Using Design Patterns

If you do it right, it can be a beautiful thing.

#1

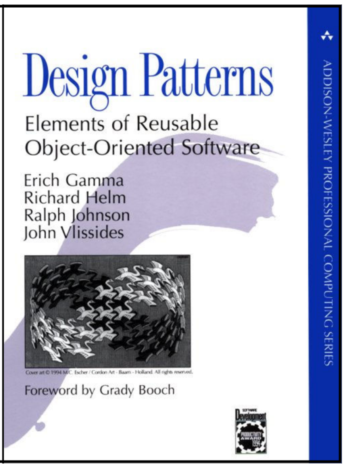
One-Slide Summary

- **Design patterns** are solutions to **recurring** OOP design problems. There are patterns for constructing objects, structuring data, and object behavior.
- Since this is PL, we'll examine how language features like (multiple) inheritance and dynamic dispatch relate to design patterns.

#2

Lecture Outline

- Design Patterns
- Iterator
- Observer
- Singleton
- Mediator



What is a design pattern?

- A solution for a *recurring* problem in a large object-oriented programming system
  - Based on Erich Gamma's Ph.D. thesis, as presented in the "gang of four" book
- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"
  - Charles Alexander

#4

Types of design patterns

- Design patterns can be (roughly) grouped into three categories:
- **Creational patterns**
  - Constructing objects
- **Structural patterns**
  - Controlling the structure of a class, e.g. affecting the API or the data structure layout
- **Behavioral patterns**
  - Deal with how the object behaves

#5

Iterator design pattern

- Often you may have to *move through a collection*
  - Tree (splay, AVL, binary, red-black, etc.), linked list, array, hash table, dictionary, etc.
- Easy for arrays and vectors
- But hard for more complicated data structures
  - Hash table, dictionary, etc.
- The code doing the iteration should not have to know the details of the data structure being used
  - What if that type is not known at compile time?
- This pattern answers the question: How do you provide a standard interface for moving through a collection of objects whose data structure is unknown?

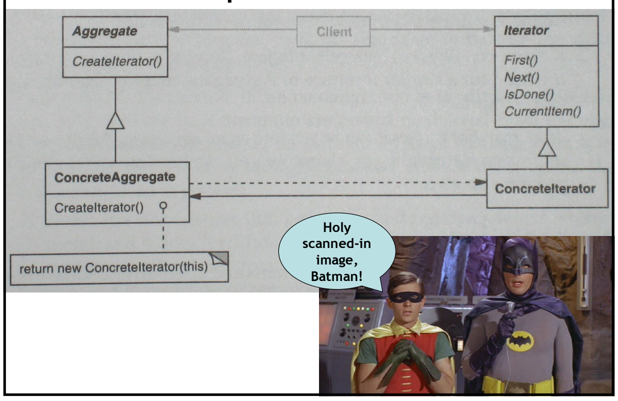
#6

Iterator pattern

- The key participants in this pattern are:
  - The **Iterator**, which provides an (virtual) interface for moving through a collection of things
  - The **Aggregate**, which defines the (virtual) interface for a collection that provides iterators
  - The **ConcreteIterator**, which is the class that inherits/extends/implements the Iterator
  - The **ConcreteAggregate**, which is the class that inherits/extends/ implements the Aggregate
- This pattern is also known as **cursor**
- Iterator is a pattern that shows why we would use multiple inheritance (or Java Interfaces) – *why?*

#7

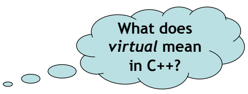
Iterator pattern: Structure



Iterator pattern: class Iterator

- We might use an abstract C++ class to define **Iterator**:

```
template <class Item>
class Iterator {
public:
  virtual void First() = 0;
  virtual void Next() = 0;
  virtual bool IsDone() const = 0;
  virtual Item CurrentItem() const = 0;
protected:
  Iterator();
}
```



- Any collection class that wants to define an iterator will define another (concrete iterator) class that inherits from this class. *How would we do this in Cool?*

#9

Language Design Segue

- In C++ you specify whether you want dynamic dispatch on a *per-method basis*
  - By saying "virtual" or not
  - It then applies to all call sites
- In Cool you specify whether you want dynamic dispatch on a *per-call-site basis*
  - By saying "@Type" for static dispatch or not
- *When is one approach "better"?*

Iterator pattern: class AbstractAggregate

- An abstract C++ class defining **AbstractAggregate**:

```
template <class Item>
class AbstractAggregate {
public:
  virtual Iterator<Item>* CreateIterator() const = 0;
  //...
}
```

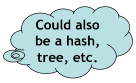
- Any collection class that wants to provide iterators will inherit from this class

#11

Iterator pattern: class List

- Example List collection class:

```
template <class Item>
class List : public AbstractAggregate {
public:
  List (long size = DEFAULT_LIST_CAPACITY);

  long Count() const;
  Item& Get (long index) const;
  // ...

  // and the method to provide the iterator...
}
```



#12

Iterator pattern: class ListIterator

- We use an abstract C++ class to define the Iterator:

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
  ListIterator (const List<Item>* aList);
  void First();
  void Next();
  bool IsDone() const;
  Item CurrentItem() const;

private:
  const List<Item>* _list;
  long _current;
}
```

- Any collection class that wants to define an iterator will define another (concrete iterator) class that inherits from this class

#13

Iterator pattern: class ListIterator

```
template <class Item>
void ListIterator<Item>::First() {
  _current = 0;
}

template <class Item>
void ListIterator<Item>::Next() {
  _current++;
}
```

#14

Iterator pattern: class ListIterator

```
template <class Item>
void ListIterator<Item>::IsDone() const {
  return _current >= _list->Count();
}

template <class Item>
void ListIterator<Item>::CurrentItem() const {
  if (IsDone())
    throw IteratorOutOfBounds;
  return _list->Get(_current);
}
```

#15

Iterator pattern: class List cont'd

- The List class now provides the concrete method for the CreateIterator() abstract method

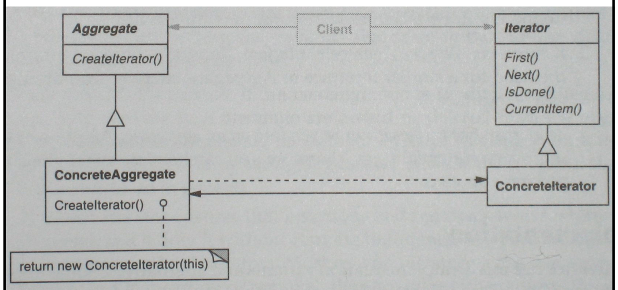
```
template <class Item>
Iterator<Item>* List<Item>::CreateIterator() const {
  return new ListIterator<Item>(this);
}
```

- We note that in the List class header:

```
  Iterator<Item>* CreateIterator() const;
```

#16

Iterator pattern: Structure again



#17

Iterator pattern: Consequences

- An iterator supports variations in transversal of an aggregate
  - The List class can provide one that iterates forward and one that iterates backward
  - Moving through a tree can be done in pre-order, in-order, or post-order
    - Separate methods can provide iterators for each transversal manner
- Iterators support the aggregate interface
- More than one transversal can be moving through an aggregate (*how?*)
  - Multiple iterators can be working at any given time

#18

Iterator pattern: Beyond Iterators

- Java defines an Iterator interface
  - Provides the hasNext(), next(), and remove() methods
- A sub-interface of that is the ListIterator
  - Sub-interface is "inheritance" for interfaces
  - Provides additional methods: hasPrevious(), nextIndex(), previous(), previousIndex(), set()
- Some methods can provide a ListIterator
  - Arrays, lists, vectors, etc.
- And some cannot
  - Hash tables, dictionaries, etc.

#19

Observer design pattern

- When a object changes state, other objects may have to be notified
  - Example: when an car in a game is moved
    - The graphics engine needs to know so it can re-render the item
    - The traffic computation routines need to re-compute the traffic pattern
    - The objects the car contains need to know they are moving as well
  - Another example: data in a spreadsheet
    - The display must be updated
    - Possibly multiple graphs that use that data need to re-draw themselves
- This pattern answers the question: *How best to notify those objects when the subject changes?*
  - And what if the list of those objects changes?

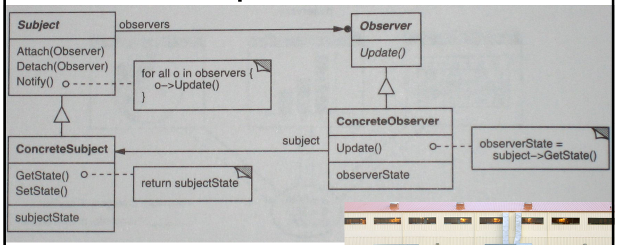
#20

Observer pattern

- The key participants in this pattern are:
  - The **Subject**, which provides an (virtual) interface for attaching and detaching observers
  - The **Observer**, which defines the (virtual) updating interface
  - The **ConcreteSubject**, which is the class that inherits/extends/implements the Subject
  - The **ConcreteObserver**, which is the class that inherits/extends/implements the Observer
- This pattern is also known as **dependents** or **publish-subscribe**
- Observer is another pattern that shows why we would use multiple inheritance

#21

Observer pattern: Structure



Observer pattern: class Observer

- Example abstract C++ Observer class:

```
class Observer {
public:
  virtual ~Observer();
  virtual void
     Update(Subject* theChagnedSubject) = 0;
protected:
  Observer();
}
```

- Any class that wants to (potentially) observe another object will inherit from this class

#23

Observer pattern: class Subject

- Abstract C++ class to define the Subject:

```
class Subject {
public:
  virtual ~Subject();
  virtual void Attach (Observer*);
  virtual void Detach (Observer*);
  virtual void Notify();
protected:
  Subject();
private:
  List<Observer*> *_observers;
};
```

What does ~ mean in C++?

- Any class that can be observed will inherit from this class

#24

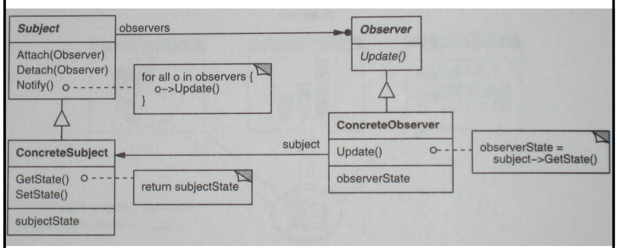
Observer pattern: class Subject

```
void Subject::Attach (Observer* o) {
  _observers->Append(o);
}

void Subject::Detach (Observer* o) {
  _observers->Remove(o);
}

void Subject::Notify() {
  ListIterator<Observer*> i(_observers);
  for ( i.First();  !i.IsDone();  i.Next() )
    i.CurrentItem()->Update(this);
}
```

Builds on iterators!
How cool are we?

#25

Observer pattern structure again



#26

Observer pattern: Consequences

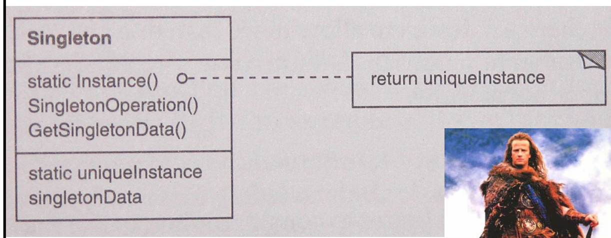
- Abstract coupling between subject and observer
  - Subject has *no idea* who the observers are (or what type they are)
- Support for broadcast communication
  - Subject can notify any number of observers
  - Observer can choose to ignore notification
- Unexpected updates
  - Subjects have no idea the *cost* of an update
  - If there are many observers (with many dependent objects), this can be an expensive operation
  - Observers do not know *what* changed in the subject, and must then spend time figuring that out

#27

Singleton design pattern

- In many systems, there should often only be one object instance for a given class
  - Print spooler
  - File system
  - Window manager
- This pattern answers the question: *How to design the class such that any client cannot create more than one instance of the class?*
- The key participants in this pattern are:
  - The **Singleton**, the class which only allows one instance to be created

Singleton pattern: Structure



Singleton pattern: class Singleton

- Example C++ Singleton class:

```
class Singleton {
public:
  static Singleton* Instance();
protected:
  Singleton();
private:
  static Singleton* _instance;
};

Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance() {
  if ( _instance == 0 )
    _instance = new Singleton();
  return _instance;
}
```

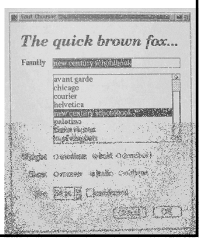
Singleton pattern: Consequences

- *Controlled access* to sole instance
  - As the constructor is protected, the class controls when an instance is created
- Reduced name space
  - Eliminates the need for *global variables* that store single instances
- Permits refinement of operations and representations
  - You can easily sub-class the Singleton
- Permits a variable number of instances
  - The class is easily modified to allow *n* instances when *n* is not 1
- More flexible than class operations
  - This pattern eliminates the need for class (i.e. static) methods
  - Note that (in C++) static methods are never virtual

#31

Mediator design pattern

- What happens if multiple objects have to communicate with each other
  - If you have many classes in a system, then each new class has to consider how to communicate with each existing class
  - Thus, you could have n^2 communication protocols
- Example
  - Elements (widgets) in a GUI
  - Each control has to modify the font
  - But we shouldn't have each widget have a separate communication means with every other widget
- This pattern answers the question: *How to define an object to encapsulate and control the communication between the various objects?*

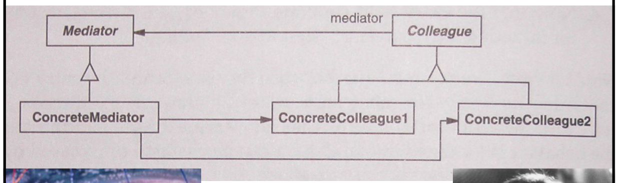


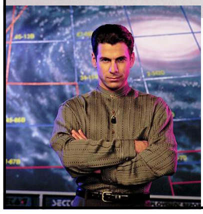
Mediator pattern

- The key participants in this pattern are:
  - The **Mediator**, which defines an abstract interface for how the Colleague classes communicate with each other
  - The **ConcreteMediator**, which implements the Mediator behavior
  - Multiple **Colleague classes**, each which know the ConcreteMediator, but do not necessarily know each other
- In the GUI example, the classes could be implemented as follows:
  - Mediator: DialogDirector
  - ConcreteMediator: FontDialogDirector
  - Colleague classes: ListBox, EntryField, RadioButton, etc.
    - All these classes inherit from the Widget class

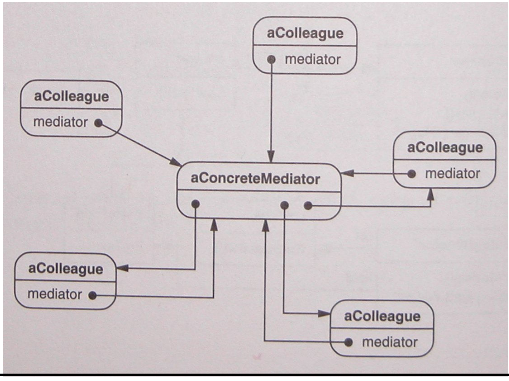
#33

Mediator pattern: Structure





Mediator pattern: Structure



Mediator pattern: class DialogDirector

- Abstract C++ class for a DialogDirector:

```
class DialogDirector {
public:
  virtual ~DialogDirector();
  virtual void ShowDialog();
  virtual void WidgetChanged(Widget*) = 0;
protected:
  DialogDirector();
  virtual void CreateWidgets() = 0;
}
```

- Whenever a widget is modified, it will call the WidgetChanged() method

Mediator pattern: class FontDialogDirector

```
class FontDialogDirector : public DialogDirector {
public:
  FontDialogDirector();
  ~FontDialogDirector();
  void WidgetChanged(Widget*);
protected:
  void CreateWidgets();
private:
  Button* _ok;
  Button* _cancel;
  ListBox* _fontList;
  EntryField* _fontName;
}
```

- Note that we probably would want to make this class a Singleton as well (via multiple inheritance)

#37

Mediator pattern: method CreateWidgets()

- An implementation of the CreateWidgets() method

```
void FontDialogDirector::CreateWidgets () {
  _ok = new Button(this);
  _cancel = new Button(this);
  _fontList = new ListBox(this);
  _fontName = new EntryField(this);

  // fill the listBox with the available font names

  // assemble the widgets in the dialog
}
```

- In the actual dialog, it would probably need more controls than the above four…

#38

Mediator pattern: method WidgetChanged()

- An implementation of the WidgetChanged() method

```
void FontDialogDirector::WidgetChanged (
  Widget* theChangedWidget
) {
  if (theChangedWidget == _fontList) {
    _fontName->SetText (_fontList->GetSelection());
  } else if ( theChangedWidget == _ok ) {
    // apply font change and dismiss dialog
    // ...
  } else if ( theChangedWidget == _cancel ) {
    // dismiss dialog
  }
}
```

- Here the actual communication between the widgets is implemented

#39

Mediator pattern: Consequences

- It *limits subclassing*
  - The communication behavior would otherwise have to be distributed among many sub-classes of the widgets
  - Instead, it's all in the Mediator
- It decouples colleagues
  - They don't have to know how to interact with each other
- It simplifies object protocols
  - A Mediator replaces many-to-many communication with a one-to-many paradigm
- It abstracts how objects cooperate
  - How objects communicate is abstracted into the Mediator class
- It centralizes control
  - Again, it's all in the Mediator
  - This can make the Mediator quite large and monolithic in a large system

#40

Creational Design Patterns

- Abstract Factory
- Builder
- **Factory Method**
- Prototype
- Singleton

#41

Structural Patterns

- **Adapter**
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- **Proxy**

The model-view-controller architectural pattern should also be mentioned!

#42

Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- **Iterator**
- Mediator
- Memento
- **Observer**
- State
- Strategy
- Template Method
- **Visitor**

#43

Homework

- WA8 Due Today
- PA5 Due Friday April 27 (8 days)

#44