CHAPTER 7:
STRUCTURAL AND BEHAVIORAL DESIGN PATTERNS
1. Structural Design Patterns
2. Behavioral Design Patterns
Session 1: Structural Design Patterns: Adapter, Composite, Facade
SESSION’S AGENDA

- Structural Patterns in Detailed Design
- Adapter
  - Character adapter example
- Composite
  - Message generator example
- Facade
  - Subsystem interface example
- What’s next…
**Structural Design Patterns**

- Structural design patterns are patterns that deal with designing larger structures from existing classes or objects at run time.
  - They play a key role in the design and evolution of systems by allowing integration of new designs with existing ones, via object composition (i.e., object structural) or inheritance (i.e., class structural).

- Popular structural design patterns include:
  - Adapter
  - Composite
  - Facade
The Adapter design pattern is a class/object structural design pattern used to adapt an exiting interface that is expected in a software system.

According to the GoF, the intent of the Adapter is to [1],

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
ADAPTER DESIGN PATTERN EXAMPLE

Problem

Consider the completed gaming system discussed in Chapter 5, which includes the design and development for all 10 levels of a gaming system, including the design and implementation of all gaming characters. At each level, the core of the gaming system (i.e., GameEngine) uses the Character interface to add enemy characters to the game, making them move, defend, and attack using the move(), defend(), and attack() interface methods respectively. Each character in the game implements the Character interface to provide specific behavior appropriate for the character and the level of the game. That is, depending on the character and the game level, the behavior for moving, defending, and attacking varies among characters. An online character developer has created a special character that is compatible with the game development’s API, but not with the particular Character interface, that is, the special character designed by the online developer includes the following interface methods: specialMove(), specialAttack(), and specialDefend(). The special character is made available freely to the gaming community; however, the special character code can only be downloaded and incorporated in other gaming systems as a binary compiled library, which can be incorporated into the existing game. Since all levels of the game are complete, it is impractical to change the code in all places to detect the new special character and make different calls for moving, attacking, and defending, therefore, the adapter design pattern is required to adapt the special character’s interface to the current character interface.
The idea is to implement the Character interface methods in terms of the BorrowedCharacter methods!

Let's see how this is done…
Notice the incompatible interfaces!

Under this current conditions, you cannot use a ConcreteBorrowedCharacter in the gaming system without changing the code!

// The Target class.
class Character {
    public:
        // Interface method for attack functionality.
        virtual void attack(void) = 0;
        // Interface method for defend functionality.
        virtual void defend(void) = 0;
        // Interface method for moving functionality.
        virtual void move(void) = 0;
};

// Interface for the borrowed character.
class BorrowedCharacter {
    public:
        // Interface methods for the borrowed character.
        virtual void borrowAttack(void) = 0;
        virtual void borrowDefend(void) = 0;
        virtual void borrowMove(void) = 0;
    }

    // Concrete borrowed character.
    class ConcreteBorrowedCharacter : public BorrowedCharacter {
        public:
            // Implementations for the BorrowedCharacter interface methods.
            void borrowAttack(void) { cout<<"borrowed attack...\n"; }
            void borrowDefend(void) { cout<<"borrowed defense...\n"; }
            void borrowMove(void) { cout<<"borrowed movement...\n"; }
    }
Character is the Target class.

The CharacterAdapter

```cpp
class CharacterAdapter : public Character {
public:
    // Constructor.
    CharacterAdapter(BorrowedCharacter* pCharacter);

    // Adapt the attack method.
    void attack(void);

    // Adapt the defend method.
    void defend(void);

    // Adapt the move method.
    void move(void);

private:
    // BorrowedCharacter that needs adapting to the Character interface.
    BorrowedCharacter* _borrowedCharacter;
};
```

We configure the Adapter with its Adaptee

The class that we are adapting
Adapter Design Pattern Example

Step 1. Configure the Adapter with its Adaptee

Step 2. Override each Character method to implement the behavior in terms of the Adaptee, in this case, the BorrowedCharacter.

```cpp
// Constructor.
CharacterAdapter::CharacterAdapter(BorrowedCharacter* pCharacter) {

    // For simplicity, assume a valid pointer.
    _borrowedCharacter = pCharacter;
}

// Adapt the attack method.
void CharacterAdapter::attack(void) {

    // Implement the attack functionality in terms of the BorrowedCharacter.
    borrowedCharacter->borrowedAttack();
}

// Adapt the defend method.
void CharacterAdapter::defend(void) {

    // Implement the defend functionality in terms of the BorrowedCharacter.
    borrowedCharacter->borrowedDefend();
}

// Adapt the move method.
void CharacterAdapter::move(void) {

    // Implement the move functionality in terms of the BorrowedCharacter.
    borrowedCharacter->borrowedMove();
}
```

With this design in place, you can now use CharacterAdapter to provide BorrowedCharacter services throughout the game!
Assuming the gaming engine has code similar to this

```cpp
class GameEngine {

public:
    // ...
    // Method to activate a character.
    void GameEngine::triggeredAction(Character* pCharacter) {
        // Activate the character and make it move it randomly for a short
        // time.
        pCharacter->move();

        // Once the character stops moving, if being attacked, defend!
        pCharacter->defend();

        // Once the characters stops defending, if others characters are
        // detected, attack!
        pCharacter->attack();
    }
    // ...
};
```

The GameEngine expects all characters to obey the Character interface

This is how you would adapt the borrowed character and use it in the GameEngine

Assuming the gaming engine has code similar to this
ADAPTER DESIGN PATTERN

The step-by-step approach for designing the object structural adapter design pattern is presented as follows:

1. Identify the source and destination interfaces that need adapting in the new system (e.g., Target and Adaptee or Character and SpecialCharacter).

2. Add a new class (e.g., Adapter or AdaptedCharacter) in the design that realizes the Target interface and implements it in terms of the Adaptee’s implementation. This requires a realization relationship between the Adapter and Target, and an association between the Adapter and the Adaptee.

3. In the new system, whenever objects that share the Target interface are expected, it can now be possible to use the Adapter objects created in step 2.

Benefits of the Adapter design pattern include:

- Allows classes with incompatible interfaces to work together, therefore it increases reusability and ease of code integration.
- Provides a standard way for integrating a plurality of different types to existing software.
The Composite design pattern is an object structural pattern that allows designers to compose (large) tree-like designs structures by strategically structuring objects that share a whole-part relationship.

According to the GoF, the intent of the Composite is to

- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and composites of objects uniformly.
COMPOSITE DESIGN PATTERN EXAMPLE

Problem

A wireless sensor system is remotely deployed to collect environmental information. The sensor system communicates via satellite to a central location, where a schedule of tasks (i.e., a mission plan) is created and sent over satellite communications. A mission plan is a composite message that contains one or more messages that command the sensor system to perform particular tasks. These messages contain information on how and when to perform particular tasks. Mission plan messages can be created with many different combinations of messages. Upon creating the mission plan message, it is sent to the wireless sensor system, which retrieves each message and message information from the mission plan, and executes them to collect environmental data, store it, and send it back to the central location, as directed by the mission plan message. The sensor system is extensible and contains many capabilities provided by numerous sensors (e.g., temperature, vibration, etc.), still shot camera, and video recording. To operate the sensor system, the operators at the central location are requesting a message generator capable of allowing them to easily create a mission plan message. The mission plan message may contain both primitive and composite messages. Numerous mission plan messages can be created to support different “missions” and it is expected that more sensing capabilities will be added in the future. Therefore, the design of the message generator must provide easy addition and removal of both messages and composite messages to a mission plan.

A graphical representation is presented in the next slide…
A Mission Plan Message dictates what happens in the system. It is a message composed of other messages.

We want to use a GUI to allow operators to create Mission Plan messages.

We want to use a GUI to allow operators to create Mission Plan messages.

Both primitive and composite messages ARE messages, so we want to treat them the same way to retrieve their IDs, print them, transform them to XML, etc., and transfer them through the system.

In this example, Message 3 is a composite message. We can include, e.g., Message 5 as part of the Message 3 composite message.

This is one message. More specifically, it is a composite message composed of other messages.

This is also a message, but, unlike the composite, this primitive message does not contain other messages!
**COMPOSITE DESIGN PATTERN EXAMPLE**

1. **Primitive message for video collection control** (Leaf)
   - Example of primitive messages

2. **Primitive message for executing a system test to ensure everything is working OK** (Leaf)

3. **Composite message**
   - This one is used to create composite messages, e.g., the mission plan message

**Message**
- id: int
- name: string

**VideoControlMessage**
- setFrameRate(framesPerSecond: int): void
- toML(): string
- clone(): Message*

**SelfTestMessage**
- toML(): string
- clone(): Message*

**CompositeMessage**
- messages: list<Message>
These apply to both primitive and composite messages

However, consider these four methods

What do you think should happen when executing a print, toXml, or Clone on composite messages?

Stop and think about this!

Remember the Prototype design pattern?
Let's examine two methods, the add and print for primitive messages...

For primitive messages, we don't have to add any other messages, so we'll have a default implementation for this:

```cpp
// Method to add messages to a composite message.
void Message::add(Message* message) {
    // The default implementation lets clients know that the operation is unsupported. This behavior is inherited by Leaf classes, but overridden by Composite classes.
    std::cout<<"Messages cannot be added to Leaf objects!\n";
}
```

We could also have a default implementation that throws an exception!

For primitive messages, printing a message entails printing its content, so we'll have this as default implementation:

```cpp
// Method to display messages to the console.
void Message::print() {
    // The default behavior for displaying a message's information. This behavior is inherited by Leaf classes, but overridden by Composite classes.
    std::cout<<"Message "<<_name.c_str()<<", Id: "<<_id<<endl;
}
```
Composite contain other messages, so we'll use a list to hold these other messages.

Nothing fancy here, simply realizing the Message interface.
COMPOSITE DESIGN PATTERN EXAMPLE

This code adds a message to the list of contained messages:

```cpp
void CompositeMessage::add(Message* message) {
    // Add this message.
    _messages.push_back(message);
}
```

When the print is called on the composite message, it must call print on all of its contained messages:

```cpp
void CompositeMessage::print() {
    // Display the Composite Message’s name and id.
    cout<<"\nComposite Message: "<<getName().c_str()
        "Id: "<<getId()<<endl;

    // Retrieve an iterator for the _messages collection.
    list<Message*>::iterator pIter = _messages.begin();

    // Iterate through the messages that make up this composite message and display their info.
    for( unsigned int i = 0; i < _messages.size(); i++ ) {
        // Display the message’s information and move the iterator to the next position.
        (*pIter++)->print();
    }
}
```

This same principle is employed on the toXml and clone methods:

```cpp
// Method to display messages to the console.
void Message::print(void) {
    // The default behavior for displaying a message’s information. This behavior is inherited by Leaf classes, but overridden by Composite classes.
    std::cout<<"Message "<<_name.c_str()<<", Id: "<<_id<<endl;
}
```
Composite Design Pattern Example

// Create a duplicate of the Composite Message using the prototype
development pattern.
Message* CompositeMessage::clone(void)
{
    return new CompositeMessage(*this);
}

// Copy constructor.
CompositeMessage::CompositeMessage(const CompositeMessage& other)
{
    setName(other.getName());
    setId(other.getId());

    // copy messages here.
    // Retrieve an iterator for the _messages collection.
    list<Message>::const_iterator pIter = other._messages.begin();

    // The size of the list to copy.
    int size = other.getMessageCount();
    for (unsigned int i = 0; i < size; i++) {
        // Make a copy of the _message list.
        _messages.push_back((pIter++)->clone());
    }
}

Override the copy constructor!

A clone on the composite means cloning of all contained messages as well!

Isn't it nice to have the Prototype design pattern? This way, we don't need to know the actual specific object that we're cloning!

Iterate through the list of contained messages and calls clone on each of them to properly create a copy of the composite message!
COMPOSITE DESIGN PATTERN EXAMPLE

// Create the initialization primitive messages.
PowerOnMessage powerOnMessage;
SelfTestMessage selfTestMessage;
TransmitStatusMessage transmitStatusMessage;

// The message to task the system to initialize properly.
CompositeMessage initializeTaskingMessage("Initialize System");

// Add copies of the power on, self test, and transmit status messages
// to the initialize tasking composite message.
initializeTaskingMessage.add( powerOnMessage.clone() );
initializeTaskingMessage.add( selfTestMessage.clone() );
initializeTaskingMessage.add( transmitStatusMessage.clone() );

// Collection Control Messages.
TemperatureSensorControlMessage temperatureSensorControlMessage;
VideoControlMessage videoControlMessage;

// The message to task the system to collect information.
CompositeMessage collectionMessage("Information Collection");

// Add the temp. sensor and video control messages to the collection
// tasking composite message.
collectionMessage.add( temperatureSensorControlMessage.clone() );
collectionMessage.add( videoControlMessage.clone() );

// Shutdown Messages.
ShutdownMessage shutdownMessage;

// The message to task the system to complete Mission 1.
CompositeMessage missionPlanMessage("Mission 1 - Temperature and Video Collection");

// Add the messages to the initialize, collection, and shutdown messages
// to the mission plan composite message.
missionPlanMessage.add( initializeTaskingMessage.clone() );
missionPlanMessage.add( collectionMessage.clone() );
missionPlanMessage.add( shutdownMessage.clone() );

// Before sending message, verify its content.
missionPlanMessage.print();

// If content is valid, send the message through the system. Before being
// sent out through the communication link, a call to missionPlanMessage.
// toXml() is made to convert all of the message's content to XML format.
COMPOSITE DESIGN PATTERN EXAMPLE

// The message to task the system to complete Mission 1.
CompositeMessage missionPlanMessage("Mission 1 - Temperature and Video Collection");

// Add the messages to the initialize, collection, and shutdown messages
// to the mission plan composite message.
missionPlanMessage.add( initializeTaskingMessage.clone() );
missionPlanMessage.add( collectionMessage.clone() );
missionPlanMessage.add( shutdownMessage.clone() );

// Before sending message, verify its content.
missionPlanMessage.print();

Sample output

```
Composite Message: Mission 1 - Temperature and Video Collection, Id: 20
Composite Message: Initialize System, Id: 20
Message Power On Message, Id: 0
Message Self Test Message, Id: 1
Message Transmit Status Message, Id: 2
Composite Message: Information Collection, Id: 20
Message Temperature Sensor Control Message, Id: 3
Message Video Control Message, Id: 4
Message Shutdown Message, Id: 5
Press any key to continue . . . .
```
The steps required to apply the composite design pattern include:

1. Identify, understand, and plan the tree-like structure required for the system.
2. With the knowledge from step 1, identify and design the Component base class (see diagram), which includes overridable methods common to both Leaf and Composite objects, as well as methods specific for Composite objects, which provide capability for adding and removing objects to the hierarchy.
3. For the methods specified in step 2 for adding/removing objects to the hierarchy, implement default behavior that if not overridden, will result in exception or error message indicating an unsupported operation.
4. Identify and design the Composite class, which overrides methods for adding and removing objects to the hierarchy. The Composite class requires an internal data structure to store Leaf nodes added to the hierarchy. In addition, the Composite class is required to override all other methods identified in step 2 to implement functionality in terms of the composite object and all of its contained Leaf objects.
5. Identify and design the Leaf classes, which overrides methods specified in step 2 to implement behavior in terms of the Leaf object. Leaf objects do not override the add and remove methods identified in step 2.
6. Identify and design the client that uses both composite and leaf objects.

Benefits of the Composite design pattern:

- Provides a design structure that supports both composite and primitive objects.
- Minimizes complexity on clients by shielding them from knowing the operational differences between primitive and composite objects. Clients that expect a primitive object will also work with a composite object, since operations are called uniformly on both primitive and composite objects.
- Easy to create and add new components objects to applications.
Facade Design Pattern

- The Facade design pattern is an object structural pattern that provides a simplified interface to complex subsystems.

- According to the GoF, the intent of the Facade is to [1]
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
FACADE DESIGN PATTERN EXAMPLE

Problem

Consider the sensor system described as part of the message generator in the previous section.

Upon field deployment, it is desirable to test the system's capabilities to ensure that the system works properly before engaging in autonomous operation. For this reason, a graphical user interface is required to monitor and control the system in the field during installation.
FACADE DESIGN PATTERN EXAMPLE
Facade Design Pattern Example

User Interface Subsystem

Client1 ──► Client2 ──► Client3 ──► Client4

SensorSystem
+enableSensor(sensorNum: int): void
+enableAllSensors(): void
+disableSensor(sensorNum: int): void
+disableAllSensors(): void
+transmitSensorData(sensorNum: int): void
+transmitAllSensorData(): void
+scheduleCollection(): void

SerialComm ──► SystemManager ──► FileSystem ──► DataAnalyzer ──► WirelessComm

The Facade
void SensorSystem::transmitSensorData(int sensorNumber) {
    // Create an object for serial communications parameters.
    SerialParams params;
    params.setCommPort( SerialParams::COM_1 );
    params.setBaudRate( SerialParams::BR_9600 );
    params.setParity( SerialParams::PARITY_NO_PARITY );
    params.setSizeBytes( SerialParams::BYTE_SIZE_8 );
    params.setStopBits( SerialParams::STOP_BIT_ONE );

    // Retrieve pointer to the serial communication object.
    SerialComm* pSerialComm = SerialComm::getInstance();

    // Open the serial communication with the specified parameters.
    if( pSerialComm->open(params) ) {
        // Ready to communicate with collection nodes, now get ready for
        // transmitting the data via the wireless link.
        TcpConnection* pConnection = TcpConnection::getInstance();

        if( pConnection->open(TcpConnection::PORT_NUMBER,
                              TcpConnection::IP_ADDRESS) ) {

            // Schedule a collection message.
            SystemManager::getInstance()->scheduleMessage(/*...*/);
        } else {
            // Log TCP error here.
            // Close serial connection.

        } // end if( pConnection->open(...)
    } else { // Log serial connection error here.
        // end if( serialComm->open(...)
    } // end transmitSensorData function.
**FAÇADE DESIGN PATTERN**

The step-by-step procedure for applying the Facade design pattern include:

1. Identify all components involved in carrying out a subsystem operation.
2. Create an ordered list of the operations required to execute the subsystem operation.
3. Design a Facade class that includes an interface method to carry out the subsystem operation. The Facade class has dependencies to all other subsystem components required to carry out the subsystem operation.
4. Implement the Facade interface method by calling operations on one or more subsystem components, in the order identified in step 2.
5. Allow one or more clients to access the objects of the Facade type so that they can gain access to the subsystem operation. This creates a many-to-one relationship between external subsystems and the Facade interface, instead of many-to-many relationships.

Benefits of the Facade design pattern include:

- Shields clients from knowing the internals of complex subsystem, therefore minimizing complexity in clients.
- Since the internals of the subsystem are prone to change, the facade provides a stable interface that hides changes to internal subsystems; therefore, client code is more stable.
- Promotes weak coupling on clients; with facade, clients depend only on one interface instead of multiple interfaces.
WHAT’S NEXT...

In this session, we presented structural design patterns, including:
- Adapter
- Composite
- Facade

The next session continues the discussion on patterns by presenting a different category of design patterns capable of encapsulating behavior. These behavioral patterns include:
- Iterator
- Observer
Session 2: Behavioral Design Patterns: Iterator, Observer
SESSION’S AGENDA

- Behavioral Patterns in Detailed Design

- Iterator
  - Example…

- Observer…
  - Example…

- What’s next…
Behavioral design patterns are patterns that deal with encapsulating behavior with objects, assigning responsibility, and managing object cooperation when achieving common tasks [1].

- They include many of the mainstream design patterns used in modern object-oriented frameworks and play a key role in the design of systems by making them independent of specific behavior, which is made replaceable with objects throughout these design patterns.

Popular behavioral design patterns include:
- Iterator
- Observer
- Others not covered:
  - Command
  - Chain of responsibility
  - Interpreter
  - Mediator
  - Memento
  - State
  - Strategy
  - Template Method
  - Visitor
The Iterator design pattern is an object behavioral pattern that provides a standardized way for accessing and traversing objects in a collection data structure.

According to the GoF, the intent of the Iterators is to [1],

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**Diagram:**

- **Aggregate**
  - +createIterator()

- **ConcreteAggregate**
  - +createIterator()

- **Iterator**
  - +first()
  - +next()
  - +isDone()
  - +currentItem()

- **ConcreteIterator**
  - +first()
  - +next()
  - +isDone()
  - +currentItem()

**Notes:**
- Iterator interface specifies a uniform way for traversing a collection data structure.
- Concrete Iterators must obey the Iterator interface!
ITERATOR DESIGN PATTERN EXAMPLE

Problem

A company’s software system manages inventory, financials, and all other information available from its two store branches. Each store carries specific computer products appropriate for their location’s demographics. During design, the software system is decomposed into several components, which includes two components for deferring and abstracting design information relevant to requirements for each computer store branch. The detailed design of each component is carried out separately by two different software engineers; this results in two different versions of data structures for managing and providing store product information. Now, anytime the software system is called upon to display information about store products, it is required to identify between the two store branches so that the correct implementation for accessing store information can be executed. This problem is encountered every time a new computer store branch is added to the system; therefore, a uniform and standardized method for accessing computer store products from different collection data structure is highly desirable.
Consider this case, where code in two different computer store classes use different collection data structures, each with its own unique interface!

```cpp
class ComputerProduct
{
public:
    ComputerProduct(void);
    virtual ~ComputerProduct(void);

    // Return the product's id.
    int getProductId(void);

    // Return the product's price.
    int getPrice(void) const;

    // Return the product's description.
    string getDescription(void) const;

    // Other methods here...
private:
    int _id;
    int _price;
    string _description;
};

class SimpleComputerStore
{
public:
    SimpleComputerStore(void);
    virtual ~SimpleComputerStore(void);

    // Return a pointer to the simple product list.
    SimpleProductList* getProducts(void) {
        // Return the simple product list.
        return &_products;
    }
private:
    // The list of simple computer products.
    SimpleProductList _products;
};
class AdvancedComputerStore
{
public:
    AdvancedComputerStore(void);
    virtual ~AdvancedComputerStore(void);

    // Computer store methods...

    // Return a pointer to the advanced product list.
    AdvancedProductList* getProducts(void) {
        // Return the advanced product list.
        return &_products;
    }
private:
    // The computer products... in ProductList form
    AdvancedProductList _products;
};
```

These could've been other collection data structures, arrays, etc.
**Iterator Design Pattern Example**

```cpp
// Simple store.
SimpleComputerStore simpleStore;
ComputerProduct* pProduct = 0;
SimpleProductList* simpleStoreProducts = simpleStore.getProducts();

// Display simple store products.
for( int i = 0; i < simpleStoreProducts->size(); i++ ) {
    // Retrieve the product at index i.
pProduct = simpleStoreProducts->getSimpleProduct(i);
    // Make sure pProduct is valid before using it!

    // Display product's information.
    cout << "Product id: " << pProduct->getProductId() << endl
        << "Product price: " << pProduct->getPrice() << endl
        << "Product Description: " << pProduct->getDescription().c_str() << endl;
}

// Advanced store.
AdvancedComputerStore advancedStore;
AdvancedProductList* advancedProducts = advancedStore.getProducts();

// Display advanced store products.
for( int i = 0; i < advancedProducts->length(); i++ ) {
    // Retrieve the product at location i.
pProduct = advancedProducts->getAdvancedProduct(i);
    // Make sure pProduct is valid before using it!

    // Display product's information.
    cout << "Product id: " << pProduct->getProductId() << endl
        << "Product price: " << pProduct->getPrice() << endl
        << "Product Description: " << pProduct->getDescription().c_str() << endl;
}
```

If a new store is added, we have to modify this code to add code for displaying products of the new store!
**Iterator Design Pattern Example**

Notice how in this example we separate the collection data structures from the computer stores!

Concrete Iterators must obey the StoreProductIterator interface to provide a uniform way for traversing a collection data structure containing Computer Products.

Notice the pattern! Applying the pattern to the computer store problem entails doing more of the same!

Let's see how this is implemented...
ITERATOR DESIGN PATTERN EXAMPLE

// The base for all store product iterators.
class StoreProductIterator {

public:
    // Constructor.
    StoreProductIterator() { /*Intentionally left blank.*/ }

    // Interface method for determining if more products are available.
    virtual bool hasNext(void) = 0;

    // Interface method for retrieving the next available product.
    virtual ComputerProduct* getNext(void) = 0;

    // Reset the iterator's position.
    virtual void reset(void) = 0;

};

<<interface>>
StoreProductIterator
+
  hasNext(); bool
  getNext(); ComputerProduct*
  reset(); void

Notice that we're using different interface names than the original pattern!

<<interface>>
Iterator
+
  first()
  next()
  isDone()
  currentItem()
ITERATOR DESIGN PATTERN EXAMPLE

// The base for all store product iterators.
class StoreProductIterator {

public:
    // Constructor.
    StoreProductIterator() : _position(0) { /*Intentionally left blank.*/}

    // Interface method for determining if more products are available.
    virtual bool hasNext() = 0;

    // Interface method for retrieving the next available product.
    virtual ComputerProduct* getNext() = 0;

    // Reset the iterator's position.
    virtual void reset() { _position = 0; }

protected:
    // Give access to derived classes for setting the iterator's position.
    void setPosition(int position) { _position = position; }

    // Allow derived classes to retrieve the iterator's position.
    int get姿态ion() { return _position; }

private:
    // The iterator’s current position.
    int _position;
};

For this problem, we've added some helper methods. It is OK to modify the pattern based on domain knowledge to fit the design to the application!
The Iterator used for the simple store

```cpp
class SimpleStoreProductIterator : public StoreProductIterator {
public:
    // Constructor.
    SimpleStoreProductIterator(SimpleProductList* products);

    // Determine if more products are available.
    bool hasNext(void);

    // If more products are available, get the next one.
    ComputerProduct* getNext(void);

private:
    // Pointer to the simple computer product list.
    SimpleProductList* _products;
};
```

Requires a simple product data collection!
**Iterator Design Pattern Example**

```cpp
// Constructor.
SimpleStoreProductIterator::SimpleStoreProductIterator(SimpleProductList* products) {
    // For simplicity, assume a valid pointer.
    _products = products;
}

// Determine if more products are available.
bool SimpleStoreProductIterator::hasNext() {
    // The return value.
    bool hasNextProductAvailable = false;
    if (getPosition() < _products->size()) {
        hasNextProductAvailable = true;
    }
    return hasNextProductAvailable;
}

// If more products are available, get the next one.
ComputerProduct* SimpleStoreProductIterator::getNext() {
    // Temporary pointer to computer product.
    ComputerProduct* pProduct = 0;
    // Get the iterator's current position.
    int nextItem = getPosition();
    // Determine if there are more products.
    if (hasNext()) {
        // Get the address of the next product and move the iterator's
        // position.
        pProduct = _products->getSimpleProduct(nextItem++);
        // Set the new position of the Iterator.
        setPosition(nextItem);
    }
    // Return the requested product.
    return pProduct;
}
```

Details / differences are hidden from clients!

Common interface allow clients to traverse through the _products objects in this collection, regardless of the difference that may exist in the _products interfaces!
**Iterator Design Pattern Example**

```cpp
// Constructor.
AdvancedStoreProductIterator::AdvancedStoreProductIterator(AdvancedProductList* products) {
    // For simplicity, assume valid pointer.
    _products = products;
}

// Determine if more products are available.
bool AdvancedStoreProductIterator::hasNext() {
    // The return value.
    bool nextProductAvailable = false;
    if (getPosition() < _products->length()) {
        nextProductAvailable = true;
    }
    return nextProductAvailable;
}

// If more products are available, get the next one.
ComputerProduct* AdvancedStoreProductIterator::getNext() {
    // Temporary pointer to computer product.
    ComputerProduct* pProduct = 0;

    // Get the iterator's current position.
    int nextItem = getPosition();

    // Determine if there are more products.
    if (hasNext()) {
        // Get the address of the next product and move the iterator's
        // position.
        pProduct = _products->getAdvancedProduct(nextItem++);
        // Set the new position of the iterator.
        setPosition(nextItem);
    }
    // Return the requested product.
    return pProduct;
}
```

Common interface allow clients to traverse through the _products objects in this collection, regardless of the difference that may exist in the _products interfaces!

Details / differences are hidden from clients!
**Iterator Design Pattern Example**

```
// The interface for all computer stores.
class ComputerStore {
    public:
        // The interface method to create an iterator.
        virtual StoreProductIterator* createIterator() = 0;
};

// Simple Computer Store
class SimpleComputerStore : public ComputerStore {
    public:
        // Override the createIterator interface method to create the
        // appropriate iterator for simple computer stores.
        StoreProductIterator* createIterator() {
            // Create and return a simple store product iterator.
            return new SimpleStoreProductIterator(&_products);
        }

    // All other methods for simple computer stores.
    private:
        // The simple product list.
        SimpleProductList _products;
};

// Advanced Computer Store
class AdvancedComputerStore : public ComputerStore {
    public:
        // Override the createIterator interface method to create the
        // appropriate iterator for advanced computer stores.
        StoreProductIterator* createIterator() {
            // Create and return an advanced store product iterator.
            return new AdvancedStoreProductIterator(&_products);
        }

    // All other methods for advanced computer stores.
    private:
        // The advanced product list.
        AdvancedProductList _products;
};
```
ITERATOR DESIGN PATTERN EXAMPLE

We can now display product information the same way for any store! With this design, if new stores are added, we can reuse the same displayProduct method to display products!

```cpp
// Display the products using the iterator.
void displayProducts(StoreProductIterator* pIterator) {

    // Temporary pointer to hold a computer product.
    ComputerProduct* pProduct = 0;

    // Determine if there are more products to browse.
    while( pIterator->hasNext() ){

        // Retrieve the next product.
        pProduct = pIterator->getNext();

        // Display the product's information.
        cout<<"\nProduct id: "<<pProduct->getProductId()<<endl
            "Product price: "<<pProduct->getPrice()<<endl
            "Product Description: "<<pProduct->getDescription().c_str();
    }
}
```

Iterators are used all over the Java framework, C#, and C++ standard template library!
### The Java Iterator and ListIterator Interface

#### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boolean hasNext()</code></td>
<td>Returns true if the iteration has more elements.</td>
</tr>
<tr>
<td><code>Object next()</code></td>
<td>Returns the next element in the iteration.</td>
</tr>
<tr>
<td><code>void remove()</code></td>
<td>Removes from the underlying collection the last element returned by the iterator (optional operation).</td>
</tr>
</tbody>
</table>

#### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void add(Object e)</code></td>
<td>Inserts the specified element into the list (optional operation).</td>
</tr>
<tr>
<td><code>boolean hasNext()</code></td>
<td>Returns true if this list iterator has more elements when traversing the list in the forward direction.</td>
</tr>
<tr>
<td><code>boolean hasPrevious()</code></td>
<td>Returns true if this list iterator has more elements when traversing the list in the reverse direction.</td>
</tr>
<tr>
<td><code>Object next()</code></td>
<td>Returns the next element in the list.</td>
</tr>
<tr>
<td><code>int nextIndex()</code></td>
<td>Returns the index of the element that would be returned by a subsequent call to next.</td>
</tr>
<tr>
<td><code>Object previous()</code></td>
<td>Returns the previous element in the list.</td>
</tr>
<tr>
<td><code>int previousIndex()</code></td>
<td>Returns the index of the element that would be returned by a subsequent call to previous.</td>
</tr>
<tr>
<td><code>void remove()</code></td>
<td>Removes from the list the last element that was returned by next or previous (optional operation).</td>
</tr>
<tr>
<td><code>void set(Object e)</code></td>
<td>Replaces the last element returned by next or previous with the specified element (optional operation).</td>
</tr>
</tbody>
</table>
JAVA ENUMERATION INTERFACE

java.util

Interface Enumeration

All Known Subinterfaces:
NamingEnumeration

All Known Implementing Classes:
 StringTokenizer

public interface Enumeration

An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the nextElement method return successive elements of the series.

For example, to print all elements of a vector v:

```java
for (Enumeration e = v.elements(); e.hasMoreElements(); ) {
    System.out.println(e.nextElement());
}
```

Methods are provided to enumerate through the elements of a vector, the keys of a hashtable, and the values in a hashtable. Enumerations are also used to specify the input streams to a SequenceInputStream.

NOTE: The functionality of this interface is duplicated by the Iterator interface. In addition, Iterator adds an optional remove operation, and has shorter method names. New implementations should consider using Iterator in preference to Enumeration.
The step-by-step approach for designing the Iterator design pattern is presented as follows:

1. Identify and design the Iterator interface.
2. For each class representing a collection data structure in the software system, design a concrete Iterator and associate it with it. Implement the concrete iterator’s methods in terms of the collection data structure.
3. Create the Aggregate interface, which includes the interface method to create Iterators.
4. For each class representing a collection data structure, implement the Aggregate interface to instantiate and return a concrete Iterator.

Benefits of the Iterator design pattern include:

- The Iterator provides a consistent way for clients to iterate through the objects in a collection.
- It abstracts the internals of the collection objects so that if they change, clients do not have to change.
- Allows client code to be extended easily; numerous iterators can be created to support different traversals from the same or different collection structure.
The Observer design pattern is an object behavioral pattern that standardizes the operations between objects that interoperate using a one-to-many relationship.

According to the GoF, the intent of the Observer is to [1]
✓ Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
**Observer Design Pattern Example**

*Problem*

A local university is designing a system for weather-alert notification that allows students, faculty, and staff to receive notifications of class cancellations (due to weather) via email, voice call, or SMS text messages. Other methods of notification may be added in the future. The system is based on the weather data decision engine that interfaces with several weather-related data sources, fuses the information, and automatically decides whether class cancellations are in effect. The university is interested in integrating the existing communication services (i.e., email, sms, and voice) with the decision engine so that these services can be triggered to initiate notification via their respective communication types. The design must be flexible so that other types of communication mechanisms can be added to the system in the future.
OBSERVER DESIGN PATTERN EXAMPLE

```
ServiceNotifier
+attach(pService: Service*): void
+detach(pService: Service*): void
#notify(): void

Service
+update(msg: string): void

DecisionEngine

EmailService
+update(msg: string): void

PhoneService
+update(msg: string): void

SmsService
+update(msg: string): void
```
**Observer Design Pattern Example**

```cpp
// Provide the registration mechanism for all observers.
void ServiceNotifier::attach(Service* pService) {
    // Add this observer to the list of registered observers. Assume a
    // valid pointer.
    _services.push_back(pService);
}

// The trigger mechanism to notify all observers of class cancellation.
void ServiceNotifier::notify(string message) {
    // Get an Iterator that points to the beginning of the observers_list.
    list<Service*>::iterator piter = _services.begin();

    // Iterate through the list of observers and notify them.
    for( int i = 0; i < _services.size(); i++ ) {
        // Pass the message along to all registered observers.
        (*piter++)->update(message);
    }
}

class EmailService : public Service {
public:
    // Once the Observable object changes, it will call this method.
    void update(string message) {
        // Open file containing all users registered for email notification.
        // Open connection to the Email server.
        // For all registered clients, notify them via email.
    }

    // ...
};
```
When a decision is made, the notify method is called to iterate through the list of registered observers to update them with the latest news!

Register observers with the subject, in this case, the DecisionEngine

When a decision is made, the notify method is called to iterate through the list of registered observers to update them with the latest news!
JAVA OBSERVER INTERFACE

java.util

Interface Observer

public interface Observer

A class can implement the observer interface when it wants to be informed of changes in observable objects.

Since:
    JDK1.0
See Also:
    Observable

Method Summary

<table>
<thead>
<tr>
<th>void</th>
<th>update(Observable o, Object arg)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This method is called whenever the observed object is changed.</td>
</tr>
</tbody>
</table>
Observers are almost always used within the MVC architecture!

```java
import java.lang.Object;
import java.util.Observable;

public class Observable extends Object {
    // ...
}
```

This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed.

An observable object can have one or more observers. An observer may be any object that implements interface Observer. After an observable instance changes, an application calling the `Observable.notifyObservers` method causes all of its observers to be notified of the change by a call to their `update` method.

The order in which notifications will be delivered is unspecified. The default implementation provided in the Observable class will notify Observers in the order in which they registered interest, but subclasses may change this order, use no guaranteed order, deliver notifications on separate threads, or may guarantee that their subclass follows this order, as they choose.

Note that this notification mechanism is has nothing to do with threads and is completely separate from the `wait` and `notify` mechanism of class `Object`.

When an observable object is newly created, its set of observers is empty. Two observers are considered the same if and only if the `equals` method returns true for them.
# Java Observable Methods

## Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void</code></td>
<td><code>addObserver(Observer o)</code></td>
<td>Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.</td>
</tr>
<tr>
<td><code>protected void</code></td>
<td><code>clearChanged()</code></td>
<td>Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the <code>hasChanged</code> method will now return <code>false</code>.</td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>countObservers()</code></td>
<td>Returns the number of observers of this <code>Observable</code> object.</td>
</tr>
<tr>
<td><code>void</code></td>
<td><code>deleteObserver(Observer o)</code></td>
<td>Deletes an observer from the set of observers of this object.</td>
</tr>
<tr>
<td><code>void</code></td>
<td><code>deleteObservers()</code></td>
<td>Clears the observer list so that this object no longer has any observers.</td>
</tr>
<tr>
<td><code>boolean</code></td>
<td><code>hasChanged()</code></td>
<td>Tests if this object has changed.</td>
</tr>
<tr>
<td><code>void</code></td>
<td><code>notifyObservers()</code></td>
<td>If this object has changed, as indicated by the <code>hasChanged</code> method, then notify all of its observers and then call the <code>clearChanged</code> method to indicate that this object has no longer changed.</td>
</tr>
<tr>
<td><code>void</code></td>
<td><code>notifyObservers(Object arg)</code></td>
<td>If this object has changed, as indicated by the <code>hasChanged</code> method, then notify all of its observers and then call the <code>clearChanged</code> method to indicate that this object has no longer changed.</td>
</tr>
<tr>
<td><code>protected void</code></td>
<td><code>setChanged()</code></td>
<td>Marks this <code>Observable</code> object as having been changed; the <code>hasChanged</code> method will now return <code>true</code>.</td>
</tr>
</tbody>
</table>
The steps required to apply the Observer design pattern include:

1. Design the Subject interface and implement code for attaching, detaching, and notifying observer objects. The code for keeping track of observers can be done using existing linked-lists data structures.
2. For classes that manage information of interests to observers, inherit from the subject class created in step 1.
3. Design the Observer interface, which includes the abstract update interface method.
4. For all observers in the system, implement the Observer interface, which requires implementing the update method.
5. At run-time, create each observer and attach them to the subject. When changes occur, the subject iterates through its list of registered objects, and calls their update method.

Benefits of the Observer design pattern:

- Flexibility for adding new services to the system.
- Since specific services are compartmentalized, maintain and modifying existing system services becomes easier.
What’s Next…

- In this session, we presented behavioral design patterns, including:
  - Iterator
  - Observer

- This concludes the presentation of design patterns in detailed design. In the next module, we will present a different form of design which occurs (mostly) at the function-level. We refer to this form of design as construction design.
REFERENCES