OOP (C++): Design Patterns

Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iaşi, Romania
dlucanu@info.uaic.ro

Object Oriented Programming 2020/2021

Plan

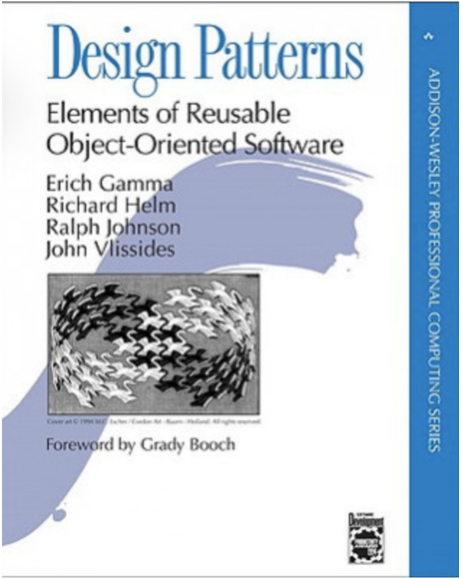
Alexander's Definition

"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"[1]

- applied first in urbanism architecture
- the first contributions in software: prima contributie in software: 1987, Kent Beck (creator of Extreme Programming) & Ward Cunningham (wrote the first wicki)

[1] C. Alexander. A Pattern Language. 1977

GoF Book



includes 23 design patterns

Full Template for a Pattern

- name and classification
- intention
- known as
- motivation
- applicability
- structure
- participants
- collaborations
- consequences
- implementation
- code
- known use cases
- related patterns

We will use a simplified template.

Classification

- creational
  used to create complex/specific objects
- structural
  used to define the structure of the classes and objects
- behavioral
  describe how the classes and their objects interact in order to distribute the responsabilities

Plan

Motivation

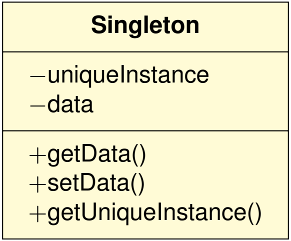
- Classification:
  creational
- Intention:
  designing a class with a single object (a single instance)
- Motivation:
  in an operating system:
  - there is a file system
  - there is only one window manager

  in a website: there is only one web page manager
- Application:
  when there must be exactly one instance
  class clients must have access to the instance from any well-defined point

Consequences

- controlled access to the single instance
- namespace reduction (global variable elimination)
- allows refinement of operations and representation
- allows a fixed number of instants (Doubleton, Tripleton, ...)
- more flexible than class-level operations (static functions)

Structure


Singleton
−uniqueInstance
−data
+getData()
+setData()
+getUniqueInstance()

Impementation (version 1, $\geq$ C++2011)

```
template <typename Data>
class Singleton {
public:
  static Singleton<Data>& getUniqueInstance() {
    return uniqueInstance;
  }
  Data getData();
  void setData(Data x);
  void operator=(Singleton&) = delete;
  Singleton(const Singleton&) = delete;
protected:
  Data data; // object state
  Singleton() { }
private:
  static Singleton<Data> uniqueInstance;
};
...
template <typename Data>
Singleton<Data> Singleton<Data>::uniqueInstance;
```

Testing

```
class SingletonTest : public testing::Test {
protected:
    virtual void SetUp() {
        s2.setValue(9);
        // s3 = s1;  // compiling error
        s3.setValue(77);
    }
    // Declares the variables your tests want to use.
    Singleton &s1 = Singleton::instance();
    Singleton &s2 = Singleton::instance();
    Singleton &s3 = s2;
    // Singleton s4 = s2;  // compiling error
};
// Tests the uniqueness.
TEST_F(SingletonTest, Uniqueness) {
    ASSERT_EQ(s1.getValue(), s2.getValue());
    ASSERT_EQ(s1.getValue(), s3.getValue());
}
```

What about the `move` constructor/assignment-operator?

From the manual (12.8):
"10 If the definition of a class X does not explicitly declare a move constructor, one will be implicitly declared as defaulted if and only if
– X does not have a user-declared copy constructor,
– X does not have a user-declared copy assignment operator,
– X does not have a user-declared move assignment operator,
– X does not have a user-declared destructor, and
– the move constructor would not be implicitly defined as deleted."

Does it make sense to declare a `move` constructor/assignment-operator?

Impementation (version 2)

```
template <typename Data>
class Singleton {
public:
  static Singleton* getUniqueInstance() {
    if (uniqueInstance == 0) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
  ...
protected:
  Data data;
  Singleton()  { }
private:
  static Singleton<Data>* uniqueInstance;
};
```

Testing

```
class SingletonTest : public testing::Test {
protected:
    virtual void SetUp() {
        s1 = Singleton::instance();
        s1->setValue(47);
        s2 = Singleton::instance();
        s2->setValue(9);
        s3 = s1;
        *s3 = *s2;
        s3->setValue(75);
    }
    // Declares the variables your tests want to use.
    Singleton *s1;
    Singleton *s2;
    Singleton *s3;
};

// Tests the uniqueness.
TEST_F(SingletonTest, Uniqueness) {
    ASSERT_EQ(s1->getValue(), s2->getValue());
    ASSERT_EQ(s1->getValue(), s3->getValue());
}
```

Testing

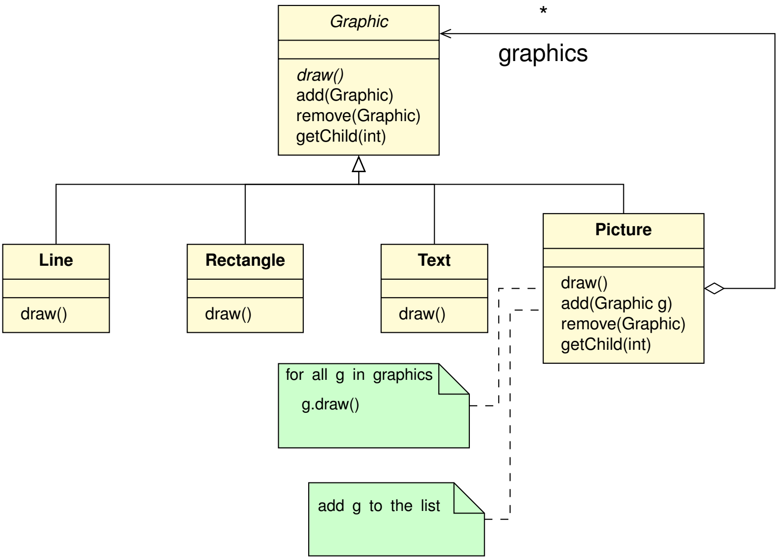
```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from SingletonTest
[ RUN      ] SingletonTest.Uniqueness
[       OK ] SingletonTest.Uniqueness (0 ms)
[----------] 1 test from SingletonTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.

Process finished with exit code 0
```

Plan

Intention

- it is a structural pattern
- composes objects in a tree structure to represent a part-whole hierarchy
- let the clients (of the structure) treat the individual and compound objects in a uniform way

Motivation

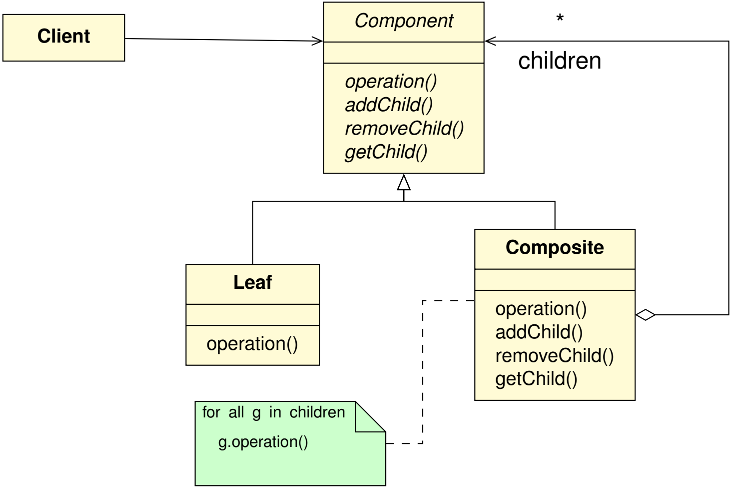


*
graphics
Graphic
draw()
add(Graphic)
remove(Graphic)
getChild(int)
Line
draw()
Rectangle
draw()
Text
draw()
Picture
draw()
add(Graphic g)
remove(Graphic)
getChild(int)
for all g in graphics
g.draw()
add g to the list

It is a Recursive Definition

- any (object) line is a graphic object
- any (object) rectangle is a graphic object
- any text (object) is a graphic object
- a picture made up of several graphic objects is a graphic object

Structure



Client
Component
operation()
addChild()
removeChild()
getChild()
*
children
Leaf
operation()
Composite
operation()
addChild()
removeChild()
getChild()
for all g in children
g.operation()

Participants 1/2

- Component (e.g., Graphic)
  - declares the interface for the objects in the composition
  - implements the default behavior for the common interface of all classes
  - declares an interface for accessing and managing child components
  - (optional) defines an interface for accessing parent components in the recursive structure
- Leaf (e.g., Rectangle, Line, Text, etc.)
  - represents primitive objects
  - a leaf has no children
  - defines the behavior of primitive objects

Participants 2/2

- Composite (e.g., Picture)
  - defines the behavior of components with children
  - memorizes child components
  - implements operations related to copies of the Component interface
- Client
  - handles the objects in the composition through the Component interface

Collaborations

- clients use the Component interface class to interact with objects in the structure
- if the container is a Leaf instance, then the request is resolved directly
- if the container is a Composite instance, then the request is forwarded to the child components; other additional operations are possible before or after forwarding

Consequences 1/2

- defines a hierarchy of classes consisting of primitive and compound objects
- primitive objects can be composed of more complex objects, which in turn can be composed of other more complex objects, etc. (recursion)
- whenever a client expects a primitive object, he can also take a composite object
- for the client it is very simple; it treats primitive and composite objects uniformly
- the client does not care if it has to do with a primitive or composite object (avoiding the use of switch-case structures)

Consequences 2/2

- it is easy to add new types of Leaf or Composite components; the new subclasses work automatically with the existing structure and the customer code. The customer does not change anything.
- makes the design very general
- drawback: it is difficult to restrict which components can appear in a composite object (a solution could be to check during execution)

Implementation: Decisions to Make

- explicit references to parents?
- shared components?
- maximize the interface? safety or transparence?
  Transparence could lead to violation of the SRP!
  Safety requires to convert a Component into a Composite!
- where to implement the operations handling children?

Plan

Expressions in Programming Languages

- arithmetic expressions: $a + b * 2 - c$
- relational expressions: $a + 2 < b * 3$
- Boolean expressions: $a < 3 \;\&\&\; (b < 0 \;||\; a < b)$

Arithmetic Expressions: Syntax

```
PrimaryExpression ::=
      IntConstant
    | VarName
    | "(" Expression ")"

IntConstant ::=
      Digit+

Digit ::=
     "0" | "1" | ... | "9"

VarName ::=
     "a" | "b" | ... | "z"

MultExpression ::=
       PrimaryExpression (("*" | "/" | "%") PrimaryExpression)*

ArithExpression ::=
       MultExpression (("+" | "-") MultExpression)*

Expression ::= ArithExpression
```

AST Classes

IntConstant **VarName** **PrimaryExpression**

MultExpression **ArithExpression** **Expression**

Relationship between Classes 1/3

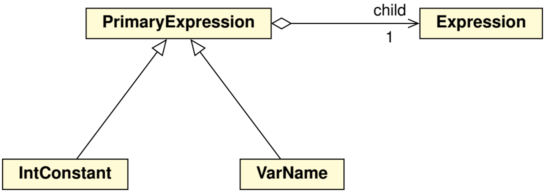
```
PrimaryExpression ::=
        IntConstant
      | VarName
      | "(" Expression ")"
```

$=$

```
PrimaryExpression ::=
        IntConstant
      | VarName
```

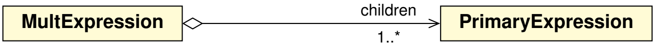
$+$

```
PrimaryExpression ::=
        "(" Expression ")"
```

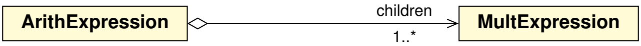


Relationship between Classes 2/3

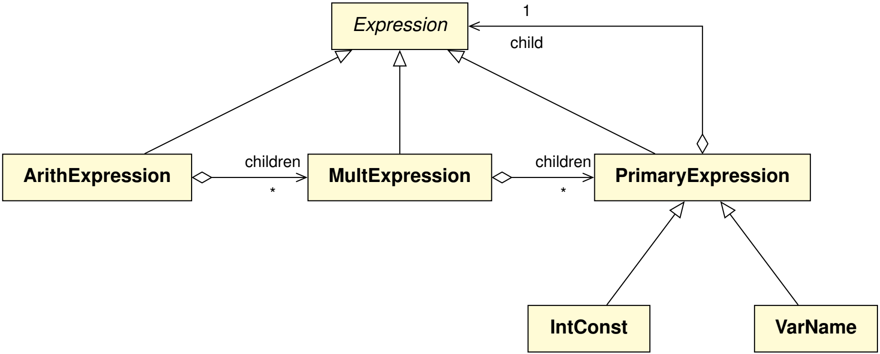
```
MultExpression ::=
         PrimaryExpression (("*" | "/" | "%") PrimaryExpression)*
```




```
ArithExpression ::=
         MultExpression (("+" | "-") MultExpression)*
```



Relationship between Classes 3/3



Ugly!

Question. What OO design principles are violated?

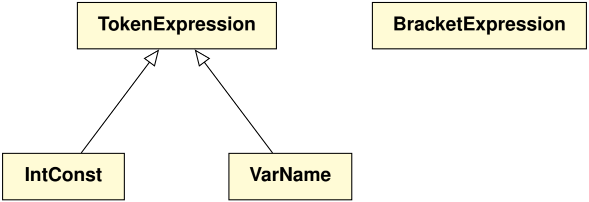
Using Composite 1/2

Leafs: IntConst, VarName

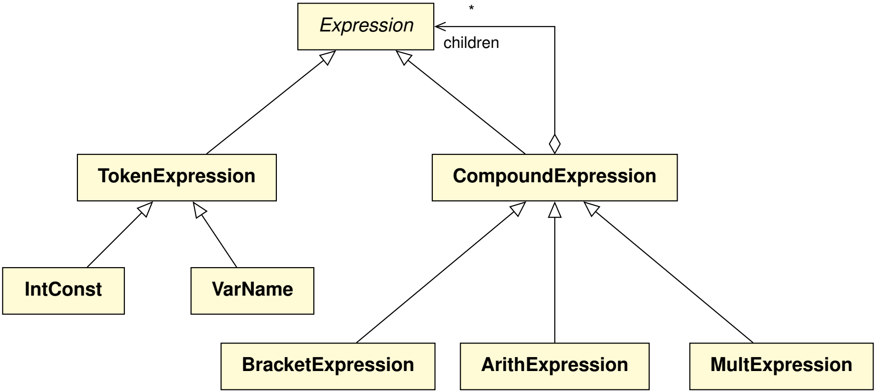
Composites: ArithExpression, MultExpression, Expression

What about PrimaryExpression? It is both

It is both, therefore we split it:



Using Composite 2/2



Class Expression in C++

```
class Expression {
 public: virtual list<string> getLabel();
 public: virtual void addLabel(string str);
 public: virtual list<string> getLabel();
 public: virtual string toString();
 protected: list<string> label;
};
```

- we opted for safety

Class CompoundExpression in C++

```
class CompoundExpression : public Expression {
 public: void addChild(Expression* pe);
 public: string toString();
 public: list<Expression*> getChildren();
 protected: list<Expression*> children;
};

class ArithExpression : public CompoundExpression {
};

class MultExpression : public CompoundExpression {
};
...
```

Expressions Parser

See the appendix.

The implementation can be found in the folder
examples/interpreter/cpp/expressions/parser-composite

Plan

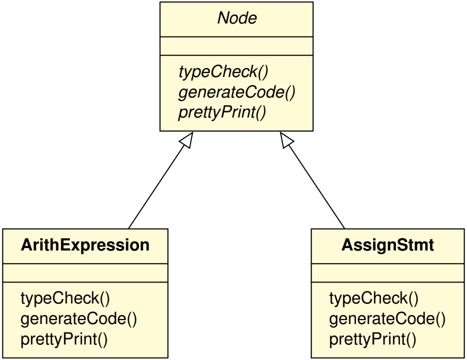
Intention

- it is a behavioral pattern
- models an operation (a set of operations) that runs over the elements of an object structure
- allows the definition of new operations without changing the classes of the elements over which the operations are executed

Motivation

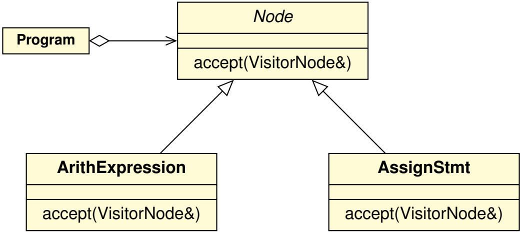
- A compiler is a program like an abstract syntactic tree (AST). This syntactic tree is used both for static semantics (e.g., type checking) and for code generation, code optimization, display.
- These operations differ from one type of instruction to another. For example, a node representing an assignment differs from a node representing an expression and consequently the operations on them will be different.
- These operations should be performed without changing the structure of the AST.
- Even if the structure of the AST differs from one language to another, the ways in which the operations are performed are similar.

Motivation: Polluting Solution

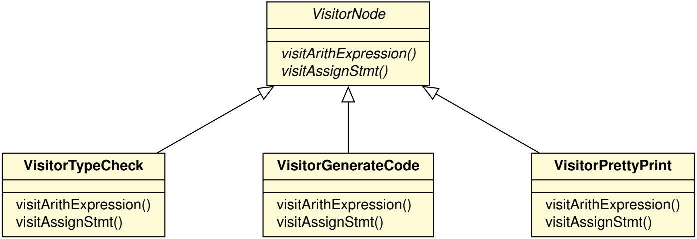


Motivation: Solution with Visitors

Hierarchy:

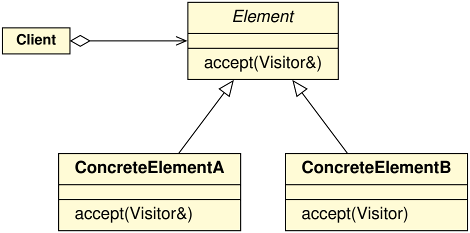


Visitors:

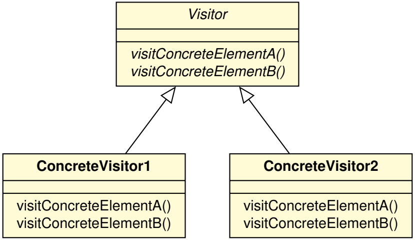


Structure

Hierarchy:



Visitors:



Participants 1/3

- Visitor (e.g., NodeVisitor)
  - declares a visit operation for each ConcreteElement class in the structure
  - the name of the operation and the signature identify the class that sends the visit request to the visitor
  - this allows the visitor to identify the specific element they are visiting
  - then, the visitor can visit the item through its interface

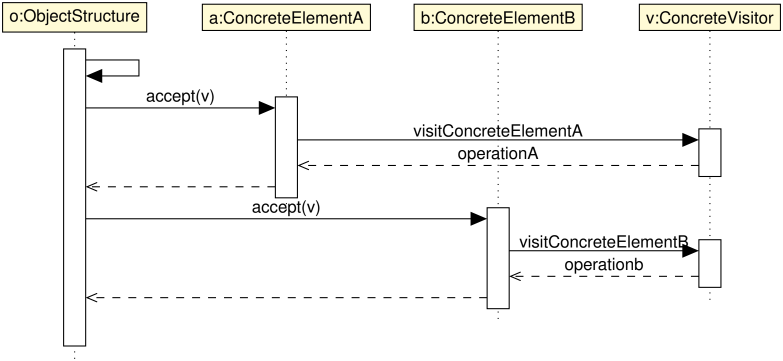
Participants 2/3

- ConcreteVisitor (e.g., TypeCheckingVisitor)
  - implements each operation declared by the visitor
  - each operation implements a fragment of the visit algorithm that corresponds to the element in the structure visited
  - it memorizes the state of the visiting algorithm, which often accumulates the results obtained while visiting the elements in the structure

Participants 3/3

- Element (Node)
  - defines accepting operations, which have a visitor as an argument
- ConcreteElement (e.g., AssignmentNode,VariableRefNode)
  - implements accepting operations
- ObjectStructure (e.g., Program)
  - it can list its elements
  - it can provide a high-level interface for a visitor visiting its elements
  - it can be a "composite"

Collaboration (Sequence Diagram)



Explanation

- after some internal computations, o sends the message accept(v) to a (in C++ this means that o calls a.accept(v))
- then a sends the message visitConcreteElementA to v (i.e., a calls v.visitConcreteElementA(this), a kind of " v, please visit me")
- then v "visits" a by executing a.operationA()
- then a similar scenario with o and b and v

Consequences 1/2

- Visitor makes adding new operations easy
- a visitor gathers the related operations and separates the unrelated ones
- adding new ConcreteElement classes to the structure is difficult
  - causes changes in the interfaces of all visitors
  - sometimes a default implementation in the Visitor abstract class can make the job easier

Consequences 2/2

- unlike iterators, a visitor can traverse multiple class hierarchies
- allows the calculation of cumulative states. Otherwise, the cumulative state must be transmitted as a parameter
- it could destroy the encapsulation
    - the concrete elements must have a strong interface capable of providing all the information requested by the visitor

Implementation 1/2

```
class Visitor {
public:
  virtual void visitElementA(ElementA*);
  virtual void visitElementB(ElementB*);
  // and so on for other concrete elements
protected:
  Visitor();
};
class Element {
public:
  virtual ~Element();
  virtual void accept(Visitor&) = 0;
protected:
   Element();
};
```

Implementation 2/2

```
class ElementA : public Element {
public:
 ElementA();
 virtual void accept(Visitor& v) {
   v.visitElementA(this);
 }
};
class ElementB : public Element {
public:
  ElementB();
  virtual void accept(Visitor& v) {
   v.visitElementB(this);
  }
};
```

Simple/Double Dispatch

- Simple dispatch. The operation that makes a request depends on two criteria: the name of the request and the type of receiver. For example, `generateCode()` depends on the type of node.
- Double dispatch. The operation that performs the request depends on the types of two receivers. For example, an `accept()` call depends on both the component and the visitor.

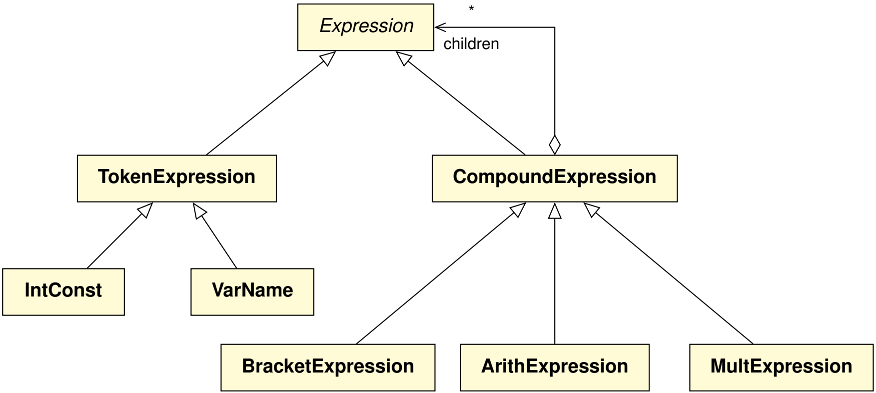
Who Traverses Object Structure?

There are several options:

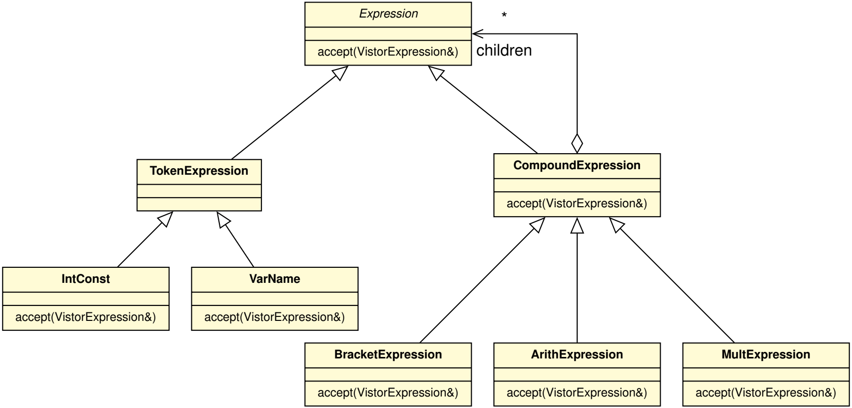
- the object structure itself
- the visitor
- an iterator

Plan

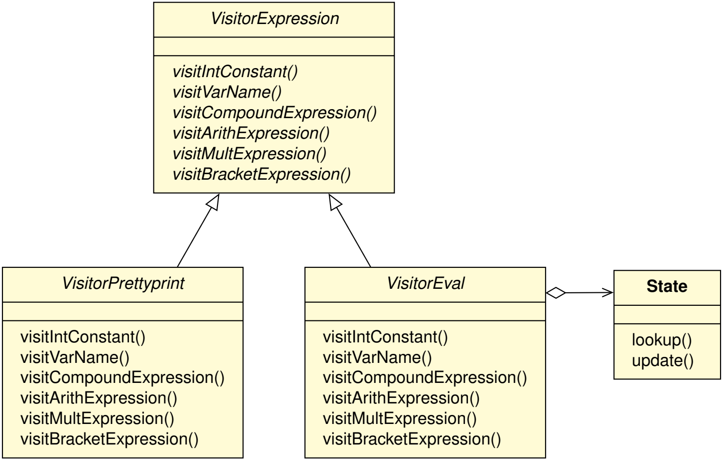
Recall Composite Diagram for Expression



Adding `accept()`



Visitors for Expression



Implementation 1/2

The implementation can be found in the folder
examples/interpreter/cpp/expressions//visitor

main.cpp:

```
std:: cout << "Input: ";
std::cin.getline(str, 80);
Parser p(str);
std::optional<Expression*> ae = p.expression();
if (ae.has_value()) std::cout << ae.value()->toString() << "\n";
else std::cout << "nothing\n";
State st;
VarName a("a"), b("b");
st.update(a, 10);
st.update(b, 5);
st.print();
VisitorEval visitorEval1(st);
if (ae.has_value()) {
      ae.value()->accept(visitorEval1);
      cout << "ae = " << visitorEval1.getCumulateVal() << endl;
}
```

Implementation 2/2

Running:

```
$ g++ *.cpp ../*.cpp -std=c++17 -o demo.exe
$ ./demo.exe
Input: a+b-2
(a + b - 2)
a |-> 10
b |-> 5
ae = 13
```

Plan

Intention

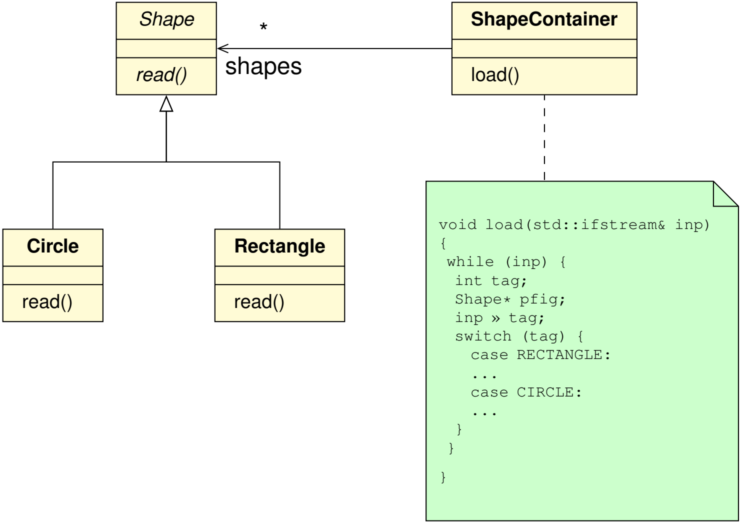
- it is a creational pattern
- to provide an interface for creating a family of intercorrelated or dependent objects without specifying their specific class

Applicabilty

- a system should be independent of how the products are created, composed or represented
- a system would be configured with multiple product families
- a family of intercorrelated objects is designed so that the objects can be used together
- you want to provide a product library and you want only the interface to be accessible, not the implementation

Motivation1/2

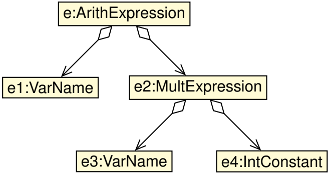
Recall the counter-example from OCP:



If we add more shapes, we have to modify `ShapeContainer::load()`.

Motivation 2/2: (De)Serialization of Expressions

$a + b * 2$



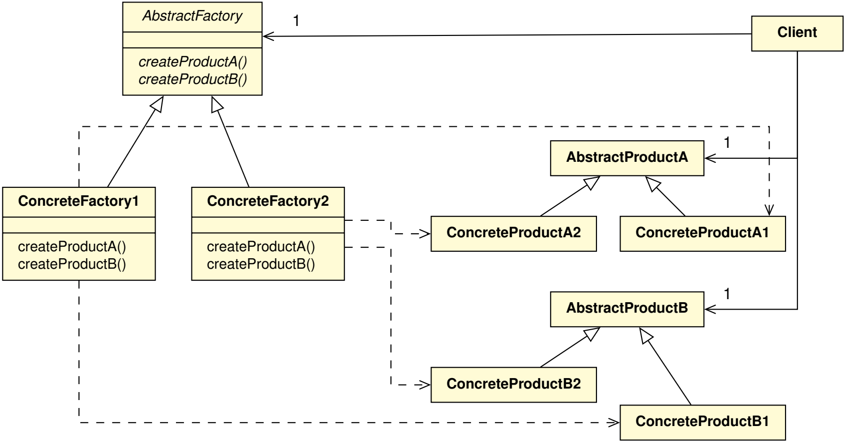
Serialization

```
<arith>
  <label> [+] </label>
  <varName> a </varName>
  <mult>
    <label> [*] </label>
    <varName> b </varName>
    <intConstant> 2 </intConstant>
  </mult>
</arith>
```

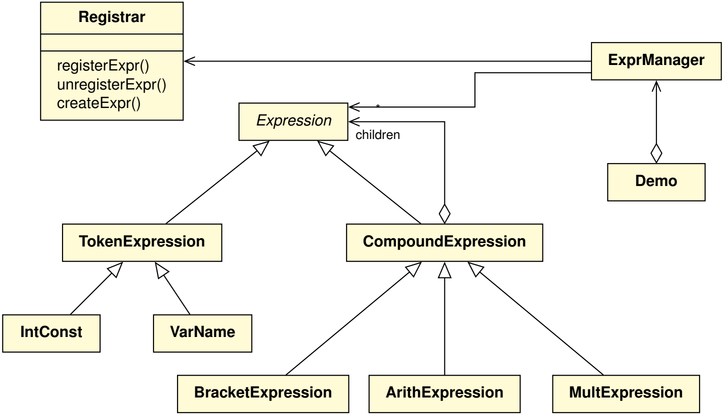
XML notation

Deserialization

Structure


AbstractFactory
createProductA()
createProductB()
1
Client
1
AbstractProductA
ConcreteFactory1
createProductA()
createProductB()
ConcreteFactory2
createProductA()
createProductB()
ConcreteProductA2
ConcreteProductA1
1
AbstractProductB
ConcreteProductB2
ConcreteProductB1

Expression Factory



Correspondens with the Standard Structure

- AbstractProductA = Expression
- AbstractProductB = Statements (not implemented yet)
- ConcreteFactory = Registrar (of expressions and statements)
- Client = ExprManager (in charge with (de)serialization)

Consequences

- isolates concrete classes
- simplifies the exchange of the product family
- promotes consistency among products
- supports new product families easily
- respects the open / closed principle

Participants: Registrar

- it is a class that manages the types of expressions
- registers a new type of expression (called whenever a new derived class is defined)
- delete a registered expression type (delete a derived class)
- creates expression objects
  - at the implementation level we use pairs (tag, createExprFn)
  - ... and callback functions (see next slide)
- Singleton template can be used to have a single factory (register)

Callback Functions

- a callback function is a function that is not explicitly invoked
- the responsibility for the call is delegated to another function that receives as parameter the address of the callback function
- the object factory uses callback functions to create objects: for each type there is a callback function that creates objects of that type
- for the "expression factory" we declare an alias for the type of Expression object creation functions:

```
typedef Expression* ( *CreateExprFn )();
```

Implementation: Registrar 1/2

```
class Registrar
{
 public:
  bool registerExpr(string tag,
                    CreateExprFn createExprFn )
  {
    return catalog.insert(
              std::pair<string, CreateExprFn>(
                            tag, createExprFn)
                         ).second;
  }
};
```

Implementation: Registrar 2/2

```
  void unregisterExpr(string tag)
  {
    catalog.erase(tag);
  }

  Expression* createExpr(string tag)
  {
    map<string, CreateExprFn>::iterator i;
    i = catalog.find(tag);
    if ( i == catalog.end() )
      throw string("Unknown expression tag");
    return (i->second)();
  }
protected:
  map<string, CreateExprFn> catalog;
};
```

Implementation: Object Creation

```
Expression* createIntConstant() {
  return new IntConstant();
}

Expression* createVarName() {
  return new VarName();
}

Expression* createMultExpression() {
  return new MultExpression();
}
...
```

Implementation: ExprManager Constructor

```
ExprManager::ExprManager() {
    reg = new Registrar();
    reg->registerExpr("<intConstant>",
                      createIntConstant);
    reg->registerExpr("<varName>",
                      createVarName);
    reg->registerExpr("<mult>",
                      createMultExpression);
    reg->registerExpr("<arith>",
                      createArithExpression);
}
```

Full Implementation

The full implementation can be found in the folder
examples/interpreter/cpp/expressions//factory
Running:

```
$ g++ *.cpp -std=c++17 -o demo.exe
$ ./demo.exe
Input: a+b-2
```

`test.xml` file created:

```
<arith>
  <label> [+,-] </label>
  <mult>
    <label> [] </label>
    <varName> a </varName>
  </mult>
  <mult>
    <label> [] </label>
    <varName> b </varName>
  </mult>
  <mult>
    <label> [] </label>
    <intConstant> 2 </intConstant>
  </mult>
</arith>
```

Conclusion

- design patterns are a way to learn how to build your programs
- a design pattern is a tip that comes from people who have distilled their most common solutions into simple, digestible and suggestive advices
- OOP and design patterns are distinct topics
- OOP teaches you how to program, is it a programming methodology or a programming concept
- design patterns teach you
  - how to think about programs,
  - suggest methods for building classes / objects to solve a certain scenario in a program,
  - proven methods to succeed
- many other useful patterns are found in GoF