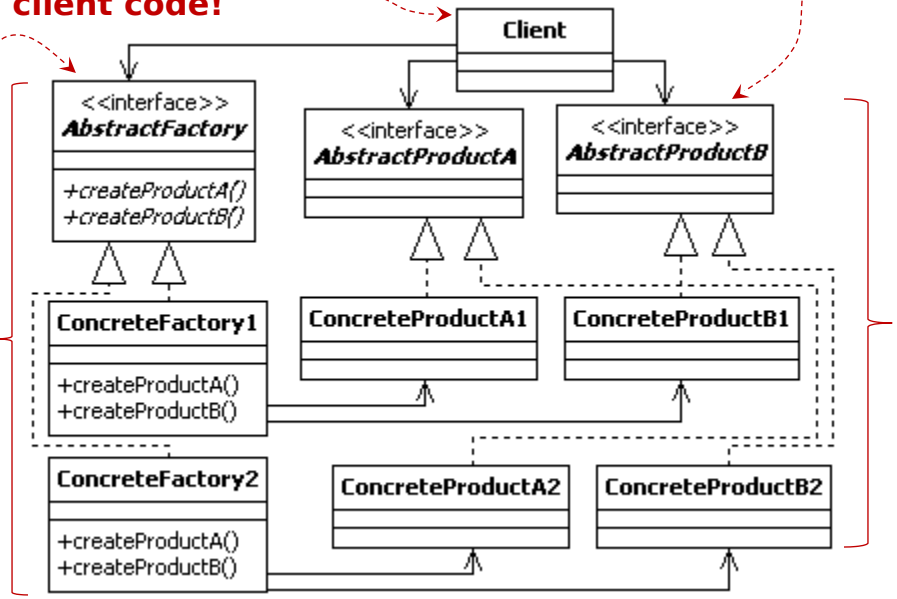
- Like all creational patterns, Abstract Factory is composed of *creator* classes and *product* classes.
  - ✓ As it will be seen, some creational patterns fuse the creator and product into one class.
  - ✓ At first, the Abstract Factory may seem confusing because of the number of classes required, however, when you take a closer look at the pattern, you'll see that the structural relationships required are modeled over and over the same way as new products are added to the design.

THE ABSTRACT FACTORY DESIGN PATTERN

Clients only know about creator and product interfaces! This allows us to vary behavior without changing client code!

Products need to obey the Product interface!

Factories need to obey the Factory interface!



Creator Classes

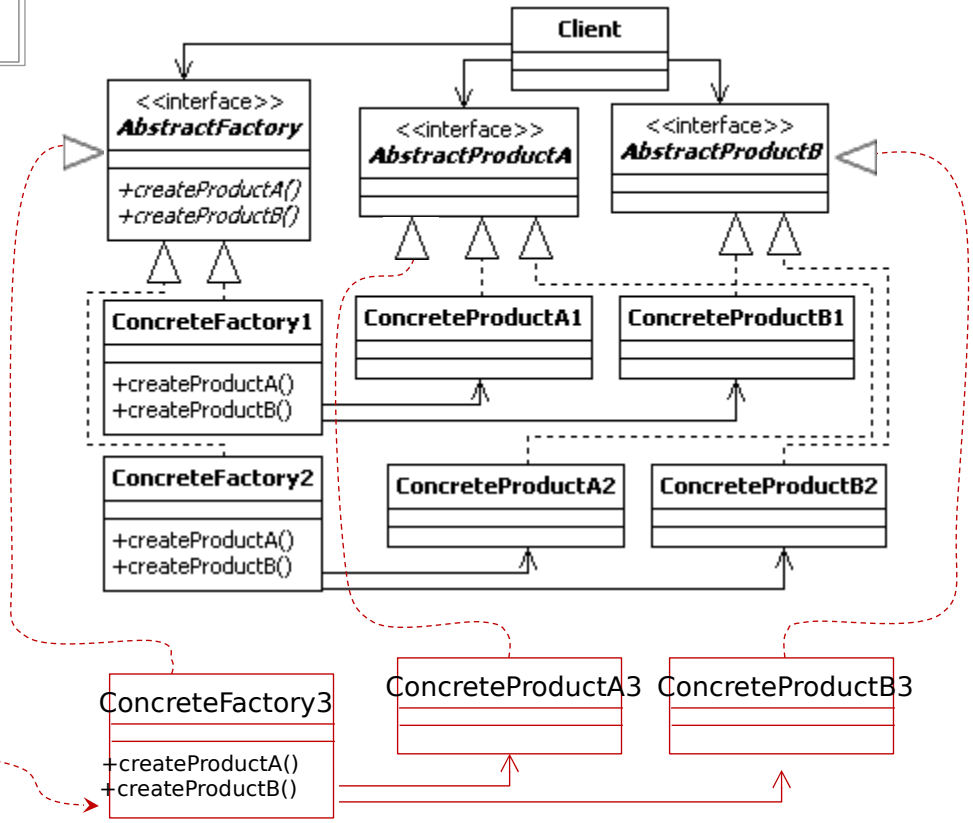
Product Classes

Important: Notice the Pattern! Adding other products for existing families requires adding another AbstractProduct interface and concrete product classes!

Important: Notice the Pattern! Adding a new family of products requires adding another Factory, AbstractProduct interface and concrete product classes!

THE ABSTRACT FACTORY DESIGN PATTERN

Important:
Adding a new family of products!

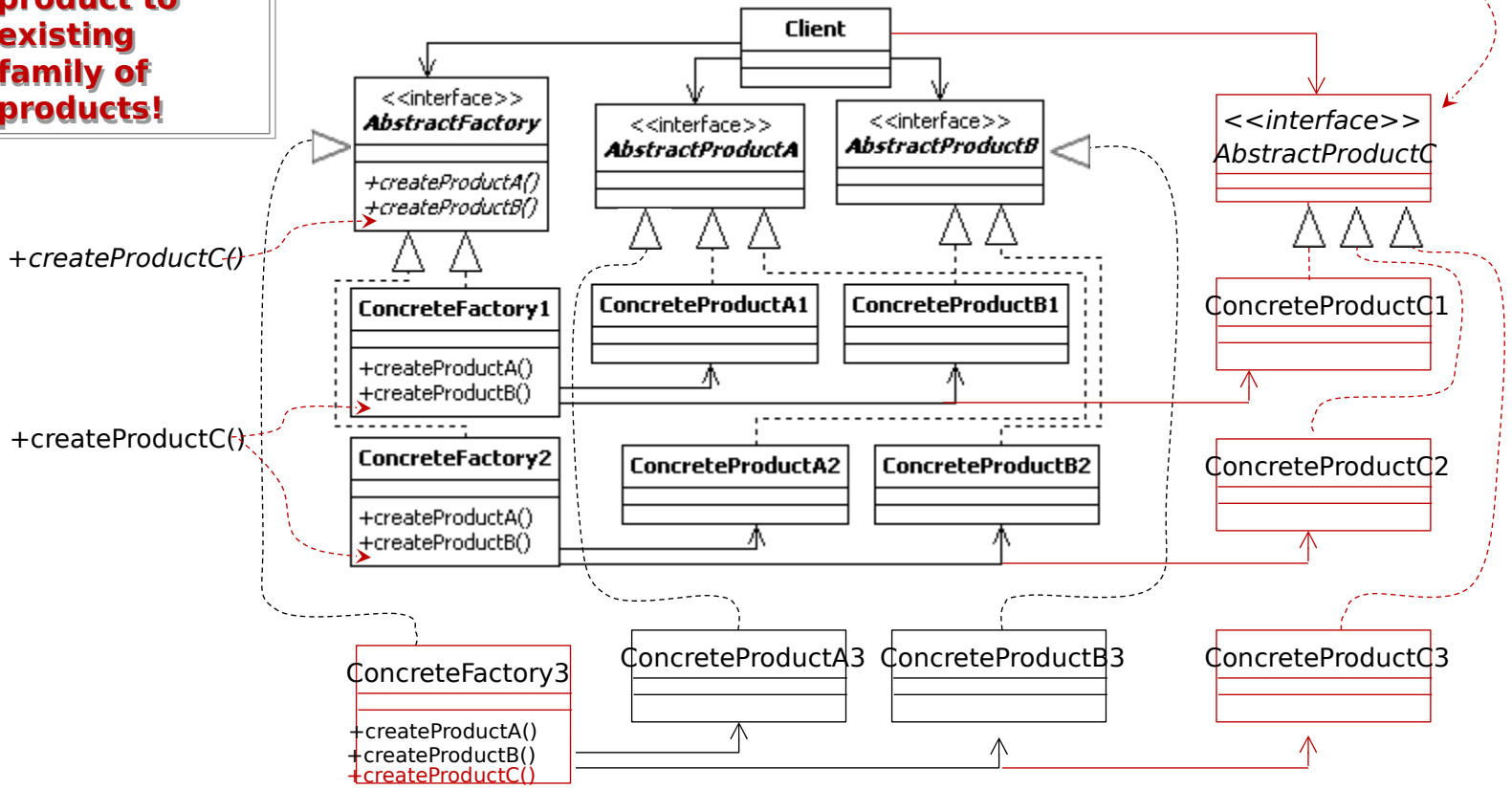


A New Family of Products has been added!

THE ABSTRACT FACTORY DESIGN PATTERN

Important:
Adding a new product to existing family of products!

A New Product has been added to existing families!



THE ABSTRACT FACTORY
— VERY SIMPLE AND FICTIONAL EXAMPLE —

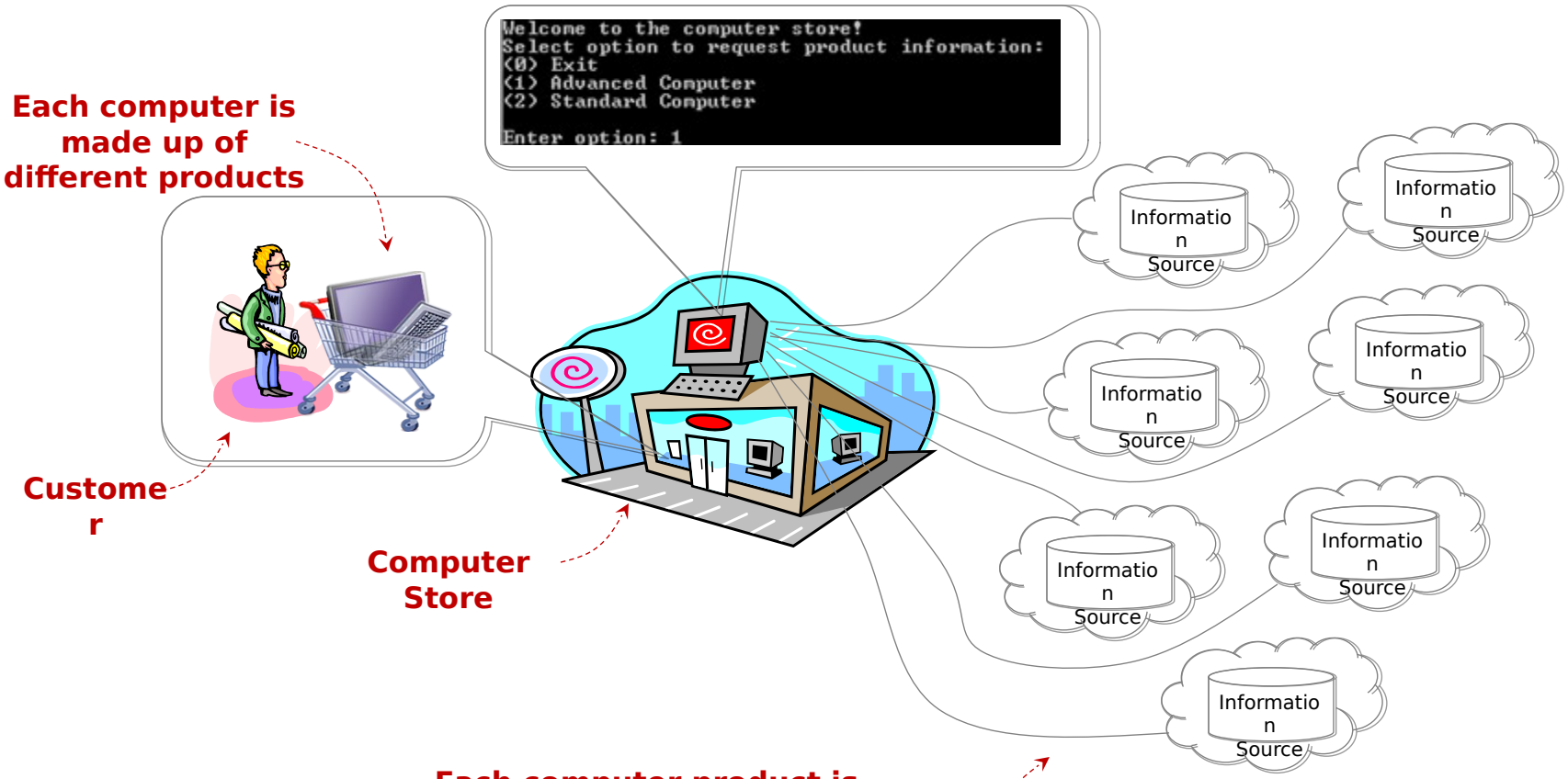
- Consider a software system for a computer store, where the store carries only two types of computers for sale:
  - ✓ Top of the line computer, we'll call these advanced computers
  - ✓ Inexpensive computers, we'll call these standard computers
  - ✓ Obviously, a computer store will need to carry more computers in the future!

- Advanced computers are made up of "advanced computer products," e.g. the latest multi-core CPU, wireless keyboard, advanced monitor (e.g., widescreen large 3D), advanced graphics & sound card, etc.
  - ✓ For simplicity, we'll only use CPU, keyboard, and Monitor for our example.

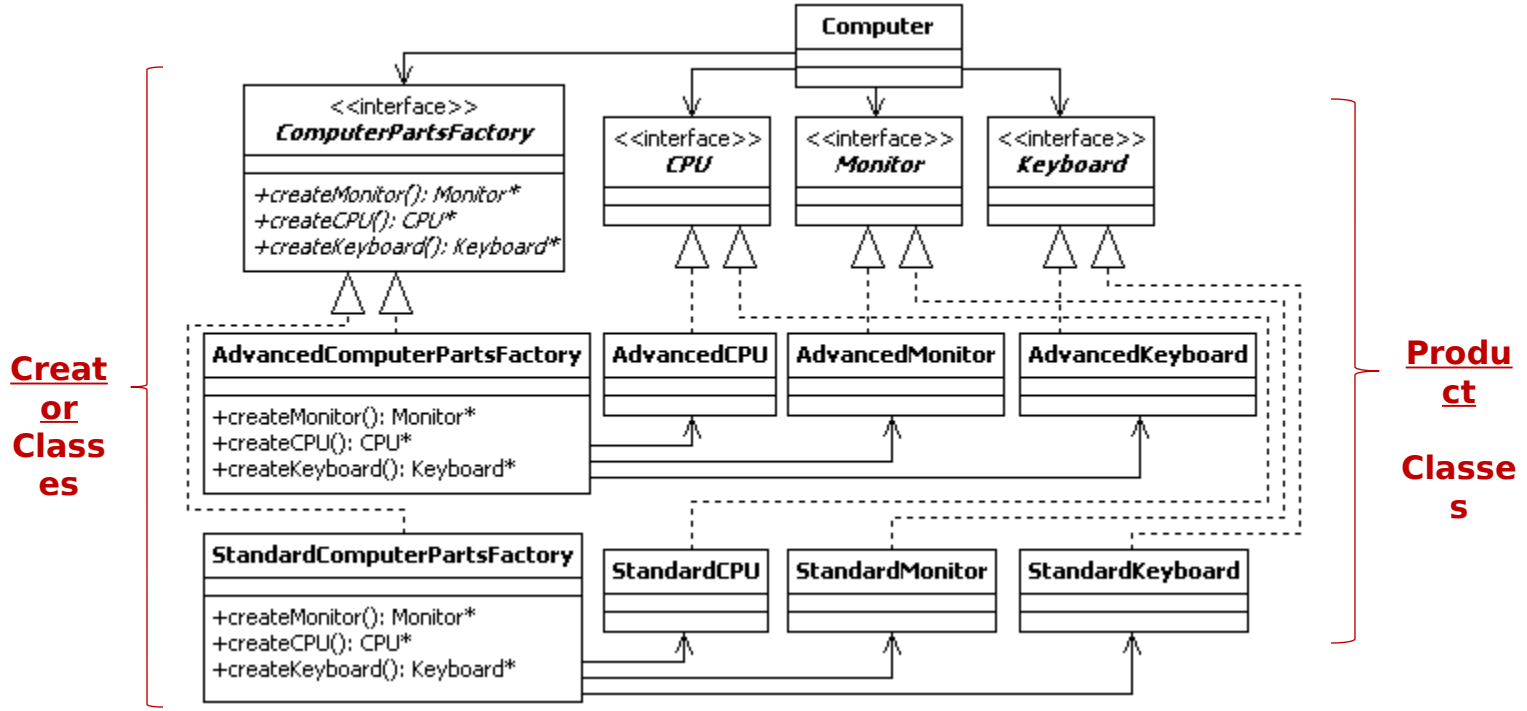
- Standard computers are made up of "standard computer products," e.g., single core CPU, wired keyboard, small screen monitor, low-grade graphics and sound, etc.
  - ✓ For simplicity, we'll only use CPU, keyboard, and Monitor for our example.

- The system is designed so that it searches remote information sources, e.g. online websites, remote databases, etc. for product information, such as:
  - ✓ Product reviews
  - ✓ Customer's comments from specific websites, e.g., Amazon.com
  - ✓ Manufacturers' comments
  - ✓ …

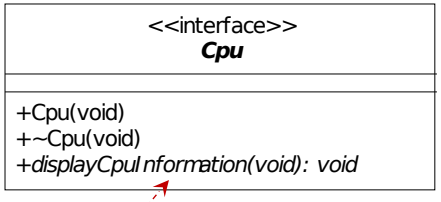
THE ABSTRACT FACTORY
— VERY SIMPLE AND FICTIONAL EXAMPLE —



ABSTRACT FACTORY FOR COMPUTER STORE



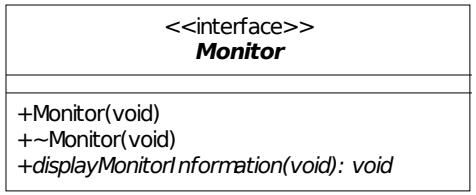
Let's break it down in the next slides…



Notice the italics to denote the abstract method

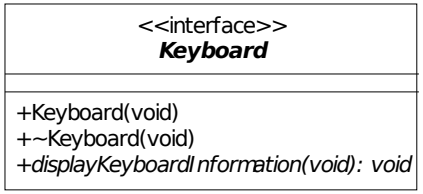
```
class Cpu
{
public:
    // Constructor.
    Cpu(void);

    // Destructor.
    virtual ~Cpu(void);

    // Interface method for retrieving the CPU's information.
    virtual void displayCpuInformation(void) = 0;

    // ... other methods for the Cpu class.
};
```

Notice how we create interfaces in C++




```
class Monitor
{
public:
    // Constructor.
    Monitor(void);

    // Destructor.
    virtual ~Monitor(void);

    // Interface method for retrieving the monitor's information.
    virtual void displayMonitorInformation(void) = 0;

    // ... other methods for the Monitor class.
};
```

This means that you cannot instantiate objects from this abstract class

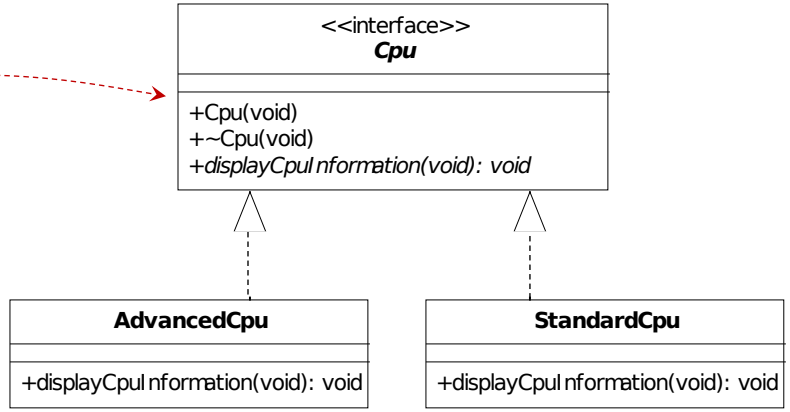



```
class Keyboard
{
public:
    // Constructor.
    Keyboard(void);

    // Destructor.
    virtual ~Keyboard(void);

    // Interface method for retrieving the keyboards's information.
    virtual void displayKeyboardInformation(void) = 0;

    // ... other methods for the Keyboard class.
};
```

Derived classes must provide implementation for this method before they can be instantiated!

THE CPU PRODUCT DESIGN

```cpp
class Cpu
{
public:
    // Constructor.
    Cpu(void);

    // Destructor.
    virtual ~Cpu(void);

    // Interface method for retrieving the CPU's information.
    virtual void displayCpuInformation(void) = 0;

    // ... other methods for the Cpu class.
};
```

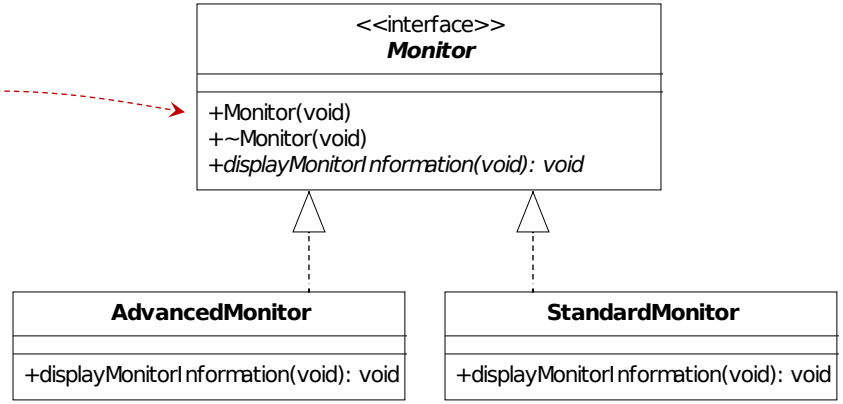



```cpp
#include "cpu.h"

class AdvancedCpu : public Cpu
{
public:
    // Constructor.
    AdvancedCpu(void);

    // Destructor.
    virtual ~AdvancedCpu(void);

    // Interface method for retrieving the CPU's information.
    virtual void displayCpuInformation(void);

    // ... other methods for the Cpu class.
};
```

```cpp
#include "cpu.h"

class StandardCpu : public Cpu
{
public:
    // Constructor.
    StandardCpu(void);

    // Destructor.
    virtual ~StandardCpu(void);

    // Interface method for retrieving the CPU's information.
    virtual void displayCpuInformation(void);

    // ... other methods for the standard Cpu class.
};
```

THE MONITOR PRODUCT DESIGN

```
class Monitor
{
public:
    // Constructor.
    Monitor(void);

    // Destructor.
    virtual ~Monitor(void);

    // Interface method for retrieving the monitor's information.
    virtual void displayMonitorInformation(void) = 0;

    // ... other methods for the Monitor class.
};
```



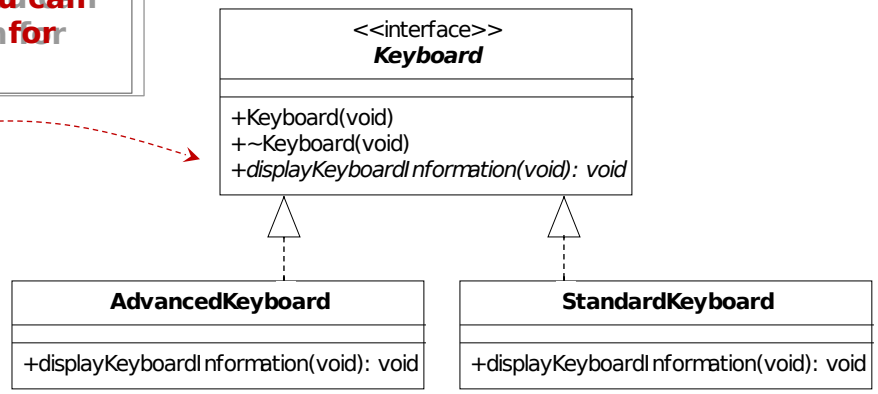

```
#include "Monitor.h"

class AdvancedMonitor :public Monitor
{
public:
    // Constructor.
    AdvancedMonitor(void);

    // Destructor.
    ~AdvancedMonitor(void);

    // Interface method for retrieving the monitor's information.
    virtual void displayMonitorInformation(void);

    // ... other advanced monitor methods.
};
```

```
#include "monitor.h"

class StandardMonitor : public Monitor
{
public:
    // Constructor.
    StandardMonitor(void);

    // Destructor.
    ~StandardMonitor(void);

    // Interface method for retrieving the monitor's information.
    virtual void displayMonitorInformation(void);
};
```

THE KEYBOARD PRODUCT DESIGN

Hopefully by this point you can start seeing the pattern for designing products!

```
class Keyboard
{
public:
    // Constructor.
    Keyboard(void);

    // Destructor.
    virtual ~Keyboard(void);

    // Interface method for retrieving the keyboards's information.
    virtual void displayKeyboardInformation(void) = 0;

    // ... other methods for the Keyboard class.
};
```




```
#include "keyboard.h"

class AdvancedKeyboard : public Keyboard
{
public:
    // Constructor.
    AdvancedKeyboard(void);

    // Destructor.
    virtual ~AdvancedKeyboard(void);

    // Interface method for retrieving the keyboards's information.
    virtual void displayKeyboardInformation(void);

    // ... other methods for the Keyboard class.
};
```

```
#include "keyboard.h"

class StandardKeyboard : public Keyboard
{
public:
    // Constructor.
    StandardKeyboard(void);

    // Destructor.
    virtual ~StandardKeyboard(void);

    // Interface method for retrieving the keyboards's information.
    virtual void displayKeyboardInformation(void);

    // ... other methods for the Keyboard class.
};
```

THE CPU PRODUCT IMPLEMENTATION

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

using std::cout;

// Constructor.
AdvancedCpu::AdvancedCpu(void)
{
    // Intentionally left blank.
}

// Destructor.
AdvancedCpu::~AdvancedCpu(void)
{
    // Intentionally left blank.
}

// Interface method for retrieving the cpu's information.
void AdvancedCpu::displayCpuInformation(void)
{
    // Since this is an example, we will assume that the advanced CPU's information
    // will be retrieved from information source A, e.g., database A, file A, etc.
    // This may require a particular database connection, file access, etc.
    cout<<"\nInformation retreived from source A.\nDisplaying the advanced cpu's information.\n\n";
}
```

The code in this function knows how to retrieve information from data source A, which can use specific format, location, etc.


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

using std::cout;

// Constructor.
StandardCpu::StandardCpu(void)
{
    // Intentionally left blank.
}

// Destructor.
StandardCpu::~StandardCpu(void)
{
    // Intentionally left blank.
}

// Interface method for retrieving the cpu's information.
void StandardCpu::displayCpuInformation(void)
{
    // Since this is an example, we will assume that the standard CPU's information
    // will be retrieved from information source B, e.g., database B, file B, etc.
    // This may require a particular database connection, file access, etc.
    cout<<"\nInformation retreived from source B.\nDisplaying the standard cpu's information.\n\n";
}
```

The code in this function knows how to retrieve information from data source B, which can use specific format, location, etc.

OTHER PRODUCT IMPLEMENTATION

```cpp
// Interface method for retrieving the cpu's information.
void AdvancedKeyboard::displayKeyboardInformation(void)
{
    // Since this is an example, we will assume that the advanced keyboard's information
    // will be retrieved from information source A, e.g., database A, file A, etc.
    // This may require a particular database connection, file access, etc.
    cout<<"\nInformation retreived from source A.\nDisplaying the advanced keyboard's information.\n\n";
}
```

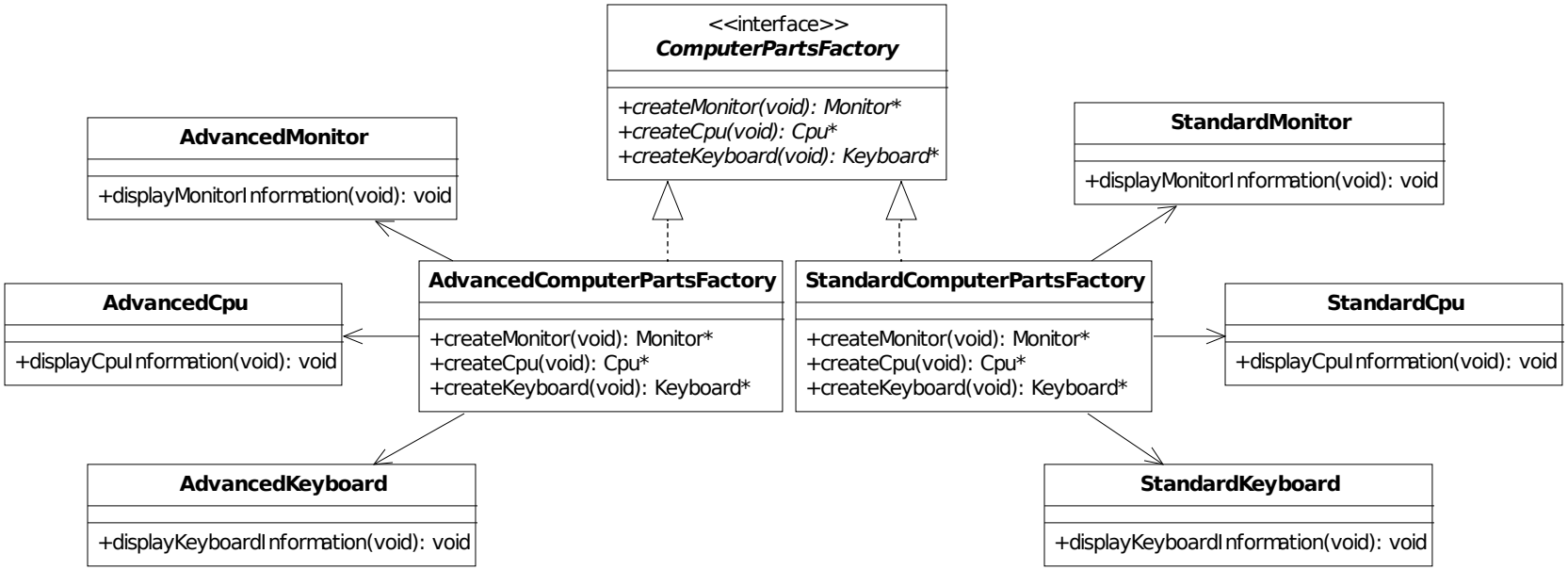
```cpp
// Interface method for retrieving the cpu's information.
void StandardKeyboard::displayKeyboardInformation(void)
{
    // Since this is an example, we will assume that the advanced keyboard's information
    // will be retrieved from information source B, e.g., database B, file B, etc.
    // This may require a particular database connection, file access, etc.
    cout<<"\nInformation retreived from source B.\nDisplaying the standard keyboard's information.\n\n";
}
```

All other products are implemented using the same pattern!

```cpp
// Interface method for retrieving the monitor's information.
void AdvancedMonitor::displayMonitorInformation(void)
{
    // Since this is an example, we will assume that the advanced monitor's information
    // will be retrieved from information source A, e.g., database A, file A, etc.
    // This may require a particular database connection, file access, etc.
    cout<<"\nInformation retreived from source A.\nDisplaying the advanced monitor's information.\n\n";
}
```

```cpp
// Interface method for retrieving the monitor's information.
void StandardMonitor::displayMonitorInformation()
{
    // Since this is an example, we will assume that the standard monitor's information
    // will be retrieved from information source B, e.g., database B, file B, etc.
    // This may require a particular database connection, file access, etc.
    cout<<"\nInformation retreived from source B.\nDisplaying the standard monitor's information.\n\n";
}
```

DESIGN THE FACTORY INTERFACE AND CONCRETE FACTORIES

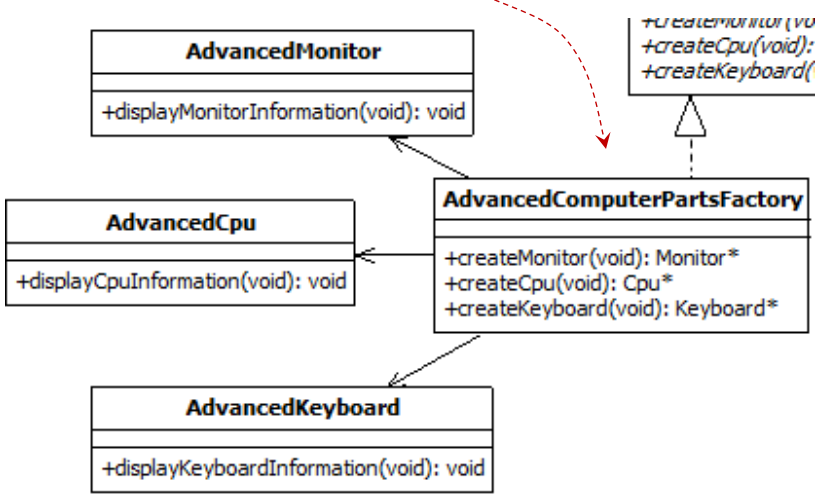


Important:
This design connects the products designed in the previous slides with the factories used to abstract their creation!

THE ADVANCED COMPUTER PARTS FACTORY

```
#include "AdvancedComputerPartsFactory.h"
#include "AdvancedMonitor.h"
#include "AdvancedCpu.h"
#include "AdvancedKeyboard.h"


AdvancedComputerPartsFactory::AdvancedComputerPartsFactory(void)
{
    // Intentionally left blank.
}

AdvancedComputerPartsFactory::~AdvancedComputerPartsFactory(void)
{
    // Intentionally left blank.
}

// Create and return an advanced monitor.
Monitor* AdvancedComputerPartsFactory::createMonitor()
{
    // This example assumes that client callers will deallocate memory.
    return new AdvancedMonitor;
}

// Create and return an advanced keyboard.
Keyboard* AdvancedComputerPartsFactory::createKeyboard()
{
    // This example assumes that client callers will deallocate memory.
    return new AdvancedKeyboard;
}

// Create and return an advanced cpu.
Cpu* AdvancedComputerPartsFactory::createCpu()
{
    // This example assumes that client callers will deallocate memory.
    return new AdvancedCpu;
}
```

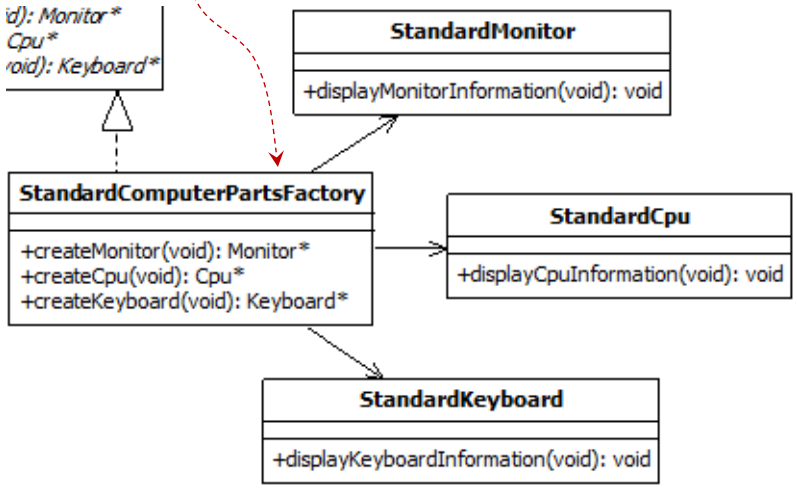
These are equivalent!



THE STANDARD COMPUTER PARTS FACTORY

```cpp
#include "StandardComputerPartsFactory.h"
#include "StandardMonitor.h"
#include "StandardKeyboard.h"
#include "StandardCpu.h"

// Constructor.
StandardComputerPartsFactory::StandardComputerPartsFactory(void)
{
    // Intentionally left blank.
}

// Destructor.
StandardComputerPartsFactory::~StandardComputerPartsFactory(void)
{
    // Intentionally left blank.
}

// Create and return the standard monitor object.
Monitor* StandardComputerPartsFactory::createMonitor(void)
{
    // This example assumes that client callers will deallocate memory.
    return new StandardMonitor;
}

// Create and return the standard keyboard object.
Keyboard* StandardComputerPartsFactory::createKeyboard(void)
{
    // This example assumes that client callers will deallocate memory.
    return new StandardKeyboard;
}

// Create and return the standard CPU object.
Cpu* StandardComputerPartsFactory::createCpu(void)
{
    // This example assumes that client callers will deallocate memory.
    return new StandardCpu;
}
```

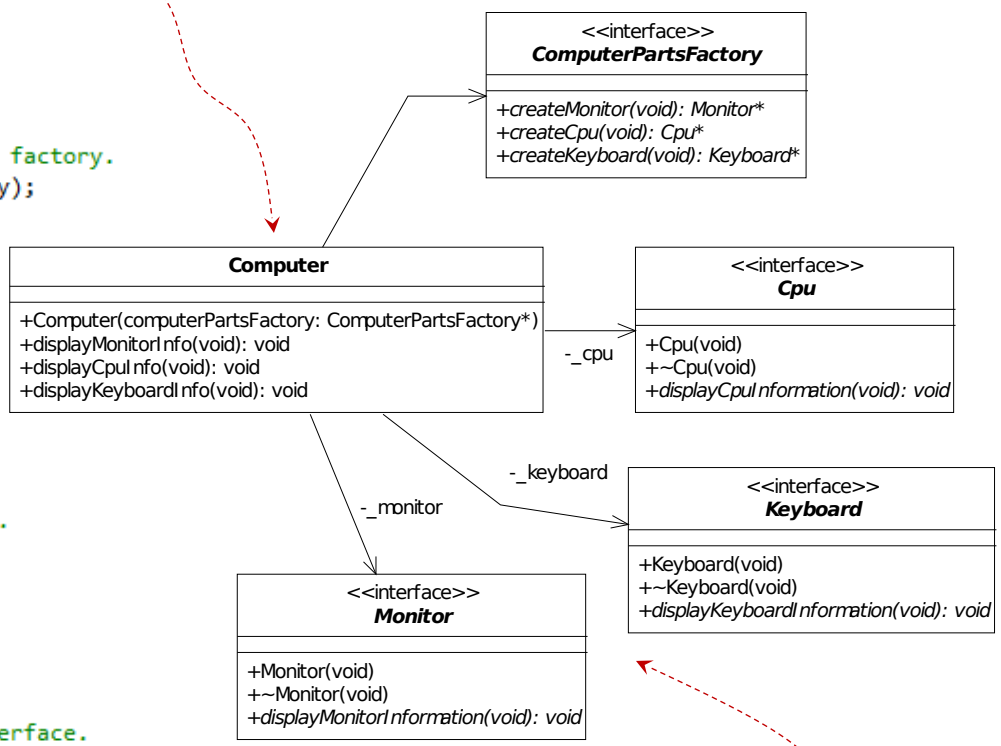
These are equivalent!



THE CLIENT COMPUTER DESIGN

```
class ComputerPartsFactory;
class Monitor;
class Cpu;
class Keyboard;

class Computer
{
public:
    // Constructor parameterized with a computer parts factory.
    Computer(ComputerPartsFactory* computerPartsFactory);

    // Destructor.
    virtual ~Computer(void);

    // Display detailed information about the monitor.
    void displayMonitorInfo(void);

    // Display detailed information about the CPU.
    void displayCpuInfo(void);

    // Display detailed information about the keyboard.
    void displayKeyboardInfo(void);

    // All other computer methods.
    // Destructor needs to clean up memory.

private:
    Monitor* _monitor;   // Pointer to the monitor interface.
    Cpu* _cpu;           // Pointer to the Cpu interface.
    Keyboard* _keyboard; // Pointer to the Keyboard interface.
};
```

These are equivalent!



Code from model!

Notice the named associations, which are specified with private visibility!

THE CLIENT COMPUTER DESIGN

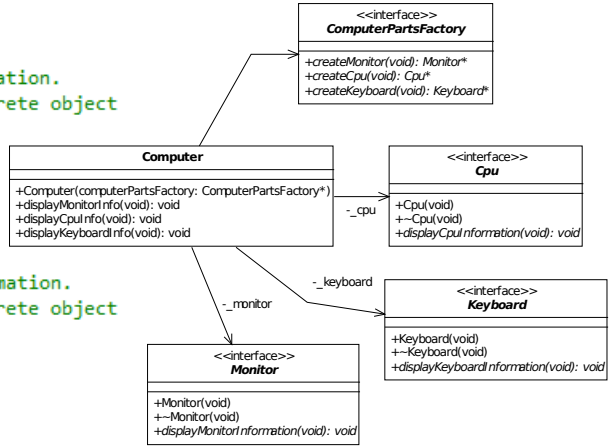
The Computer object is configured with a Factory object. The Computer object delegates creation of products to its Factory!

If you want and advanced computer, pass in an AdvancedComputerFactory, otherwise, pass in a StandardComputerFactory

```
#include "Computer.h"
#include "ComputerPartsFactory.h"
#include "Monitor.h"
#include "Keyboard.h"
#include "Cpu.h"

Computer::Computer(ComputerPartsFactory* computerPartsFactory)
{
    // This example assumes a valid pointer is passed in.

    // Retrieve the monitor object.  This depends on the factory passed in.
    _monitor = computerPartsFactory->createMonitor();

    // Retrieve the keyboard object.  This depends on the factory passed in.
    _keyboard = computerPartsFactory->createKeyboard();

    // Retrieve the cpu object.  This depends on the factory passed in.
    _cpu = computerPartsFactory->createCpu();
}
```

Since our design relies on interfaces only, this code works for both standard and advanced computers!

```
void Computer::displayMonitorInfo(void)
{
    // Use  the interface method to display the monitor information.
    // Note that at this point, we don't know what actual concrete object
    // will be providing this service.
    _monitor->displayMonitorInformation();
}

void Computer::displayKeyboardInfo(void)
{
    // Use  the interface method to display the keyboard information.
    // Note that at this point, we don't know what actual concrete object
    // will be providing this service.
    _keyboard->displayKeyboardInformation();
}

void Computer::displayCpuInfo(void)
{
    // Use  the interface method to display the CPU information.
    // Note that at this point, we don't know what actual concrete object
    // will be providing this service.
    _cpu->displayCpuInformation();
}
```



ABSTRACT FACTORY EXAMPLE – PUTTING IT ALL TOGETHER

1

```
int main(int argc, char* argv[])
{
    // Create the advanced computer parts factory.
    AdvancedComputerPartsFactory advancedFactory;

    // Create the standard computer parts factory.
    StandardComputerPartsFactory standardFactory;

    // The pointer to the computer object.
    Computer* pComputer = 0;
```

2

```
int option = 1;
cout<<"Welcome to the computer store!\n";

while( option != 0 )
{
    cout<<"Select option to request product information:\n"
        <<"(0) Exit\n"
        <<"(1) Advanced Computer\n"
        <<"(2) Standard Computer\n\n"
        <<"Enter option: ";

    cin>>option;
```

3

```
if(option == 1)
    pComputer = new Computer(&advancedFactory);
else if( option == 2 )
    pComputer = new Computer(&standardFactory);

// Notice that regardless of the type of computer, we
// can obtain information via its well-defined interfaces!
pComputer->displayMonitorInfo();
pComputer->displayKeyboardInfo();
pComputer->displayCpuInfo();

delete pComputer;
```

4

```
Welcome to the computer store!
Select option to request product information:
(0) Exit
(1) Advanced Computer
(2) Standard Computer

Enter option: 1

Information retreived from source A.
Displaying the advanced monitor's information.


Information retreived from source A.
Displaying the advanced keyboard's information.


Information retreived from source A.
Displaying the advanced cpu's information.

Select option to request product information:
(0) Exit
(1) Advanced Computer
(2) Standard Computer

Enter option:
```

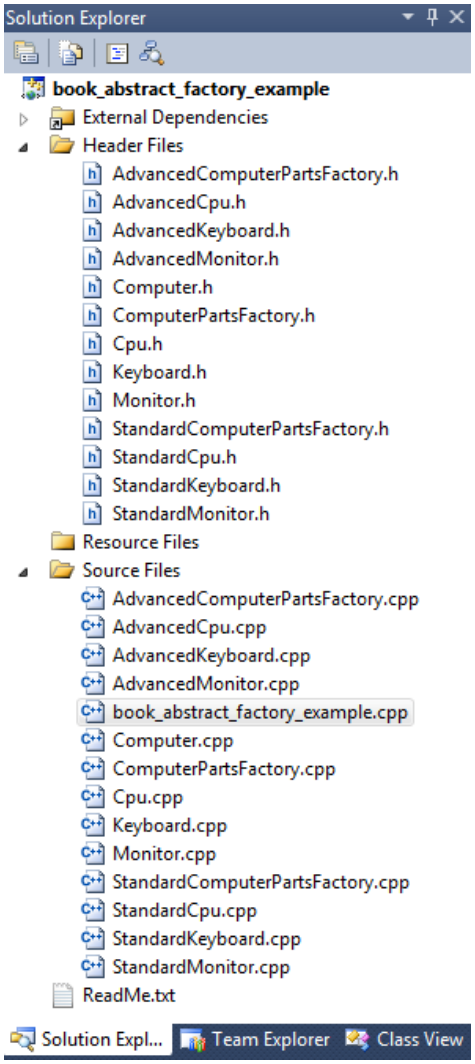
Notice how we configure the Computer object with a Factory object!

ABSTRACT FACTORY STEP-BY-STEP SUMMARY

- As seen, the Abstract Factory pattern can be used over and over to support new family of products or to add new products to existing ones. When designing with the Abstract Factory, execute the following steps:
  1. Design the product interfaces (e.g., Cpu, Monitor, and Keyboard)
  2. Identify the different families or groups required for the problem (e.g., standard vs. advanced computers)
  3. For each group identified in step 2, design concrete products that realize the respective product interfaces identified in step 1.
  4. Create the factory interface (e.g., ComputerPartsFactory). The factory interface contains *n* interface methods, one for each product interface identified in step 1.
  5. For each family or group identified in step 2, create concrete factories that realize the factory interface created in step 4.
  6. Associate each concrete factory from step 5 with their respective products from step 3.
  7. Create the Client (e.g., Computer) which is associated with both product and factory interfaces created in steps 1 and 4, respectively.

CONSEQUENCES OF ABSTRACT FACTORY

- Cons
  - ✓ Large number of classes are required
- Pros
  - ✓ Isolates concrete product classes so that reusing them becomes easier
  - ✓ Promotes consistency within specific product families.
  - ✓ Adding new families of products require no modification to existing code.
    - Additions are made through extension, therefore, obeying the OCP.
  - ✓ Helps minimize the degree of complexity when changing the system to meet future needs.
    - i.e., increases modifiability



WHAT'S NEXT…

- In this session, we presented fundamentals concepts of design patterns and creational design patterns, including:
  - ✓ Abstract Factory

- In the next sessions, we will continue the presentation on creational design patterns, including:
  - ✓ Factory method
  - ✓ Builder
  - ✓ Prototype
  - ✓ Singleton

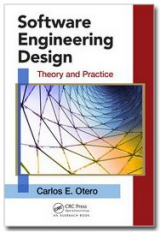
REFERENCES

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

CHAPTER 6: CREATIONAL DESIGN PATTERNS

SESSION II: FACTORY METHOD

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



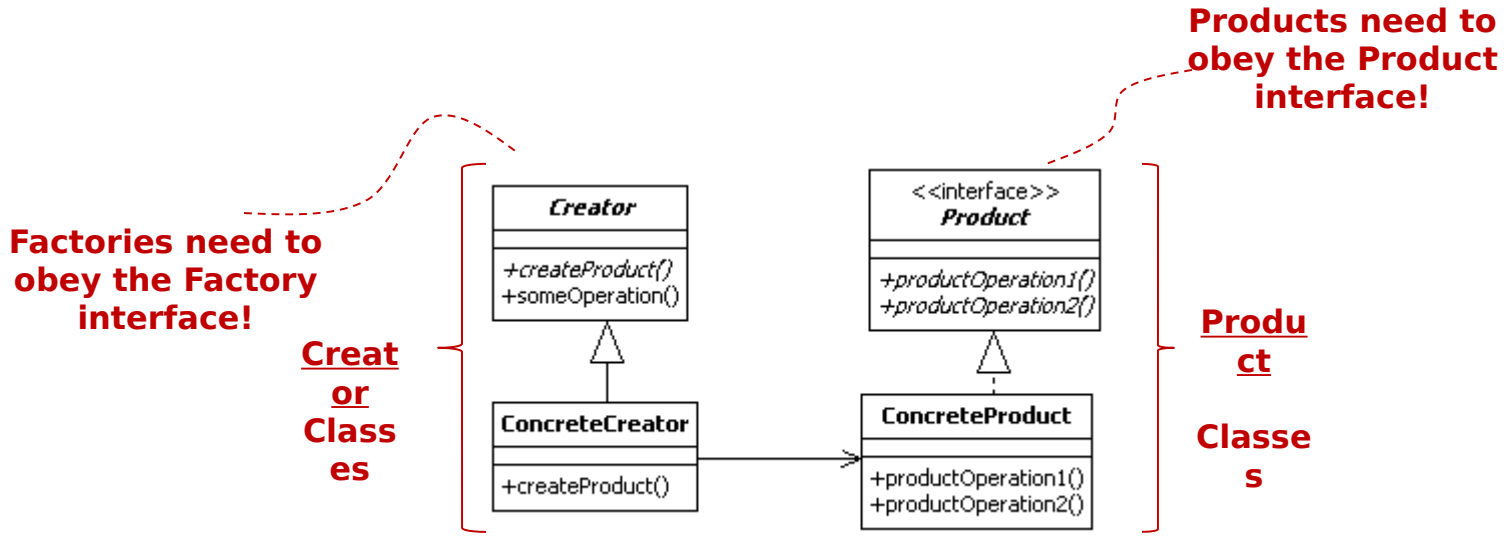
SESSION'S AGENDA

- Creational Patterns in Detailed Design
- Factory Method Design Pattern
  - ✔ Java Example
  - ✔ C++ Example
- Null Object Design Pattern
- Benefits of Factory Method
- What's next…

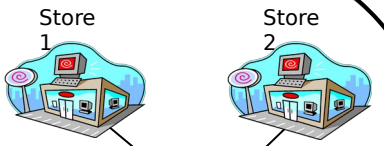
FACTORY METHOD DESIGN PATTERN

- The Factory Method design pattern is a class creational pattern used to encapsulate and defer object instantiation to derived classes.

- According to the GoF, the intent of the factory method is to [1]
  - ✔ Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.



FACTORY METHOD EXAMPLE

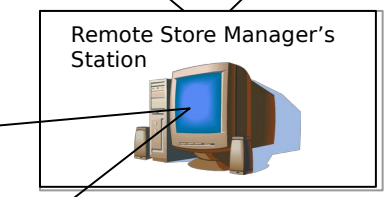


Consider this code, which handles all computers from stores #1 and #2. What's

When you see code like this, you know that when it comes time for changes or extension, you'll have to reopen this code and examine what needs to be added or deleted. In addition, more often than not, this kind of code ends up in several parts of the application, making maintenance more difficult and error-prone.

Java code

```
public class ComputerStore {

    public void displayComputer(String type) {

        // The computer object.
        Computer computer = null;

        if( type.equals("DellPC")) {

            // Instantiate a Dell computer.
            computer = new DellPC();

        } else if( type.equals("GatewayPC")) {

            // Instantiate a Gateway computer.
            computer = new GatewayPC();

        } else if(type.equals("Mac")) {

            // Instantiate a MAC computer.
            computer = new Mac();
        }

        // Display the computer's information.
        computer.displayMemoryInfo();
        computer.displayMonitorInfo();
        computer.displayProcessorInfo();
        computer.displayCustomerRatings();
        computer.displayCost();
    }
```




```
public Computer orderComputer(String type) {
    // The computer object.
    Computer computer = null;

    if( type.equals("DellPC")) {

        // Instantiate a Dell computer.
        computer = new DellPC();

    } else if( type.equals("GatewayPC")) {

        // Instantiate a Gateway computer.
        computer = new GatewayPC();

    } else if(type.equals("Mac")) {

        // Instantiate a MAC computer.
        computer = new Mac();
    }

    // Process the computer sale.
    processSale(computer.getProductId());

    return computer;
}
```

This code would have to support all types of computer in all sites!

Let's take a closer look in the next slide…

FACTORY METHOD EXAMPLE

This code is not closed for modification. If a new computer is added to the inventory of store #1, we have to get into this code and modify it for all other stores!

```
public void displayComputer(String type) {

    // The computer object.
    Computer computer = null;

    if( type.equals("DellPC")) {

        // Instantiate a Dell computer.
        computer = new DellPC();

    } else if( type.equals("GatewayPC")) {

        // Instantiate a Gateway computer.
        computer = new GatewayPC();

    } else if(type.equals("Mac")) {

        // Instantiate a MAC computer.
        computer = new Mac();
    }

    // Display the computer's information.
    computer.displayMemoryInfo();
    computer.displayMonitorInfo();
    computer.displayProcessorInfo();
    computer.displayCustomerRatings();
    computer.displayCost();
}
```

Design Principle: Encapsulate what varies!

This is what varies. As the computer selection changes over time (for all computer stores), you'll have to modify this code over and over.

This is what we expect to stay the same.

Important: By coding to an interface, we can insulate ourselves from future changes!

Let's see how the Factory Method can be used to solve this problem…

FACTORY METHOD EXAMPLE

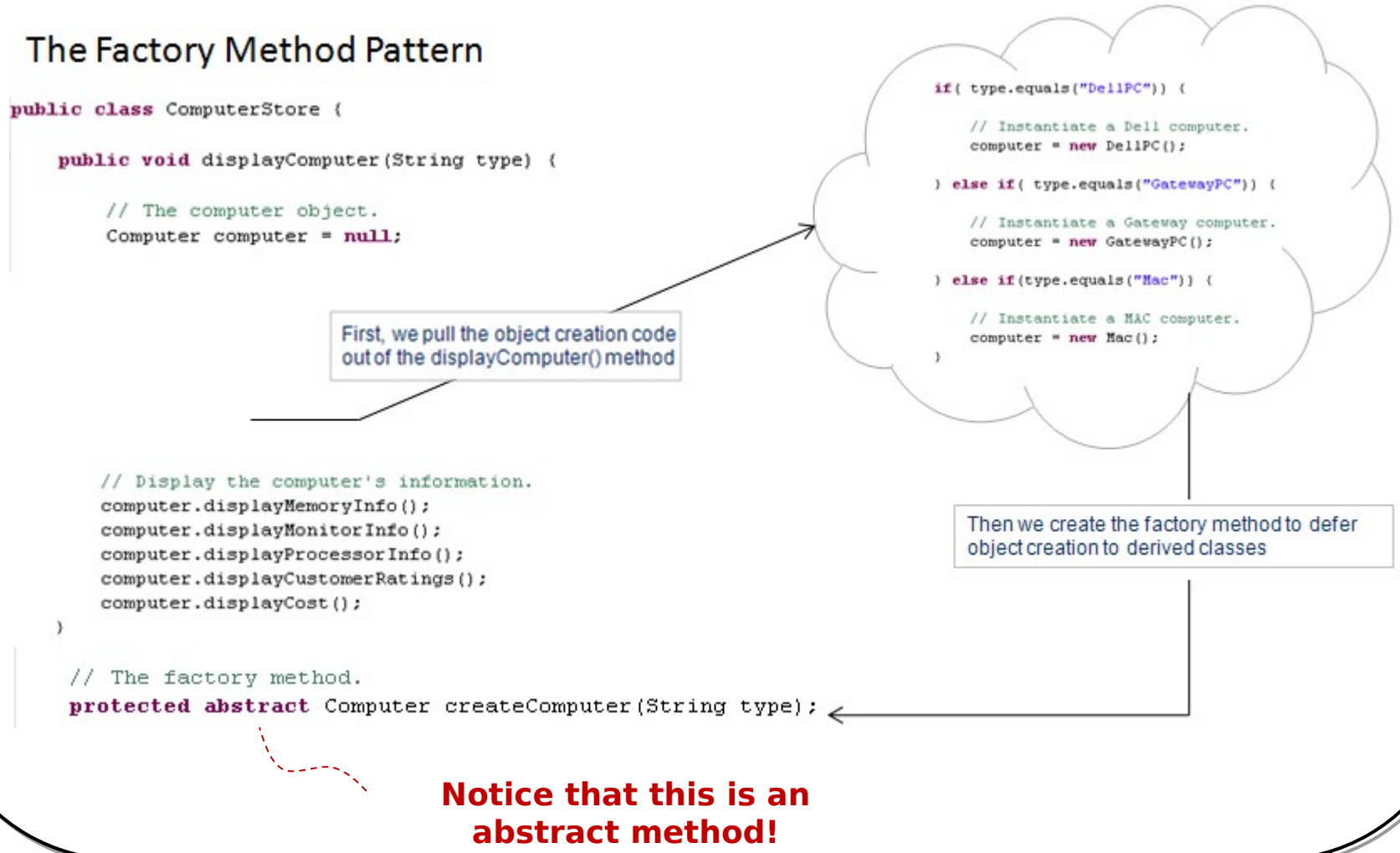
The Factory Method Pattern

```
public class ComputerStore {

    public void displayComputer(String type) {

        // The computer object.
        Computer computer = null;

        // Display the computer's information.
        computer.displayMemoryInfo();
        computer.displayMonitorInfo();
        computer.displayProcessorInfo();
        computer.displayCustomerRatings();
        computer.displayCost();
    }
```

First, we pull the object creation code out of the displayComputer() method

```
if( type.equals("DellPC")) {

    // Instantiate a Dell computer.
    computer = new DellPC();

} else if( type.equals("GatewayPC")) {

    // Instantiate a Gateway computer.
    computer = new GatewayPC();

} else if(type.equals("Mac")) {

    // Instantiate a MAC computer.
    computer = new Mac();
}
```

FACTORY METHOD EXAMPLE



FACTORY METHOD EXAMPLE

The Factory Method Pattern

```
public abstract class ComputerStore {

    public void displayComputer(String type) {

        // The computer object.
        Computer computer = createComputer(type);

        // Display the computer's information.
        computer.displayMemoryInfo();
        computer.displayMonitorInfo();
        computer.displayProcessorInfo();
        computer.displayCustomerRatings();
        computer.displayCost();
    }
```

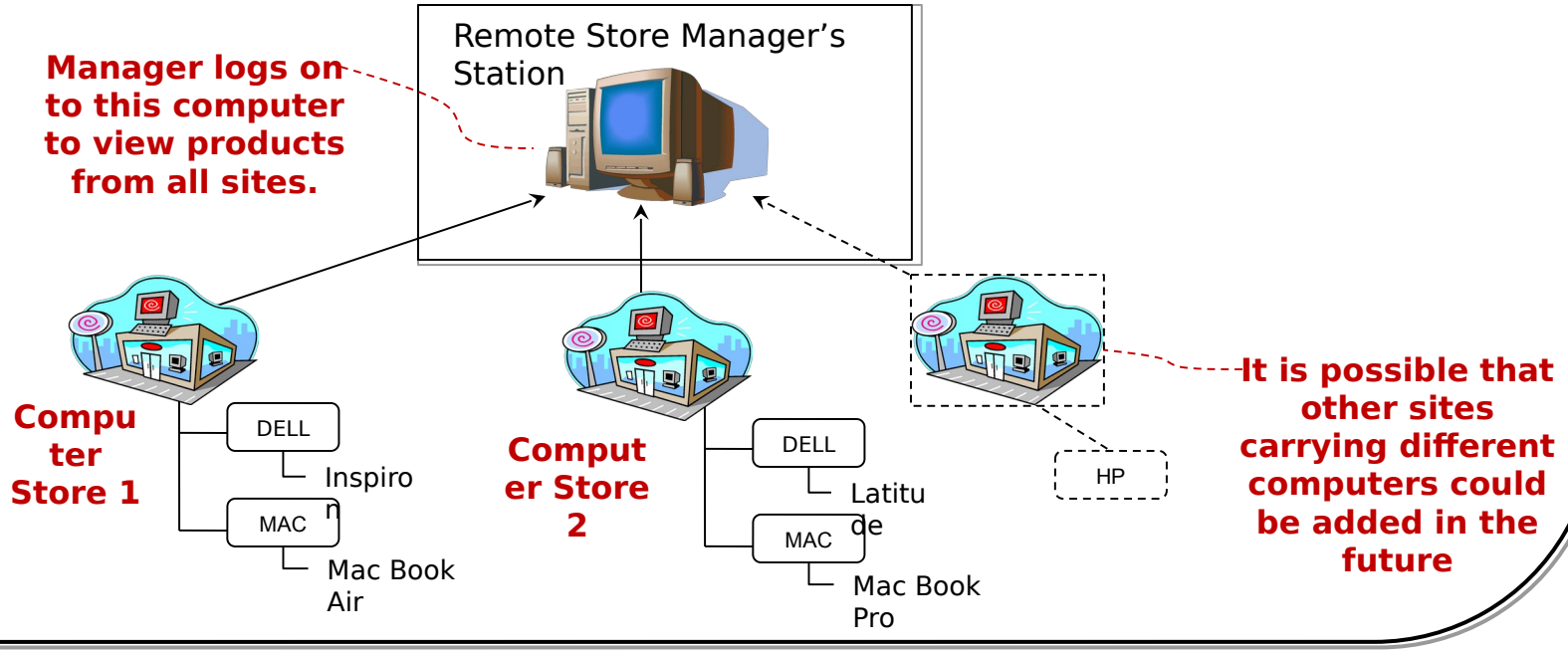
Finally, we add the factory method to our code. Keep in mind that by adding the abstract factory method, the ComputerStore now is abstract as well. This enforces object creation in derived classes.

```
    // The factory method.
    protected abstract Computer createComputer(String type);
```

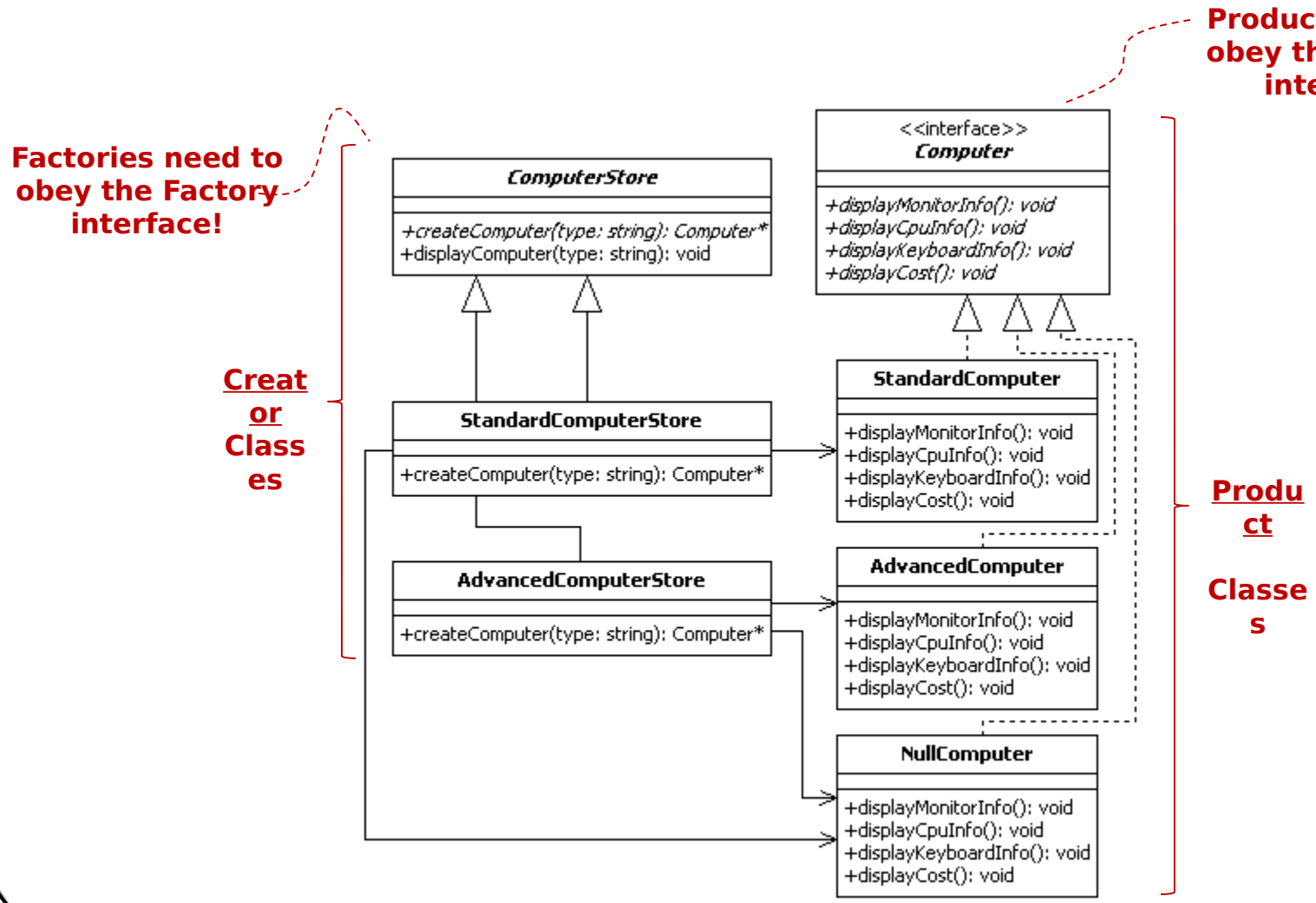
Notice that this is an abstract method!

FACTORY METHOD EXAMPLE

- Consider an information system for a chain of Computer Stores. The chain is currently composed of two sites and the manager wants to have a centralized software that manages all two sites. Mainly, the manager wants this centralized system to keep track of computer sales and display computer information for all sites. Not all sites carry the same computers and new sites could be added in the future, therefore support for computers should be site-specific!



FACTORY METHOD DESIGN PATTERN EXAMPLE



FACTORY METHOD EXAMPLE

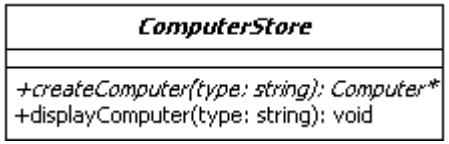
- Sample code for the displayComputer (…)method:

Factory method!

```
// Method to display a computer's information.
void ComputerStore::displayComputer(string type) {

  // Delegate the responsibility of creating a computer object to
  // derived classes using the factory method.
  Computer * computer = createComputer(type);

  // Display the computer information, including its cost. This
  // information varies according to the factory object used to create
  // the computer.
  computer->displayMonitorInfo();
  computer->displayCpuInfo();
  computer->displayKeyboardInfo();
  computer->displayCost();

  // Do more stuff with the computer object here.
  // Clean up the pComputer and pFactory objects when done.
}
```

This is what we expect to stay the same.

Optional parameter



FACTORY METHOD EXAMPLE

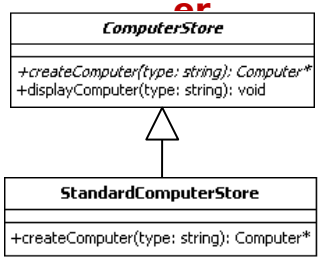
Factory method implementation for standard computer store

Optional parameter

```
// The factory method for creating computer products.
Computer* StandardComputerStore::createComputer(string type)
{
    // Pointer to a computer object.
  Computer* computer = 0;

  // Determine which computer needs to be created.
  if( type.compare("standard") == 0 ) {
    // Create the StandardComputer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new StandardComputer;
  }
  else {
    // Computer type not supported at this store. Create a null computer object.
    computer = new NullComputer(type);
  }

  // Return the newly created computer object. Clients are responsible
  // for cleaning up the computer object.
  return computer;
}
```

Rarely a computer store will carry only one computer!

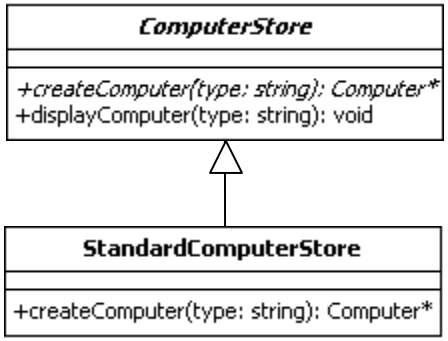
A generic standard computer



We use the optional parameter to support computers that may be carried in the future by the standard computer store

Let's see how to support new standard computers…

FACTORY METHOD EXAMPLE

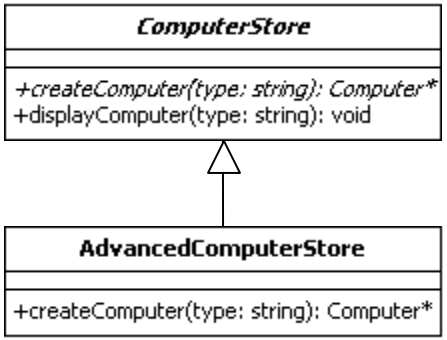


The standard computer store now carries Dell Inspiron and Mac Book Air computers

Important:
Notice that these changes are independent from what happens in the AdvancedComputer Store!

```
// The factory method for creating computer products.
Computer* StandardComputerStore::createComputer(string type)
{
    // Pointer to a computer object.
  Computer* computer = 0;

  // Determine which computer needs to be created.
  if( type.compare("standard") == 0 ) {
    // Create the StandardComputer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new StandardComputer;
  }
  else if( type.compare("DELL") == 0 ) {
    // Create a standard DELL Computer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new DellComputer("DellInspiron");
  }
  else if( type.compare("MAC") == 0 ) {
    // Create a standard MAC Computer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new MacComputer("MacBookAir");
  }
  else {
    // Computer type not supported at this store. Create a null computer object.
    computer = new NullComputer(type);
  }

  // Return the newly created computer object. Clients are responsible
  // for cleaning up the computer object.
  return computer;
}
```

FACTORY METHOD EXAMPLE



Currently, the advanced computer store only carries Mac Book Pro computers

Optional parameter

```
// The factory method for creating computer products.
Computer* AdvancedComputerStore::createComputer(string type)
{
    // Pointer to a computer object.
  Computer* computer = 0;

  if( type.compare("MAC") == 0 ) {
    // Create the advanced MAC computer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, AdvancedComputer
    // uses AdvancedComputerPartsFactory to create the advanced computer.
    computer = new MacComputer("MacBookPro");
  }
  else {
    // Computer type not supported at this store. Create a null computer object.
    computer = new NullComputer(type);
  }

  // Return the newly created computer object. Clients are responsible
  // for cleaning up the computer object.
  return computer;
}
```

Hopefully, by now, you are able to see how the pattern works to localize changes and support the addition of future computers with minimal impact!

FACTORY METHOD EXAMPLE

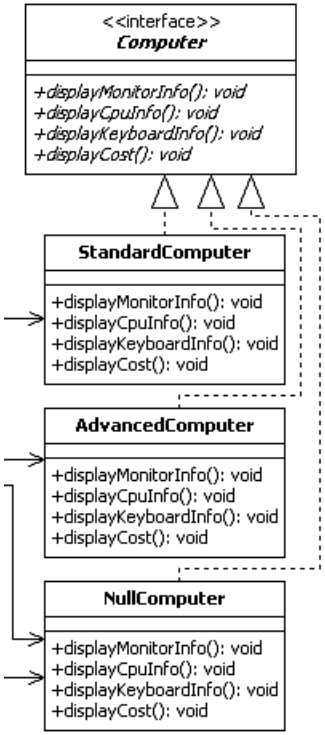
```
int main(int argc, char* argv[])
{
    // User input.
    int option = 0;

    // A computer store.
    ComputerStore* pStore = 0;

    // Type of computer.
    string type;

    // Display welcome message.
    cout<<"Welcome to the Computer Store Manager Software!\n\n";

    while(true)
    {
        cout<<"Store locations:\n"
            <<"1) New York store\n"
            <<"2) Florida store\n\n"
            <<"Enter store (0 to exit):";

        cin>>option;

        if( option == 0 || option < 1 || option > 2 )
        {
            cout<<"\nGood bye!\n";
            break;
        }
        else if( option == 1 )
            pStore = new AdvancedComputerStore;
        else if( option == 2 )
            pStore = new StandardComputerStore;

        cout<<"Enter computer type to search:";
        cin>>type;

        pStore->displayComputer(type);
        delete pStore;
    }

    return 0;
}
```

```
Welcome to the Computer Store Manager Software!

Store locations:
1) New York store
2) Florida store

Enter store (0 to exit):1
Enter computer type to search:DELL


This store does not carry the DELL computer. No monitor information available!
This store does not carry the DELL computer. No cpu information available!
This store does not carry the DELL computer. No keyboard information available!
This store does not carry the DELL computer. No cost information available!

Store locations:
1) New York store
2) Florida store

Enter store (0 to exit):2
Enter computer type to search:DELL


Displaying monitor information for DellInspiron computer...
Displaying cpu information for DellInspiron computer...
Displaying keyboard information for DellInspiron computer...
Displaying cost information for DellInspiron computer...

Store locations:
1) New York store
2) Florida store

Enter store (0 to exit):0

Good bye!
Press any key to continue . . .
```

Sample output

Just enough code to demonstrate the Factory Method in action!

FACTORY METHOD EXAMPLE




```
Welcome to the Computer Store Manager Software!

Store locations:
1) New York store
2) Florida store

Enter store (0 to exit):1
Enter computer type to search:HP


This store does not carry the HP computer. No monitor information available!
This store does not carry the HP computer. No cpu information available!
This store does not carry the HP computer. No keyboard information available!
This store does not carry the HP computer. No cost information available!
Store locations:
1) New York store
2) Florida store

Enter store (0 to exit):
```

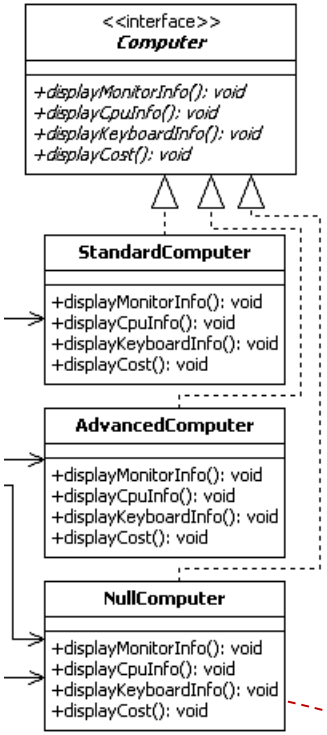
```
// The factory method for creating computer products.
Computer* AdvancedComputerStore::createComputer(string type)
{
    // Pointer to a computer object.
  Computer* computer = 0;

  if( type.compare("MAC") == 0 ) {
    // Create the advanced MAC computer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, AdvancedComputer
    // uses AdvancedComputerPartsFactory to create the advanced computer.
    computer = new MacComputer("MacBookPro");
  }
  else {
    // Computer type not supported at this store. Create a null computer object.
    computer = new NullComputer(type);
  }
```

The Null Object Pattern!

```
void NullComputer::displayCpuInfo()
{
    cout<<"This store does not carry the " + _type + " computer. No cpu information available!\n";
}

void NullComputer::displayKeyboardInfo()
{
    cout<<"This store does not carry the " + _type + " computer. No keyboard information available!\n";
}
```

Did you notice this NullComputer class?

THE NULL OBJECT PATTERN EXAMPLE




```
// The factory method for creating computer products.
Computer* StandardComputerStore::createComputer(string type)
{
    // Pointer to a computer object.
  Computer* computer = 0;

  // Determine which computer needs to be created.
  if( type.compare("standard") == 0 ) {
    // Create the StandardComputer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new StandardComputer;
  }
  else if( type.compare("DELL") == 0 ) {
    // Create a standard DELL Computer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new DellComputer("DellInspiron");
  }
  else if( type.compare("MAC") == 0 ) {
    // Create a standard MAC Computer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new MacComputer("MacBookAir");
  }
  else {
    // Computer type not supported at this store. Create a null computer object.
    computer = new NullComputer(type);
  }

  // Return the newly created computer object. Clients are responsible
  // for cleaning up the computer object.
  return computer;
}
```

The Null Object Pattern!

With this pattern, we treat invalid inputs as objects, therefore allowing us to treat them as other valid expected objects!

FACTORY METHOD DESIGN PATTERN

- Steps for designing with the Factory Method:
  1. Identify and design the product interface (e.g., Computer)
  2. Identify and design the concrete products that realize the interface from step 1 (e.g., StandardComputer, AdvancedComputer, DellComputer, etc.)
  3. Design the factory interface (e.g., ComputerStore), which contains one abstract factory interface method for delegating product creation to derived classes.
  4. Design one or more concrete factories for each product identified in step 2.
  5. Associate each factory from step 4 with its respective product from step 2.

- Benefits of the Factory Method pattern
  - ✔ Separates code from product-specific classes; therefore, the same code can work with various existing or newly created product classes.
  - ✔ By separating the code, development becomes efficient, since different developers can work on the different parts of the project at the same time.
  - ✔ By separating the code, it becomes easier to reuse and maintain specific parts of the code.

WHAT'S NEXT…

- In this session, we continued the discussion on creational design patterns, including:
  - ✔ Factory Method

- In the next sessions, we will finalize the presentation on creational design patterns. Specifically, we will cover:
  - ✔ Builder
  - ✔ Prototype
  - ✔ Singleton

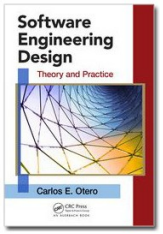
REFERENCES

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

CHAPTER 6: CREATIONAL DESIGN PATTERNS

SESSION III: BUILDER, PROTOTYPE, SINGLETON

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



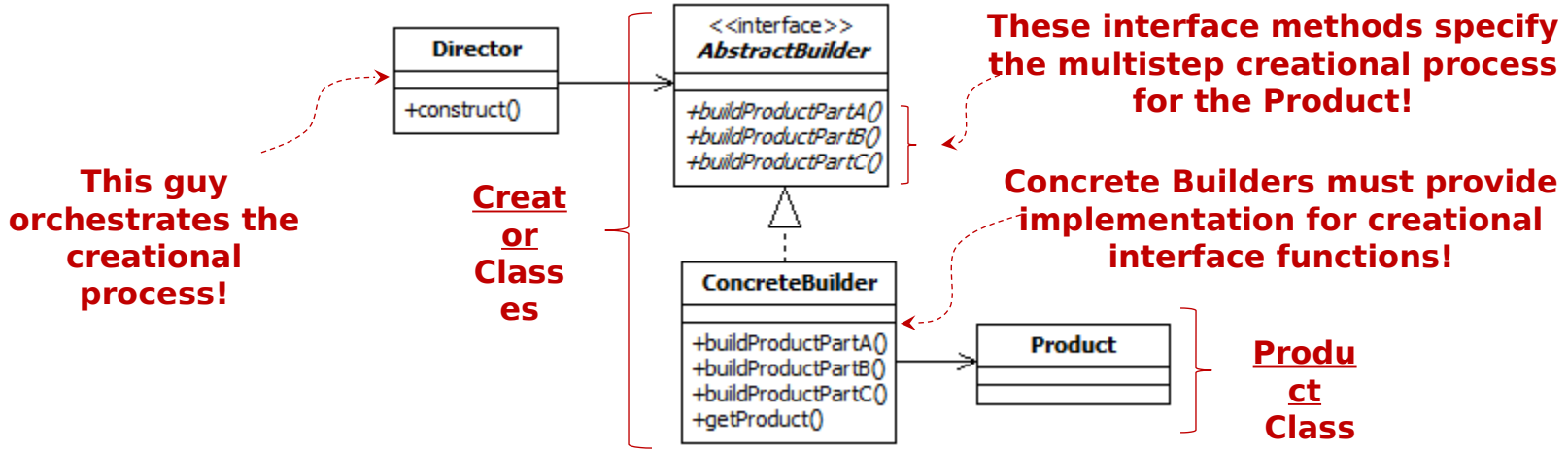
SESSION'S AGENDA

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

BUILDER DESIGN PATTERN

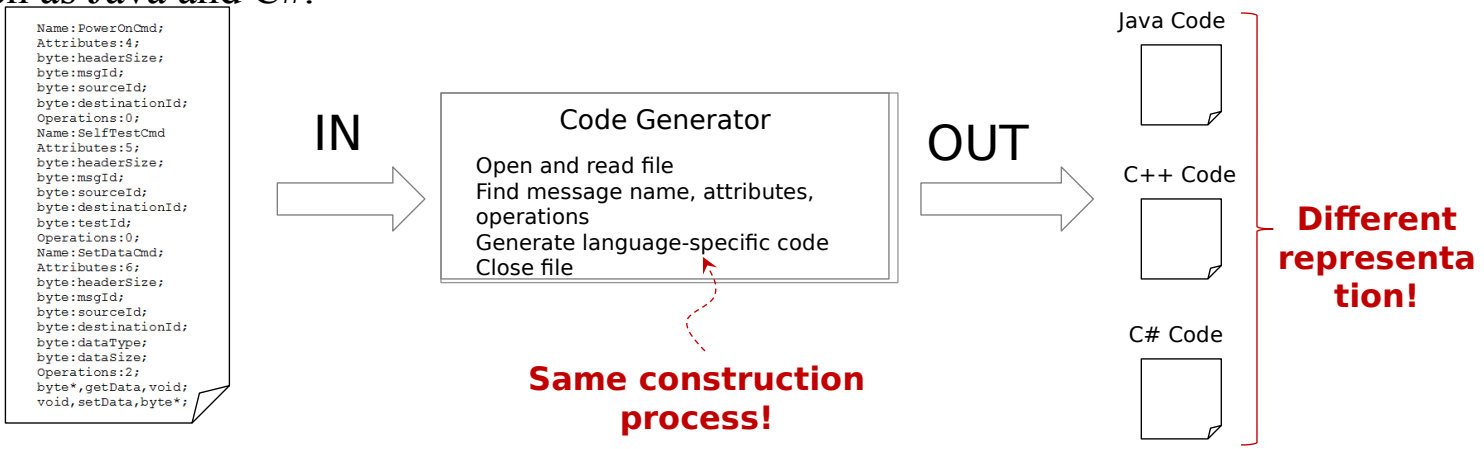
- The Builder Design pattern is an object creational pattern that encapsulates both the creational process and the representation of product objects.
  - ✓ Unlike the Abstract Factory, in which various product objects are created all at once, Builder allows clients to control the (multistep) creational 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.

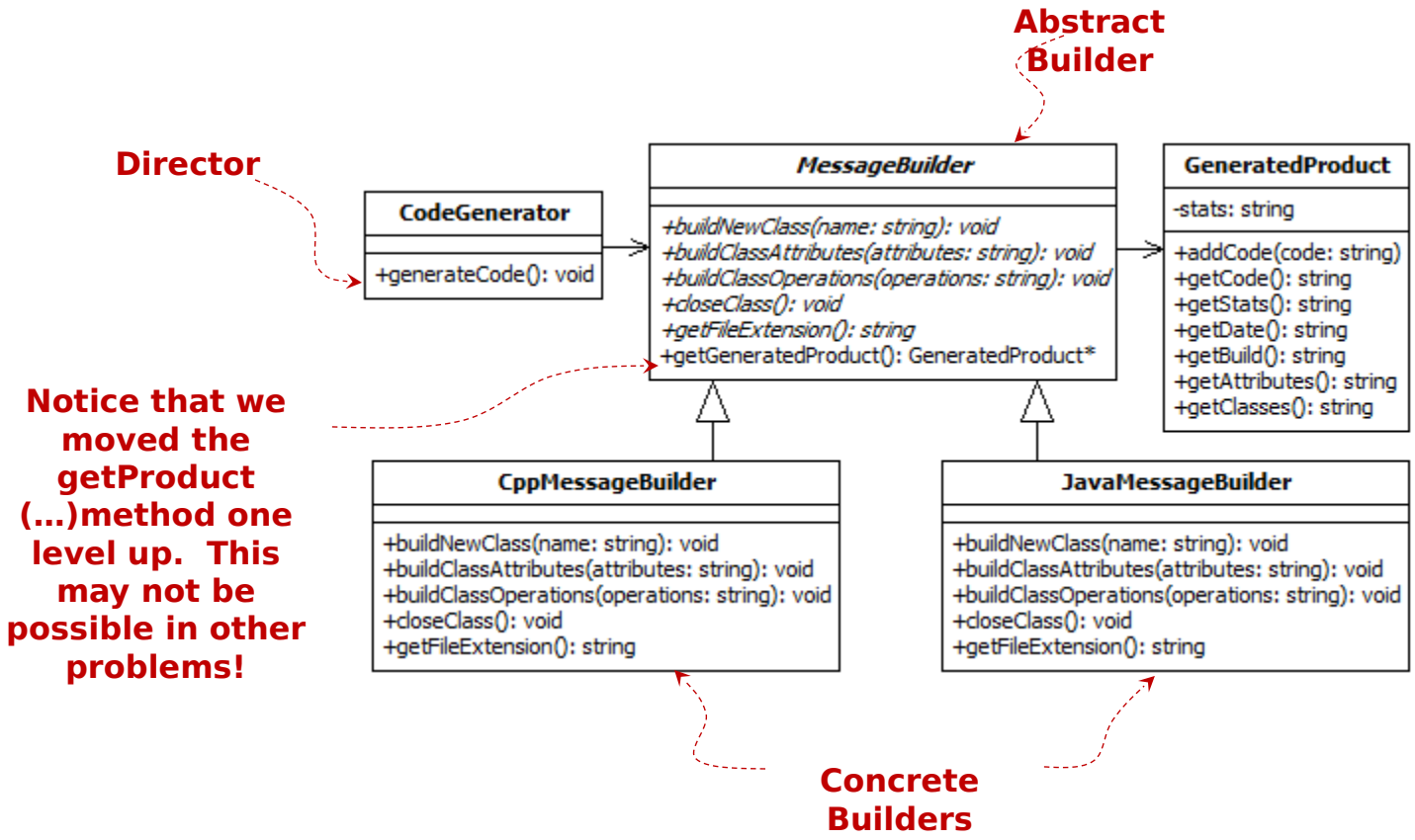


BUILDER DESIGN PATTERN EXAMPLE

- 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#.



BUILDER DESIGN PATTERN EXAMPLE



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;
};
```

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

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

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

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

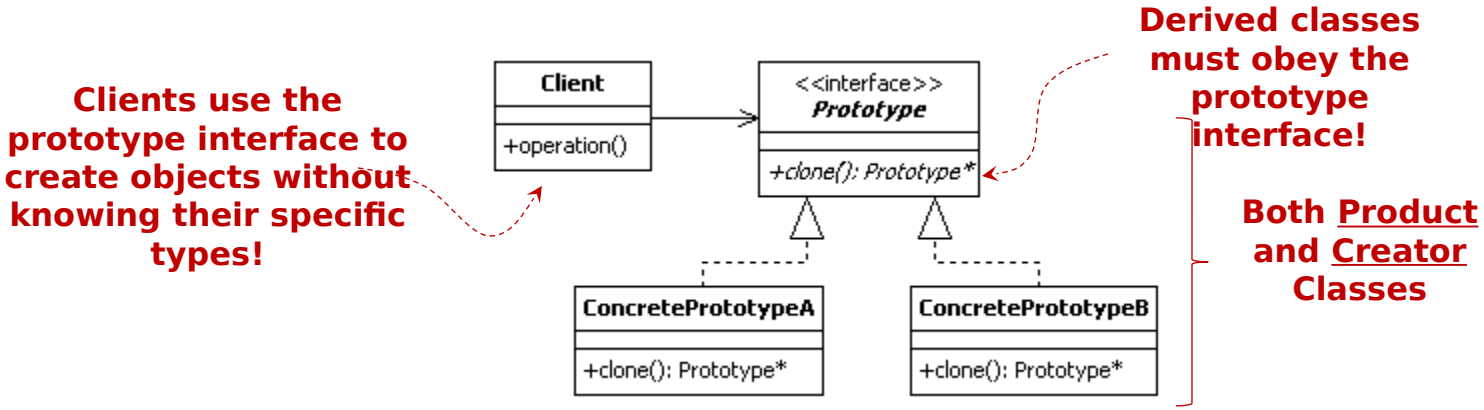
- Steps for designing using the Builder pattern:
  1. Identify and design the product class (e.g., GeneratedProduct)
  2. Identify the product's creational process and algorithm, and design a class for its execution (e.g., CodeGenerator).
  3. Using the knowledge acquired from steps 1 and 2, design the 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.



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;
}
```

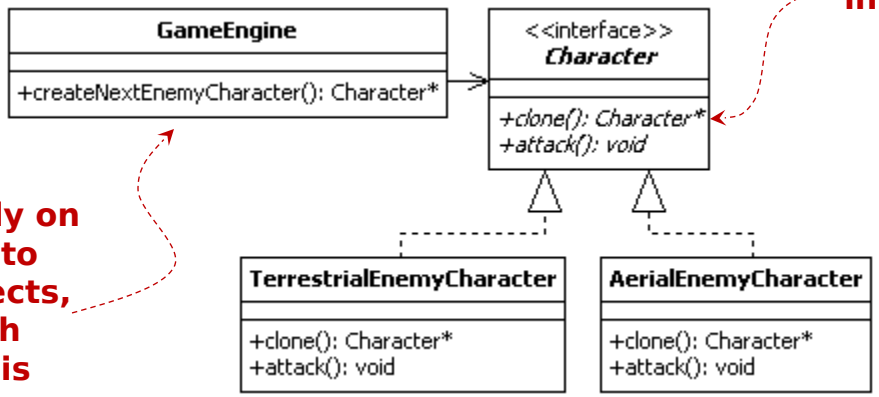
Yikes!

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

```
// 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.
};
```

```
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);
  }
};
```

```
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

```
// 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;
}
```

```
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

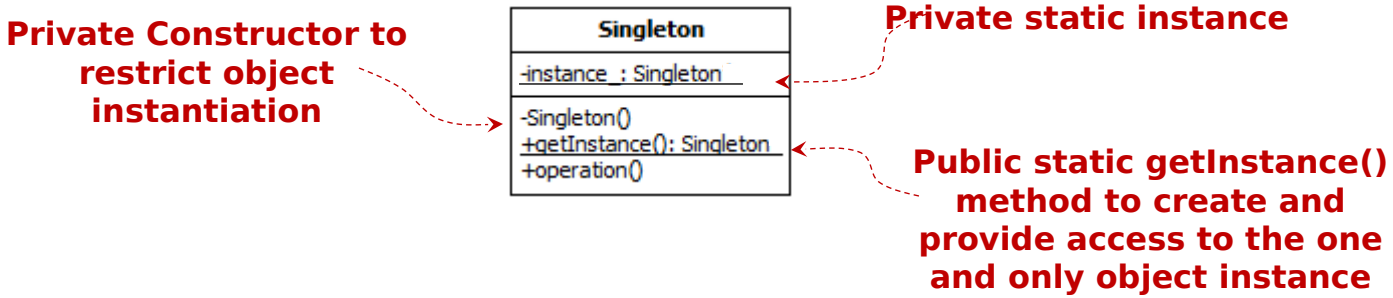
- Steps to design using the Prototype design pattern include:
  1. Identify and design the common interface that needs duplication. As part of the interface, the clone method (or equivalent) needs to be specified.
  2. Identify and design concrete products, which realize the interface created in Step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



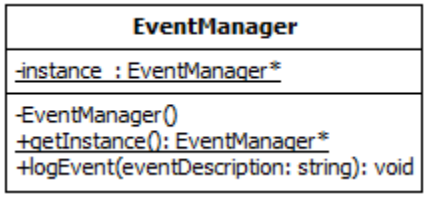
SINGLETON DESIGN PATTERN EXAMPLE

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;
}
```




```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.