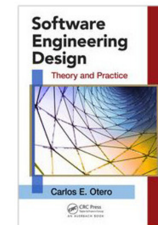
CHAPTER 6: CREATIONAL DESIGN PATTERNS

SESSION I: OVERVIEW OF DESIGN PATTERNS, ABSTRACT FACTORY

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



Slides copyright © 2012 by Carlos E. Otero

For non-profit educational use only

May be reproduced only for student use when used in conjunction with *Software Engineering Design: Theory and Practice.* Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information must appear if these slides are posted on a website for student use.

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, the 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.

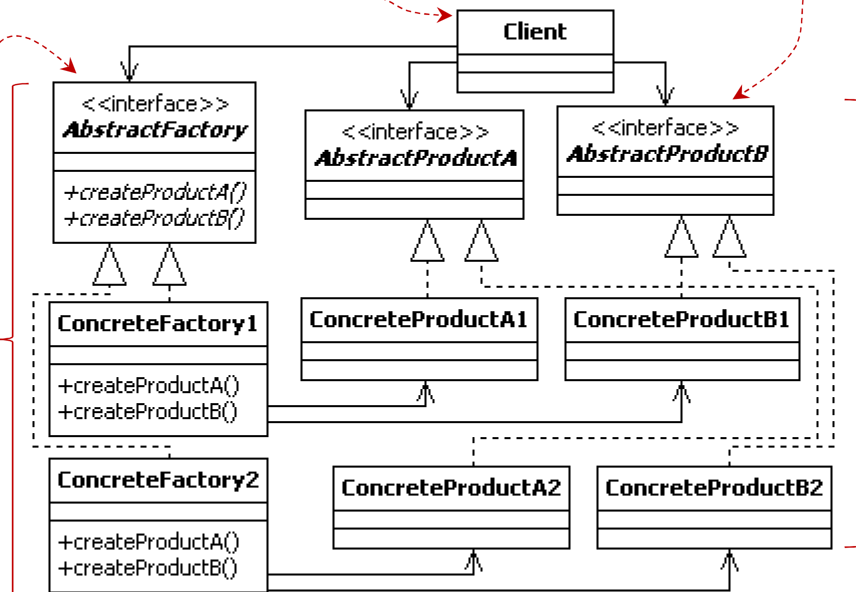
THE ABSTRACT FACTORY DESIGN PATTERN

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

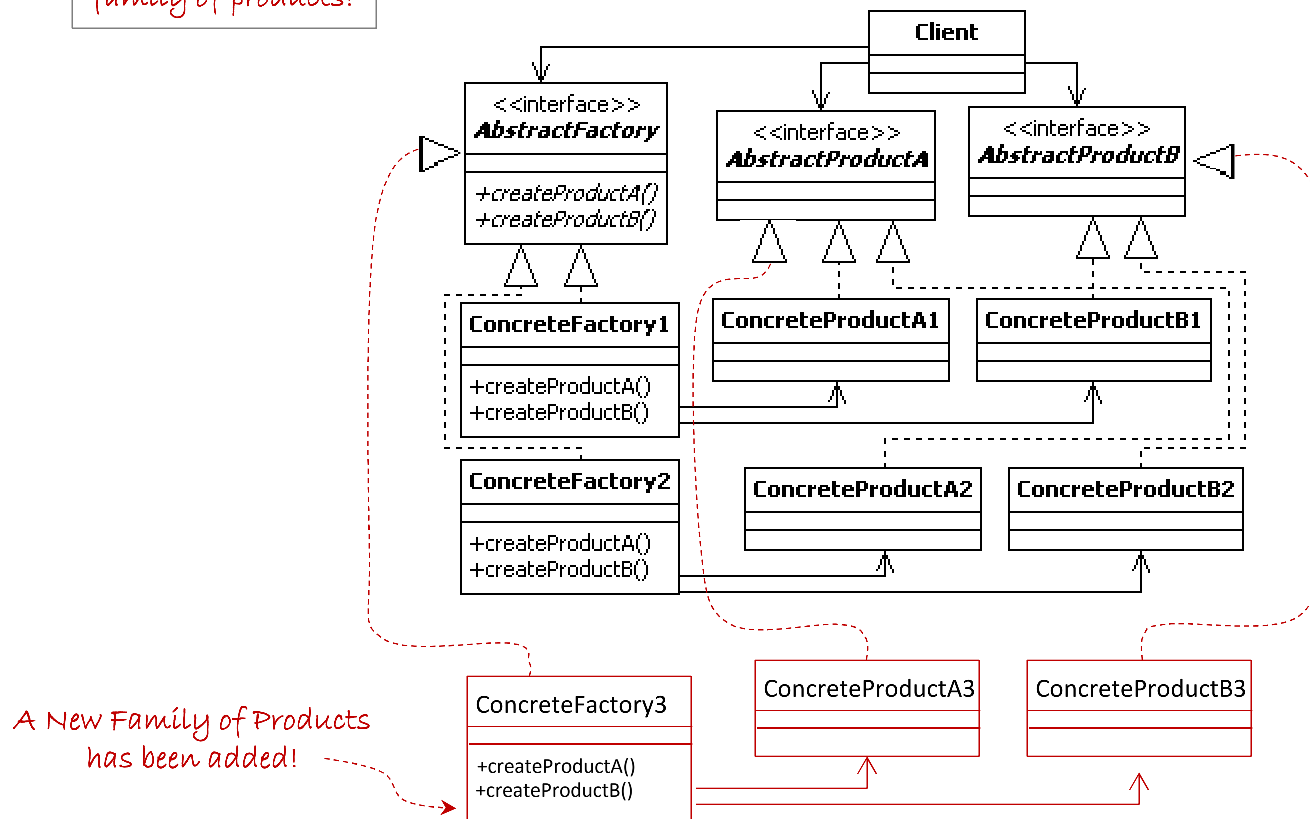


Product Classes

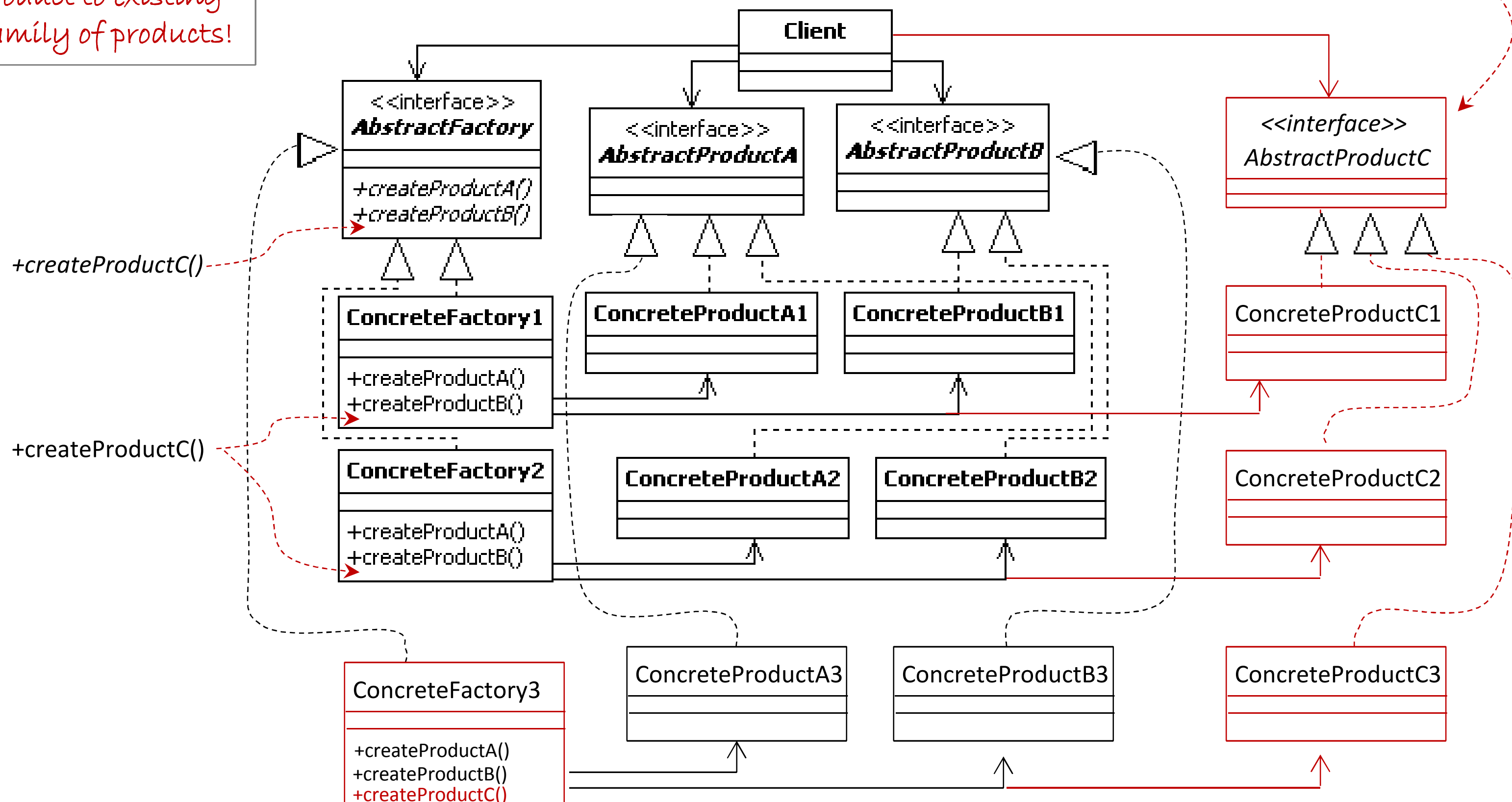
Important: Notice the Pattern!
Adding other products for existing families requires adding another AbstractProduct interface and concrete product classes!

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



THE ABSTRACT FACTORY DESIGN PATTERN



THE ABSTRACT FACTORY
— VERY SIMPLE AND FICTIONAL EXAMPLE —

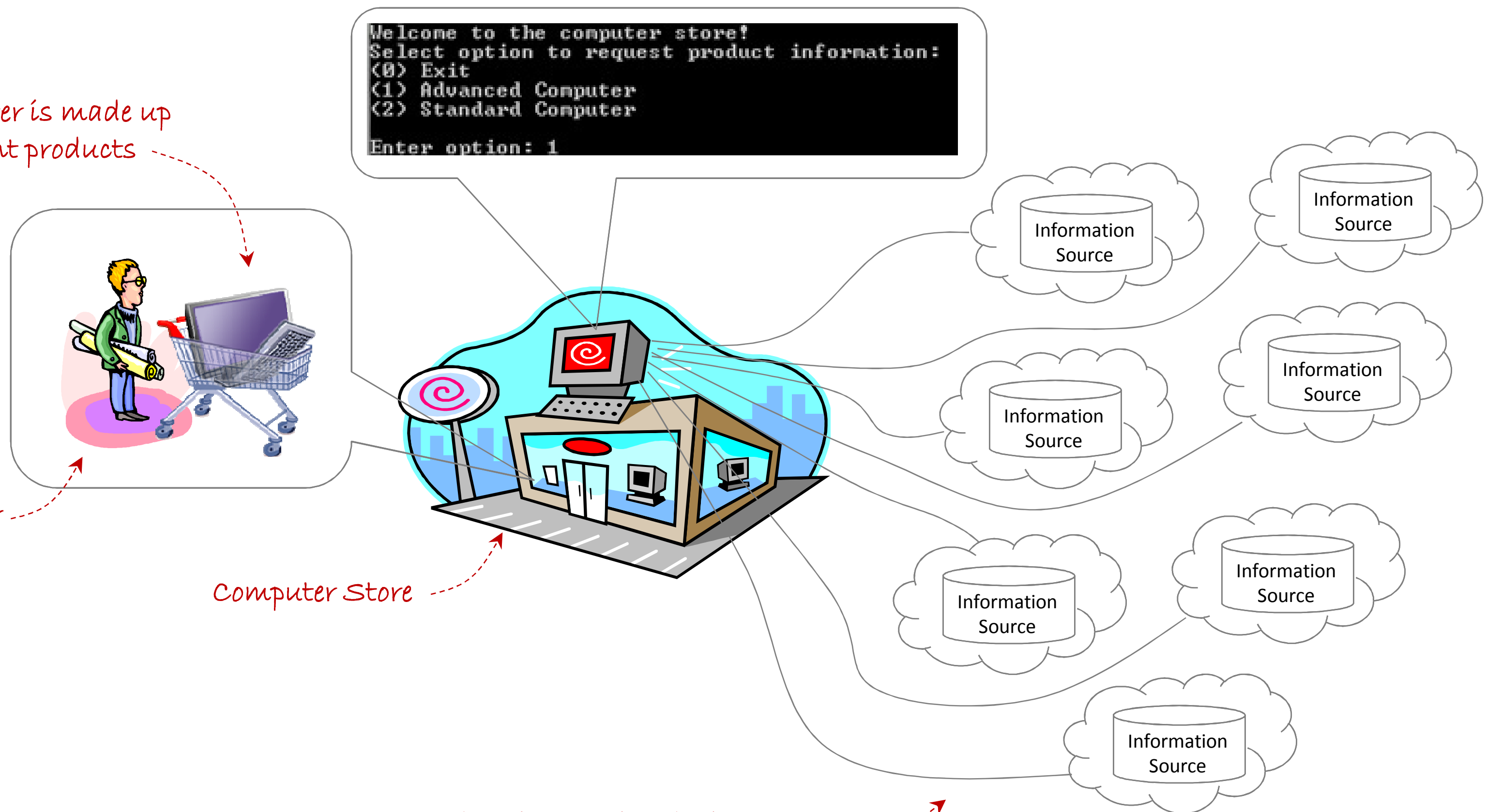
- Consider a software system for a computer store, where the store carries only two types of computers for sale:
  - ✓ Top of the line computer, we'll call these advanced computers
  - ✓ Inexpensive computers, we'll call these standard computers
  - ✓ Obviously, a computer store will need to carry more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

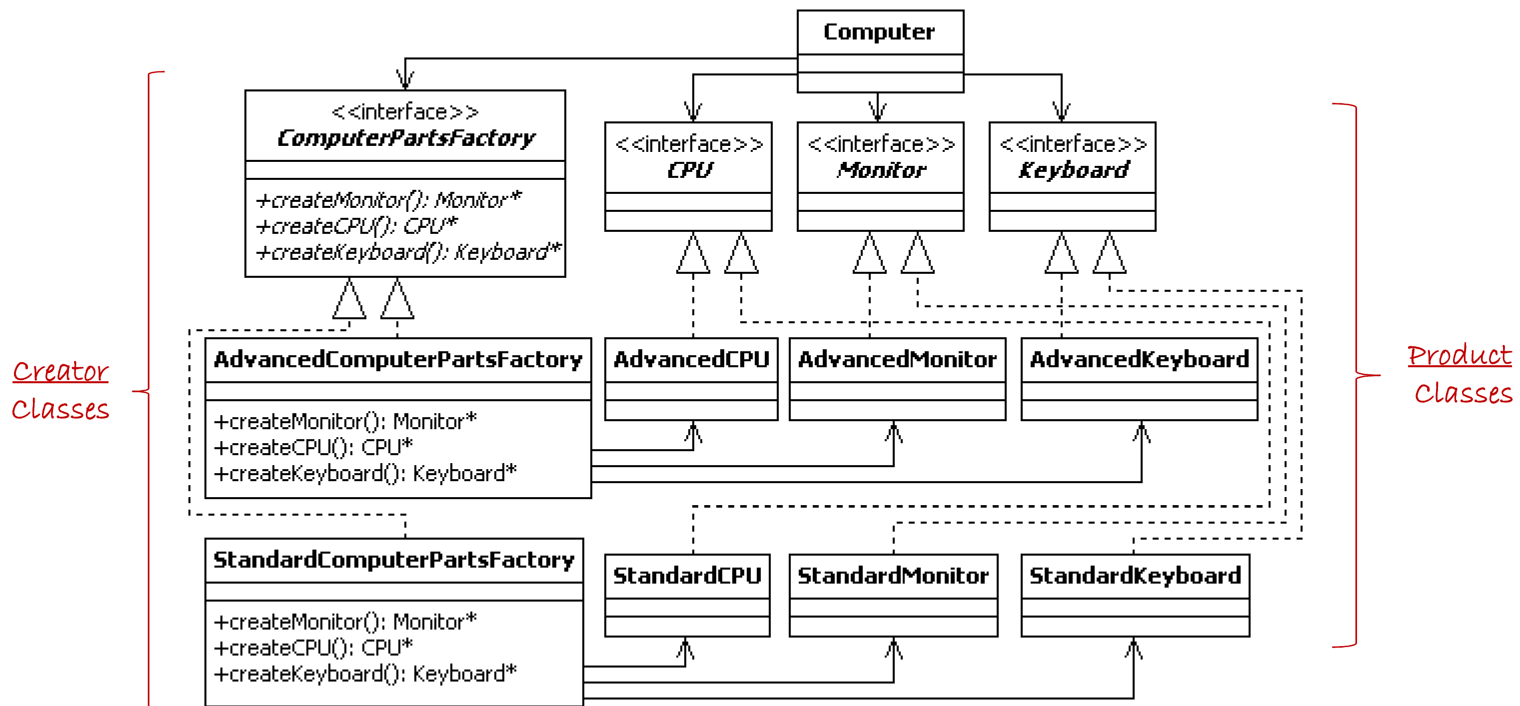
- Standard computers are made up of "standard computer products," e.g., single core CPU, wired keyboard, small screen monitor, low-grade graphics and sound, etc.
  - ✓ For simplicity, we'll only use CPU, keyboard, and Monitor for our example.

- 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
  - ✓ …

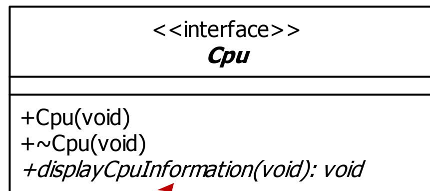
The Abstract Factory
— Very Simple and Fictional Example —



Abstract Factory for Computer Store



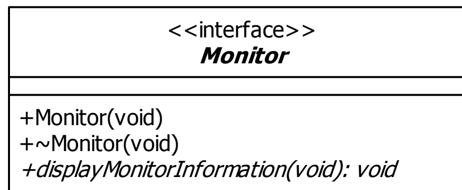
Let's break it down in the next slides...



Notice the italics to denote the abstract method

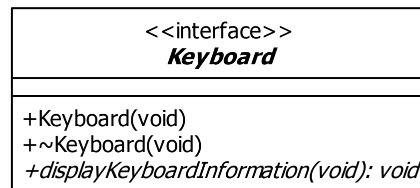
```
class Cpu
{
public:
    // Constructor.
    Cpu(void);

    // 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

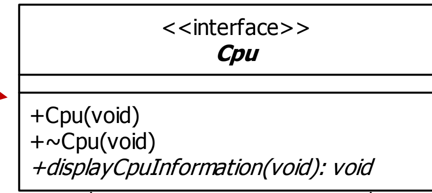


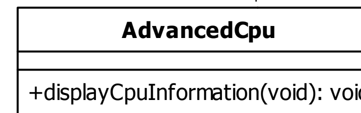

```
class Keyboard
{
public:
    // Constructor.
    Keyboard(void);

    // Destructor.
    virtual ~Keyboard(void);

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

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

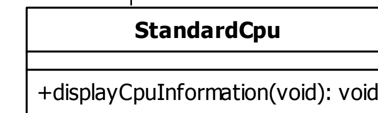
Derived classes must provide implementation for this method before they can be instantiated!

The Cpu Product Design

```
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.
};
```





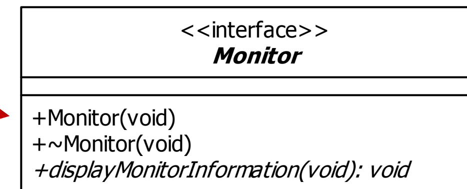


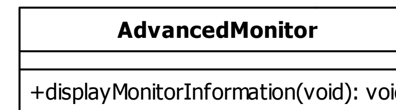

```
#include "cpu.h"

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.
};
```

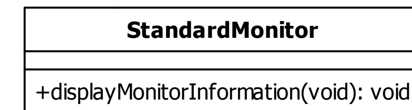
```
#include "cpu.h"

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

```cpp
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.
};
```







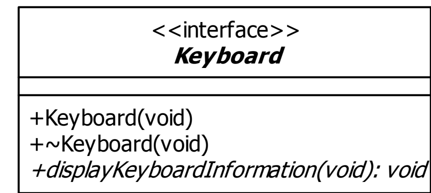

```cpp
#include "Monitor.h"

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.
};
```

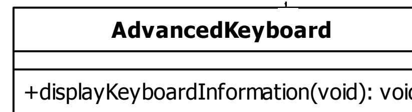
```cpp
#include "monitor.h"

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

    // Destructor.
    ~StandardMonitor(void);

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

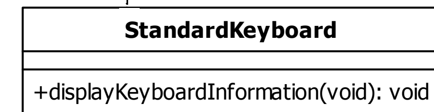
The Keyboard Product Design

```
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.
};
```

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








```
#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.
};
```

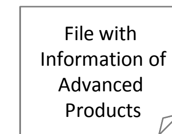
```
#include "keyboard.h"

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

```
#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.

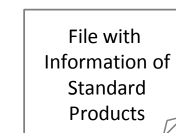
Information Source A




```
#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.

Information Source B



Other Product Implementation

```
// 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";
}
```

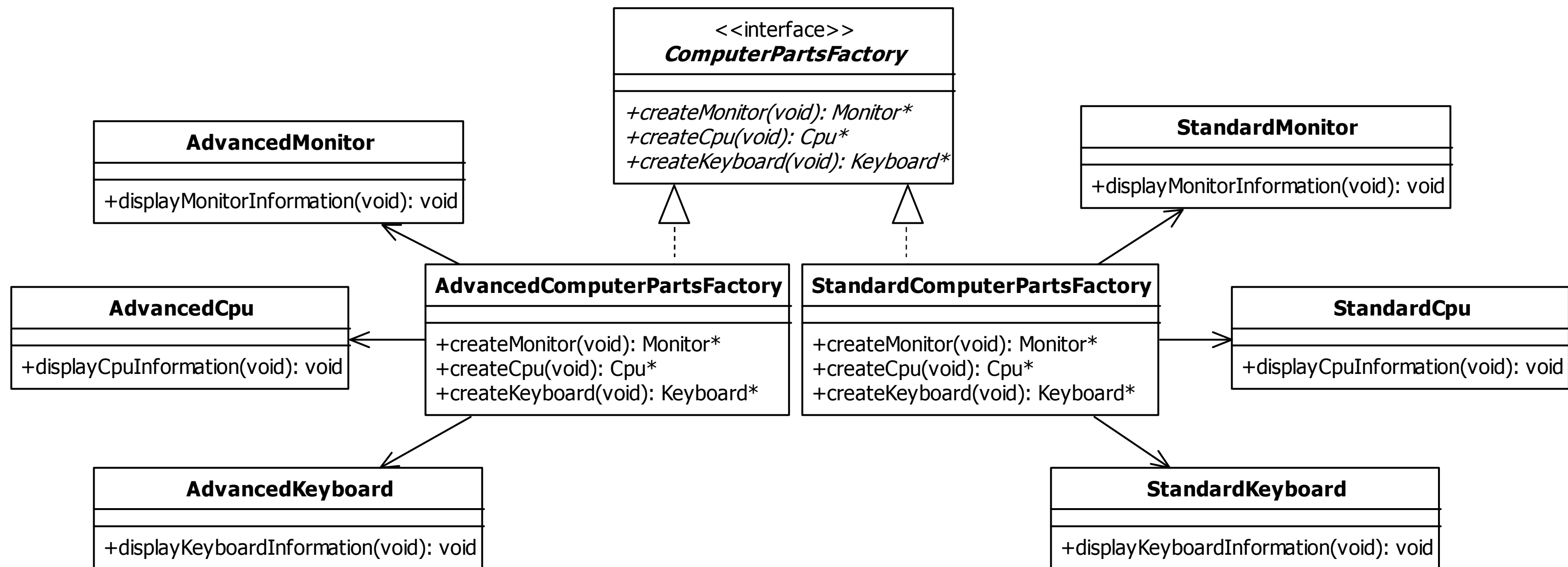
```
// Interface method for retrieving the cpu's information.
void StandardKeyboard::displayKeyboardInformation(void)
{
    // Since this is an example, we will assume that the advanced keyboard's information
    // will be retrieved from information source B, e.g., database B, file B, etc.
    // This 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!

```
// 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";
}
```

```
// 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

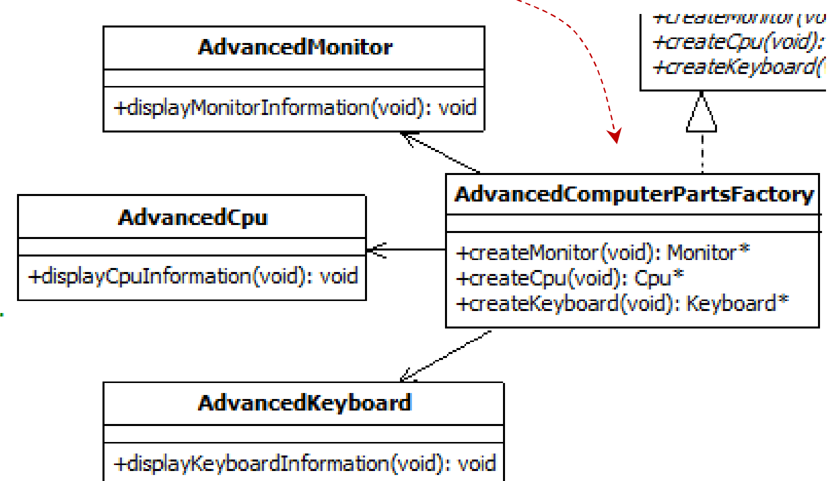


Important:
This design connects the products designed in the previous slides with the factories used to abstract their creation!

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;
}
```

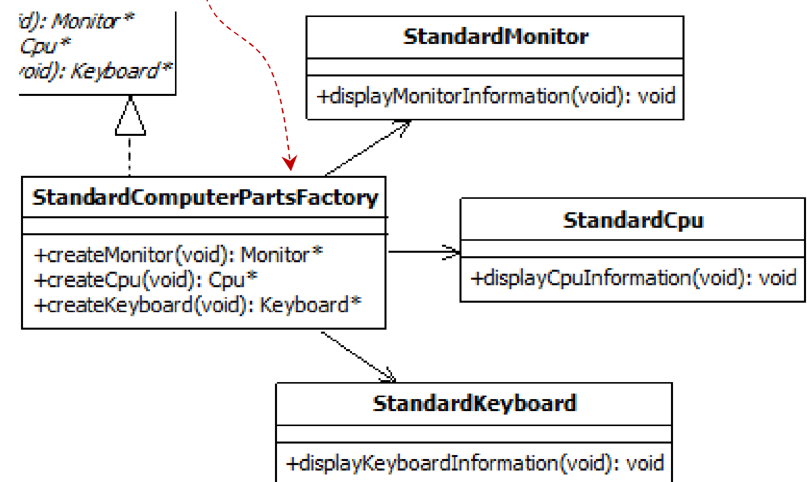
These are equivalent!



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;
}
```

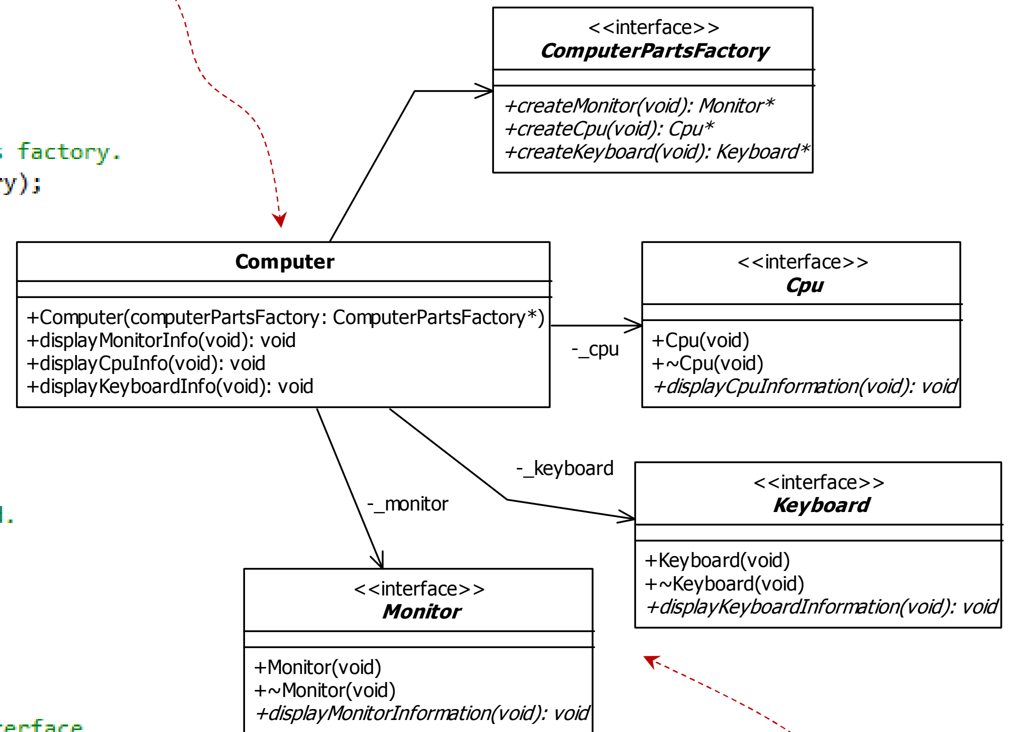
These are equivalent!



THE CLIENT COMPUTER DESIGN

These are equivalent!

```
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.
};
```



Code from model!

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

The Client Computer Design

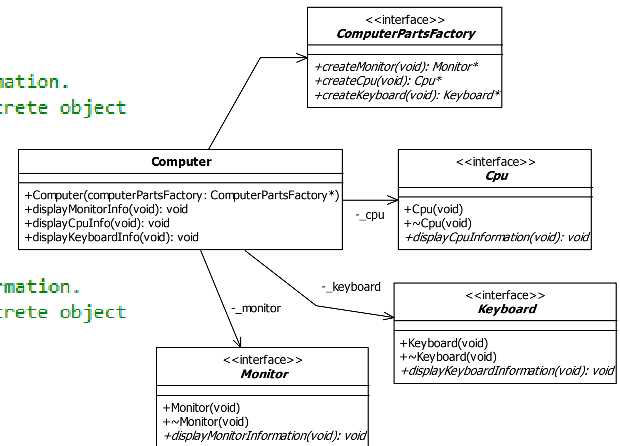
```
#include "Computer.h"
#include "ComputerPartsFactory.h"
#include "Monitor.h"
#include "Keyboard.h"
#include "Cpu.h"

Computer::Computer(ComputerPartsFactory* computerPartsFactory) : _monitor(0), _keyboard(0), _cpu(0)
{
    // 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();
}
```

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

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

```
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();
}
```

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



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:
```

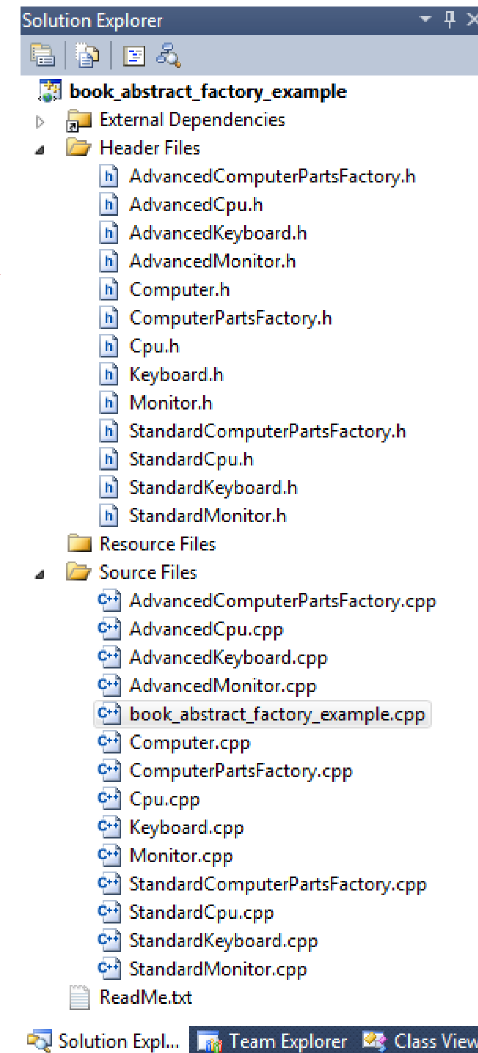
Notice how we configure the Computer object with a Factory object!

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.