

Software Design
CSE 335, Fall 2004

The Template-Method Pattern

Stephen Wagner

Michigan State University

Expressions

- In the project, Sum, Product and Difference are all very similar classes
- We developed a BinaryExpr class and derived all the binary expressions from it.

```
class BinaryExpr : public Expr {
protected:
    const Expr* left;
    const Expr* right;
    BinaryExpr( const Expr* l, const Expr *r)
        : leftOperand(l), rightOperand(r) {}
};
```

- evaluate looks nearly identical in all the classes derived from BinaryExpr

```
int Sum::evaluate(int & v, const Store & st)
{  int leftval, rightval;
   if (left->evaluate(leftval,st) && right->evaluate(rightval,st)) {
       v=leftval+rightval;
       return true; }
   return false; }
```

```
int Product::evaluate(int & v, const Store & st)
{  int leftval, rightval;
   if (left->evaluate(leftval,st) && right->evaluate(rightval,st)) {
       v=leftval*rightval;
       return true; }
   return false; }
```

```
int Difference::evaluate(int & v, const Store & st)
{  int leftval, rightval;
   if (left->evaluate(leftval,st) && right->evaluate(rightval,st)) {
       v=leftval-rightval;
       return true; }
   return false; }
```

First Factoring Attempt

- In all cases we compute the left, then compute the right, and then perform an operation.
- Can we move this to BinaryExpr?

```
int BinaryExpr::evaluate(int & v, const Store & st)
{  int leftval, rightval;
   if (left->evaluate(leftval,st) && right->evaluate(rightval,st)) {
       if (type==SUM)
           v=leftval+rightval;
       if (type==PRODUCT)
           v=leftval*rightval;
       if (type==DIFFERENCE)
           v=leftval-rightval;
       return true; }
   return false;
}
```

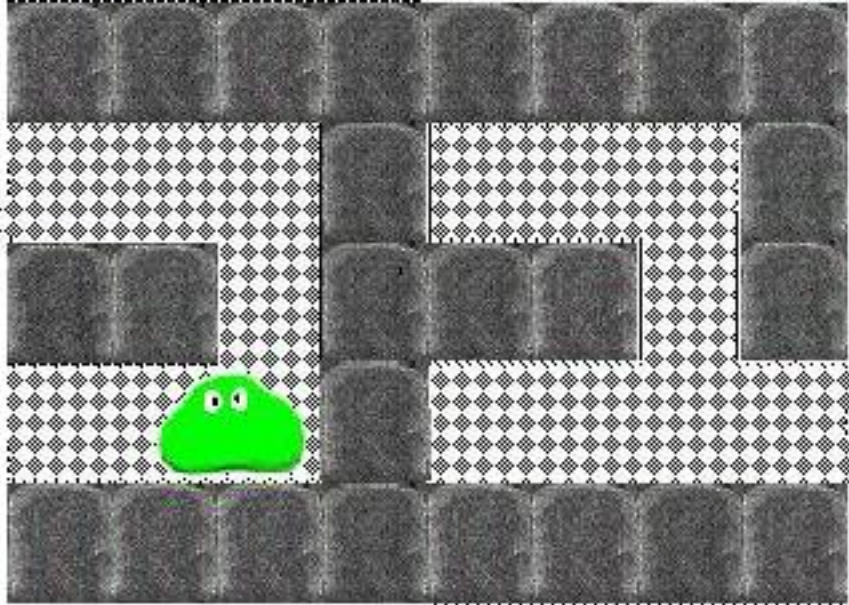
A Better Approach

- Lots of if statements are ugly

```
int BinaryExpr::evaluate(int & v, const Store & st)
{  int leftval, rightval;
   if (left->evaluate(leftval,st) && right->evaluate(rightval,st)) {
       v=operation(leftval,rightval);
       return true; }
   return false;
}
```

- How and where should operation be defined?
- Why is this better?

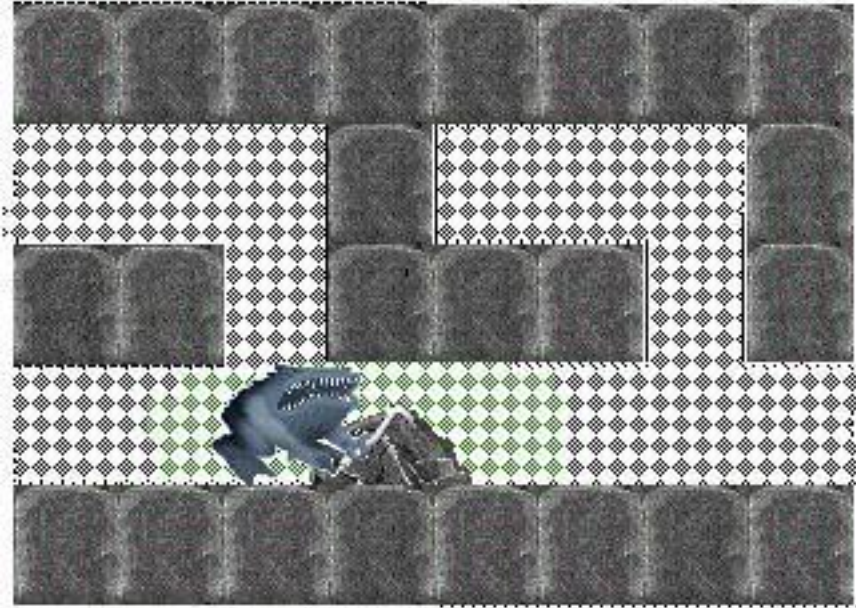
A More “Graphic” Example



A Normal Monster

- Wanders around maze
- Looks for and reacts to opponents
- Bounces off of walls
- What if we want to add additional monsters?

A Different Monster



Wall Busting Monster

- Has nearly identical behavior to a normal monster
- Can smash through walls
- How do we design our code so that we can reuse as much as possible?

The Template-Method Pattern

- Motivation: we want to implement an algorithm, but the exact details of the algorithm can change
 - we want to be able to describe a skeleton or outline for the algorithm
 - we want to be able to specialize the algorithm
- In the template-method pattern
 - the code that describes the algorithm is defined in the base class
 - the code that specializes the algorithm is defined in the derived classes
- This has nothing to do with C++ templates.

The Template Pattern

- The template pattern uses inheritance differently
- Traditional
 - operations of derived classes are invoked directly
 - operations defined in derived classes will make use of primitive operations defined in base class
- Template Pattern
 - operation of base class is invoked directly
 - the base class calls methods in the derived class
 - variations in the algorithm are achieved by using *hooks*

Using the Template Pattern

- Making use of template pattern requires a lot of foresight
- It is often easier to take two existing classes and redesign them using the template pattern
 - factors out common code
 - gives us a starting point to determine what the algorithms are

Expressions and the Template Pattern

- The basic algorithm is defined in the base class
 - compute the left expression, compute the right expression, perform an operation and return the result
- ```
int BinaryExpr::evaluate(int & v, const Store & st)
{ int leftval, rightval;
 if (left->evaluate(leftval,st) && right->evaluate(rightval,st)) {
 v=operation(leftval,rightval);
 return true; }
 return false;
}
```
- operation is the hook
- Each derived class defines its own operation method
- BinaryExpr::evaluate will invoke the operation method of one of the derived classes.

# The Template-Method Pattern and `simplify`

- In the project `simplify` was very similar for all three of the binary expressions
  - Each function was 30+ lines long
  - Each had to worry about negative results
- Using the Template-Method pattern
  - Most of the code can be put in once place
  - The tricky bit involving negative results is in one place
  - Additional operations could be easily added

```
const Expr * Sum::simplify() const
{
 int leftv, rightv;
 const Expr *leftsimp=left->simplify();
 const Expr *rightsimp=right->simplify();
 bool leftb=leftsimp->evaluate(leftv);
 bool rightb=rightsimp->evaluate(rightv);
 if (leftb && rightb) { // simplify constant expressions
 delete leftsimp; delete rightsimp;
 if (leftv+rightv>=0) return new Literal(leftv+rightv);
 else return new Negation(new Literal(-(leftv+rightv))); }
 else if (leftb && leftv==0) {
 delete leftsimp;
 return rightsimp; }
 else if (rightb && rightv==0) {
 delete rightsimp;
 return leftsimp; }
 return new Sum(leftsimp, rightsimp);
}
```

```

const Expr * Product::simplify() const
{
 int leftv, rightv;
 const Expr *leftsimp=left->simplify();
 const Expr *rightsimp=right->simplify();
 bool leftb=leftsimp->evaluate(leftv);
 bool rightb=rightsimp->evaluate(rightv);
 if (leftb && rightb) { // simplify constant expressions
 delete leftsimp; delete rightsimp;
 if (leftv*rightv>=0) return new Literal(leftv*rightv);
 else return new Negation(new Literal(-(leftv*rightv))); }
 else if (leftb && leftv==1) {
 delete leftsimp;
 return rightsimp; }
 else if (rightb && rightv==1) {
 delete rightsimp;
 return leftsimp; }
 if (leftb && leftv==0 || rightb && rightv==0) {
 delete leftsimp; delete rightsimp;
 return new Literal(0); }
 return new Product(l,r);
}

```

# Hook #1

- The first 10 lines are identical except for the operation performed.

```
const Expr * BinaryExpr::simplify() const
{
 int leftv, rightv;
 const Expr * leftsimp=left->simplify();
 const Expr * rightsimp=right->simplify();
 bool leftb=leftsimp->evaluate(leftv);
 bool rightb=rightsimp->evaluate(rightv);
 if (leftb && rightb) { // simplify constant expressions
 delete leftsimp; delete rightsimp;
 int val=operation(leftv,rightv);
 if (val>=0) return new Literal(val);
 else return new Negation(new Literal(val)); }
}
```

## Hook #2

- Code from `Sum::simplify`

```
else if (leftb && leftv==0) {
 delete leftsimp;
 return rightsimp; }
}
```

- Code from `Product::simplify`

```
else if (leftb && leftv==1) {
 delete leftsimp;
 return rightsimp; }
}
```

- What should the hook be?
- What mathematical property are we using?

## Hook #2 and #3

- We have more flexibility if we allow for left identities and right identities

```
else if (leftb && leftidentity(leftv))
{
 delete leftsimp;
 return rightsimp;
}
else if (rightb && rightidentity(rightv))
{
 delete rightsimp;
 return leftsimp;
}
```

## Hook #4

- Allow for additional simplifications to be performed

```
const Expr * BinaryExpr::simplify() const {
 int leftv,rightv;
 const Expr * leftsimp=left->simplify();
 const Expr * rightsimp=right->simplify();
 bool leftb=leftsimp->evaluate(leftv);
 bool rightb=rightsimp->evaluate(rightv);
 if (leftb && rightb) {
 delete leftsimp; delete rightsimp;
 int val=operation(leftv,rightv);
 if (val>=0) return new Literal(val);
 else return new Negation(new Literal(val)); }
 else if (leftb && leftidentity(leftv)) {
 delete leftsimp; return rightsimp; }
 else if (rightb && rightidentity(rightv)) {
 delete rightsimp; return leftsimp; }
 else
 return othersimplify(leftsimp, rightsimp, leftb, rightb, leftv, rightv);
```

## BinaryExpr

```
class BinaryExpr : public Expr {
protected:
 const Expr* left;
 const Expr* right;
public:
 BinaryExpr(const Expr* l, const Expr *r)
 : leftOperand(l), rightOperand(r) {}
 virtual int operation(int l, int r) const = 0;
 virtual int leftidentity(int v) const { return false;}
 virtual int rightidentity(int v) const { return false;}
 virtual const Expr * othersimplify(const Expr *, const Expr *, bool, bool,
 int, int) const = 0;
};
```

## Sum

```
class Sum : public BinaryExpr {
public:
 Sum(const Expr* l, const Expr *r)
 : BinaryExpr(l,r) {}
 virtual int operation(int l, int r) { return l+r; }
 virtual int leftidentity(int v) const { return v==0;}
 virtual int rightidentity(int v) const { return v==0;}
 virtual const Expr * othersimplify(const Expr * l, const Expr * r, bool, bool
 int, int)
 { return new Sum(l,r); }
};
```

# Difference

```
class Difference : public BinaryExpr {
public:
 Difference(const Expr* l, const Expr *r)
 : BinaryExpr(l,r) {}
 virtual int operation(int l, int r) { return l-r; }
 virtual int rightidentity(int v) const { return v==0;}
 virtual const Expr * othersimplify(const Expr * l, const Expr * r,
 bool leftb, bool rightb, int leftv, int rightv)
 { if (leftb && leftv==0)
 return new Negation(r);
 else
 return new Difference(l,r); }
};
```

# Product

```
class Product : public BinaryExpr {
public:
 Product(const Expr* l, const Expr *r)
 : BinaryExpr(l,r) {}
 virtual int operation(int l, int r) { return l*r; }
 virtual int leftidentity(int v) const { return v==1;}
 virtual int rightidentity(int v) const { return v==1;}
 virtual const Expr * othersimplify(const Expr * l, const Expr * r,
 bool leftb, bool rightb, int leftv, int rightv)
 {
 if (leftb && leftv==0 || rightb && rightv==0) {
 delete l; delete r;
 return new Literal(0); }
 else
 return new Product(l,r);
 }
};
```

## Back to the Monsters

```
void BasicMonster::move()
{
 if (seesOpponent())
 reactToOpponent();
 if (attacked)
 reactToAttack();
 if (wall in the way)
 hitWall();
}
```