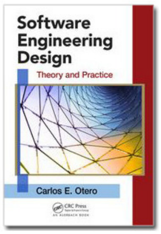
CHAPTER 7: STRUCTURAL AND BEHAVIORAL DESIGN PATTERNS

SESSION I: ADAPTER, COMPOSITE, FACADE

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



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

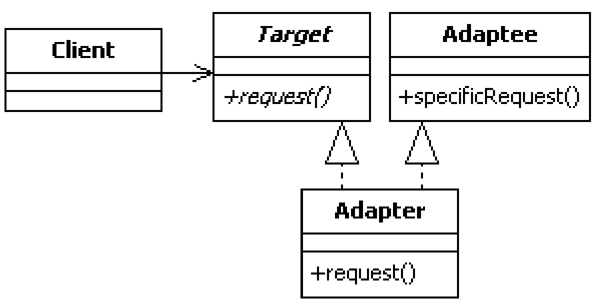
- Structural design patterns are patterns that deal with designing larger structures from existing classes or objects at run time.
  - ✓ They play a key role in the design and evolution of systems by allowing integration of new designs with existing ones, via object composition (i.e., object structural)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

- Popular structural design patterns include:
  - ✓ Adapter
  - ✓ Composite
  - ✓ Facade

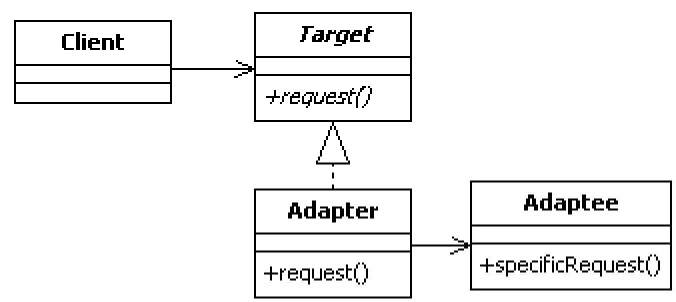
ADAPTER DESIGN PATTERN

- 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.



Class Structural version of Adapter Pattern



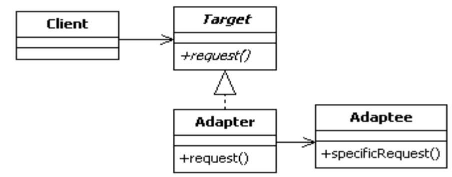
Object Structural version of Adapter Pattern

Adapter Design Pattern Example

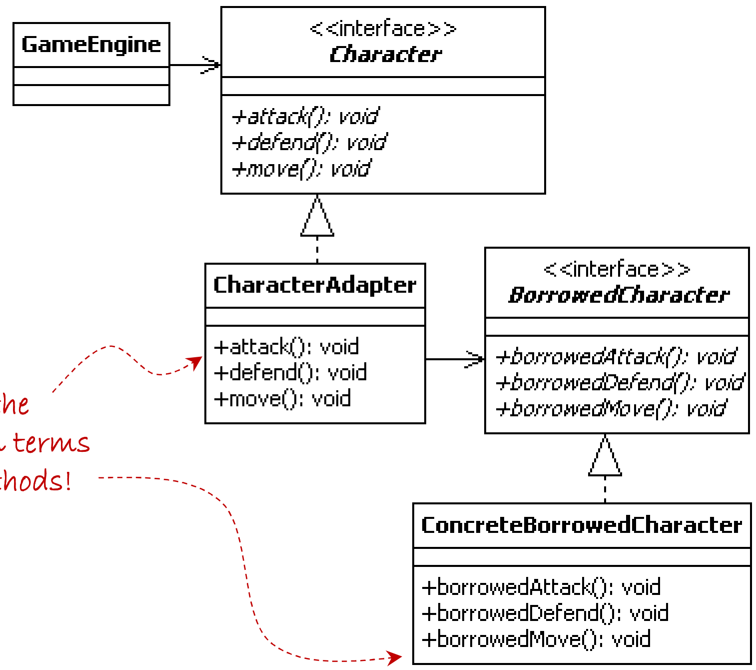
Problem

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.

ADAPTER DESIGN PATTERN EXAMPLE



Here's the object Structural version of Adapter Pattern



The idea is to implement the Character interface methods in terms of the BorrowedCharacter methods!

Here's the object Structural version of Adapter Pattern applied to the gaming problem

Let's see how this is done…

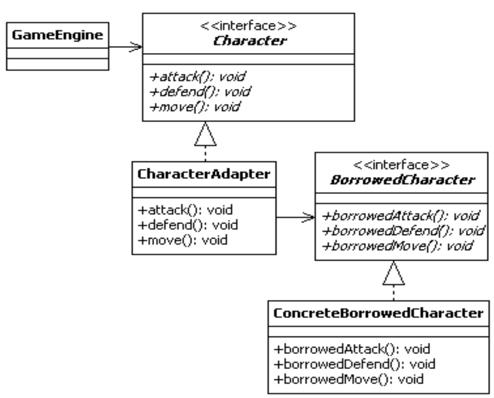
Adapter Design Pattern Example

```
// 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;
};
```

Notice the incompatible interfaces!

```
// Interface for the borrowed character.
class BorrowedCharacter {

public:
  // Interface methods for the borrowed character.
  virtual void borrowedAttack(void) = 0;
  virtual void borrowedDefend(void) = 0;
  virtual void borrowedMove(void) = 0;
};
```

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




```
// Concrete borrowed character.
class ConcreteBorrowedCharacter : public BorrowedCharacter {

public:
  // Implementations for the BorrowedCharacter interface methods.
  void borrowedAttack(void) { cout<<"borrowed attack...\n"; }
  void borrowedDefend(void) { cout<<"borrowed defense...\n"; }
  void borrowedMove(void) { cout<<"borrowed movement...\n\n"; }
};
```

Adapter Design Pattern Example

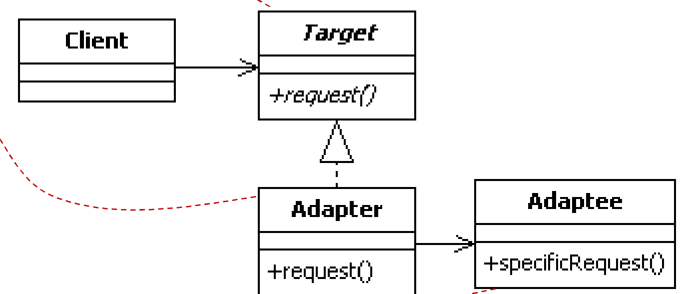
Character is the Target class

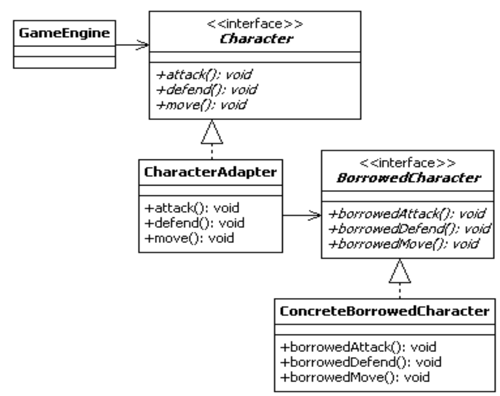
```
class CharacterAdapter : public Character {
public:
  // Constructor.
  CharacterAdapter(BorrowedCharacter* pCharacter);

  // 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;
};
```

The CharacterAdapter

We configure the Adapter with its Adaptee

The class that we are adapting





Adapter Design Pattern Example

```
// 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();
}
```

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 BorrowedChracter

With this design in place, you can now use CharacterAdapter to provide BorrowedCharacter services throughout the game!

Adapter Design Pattern Example

```
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

Assuming the gaming engine has code similar to this

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

```
// Instantiate the game engine.
GameEngine engine;

// Create the borrowed character that needs adapting.
ConcreteBorrowedCharacter borrowedCharacter;

// Create the character adapter and pass in the borrowed character.
// From this point on, the adapterCharacter object can be used
// throughout the game engine as if it were a Character!
CharacterAdapter adaptedCharacter(&borrowedCharacter);

// Move, attack, and defend with the borrowed character's features!
engine.triggeredAction(&adaptedCharacter);
```

ADAPTER DESIGN PATTERN

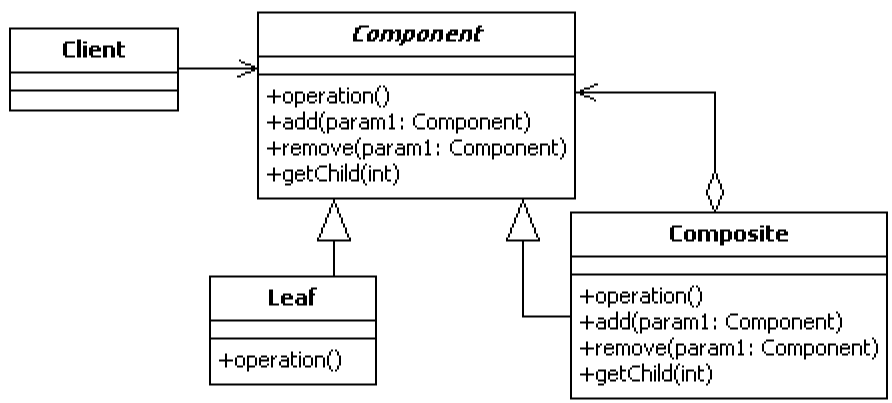
- The step-by-step approach for designing the object structural adapter design pattern is presented as follows:
  1. Identify the source and destination interfaces that need adapting in the new system (e.g., Target and Adaptee or Character and SpecialCharacter).
  2. Add a new class (e.g., Adapter or 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.

COMPOSITE DESIGN PATTERN

- 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 [1]
  - ✓ 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.

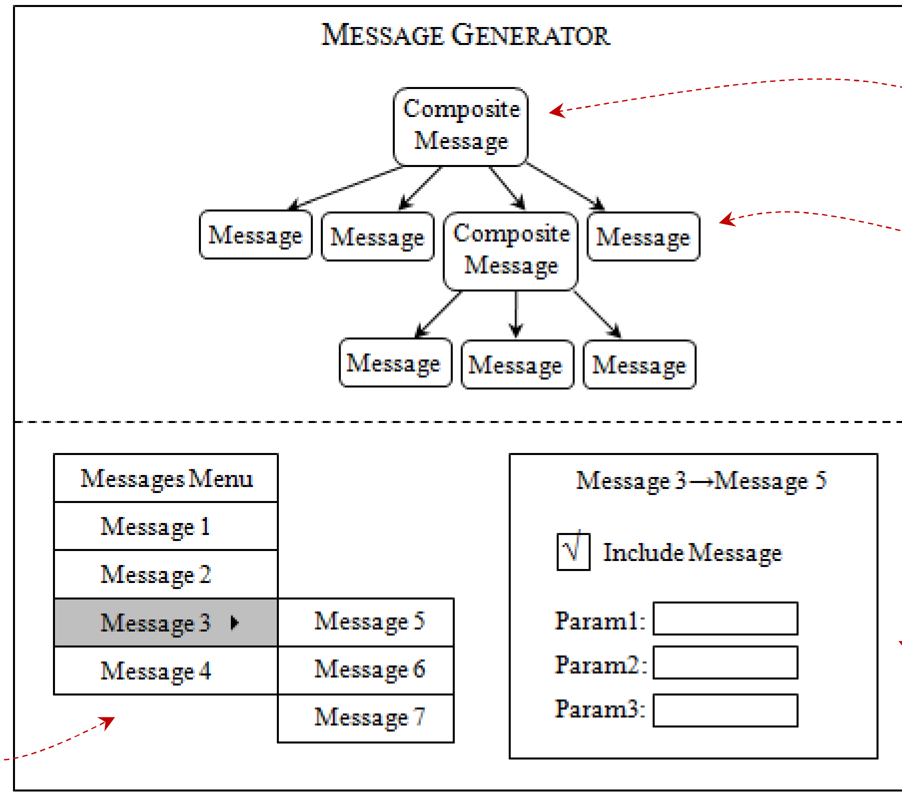
A graphical representation is presented in the next slide…

Composite Design Pattern Example

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



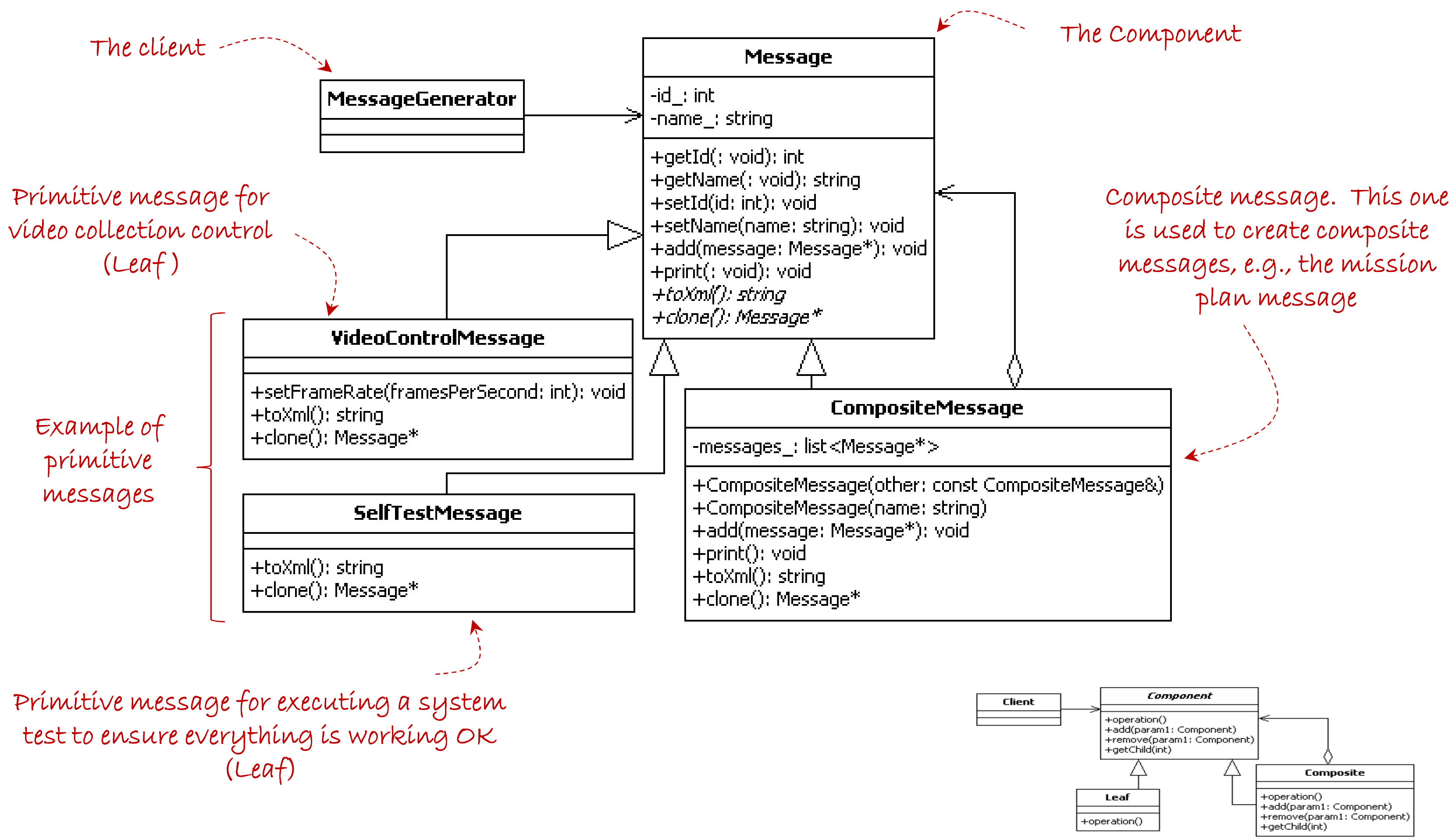
This is one message. More specifically, it is a composite message composed of other messages

This is also a message, but, unlike the composite, this primitive message does not contain other messages!

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

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.

COMPOSITE DESIGN PATTERN EXAMPLE



COMPOSITE DESIGN PATTERN EXAMPLE

```
class Message {

public:

    // Constructor.
    Message(void);

    // Destructor.
    virtual ~Message(void);

    // Method to retrieve the message's id.
    int getId(void) const;

    // Method to retrieve the message's name.
    string getName(void) const;

    // Method to set the message's id.
    void setId(int id);

    // Method to set the message's name.
    void setName(string name);

    // Method to add messages to a composite message.
    virtual void add(Message* message);

    // Method to display messages to the console.
    virtual void print(void);

    // Method to transform the contents of this message to XML format.
    virtual string toXml(void) = 0;

    // Duplicate Messages using the prototype design pattern.
    virtual Message* clone(void) = 0;

private:
    // The message's id.
    int _id;

    // The message's name.
    string _name;
};
```

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?

Composite Design Pattern Example

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

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

```
// 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";
}
```

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

```
// 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

```
class CompositeMessage : public Message
{
public:
    // Overloaded constructor to set the message's name.
    CompositeMessage(string name);

    // Copy constructor.
    CompositeMessage(const CompositeMessage& other);

    // Destructor to clean up memory for messages in _message.
    virtual ~CompositeMessage(void);

    // Method to add messages to this Composite Message.
    void add(Message* message);

    // Method to access the list of messages contained.
    list<Message*> getMessages(void) const;

    // Override the print method to display all messages in _message.
    virtual void print(void);

    // Method to transform the contents of this message to XML format.
    string Message::toXml(void);

    // Create a duplicate of the Composite Message using the prototype
    // design pattern.
    Message* clone(void);

private:

  // The Messages that make up the Composite Message.
  list<Message*> _messages;
};
```

Nothing fancy here, simply realizing the Message interface

Composite contain other messages, so we'll use a list to hold these other messages

Composite Design Pattern Example

This code adds a message to the list of contained messages

```
// Add a message to the collection of messages in the Composite Message.
void CompositeMessage::add(Message* message) {

  // Add this message.
  _messages.push_back(message);
}
```

This code displays the composite message's id and name, but also, it needs to iterate through the list of contained messages and calls print on each of them!

Compare this with the print for primitive messages! Yikes!

```
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();
  }
}
```

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

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

```
// 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
// design 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.getMessages().size();

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

Override the copy constructor!

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

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!

Composite Design Pattern Example

Create primitive messages

Add messages to the composite message to initialize the system

More primitive messages

Another composite message. This one is used for controlling the collection of information by the sensor system.

Add messages to the composite message for scheduling data collection

Schedule a shutdown system message

The mission plan message, which contains two composite messages and one primitive message

Print to the console the mission plan message

```
// 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

```java
// 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

C:\Windows\system32\cmd.exe

```
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 . . .
```

COMPOSITE DESIGN PATTERN

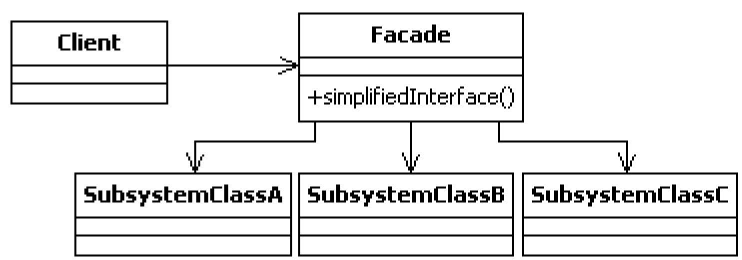
- The steps required to apply the composite design pattern include:
  1. Identify, understand, and plan the tree-like structure required for the system.
  2. With the knowledge from step 1, identify and design the Component base class (see diagram), which includes overridable methods common to both Leaf 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.

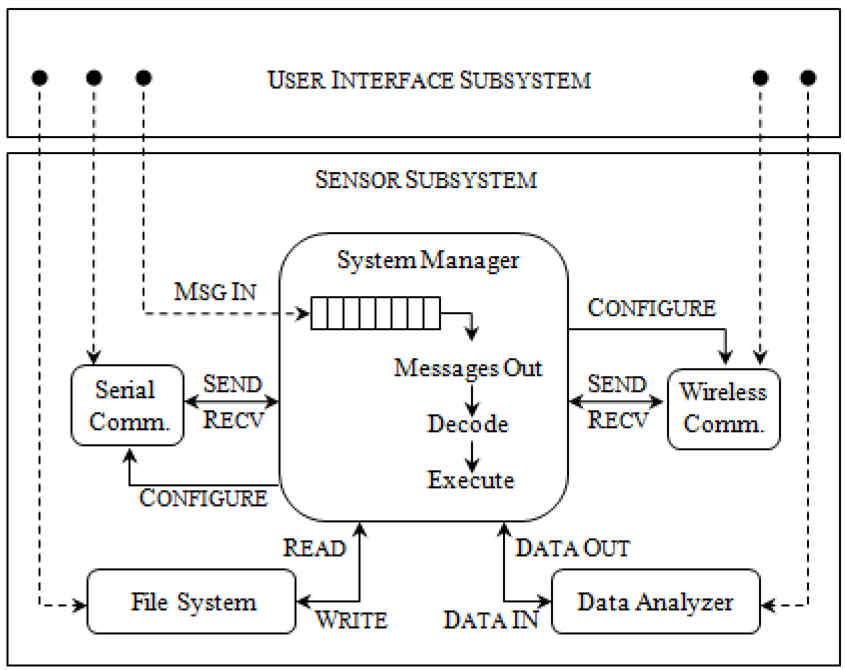
- According to the GoF, the intent of the Facade is to [1]
  - ✓ Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



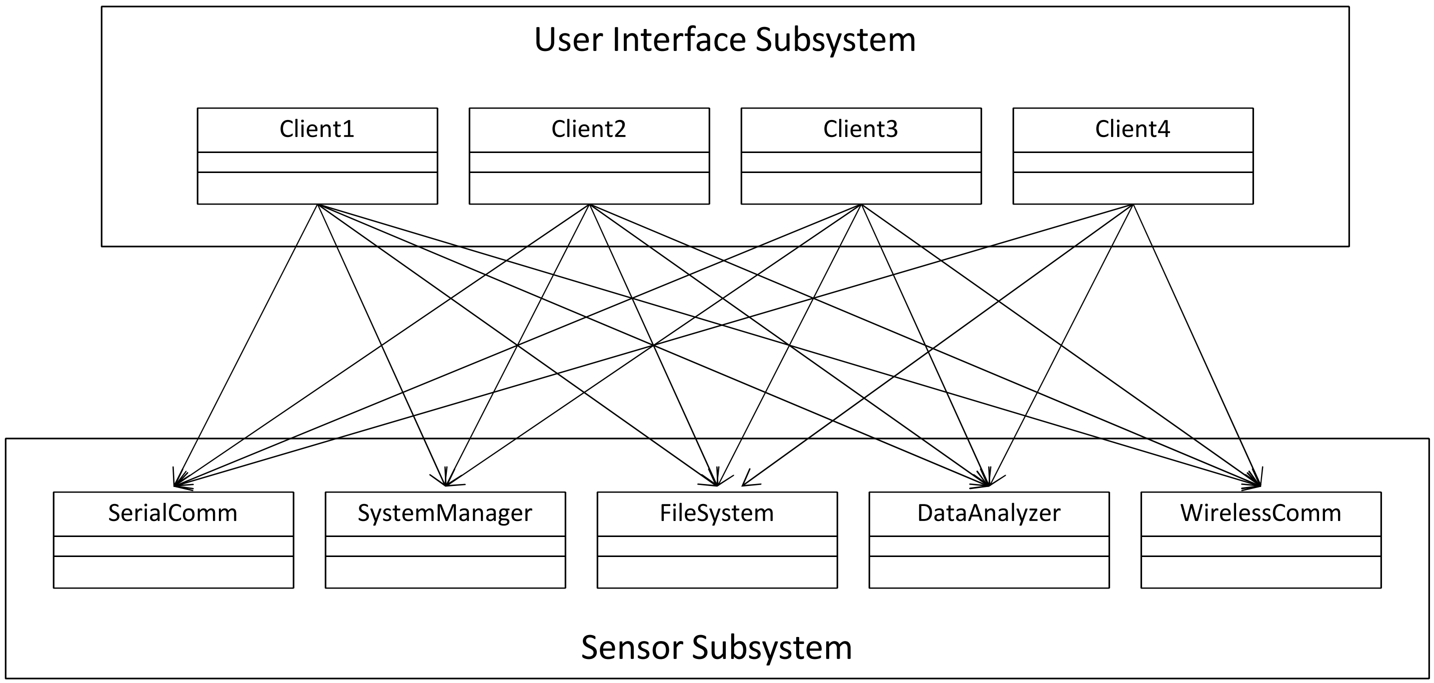
FACADE DESIGN PATTERN EXAMPLE

Problem

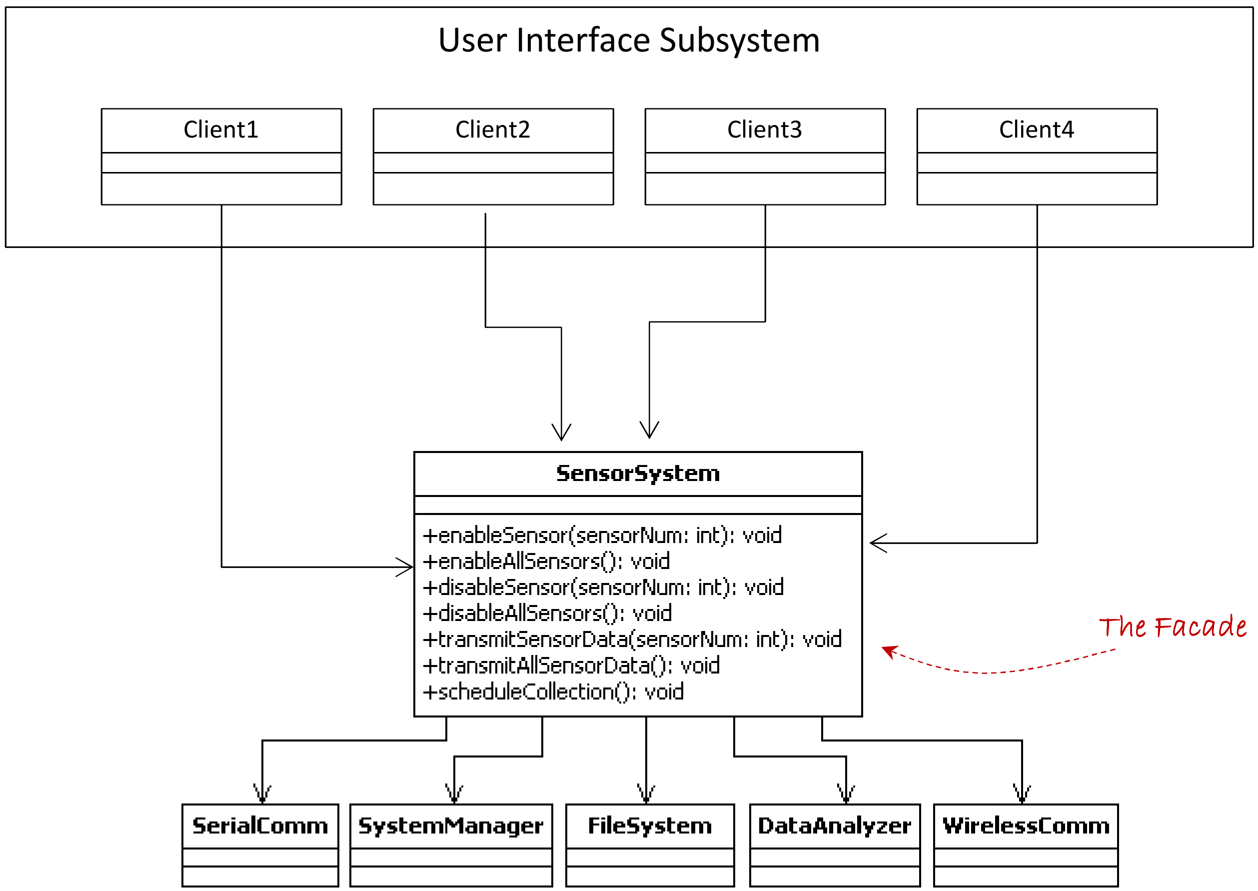
Consider the sensor system described as part of the message generator in the previous section. Upon field deployment, it is desirable to test the system's capabilities to ensure that the system works properly before engaging in autonomous operation. For this reason, a graphical user interface is required to monitor and control the system in the field during installation.



FACADE DESIGN PATTERN EXAMPLE



FACADE DESIGN PATTERN EXAMPLE



FACADE DESIGN PATTERN EXAMPLE

```
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.setByteSize( 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( serialComm->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.
```

The Façade hides all of these details and dependencies from clients, therefore simplifying the client side.

FACADE 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 ore 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

REFERENCES

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