Overview

- Design pattern concepts
- Kinds of patterns
- Some specific patterns
- Pattern resources

Design Pattern

- Solution to a particular kind of problem
- How to combine classes and methods
- Based on design experience
- Use requires understanding of the appropriate problem and being able to recognize when such problems occur

Kinds of Patterns

- Factory patterns – creation of objects
- Delegation patterns – coordination of objects
- Control patterns – organization of control in application, control structures, algorithmic patterns

Factory Pattern Rationale

- Examples:
  - When using objects polymorphically: base class cannot create derived objects
  - Want to control number of copies: "copies" are pointers to actual object
  - Cloning prototype object: can make copies of existing object

Kinds of Factory Patterns

- Builder – simple factory object
- Abstract factory – different implementations
- Flyweight – constructing shared objects
- Singleton – “global objects”
- Factory method – adding factory to a class
- Prototype – creating objects by cloning
Builder Pattern

• Generic Class Diagram

```
  Base
  +--- DerivedA
  +--- DerivedB

  BaseFactory
   +-- createA() / createB()
```

• Creation method prototype
  Base* createA(/*parameters needed for creation*/);

Builder Example

• Game with multiple levels, each of which is distinguished by characters of different appearance and capabilities.
• Code for game control is same for all levels
• Use inheritance to define hierarchy of character classes
• Factory object can build any kind of character

Abstract Factory Pattern

• Similar to builder except have hierarchy of factory classes
• Can change behavior of factory by changing which derived factory object is used

Flyweight Pattern

• Goal: minimize number of copies of objects
• Factory maintains a table of objects
• Request to create object handled by testing
  – if object exists, return pointer (also see Bridge)
  – if object does not exist, create, add to table and return pointer
• Could use map; hash table faster lookups

Flyweight Example

• Computer algebra system (e.g., Mathematica)
• Representing rational numbers (1/2, 33/56)
• No need to have multiple copies of 1/2
• Factory maintains table of objects
• Number objects are lightweight (hold pointers to heavyweight object in table)

Prototype Pattern

• No separate factory class
• Class has clone function that creates exact copy of object
• Example:
  – Palette of drawing program includes an object.
  – User selects object to copy into drawing
  – Placing object creates clone.
Delegation Patterns

- **Adaptor** – using functions of another class
- **Bridge** – treating class as pointer to separate interface from implementation
- **Decorator** – change behavior of object
- **Facade** – encapsulating subsystem
- **Proxy** – placeholder for remote objects

Adaptor Pattern

- Adapt existing class to needs of new class
- Example: implementing a stack class using a list class
  - push uses list function insert
  - pop uses list function remove
- Either use aggregation or association

Bridge Pattern

- Define class to act as interface
- Holds pointer to implementation class
- Can control use of pointer
  - Check for undefined, null
  - Aliasing
  - Reference counting
- Also called a wrapper class

Bridge Example

- Reference counting - similar motivation to flyweight: avoid copying object
- Implementation object holds count of the number of wrapper objects that point to it
- “Copying” wrapper object actually copies reference count
- Care required with modification because of aliasing problems

Reference Counting Example

```cpp
class RationalNum {
public:
RationalNum() : rep(new RationalRep()) {
    rep->incr();
}
RationalNum(int n, int d) : rep(new RationalRep(n,d)) {
    rep->incr();
}
RationalNum(const RationalNum& r) : rep(r.rep) {
    rep->incr();
}
RationalNum operator+(const RationalNum&) const;
~RationalNum() { if (decr()== 0) delete rep; }
private:
RationalRep* rep;
RationalNum(RationalRep* r) : rep(r) {
    rep->incr();
};
}
```

Reference Counting (2)

```cpp
class RationalRep {
public:
RationalRep() : num(0), denom(0), refCount(0) {}
RationalRep(int n, int d) : num(n), denom(d), refCount(0) {}
RationalRep(const RationalRep& r) : num(r.num),
    denom(r.denom), refCount(0) {};
void add(const RationalRep&, const RationalRep&);
int incr() { refCount++; return refCount;};
int decr() { refCount--; return refCount;};
private:
int num, denom;
int refCount;
};
```
Reference Counting (3)

```c++
void RationalRep::
    add(const RationalRep& a, const RationalRep& b) {
        if (a.denom == b.denom) {
            denom = a.denom;
            num = a.num + b.num;
        } else {
            num = a.num * b.denom + b.num * a.denom;
            denom = a.denom * b.denom;
        }
    }
```

Reference Counting (4)

```c++
RationalNum RationalNum::
    operator+(const RationalNum& a) const {
        RationalRep* res = new RationalRep();
        res->add(*rep,*a.rep);
        res->refCount++;
        return RationalNum(res);
    }
```

Reference Counting (5)

- Assignment operator would look like

```c++
RationalNum& RationalNum::
    operator= (const RationalNum& r) {
        if (this != &r) {
            if (--rep->refCount == 0) delete rep;
            rep = r.rep;
            rep->refCount++;
        }
    return *this;
    }
```

Decorator Pattern

- Behavior of object may need to change significantly during execution
- Object has different states in which behavior is different
- Decorator object holds pointer
- Change in state changes object pointed to
- Pointed to object provides behavior

Decorator Example

- In game an object representing a door
- Door has locked and unlocked states
- Door is locked until something in game unlocks it
- Use inheritance to define door objects with different behavior, and initially use a locked door object, and replace by unlocked door

Façade Pattern

- Façade object delegates responsibilities to multiple objects
- Façade is effectively the interface for a subsystem to support information hiding
- Generally implemented with aggregation
Façade Example

- Subsystem of game for handling display
- Façade is interface for game commands
- Façade delegates responsibility to different commands
- Hides details of GUI, so localizes changes needed when change GUI

Proxy Pattern

- **Problem**: need interaction with object that exists as data on disk, or in another process, or on another computer (*remote* object).
- Proxy object represents the remote object
- Forwarding may involve
  - reading from or writing to disk
  - interprocess communication
  - sending/receiving network messages

Proxy Example

- Persistent objects
  - Objects that exist between invocations of program
  - Stored on disk (in special database if supported)
- Proxy object represents object on disk
- Proxy reads data from disk when first needed, writes to disk when program ends

Control Patterns

- **Composite** – treat group of objects as one
- **Interpreter** – represent text input as object
- **Command** – user interface command objects
- **Iterator** – iteration through containers
- **Strategy** – encapsulate algorithm
- **Template** – algorithm with steps from derived class
- **Visitor** – operations applied to elements of heterogeneous container

Composite Pattern

- Treat group of objects as a single object
- Method of group object forwards call to methods of group members (broadcast)
- Method implemented by iterating through elements of group and calling method
- “Grouping” is either association or aggregation

Example Composite Pattern

- Drawing program - can form group of shapes

  - Group::draw() calls draw() for all members in group
**Interpreter Pattern**

- Represent textual input as object that can be given to other parts of program
- Parser constructs object based on grammar rules – similar to how compiler works
- Separates parsing from interpretation of input
- Proper use requires knowing about language grammars

**Abstract Syntax Trees**

- Example: tree for expression $3 \times x + y \times z$

**Interpreter Example**

- Expressions with binary operators: $+,-,\times,\div$

**Command Pattern**

- Represent command from user interface as object
- Typically has methods `do()`, `undo()`
- Separate interface details from rest of program by passing command object
- Can construct command macros by combining commands together

**Example Command Pattern**

- Commands for an editor

**Iterator Patterns**

- Simplify traversal of complex data structures
- Two common forms:
  - Iterator type owned by container class
    void list<T>::erase(list<T>::iterator);
  - Container is attribute of iterator
    bool list<iterator<T> remove();
Strategy Pattern
- (Complex) Algorithm implemented as class
- Different algorithmic approaches can be implemented using the same interface
- Can be used with inheritance to allow runtime changes to behavior.

Template Pattern
- Algorithm is a generic template solution
- Three forms:
  - Use templates directly (STL)
  - External “callback” objects (abstract solution) supplies algorithm, (does not use templates)
  - Use inheritance, general algorithm in base class is virtual & redefined in hierarchy for specialization.

Template (callback) Pattern
- Callback FN Organization
  - FrontEnd (interface FNs), invokes BackEnd to do work
  - Contains a displayCallBack FN for BackEnd
  - BackEnd (class) contains FNs to perform work
  - Whenever changes require display update the CallBack FN parameter is invoked

Template (callback) Inheritance
- Inheritance provides cleaner implementation of callbacks than C
  - Avoids generic (void *) pointers
- Back end defines abstract callback base class with pure virtual callback method prototypes
- Front end defines callback subclass with callback FN implementation & required data.
- Front end passes a pointer to callback subclass object to back end.
- Backend uses subclass callback object pointer to invoke specific callback method.

Visitor Pattern
- Abstract type (visitor) outside class hierarchy implements abstract operations.
  - For each hierarchy subclass, visitor class defines virtual method for subclass abstract operation.
  - Each hierarchy subclass contains a virtual method: accept(visitor)
  - Visitor object is used to invoke its visitor abstract operation, passing itself.
  - Visitor object invokes method for the specific type of subclass object.
- Permits adding new operations w/o changing class hierarchy.
Algorithmic Patterns

- **Mediator** – coordinates interactions between associates
- **Memento** – captures and restores object state privately
- **Observer** – automatically update dependent objects when state changes

Mediator Pattern

- Used to control complex interaction between multiple objects
- A mediator object is used update objects as other objects change.
- Cleans up inter-related object communication.
  - e.g., a dialog-box object controls (mediates) contained interface objects.
  - The dialog box object controls the overall window interaction, relationships between the objects/application.

Memento Pattern

- Objects used to save & restore the state of other objects.
- Requires the saved object to have a method for creating the memento object.
- The memento class must have a method for recreating the saved object.
  - e.g., implementing an undo operation could use a memento object to hold only the necessary undo information for an object prior to an operation for later possible undo’ing.

Observer Pattern

- One subject object has many dependent objects called observers
- Change of subject’s state implies reaction by observers
- Different observers may react differently

Observer Pattern

```
for each o in observers
    o.update()
```

Other Pattern Sources

- Gamma, Helm, Johnson, Vlissides. *Design Patterns*
- [www.cs.wustl.edu/~schmidt/tutorials-patterns.html](http://www.cs.wustl.edu/~schmidt/tutorials-patterns.html)
- List of books about patterns: [hillside.net/patterns/books](http://hillside.net/patterns/books)