6.S096 Lecture 6 – Design Patterns
Higher-level program design

Andre Kessler

January 22, 2014
Outline

1. Code Review
2. Design Patterns
3. Wrap-up
Don’t overuse this->

- No need for this->_member, just write _member
- (that’s why we use a leading underscore - to distinguish member variables)

How not to do it:

```cpp
void Rational::normalize() {
    auto abs_num = std::abs( this->num );
    auto abs_den = std::abs( this->den );
    auto theSign = this->sign();
    // ..etc, we don’t need ‘this’!
}
```
Don’t overuse this->

- No need for this->_member, just write _member
- (that’s why we use a leading underscore - to distinguish member variables)

Much better:

```cpp
void Rational::normalize() {
    auto abs_num = std::abs( _num );
    auto abs_den = std::abs( _den );
    auto theSign = sign();
    // ..etc, ^^^ better
}
```
Scope issues

SomeClass::whatIsThis()?
What are design patterns?

- “Distilled wisdom” about object-oriented programming
- Solutions to common problems that arise
- Anti-patterns: bad solutions to common problems that arise.
Gang of Four (GoF)
The 23 standard patterns

Design Pattern Relationships
We’ll be covering:

- Strategy (behavioral)
- Composite (structural)
- Factory Method (creational)
class IndexingScheme {
public:
    virtual size_t idx(size_t r, size_t c) = 0;
    virtual ~IndexingScheme() {};
};

class RowMajor : public IndexingScheme {
    size_t _nCols;
    RowMajor() = delete;

    public:
    RowMajor(size_t, size_t numCols) :
        _nCols{numCols} {}
    size_t idx(size_t r, size_t c) {
        return c + r * _nCols;
    }
};
class ColMajor : public IndexingScheme {
    size_t _nRows;
    ColMajor() = delete;

public:
    ColMajor( size_t numRows, size_t ) :
        _nRows{numRows} {}
    size_t idx( size_t r, size_t c ) {
        return r + c * _nRows;
    }
};

Let’s look at the example code...
Composite Pattern

**Component**

- operation()

**Leaf**

- operation()

**Composite**

- operation()
- add()
- remove()
- getChild()
Composite Pattern

We’ll consider the example of a file system.

- Need to represent directories and files
- Directories can contain other files or directories
- Files are “leaf” nodes, probably contain pointers to data.
- This example will also use the **factory pattern**.
class Node {

public:
    virtual ~Node() {}
    virtual Directory* getDirectory() { return nullptr; }
    // ...etc

};

class Directory : public Node {
    std::string _name;
    std::vector<NodePtr> _child; // ...etc

public: // ...etc
    virtual Directory* getDirectory() { return this; }
    void add( NodePtr item ) { _child.push_back( item ); }
    static NodePtr create( const std::string &dirname );

};
// the ‘‘leaf’’ class

class File : public Node {
    std::string _name;
    File() = delete;
    void lsIndented( int indent ) const;

public:
    File( std::string filename ) : _name(filename) {}
    void ls() const;

    static NodePtr create( const std::string &filename );
};

Let’s look at the example code...
Let’s see some examples...
Wrap-up & Friday

Second assignment due tonight at midnight

Third assignment (small) due Saturday 1/25 at midnight

Class on Fri. 1/24 is in 54-100 at 2pm.

Will cover …

- Design patterns and anti-patterns

Questions?

- I’m available after class or on Piazza.
- Office hours 5-7pm Mon, Tues in 26-142