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: 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       |        |     BaseFactory |
|---------------|        |----------------|
|  Derived A    |        |  Derived B     |
|---------------|        |                |
| BaseFactory   |        |                |
|---------------|        |                |
| Derived A     |        | Derived B      |
|               |        |                |
```

- Creation method prototype

```cpp
Base* createA() /*parameters needed for creation*/
```

Builder Example

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

Flyweight Pattern

- Goal: minimize number of copies of objects
- Factory maintains a table of objects
- Request to create object handled by
  - if object exists, return pointer (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 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

- Adapter - using functions of another class
- Bridge - treating class as pointer to separate interface from implementation
- Decorator - change behavior of object
- Façade - 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->refCount++;
    }
    RationalNum(int n, int d) : rep(new RationalRep(n,d)) {
        rep->refCount++;
    }
    RationalRep(const RationalRep& r) : rep(r.rep) {
        rep->refCount++;
    }
    RationalNum(const RationalNum& r) {
        if (!rep->refCount + 0) delete rep;
    }
private:
    RationalRep* rep;
    RationalNum(RationalRep* & rep) {
        rep->refCount++;
    }
}
```

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& a, const RationalRep& b) {
        int num = a.num + b.num;
        int denom = a.denom;
    }
private:
    int num, denom;
    int refCount;
};
```

Reference Counting (3)

```cpp
void RationalRep::
    add(const RationalRep&, const RationalRep& b) {
        num = a.num + b.num;
        denom = a.denom;
    }
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->refCount += 1;
        return RationalNum(res);
    }
```

Reference Counting (5)

```c
RationalNum& RationalNum::
    operator=(const RationalNum& r) {
        if (this != &r) {
            if ((-rep->refCount == 0) delete rep;
            rep = r.rep;
            rep->refCount += 1;
        }
        return *this;
    }
```

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
- Visitor

Composite Pattern

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

- Drawing program - can form group of shapes

```
Shape
  |__Rectangle
  |__Ellipse
  |__Line
  |__Group

• 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 $3 \times x + y \times z$

```
+   
3  x  y  z
```

- Represents structure of expression

Interpreter Example

- Expressions with binary operators: +,-,*,/.

Example Command Pattern

- Commands for an editor

```
EditCommand
  |__EditBuffer
  |__EditMove
  |__EditTyping
  |__EditCut
  |__EditPaste
  |__EditCopy
  |__...
```

- Commands for an editor

```
Add
  |__Constant
  |__Variable
  |__Expression
  |__Operator
  |__...
```

- Expressions with binary operators: +,-,*,/.

- Separates parsing from interpretation of input
- Proper use requires knowing about language grammars
**Iterator Patterns**

- Simplify traversal of complex data structures
- Two common forms:
  - Iterator type owned by container class
    ```cpp
    void list<T>::erase(list<T>::iterator);
    ```
  - Container is attribute of iterator
    ```cpp
    bool listIterator<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.
- Function objects an example.