Outline

- Facade Pattern
- Chain of Responsibility
- Interpreter pattern

Facade Pattern

- Motivation
  - Most users want a subsystem to have a simplified interface.
  - Whereas other users want direct access to all components of a subsystem.
- An example
  - A compiler system

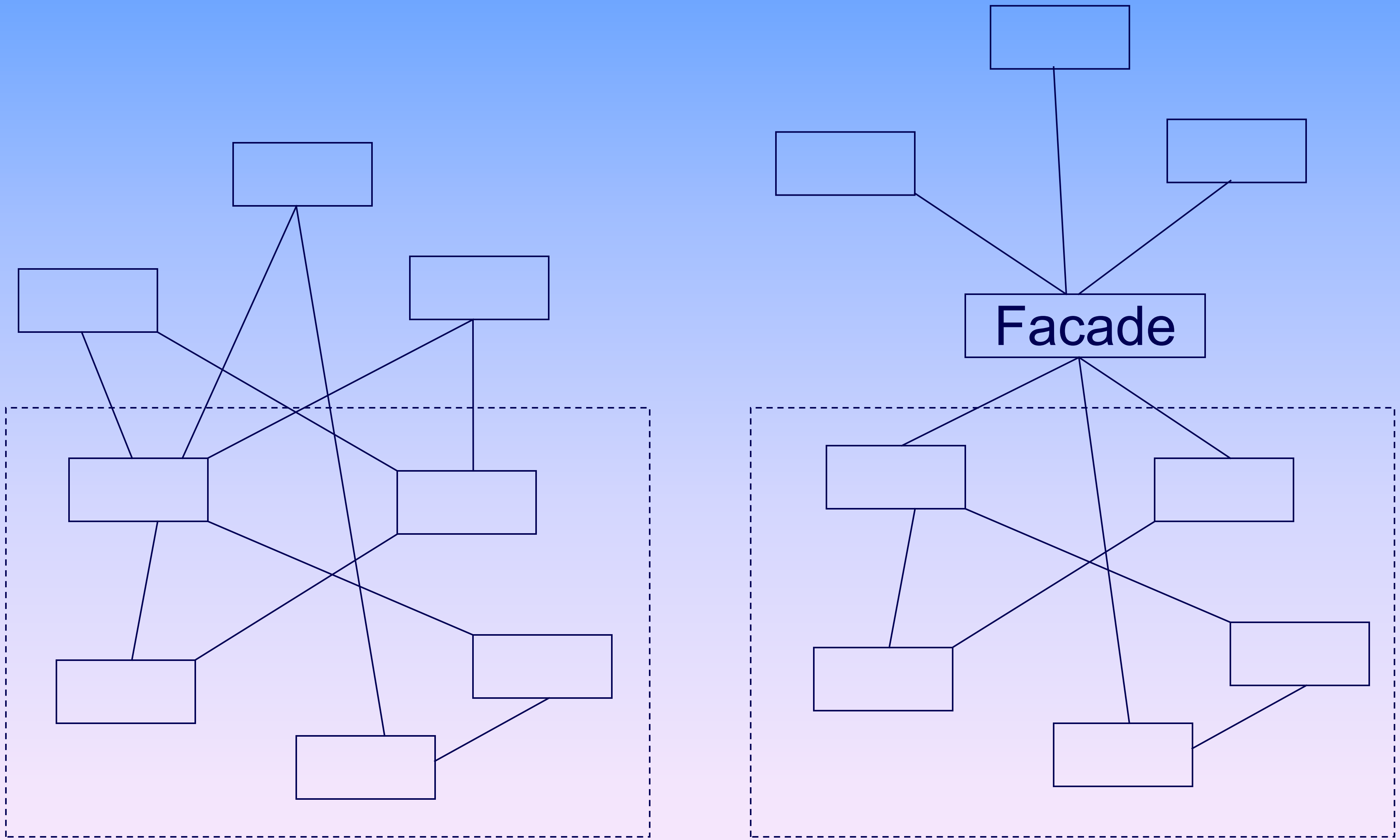
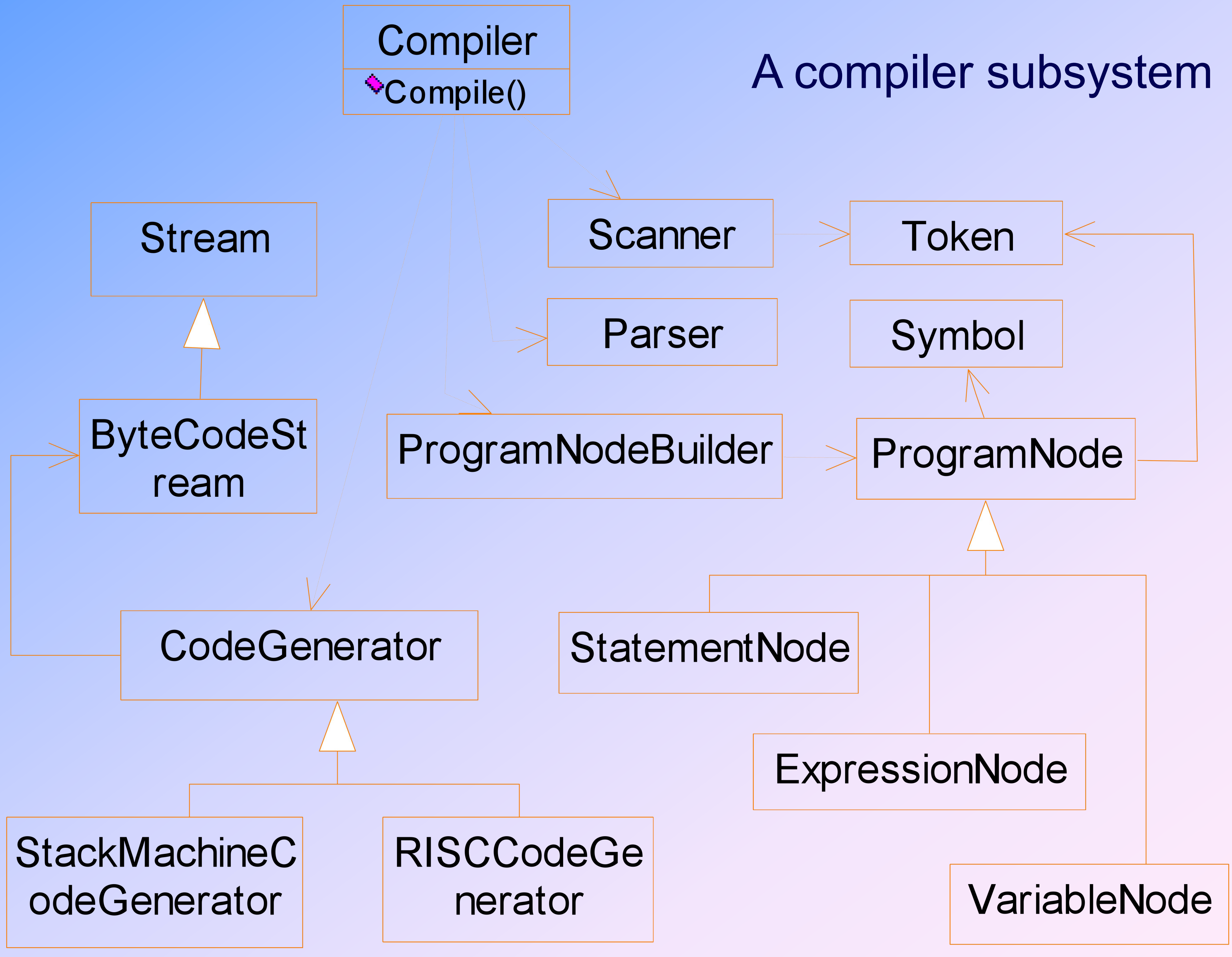


Illustration of a facade pattern

A compiler subsystem

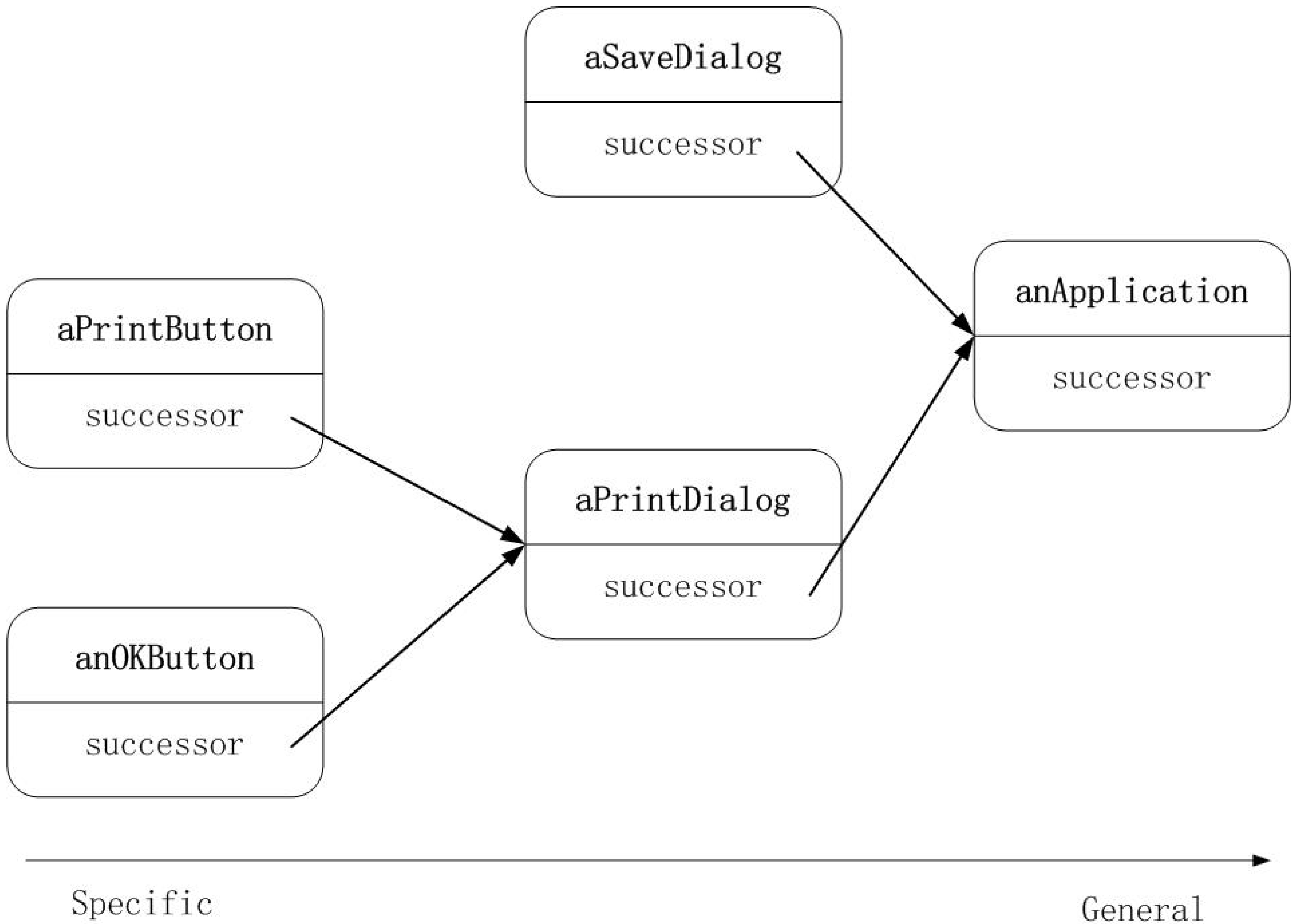


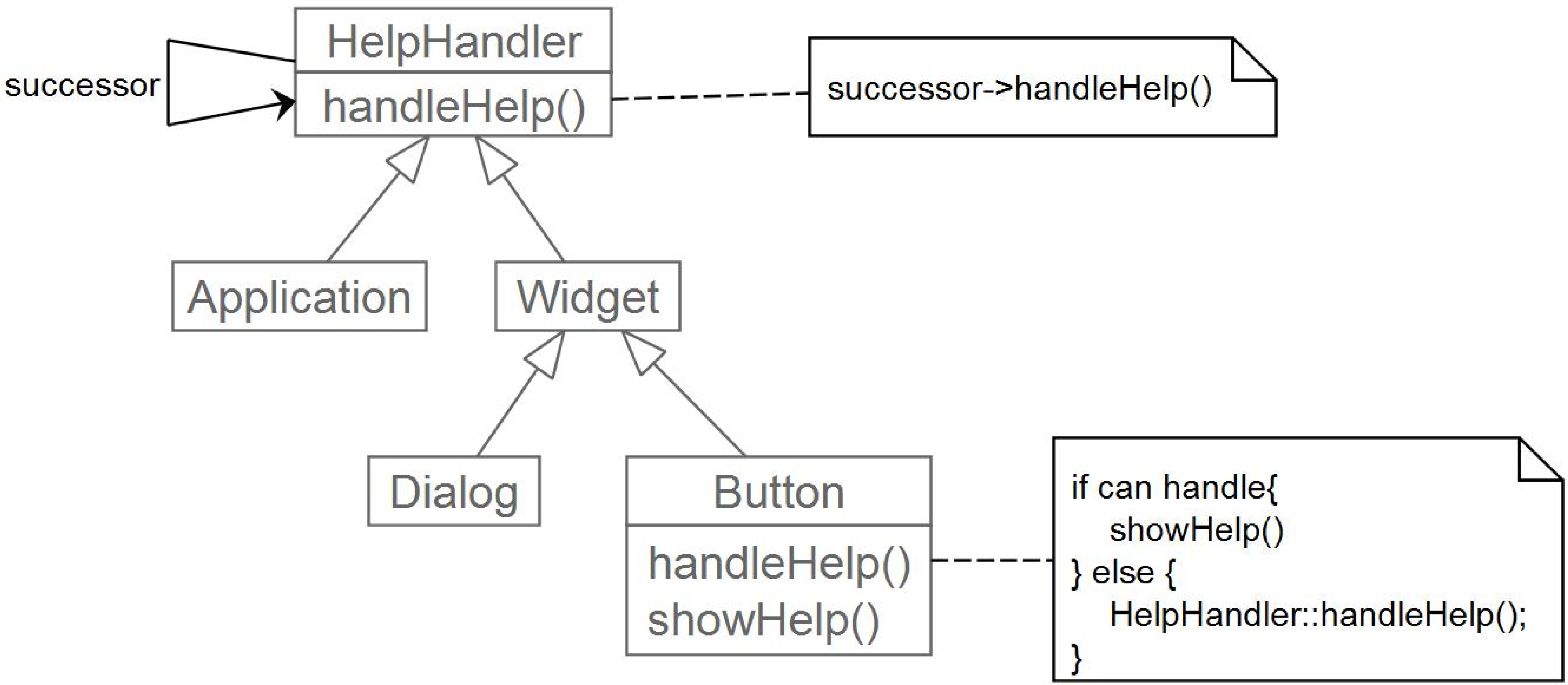
Applicability

- “Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for the clients that do not need to customize it.”
- A facade can provide a simple default interface.

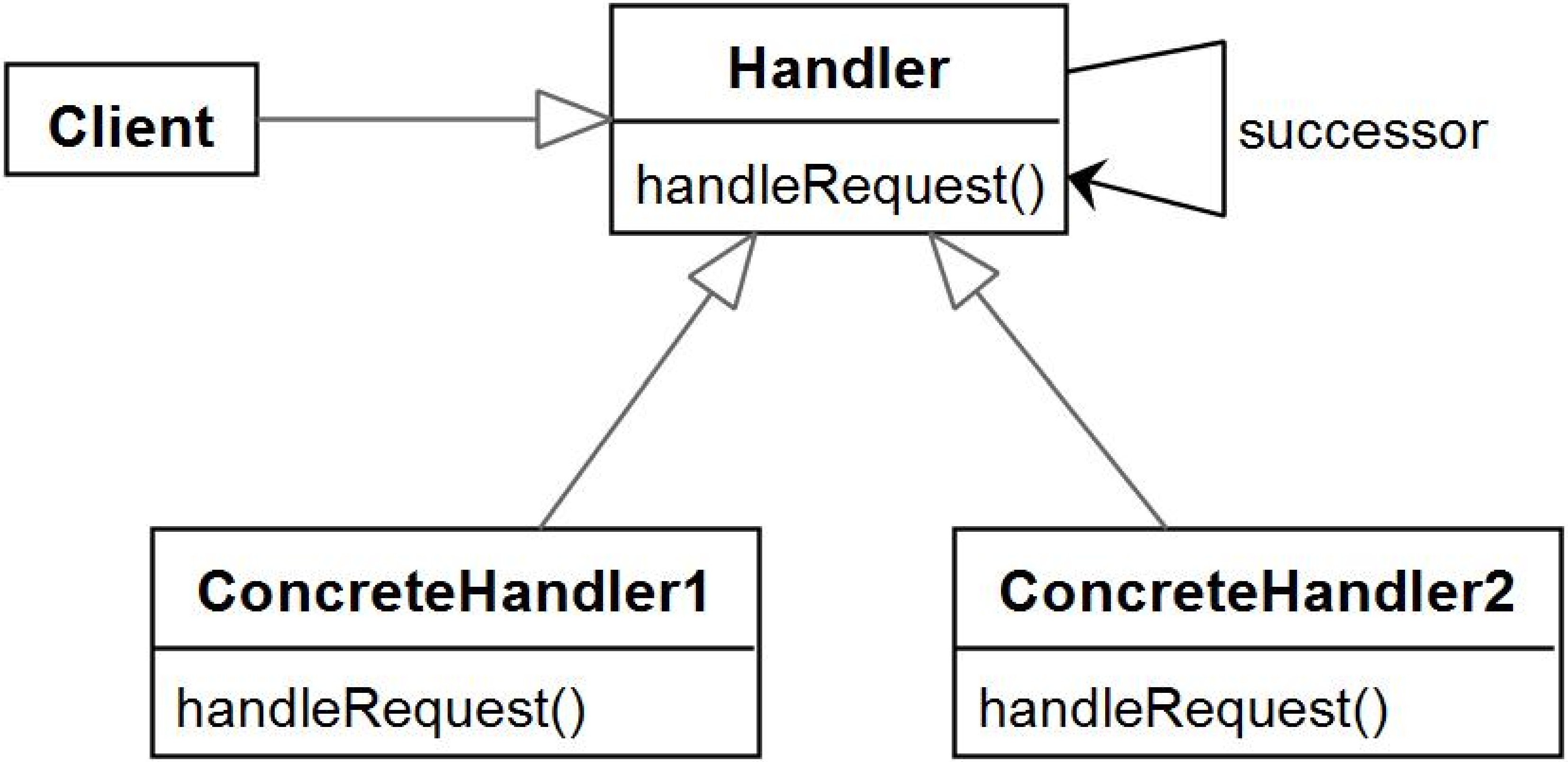
Chain of responsibility

- Motivation
  - Context sensitive help
  - Specific help ➔ general help ➔ even general help
  - The sender does not know explicitly who will response to the request
- Illustration of the solution


aSaveDialog
successor
aPrintButton
successor
anApplication
successor
aPrintDialog
successor
anOKButton
successor
Specific
General



Solution to the example



general structure of the chain of responsibility pattern

Consequence

- Reduced coupling
  - The sender just simply knows its request will be processed appropriately.
- Responsibility of the objects can be changed dynamically

Implementation

- Implementation of the chain
  - Define new links
  - Using existing ones
    - The parent pointers in the composite pattern.
- Representation of the request
  - Just one kind of request
  - Request code (an integer or a string)
    - Parameters must be packed and unpacked
  - Request objects

```
void Handler::handleRequest (Request * theRequest ) {
    switch (  theRequest->getKind( )   ) {
    case Help:
        // cast argument to appropriate type
        handleHelp(   (HelpRequest * ) theRequest   );
        break;

    case Keyboard_Message:
        handlePrint(   (Keyboard_Message *) theRequest    );
        break;

    default:
        // ...
        break;
    }
```

Code segment to illustrate the use of request objects 1/1

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1

class HelpHandler {
public:
    HelpHandler ( HelpHandler* successor = 0,
                        Topic= NO_HELP_TOPIC );
    virtual bool hasHelp( );
    virtual void setHandler(
            HelpHandler * successor, Topic );
    virtual void handleHelp( );

private:
    HelpHandler * _successor;
    Topic _topic;
};
```

```
HelpHandler::HelpHandler (
                    HelpHandler *successor , Topic t):
  _successor(successor), _topic(t) { }

bool HelpHandler::hasHelp() {
    return _topic != NO_HELP_TOPIC;
}

void HelpHandler::handleHelp() {
    if ( _successor != 0 ) {
        _successor->handleHelp();
    }
}
```

```
class Widget: public HelpHandler {
protected:
    Widget( Widget * parent_,
                Topic t = NO_HELP_TOPIC );
private:
    Widget* parent;
}
Widget::Widget ( Widget * parent_, Topic t)
     :HelpHandler( parent_, t )
{
     parent  = parent_;
}

class Button: public Widget {
public:
    Button (Widget* parent_,
                Topic t= NO_HELP_TOPIC );
    virtual void handleHelp();
}
```

```
Button::Button(Widget * parent_, Topic t)
        :Widget (parent_, t ) { }

void Button::handleHelp() {
    if ( hasHelp() ) {
        // Offer help on the button
    } else {
        HelpHandler::handleHelp();
    }
}
```

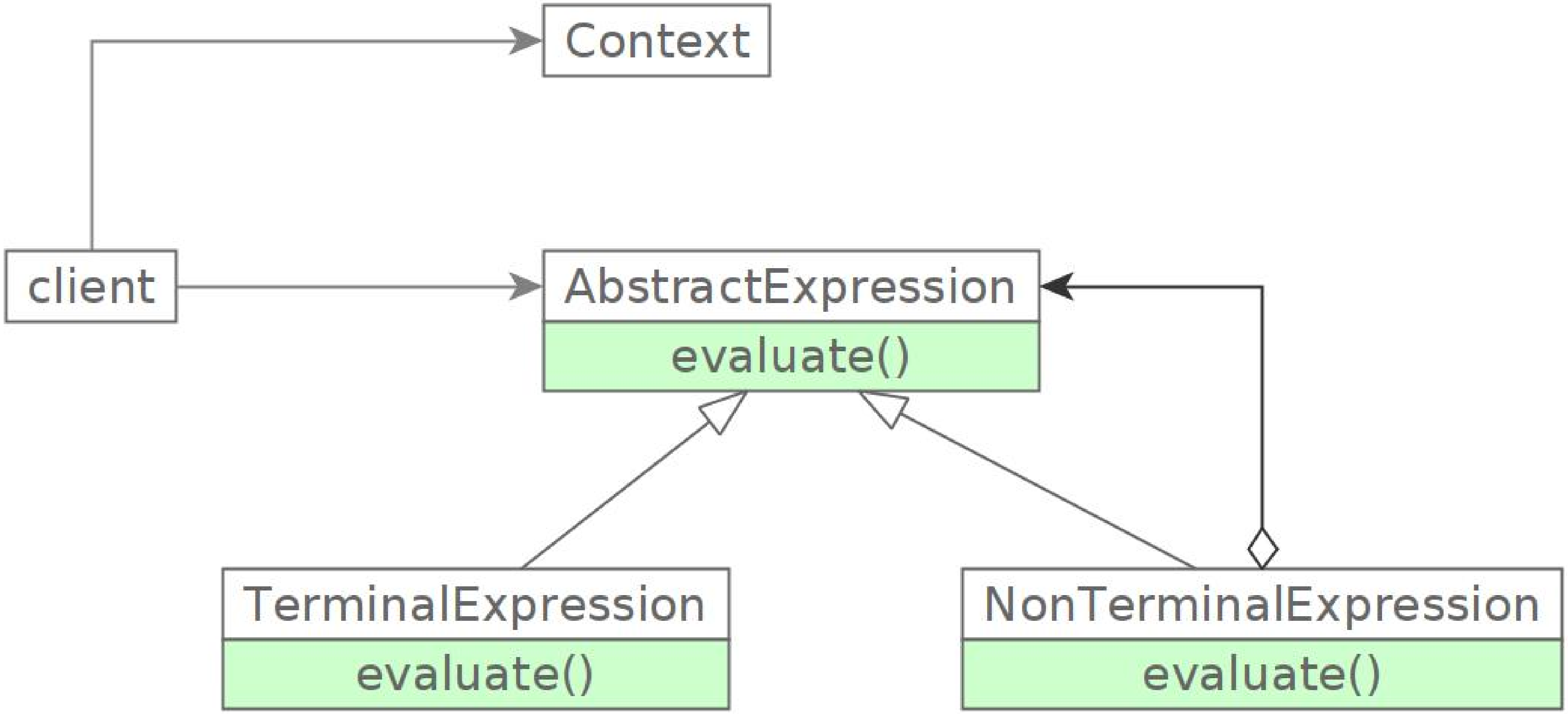
```
class Dialog: public Widget {
public:
    Dialog( HelpHandler * parent_,
               Topic t = NO_HELP_TOPIC );
    virtual void handleHelp();
}
Dialog::Dialog (HelpHandler* parent_, Topic t)
     :Widget(0)
{
    setHandler( parent_, t );
}
void Dialog::handleHelp() {
    if ( hasHelp() ) {
        // offer help on the dialog
    } else {
        HelpHandler::HandleHelp();
    }
}
```

```
class Application: public HelpHandler {
public:
    Application(Topic t): HelpHandler(0,t) {}

    virtual void handleHelp();
}

void Application::handleHelp() {
    // application-specific operations, eg.
    // show a list of help topics
}
```

```
const Topic APPLICATION_TOPIC = 1;
const Topic PRINT_TOPIC =2;
const Topic PAPER_ORIENTATION_TOPIC =3 ;

Application* application =
        new Application(APPLICATION_TOPIC);
Dialog * dialog =
        new Dialog ( application, PRINT_TOPIC);
Button * button =
        new Button( dialog, PAPER_ORIENTATION_TOPIC  );
```

Interpreter

- When the composite pattern is applied, how to implement an operation defined in the base class?
- Recursive operations on the object tree are often required.
- Interpreter pattern is used for such kind of operations.

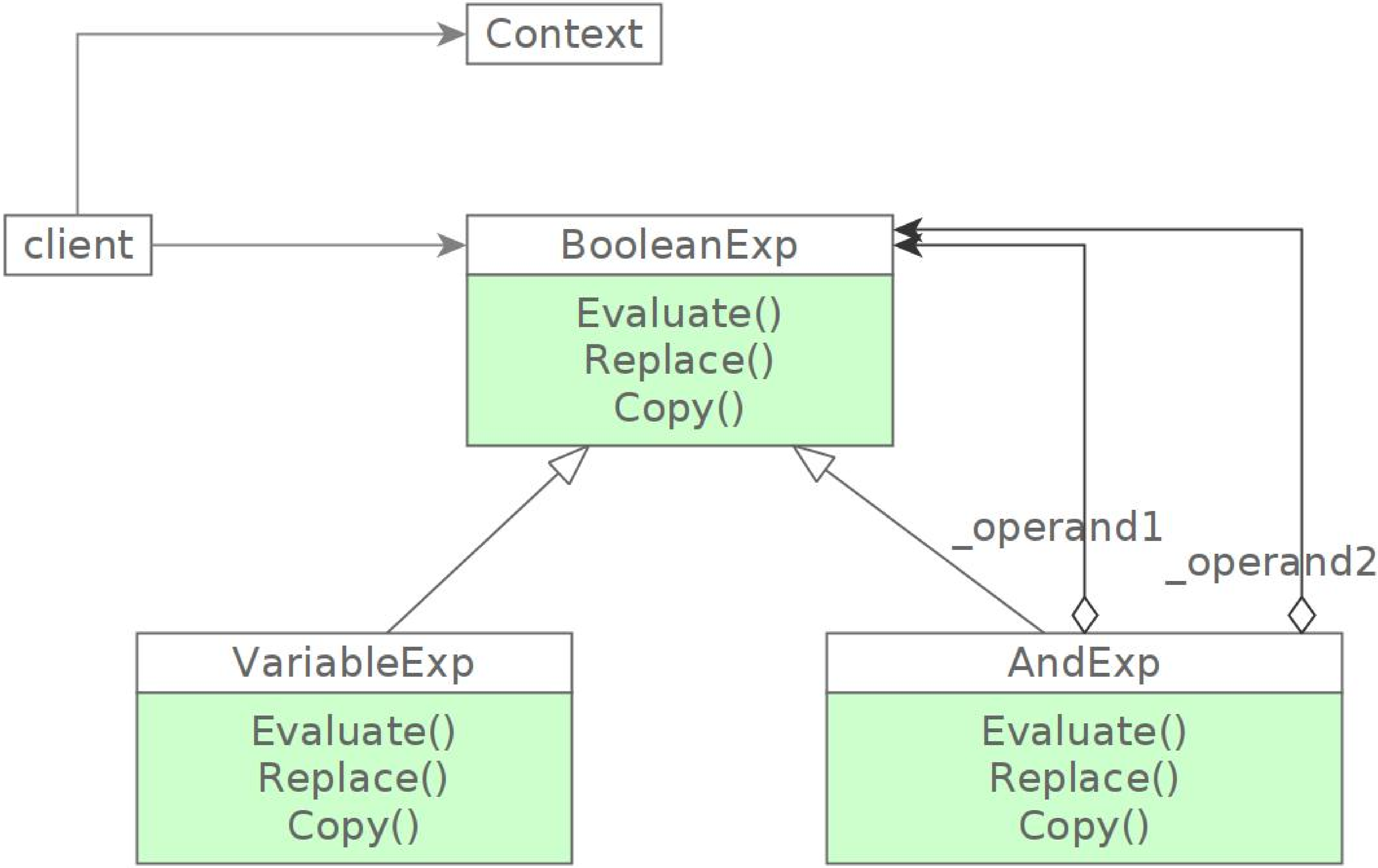


General structure of the Interpreter pattern

An example: boolean expressesion

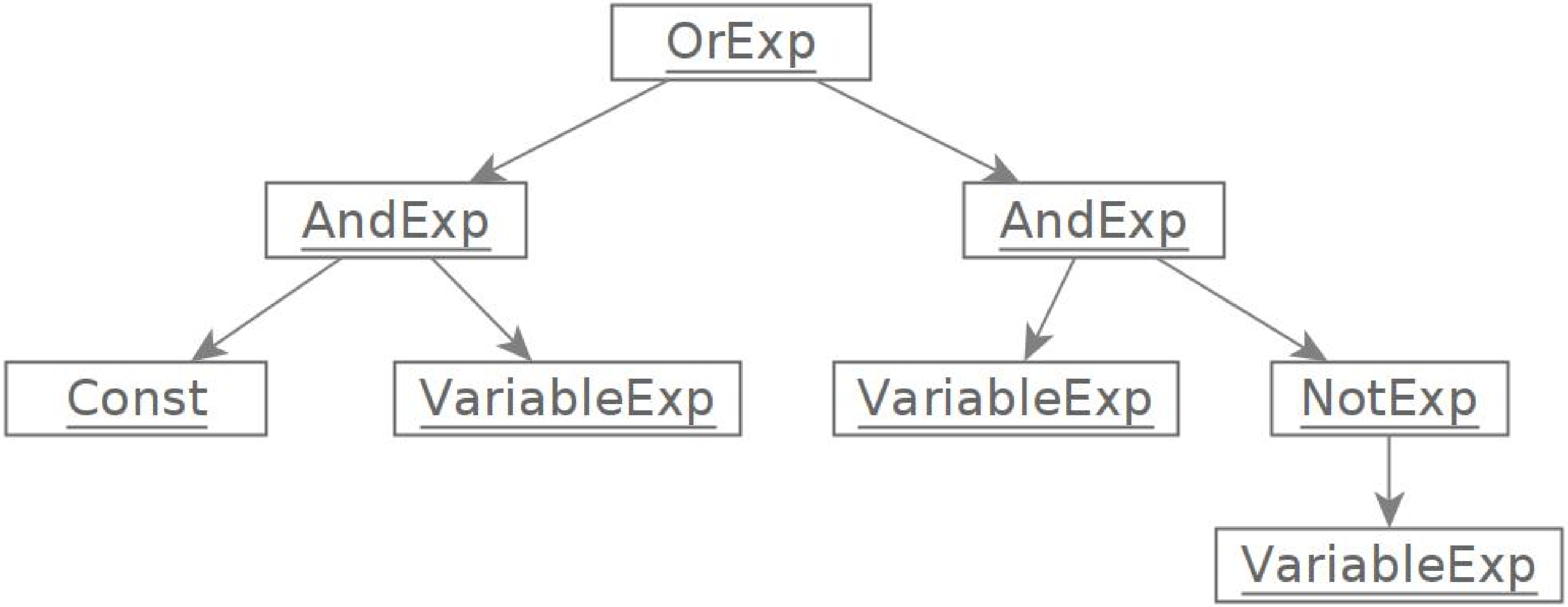
// (true and X) or (Y and (not X))

```
BooleanExp ::= VariableExp | Constant | OrExp |
               AndExp      | NotExp   | '(' BooleanExp ')'
AndExp     ::= BooleanExp 'and' BooleanExp
OrExp      ::= BooleanExp 'or'  BooleanExp
NotExp     ::= 'not' BooleanExp
Constant   ::= 'true' | 'false'
VariableExp::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'
```



Context
client
BooleanExp
Evaluate()
Replace()
Copy()
VariableExp
Evaluate()
Replace()
Copy()
AndExp
Evaluate()
Replace()
Copy()
_operand1
_operand2

(true and X) or (Y and (not X))



```
class BooleanExp {
  public:
    BooleanExp();
    virtual ~BooleanExp();

    virtual bool Evaluate(Context &) = 0;
    virtual BooleanExp * Replace( const char*,
                                  BooleanExp & ) = 0;
    virtual BooleanExp * Copy() const = 0;
}

class Context {
  public:
    bool Lookup(const char*) const;
    void Assign(VariableExp *, bool);
}
```

```
class VariableExp: public BooleanExp {
 public:
   VarialeExp( const char * name)   {   _name = strdup(name);   }
   virtual ~VariableExp();
   virtual bool Evaluate( Context & aContext) {
           return aContext.Lookup( _name );     }
   virtual BooleanExp * Replace( const char* name,
                                           BooleanExp & exp)  {
     if (strcmp(name, _name) == 0 )
          return exp.Copy();
     else
          return new VariableExp(_name);
   }
   virtual BooleanExp * Copy() const  {
      return new VariableExp(_name);    }
 private:
      char * _name;
}
```

```
class AndExp: public BooleanExp {
  public:
    AndExp(BooleanExp* op1, Boolean* op2) {
        _operand1 = op1;  _operand2 = op2;
    }
    virtual ~AndExp();
    virtual bool Evaluate( Context & aContext) {
            return  _operand1->Evaluate(aContext) &&
                    _operand2->Evaluate(aContext);
    }
    virtual BooleanExp * Replace( const char* name,
                                  BooleanExp & exp)  {
            return new AndExp(
                    _operand1->Replace(name,exp),
                    _operand2->Replace(name,exp)  )
    }
```

```
    virtual BooleanExp * Copy( )  const {
            return new AndExp(
                            _operand1->Copy(),
                            _operand2->Copy() )
    }
}
```

```
//   The expression is:   (true and X ) or (Y and (not X) )
BooleanExp * expression;
Context context;

VariableExp* x = new VariableExp(“X”);
VariableExp* y = new VariableExp(“Y”);
Expression = new OrExp(
        new AndExp( new Constant ( true), x),
        new AndExp( y, new NotExp(x) )
);

context.Assign( x, false);
context.Assign( y, true );

Bool result = expression->Evaluate( context );
```

```
// Now we replace the variable y with a new expression and
//      re-evaluate the expression

VariableExp* z = new VariableExp(“Z”);
NotExp not_z  (z);

BooleanExp*replacement = expression->Replace(“Y”, not_z);
context.Assign(z, true);
result = replacement->Evaluate ( context );
```

Implementation

- Dependency analysis
  - Examples: if an expression inside a loop does not depend on the local variables of the loop, it can be evaluated only once.
  - Effect caching
  - Parallelism
  - Lazy evaluation: compute nothing until it is explicitly asked for.