CS 6015: Software Engineering

Spring 2024

Lecture 14: Design Patterns

This Week

- Test Automation
- Design patterns
- Participation (First half)

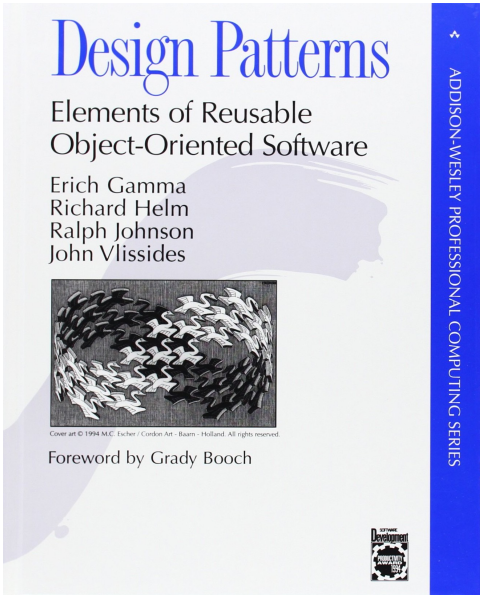
Next Week

- Conditionals (Project related)
- Midterm

Resources

- Design Patterns (**Gang of four**)
- Wikipedia page
- Pluralsight: “Design Patterns in Java”
- Head First Design Patterns

“Gang of four”



Design Patterns

Design Patterns

- Like a template – can be applied in different occasions
- Arrangement of elements to solve a problem
- Capture solutions to design problems
- Reuse of successful design
- Abstract description of a design problem

A pattern describes a recurring software structure

Describing a Design Pattern

- Pattern name
  - Names make it easier to talk about and think about designs.
- *Problem:* In what situation should this pattern be used?
- *Solution:* What should you do? What is the pattern?
  - Describe details of the objects/classes/structure needed
- *Advantages:* Why is this pattern useful?
- *Disadvantages:* Why might someone not want this pattern?

A pattern is NOT an implementation

Benefits of using patterns

- Patterns give a design ***common vocabulary*** for software design
  - Powerful - Say more with less
  - Allows abstraction of a problem
- Help structure applications in ways easier to understand
- Improve understandability → improve documentation

Classifying Patterns

- **Creational Patterns** *(*how objects can be created*)*
  - Factory Method
  - Builder
  - Abstract Factory
  - Prototype
  - Singleton
- **Structural Patterns** *(how objects/classes can be combined)*
  - Adapter
  - Decorator
  - Proxy
  - Bridge
  - Facade
  - Composite
  - Flyweight
- **Behavioral Patterns** *(*how methods can be implemented *)*
  - Command
  - Mediator
  - Strategy
  - Template Method
  - Interpreter
  - Observer
  - Chain of Responsibility
  - Iterator
  - State
  - Visitor

Singleton Pattern

- A class that has only a single instance
- We would like to make it illegal to have more than 1 instance
- Example ?

Singleton Pattern

- A class that has only a single instance
- We would like to make it illegal to have more than 1 instance
- Example: keyboard reader, game, GUI
- Why single instance for such classes?

Singleton Pattern

- A class that has only a single instance
- We would like to make it illegal to have more than 1 instance
- Example: keyboard reader, game, GUI
- Why single instance for such classes?
  - Creating lots of objects can take a lot of time.
  - Extra objects take up memory.
  - Pain to deal with different objects if they are essentially the same.
  - What happens if we have more than 1 GUI?

Singleton Pattern

- **singleton**: An object that is the only object of its type.
  (one of the most known / popular design patterns)

- *Benefits:*
  - Saves memory.
  - Avoids bugs arising from multiple instances.

- How to avoid creating many objects?

Singleton Pattern

- **singleton**: An object that is the only object of its type.
  (one of the most known / popular design patterns)

- *Benefits:*
  - Saves memory.
  - Avoids bugs arising from multiple instances.

- How to avoid creating many objects?
  - Use static methods

Singleton Pattern: Implementation

- Make constructor(s) `private` so that they can not be called from outside by clients.
- Remove any copy constructor
- Declare a single `private static` instance of the class.
- Write a public `getInstance()`

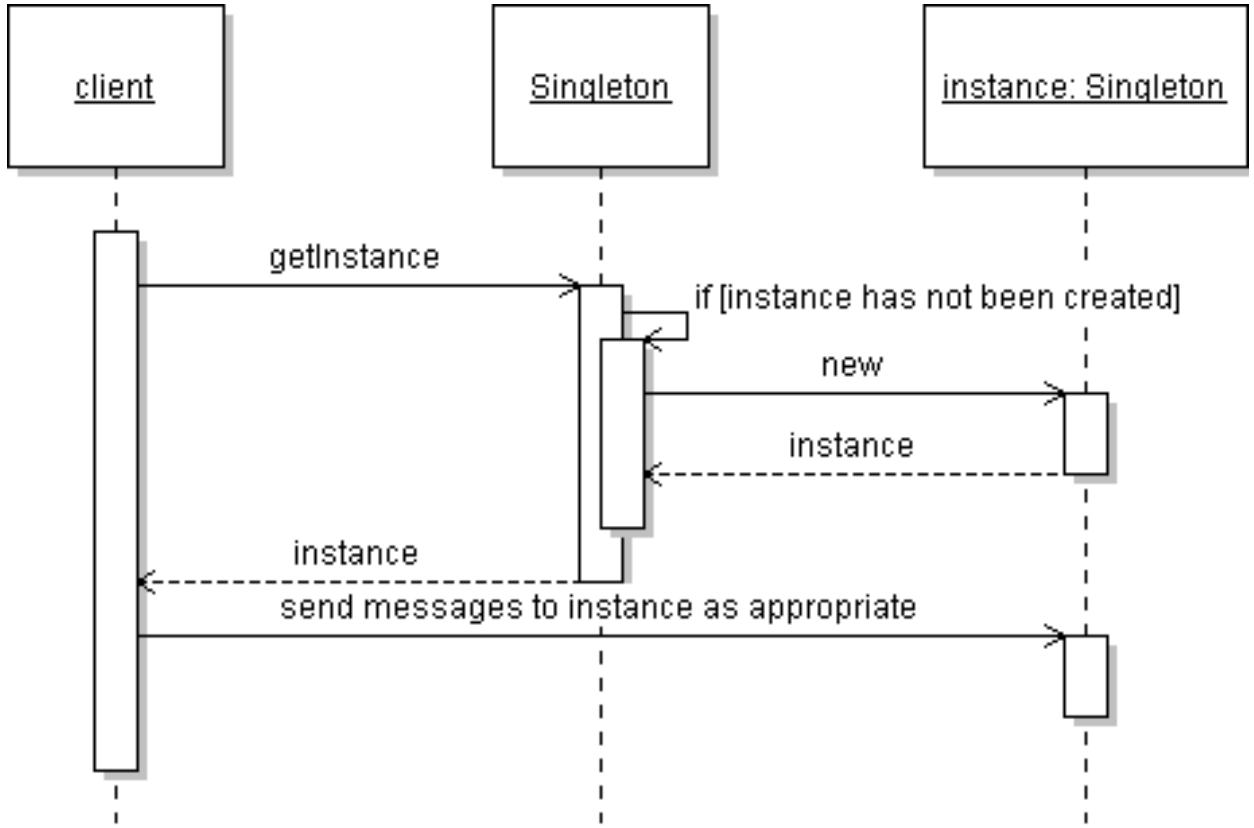
Singleton Pattern: C++ Implementation

```
class Singleton {
   private:
    std::string str;
    Singleton(std::string str) {
        this->str = str;
    }

    public:
    static Singleton *getInstance() {
        static Singleton *instance1 = new Singleton("Singleton");
        return instance1;
    }

};
```

Singleton Pattern: Implementation



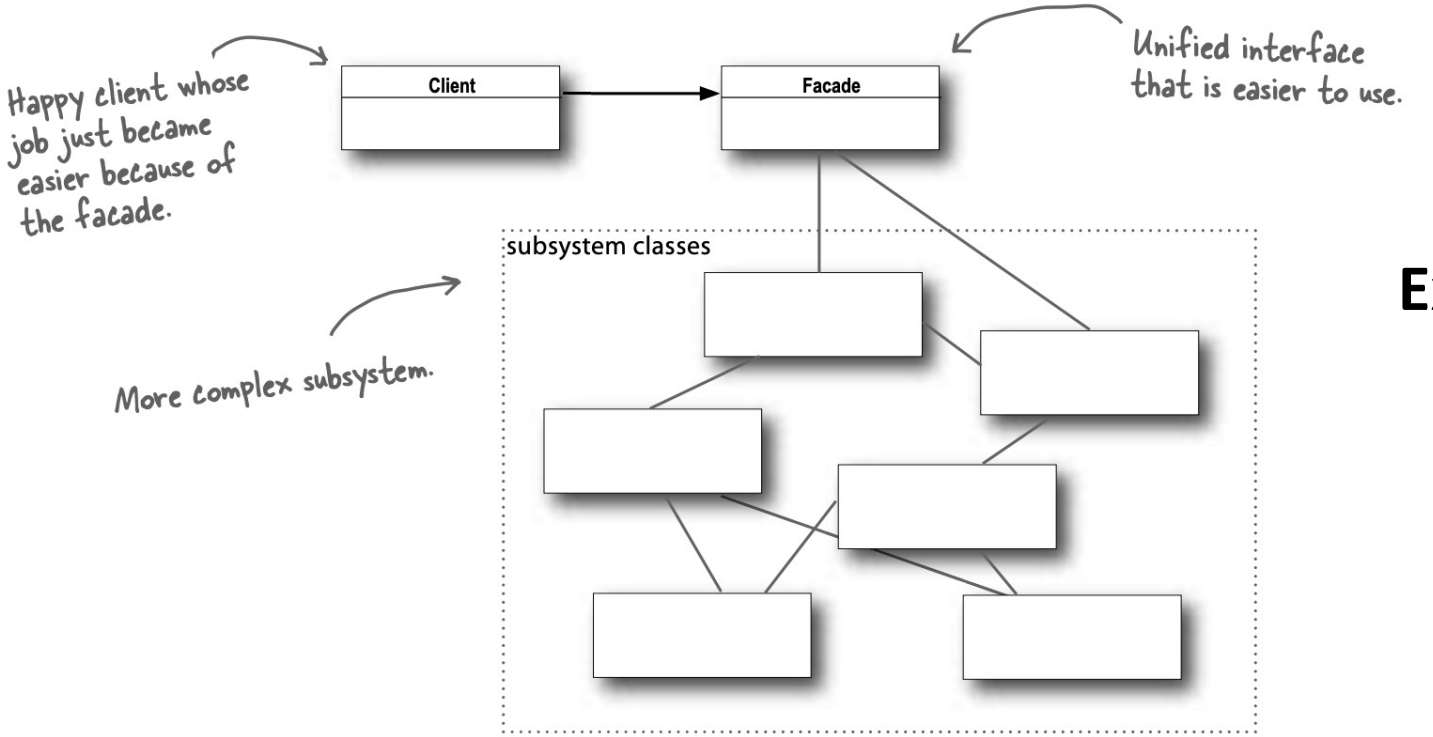
Builder Pattern

- To construct a complex object.
  - No complex constructor member / Less arguments.
- To encapsulate the construction of a product and allow it to be constructed in steps.

Facade Pattern

- Provide a simple unified interface to a set of interfaces in a subsystem that makes the subsystem easier to use.
- Straightforward pattern
- Minimize the communication and dependencies between subsystems.

Facade Pattern



Example: Compiler subsystem

- Scanner
- Parser
- ProgramNode
- BytecodeStream
- ProgramNodeBuilder

Facade Pattern

```
class Scanner {
        public:
                Scanner(istream&);
                virtual ~Scanner();
                virtual Token& Scan();
        private:
                istream& _inputStream;
};
```

```
class ProgramNodeBuilder {
        public:
                ProgramNodeBuilder();
                virtual ProgramNode* NewVariable( const char* variableName ) const;
        . . .
        private:
                ProgramNode* _node;
}
```

```
class CodeGenerator {
        public:
                virtual void Visit(StatementNode*);
                virtual void Visit(ExpressionNode*);
        protected:
                CodeGenerator(BytecodeStream&);
                BytecodeStreamk _output;
};
```

```
class ProgramNode {
        public:
                virtual void GetSourcePosition(int& line, int& index);
                virtual void Add(ProgramNode*);
                virtual void Remove(ProgramNode*);
                virtual void Traverse(CodeGenerator&);
        protected:
                ProgramNode();
};
```

```
class Parser {
        public:
                Parser();
                virtual ~Parser();
                virtual void Parse(Scanners, ProgramNodeBuilder&);
 };
```

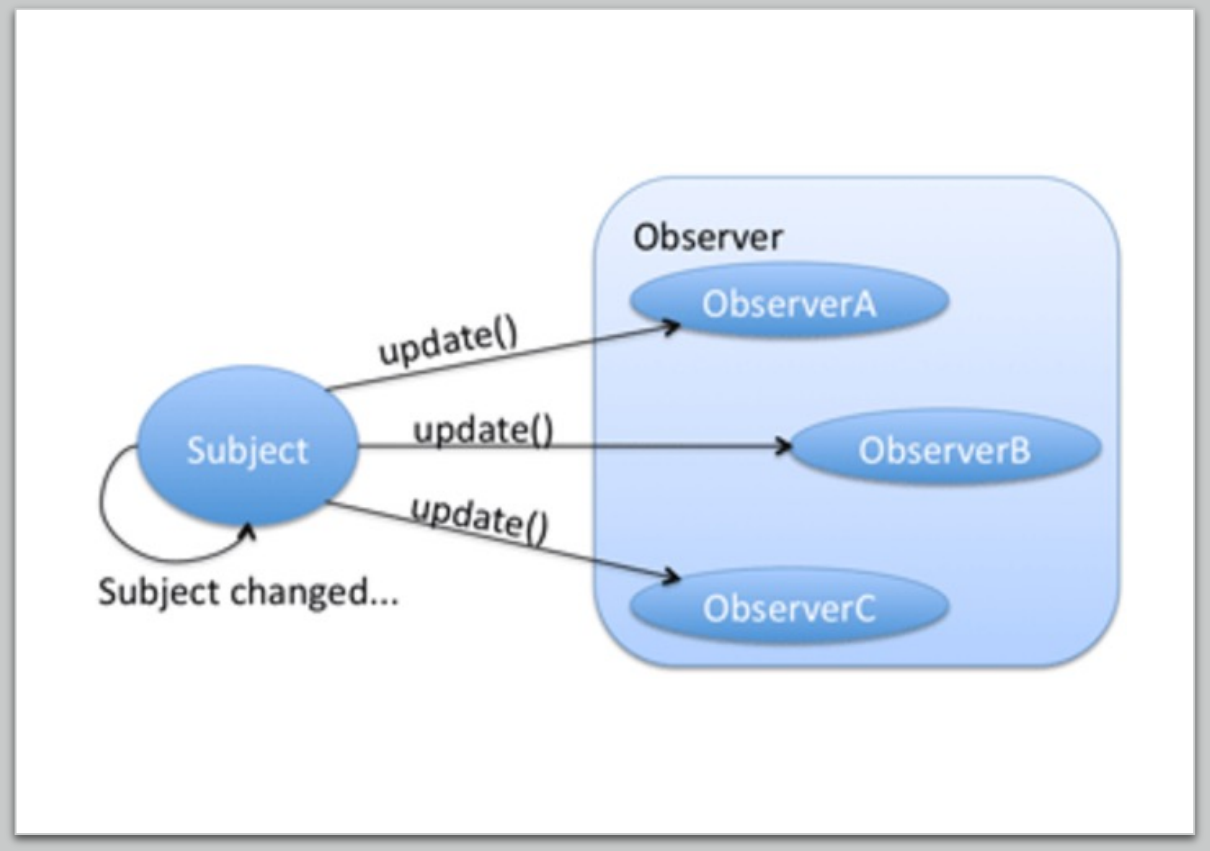
Facade Pattern

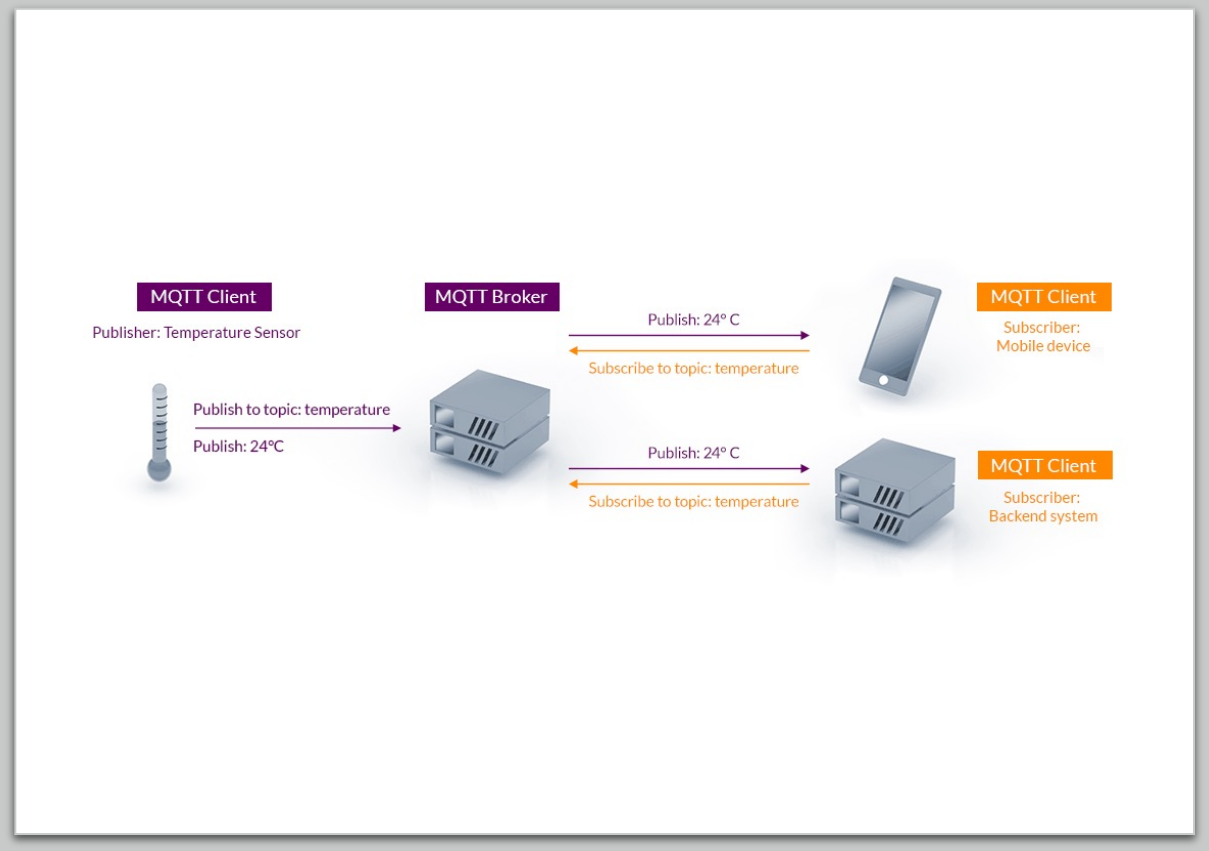
```
class Compiler {
        public:
                Compiler();
                virtual void Compile ( istream& input, BytecodeStreamk output ) {
                Scanner scanner(input);
                ProgramNodeBuilder builder;
                Parser parser;
                parser.Parse(scanner, builder);
                RISCCodeGenerator generator(output);
                ProgramNode* parseTree = builder.GetRootNode();
                parseTree->Traverse(generator);
 };
```

Observer Pattern

- **Observer Pattern**'s intent is to define a **one-to-many** dependency between objects.
- When the state of one object changes, all its dependents are notified automatically.

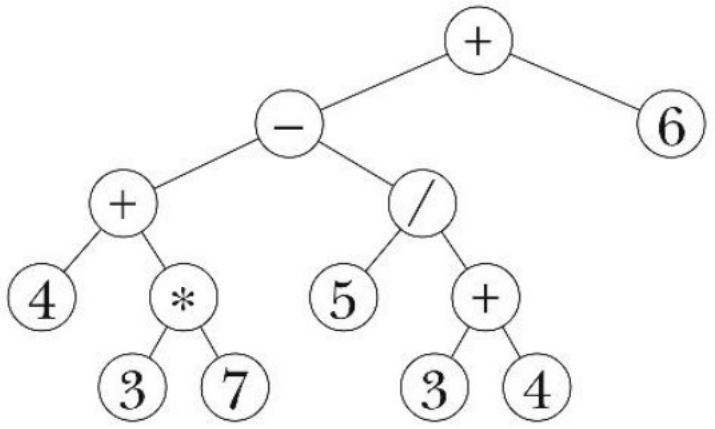
Observer Pattern





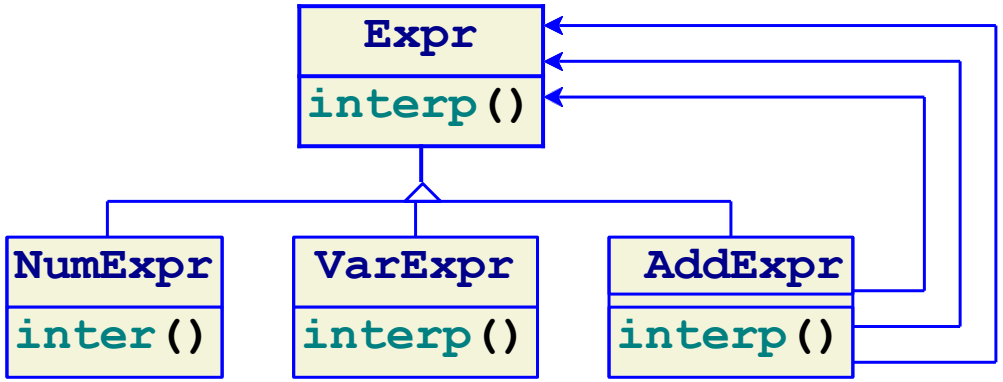
Interpreter Pattern

- Given a language, define class hierarchy for parse tree, recursive method to interpret it
- Uses Composite pattern



Interpreter Pattern

- Given a language, define class hierarchy for parse tree, recursive method to interpret it
- Uses Composite pattern



Highlights: interpret nested data recursively

MSDscript design

Advantages:

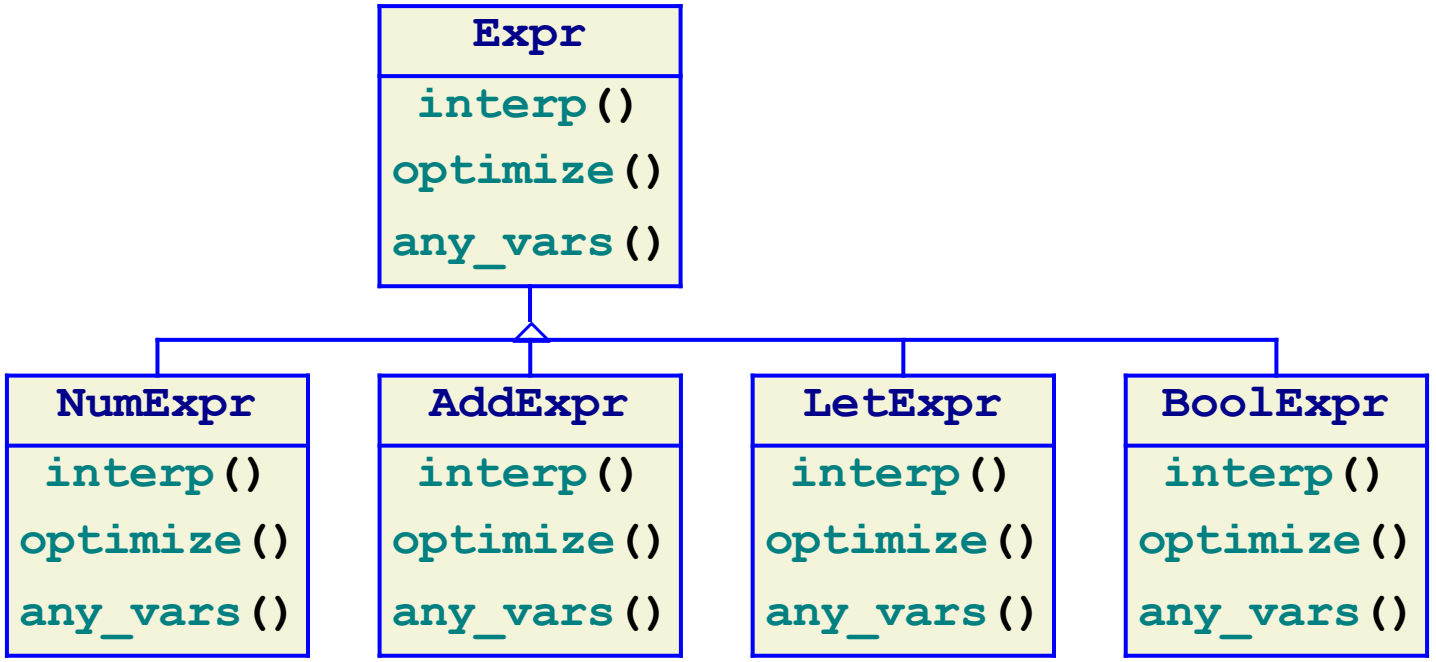
- Adding new variants is easy

Drawback

- Adding new operations/methods requires adding in all subclasses
- Can not add independent operations

Solution ?

Interpreter Pattern



- Each operation is spread across classes
- New operation means changing every class

MSDscript design

Advantages:

- Adding new variants is easy

Drawback

- Adding new operations/methods requires adding in all subclasses
- Can not add independent operations

Solution:

- Introducing new design

Visitor Pattern

- Represent an operation to be performed on elements of an object structure.
- Visitor lets you define a new operation without modifying the type hierarchy.

Visitor Pattern

- Represent an operation to be performed on elements of an object structure.
- Visitor lets you define a new operation without modifying the type hierarchy.

```
class Expr {
  virtual void* visit(ExprVisitor *visitor) = 0;
};
```

```
class ExprVisitor {
  virtual void* visit_num(int rep) = 0;
  virtual void* visit_add(Expr *lhs, Expr *rhs) = 0;
  virtual void* visit_let(string name, Expr *rhs, Expr* body) = 0;
  ....
} ;
```

Visitor Pattern

```
class Expr {
  virtual void* visit(ExprVisitor *visitor) = 0;
};
```

```
class ExprVisitor {
  virtual void* visit_num(int rep) = 0;
  virtual void* visit_add(Expr *lhs, Expr *rhs) = 0;
  virtual void* visit_let(string name, Expr *rhs, Expr* body) = 0;
  ....
} ;
```

Visitor Pattern

```
class NumExpr:Expr
  int rep;
  void* visit(ExprVisitor *visitor) {  return
    visitor->visit_num(rep);
  }
};
```

Implement `visit` once and for all

```
class AddExpr:Expr {
  Expr *lhs;  Expr
  *rhs;
  void* visit(ExprVisitor *visitor) {  return
    visitor->visit_add(lhs, rhs);
  }
};
```

Visitor Pattern

```
class InterpVisitor : public ExprVisitor {

  void* visit_num(int rep) { return new NumVal(rep); }

  void* visit_add(Expr *lhs, Expr *rhs) {
     int *lhs_val = (int *)lhs->visit(this);
     int *rhs_val = (int *)rhs->visit(this);
     return lhs_val->add_to(rhs_val);
  }

  ....
};
```

```
expr->visit(new InterpVisitor())
```

Visitor Pattern

```
class InterpVisitor : public ExprVisitor {

  void* visit_num(int rep) { return new NumVal(rep); }

  void* visit_add(Expr *lhs, Expr *rhs) {
     Val *lhs_val = (Val *)lhs->visit(this);
     Val *rhs_val = (Val *)rhs->visit(this);
     return lhs_val->add_to(rhs_val);
  }

  ....
};
```

```
expr->visit(new InterpVisitor())
```

Visitor Pattern

```
class PrintVisitor : public ExprVisitor {
  ....
};
```

```
class AnyOtherVisitor : public ExprVisitor {
  ....
};
```

Resources

- Design Patterns (**Gang of four**)
- Wikipedia page
- Pluralsight: “Design Patterns in Java”
- Head First Design Patterns

“Gang of four”