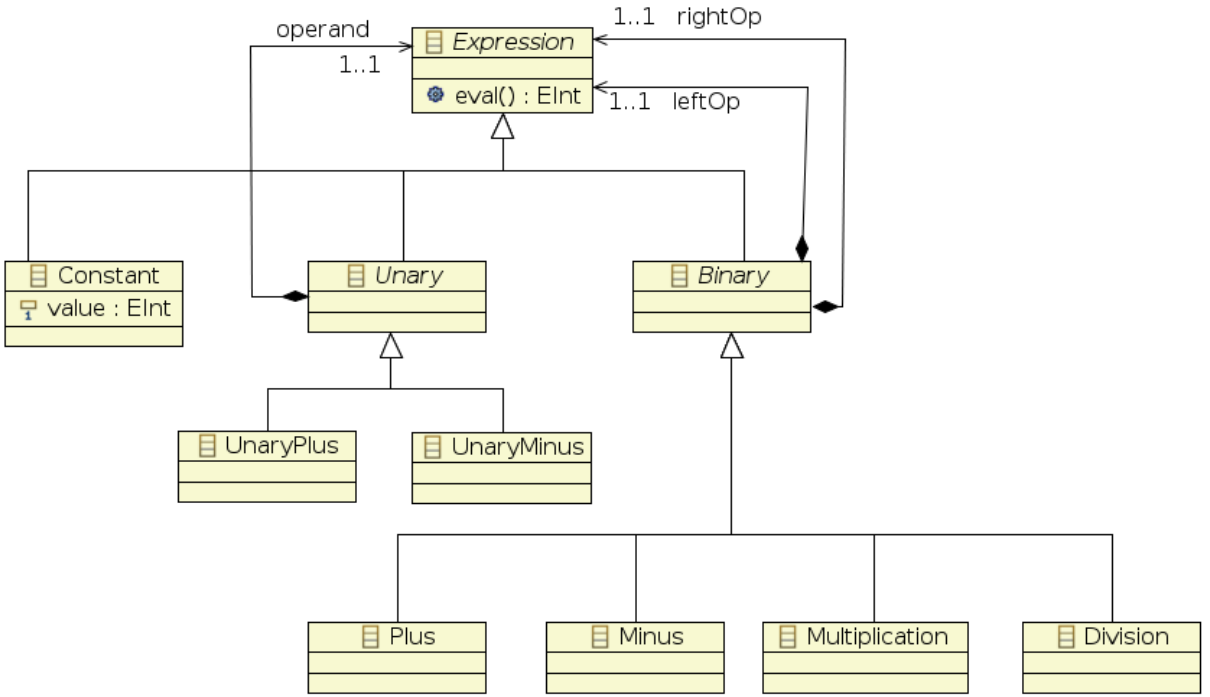
A <Basic> C++ Course

10 - SumUp

Julien Deantoni

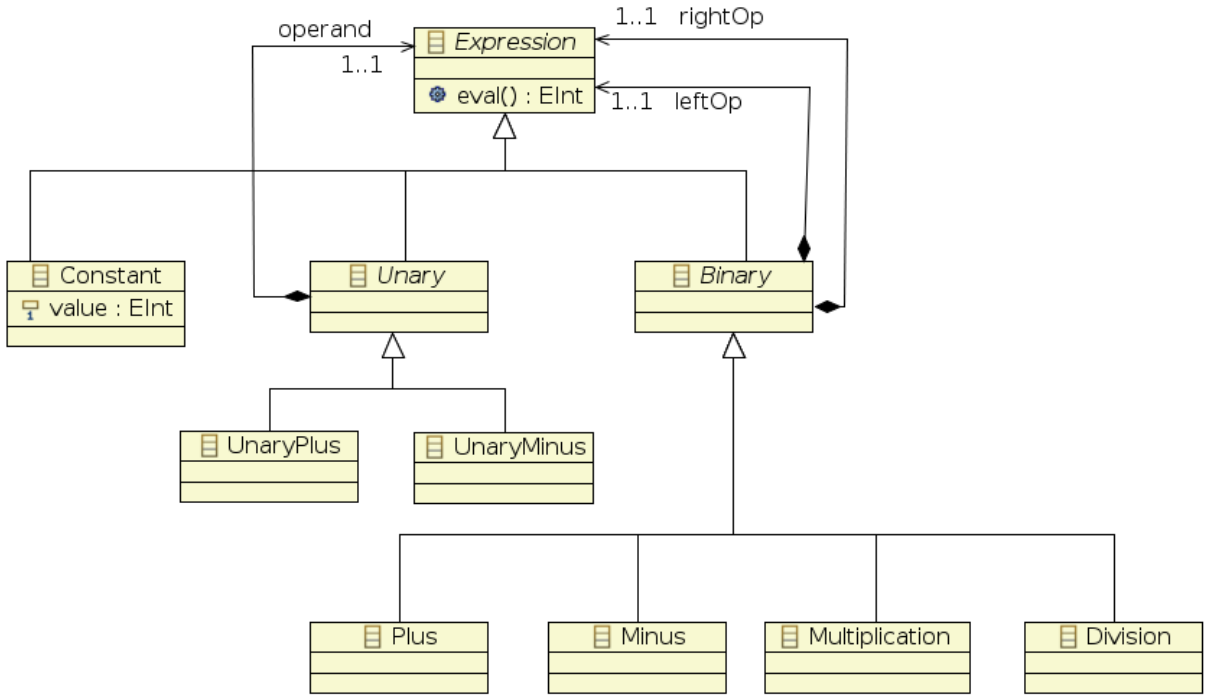
Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects



Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects



No description of the behaviours

How does `eval()` behave ?

Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects

No description of the behaviours

How does `eval()` behave ?

```
class Expression
{
public:

    class ZeroDivide : public exception
    {
    public:
        const char* what() const throw () {return "Division
         by 0";}
    };

    virtual ~Expr() {}
    virtual int eval() const = 0;
};
```

Expression.h

Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects

No description of the behaviours

How does `eval()` behave ?

```
class Expression
{
public:

    class ZeroDivide : public exception
    {
    public:
        const char* what() const throw () {return "Division
         by 0";}
    };

    virtual ~Expr() {}
    virtual int eval() const = 0;
};
```

Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects

No description of the behaviours

How does `eval()` behave ?

```
class Expression
{
public:

    class ZeroDivide : public exception
    {
    public:
        const char* what() const throw () {return "Division
         by 0";}
    };

    virtual ~Expr() {}
    virtual int eval() const = 0;
};
```

Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects

No description of the behaviours

How does `eval()` behave ?

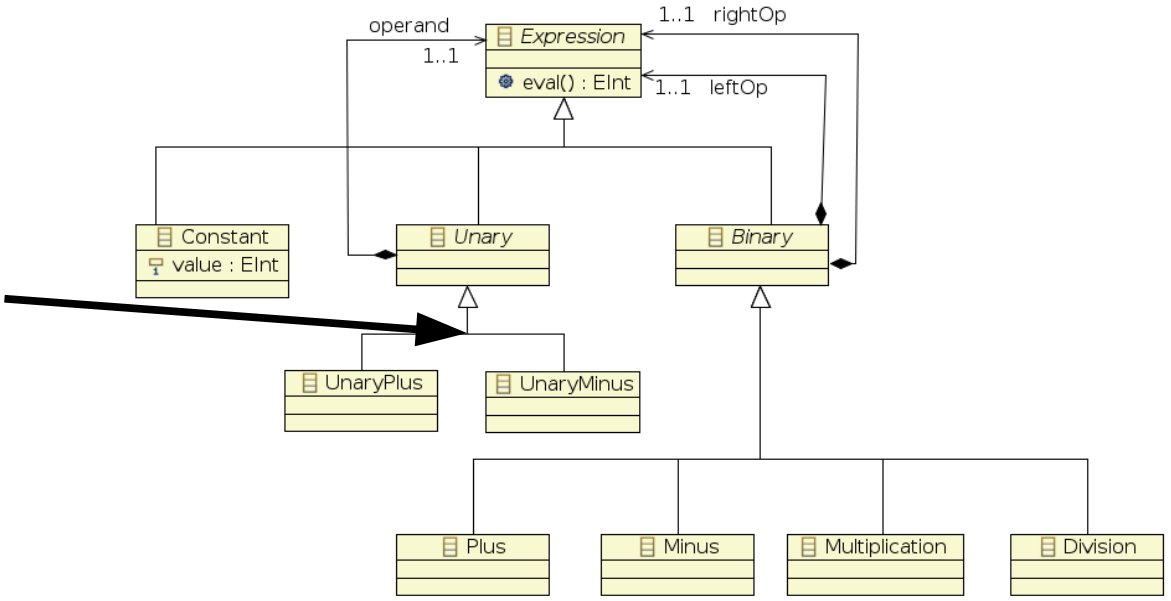
```
class Expression
{
public:

    class ZeroDivide : public exception
    {
    public:
        const char* what() const throw () {return "Division
         by 0";}
    };

    virtual ~Expr() {}
    virtual int eval() const = 0;
};
```

Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects

No description of the behaviours

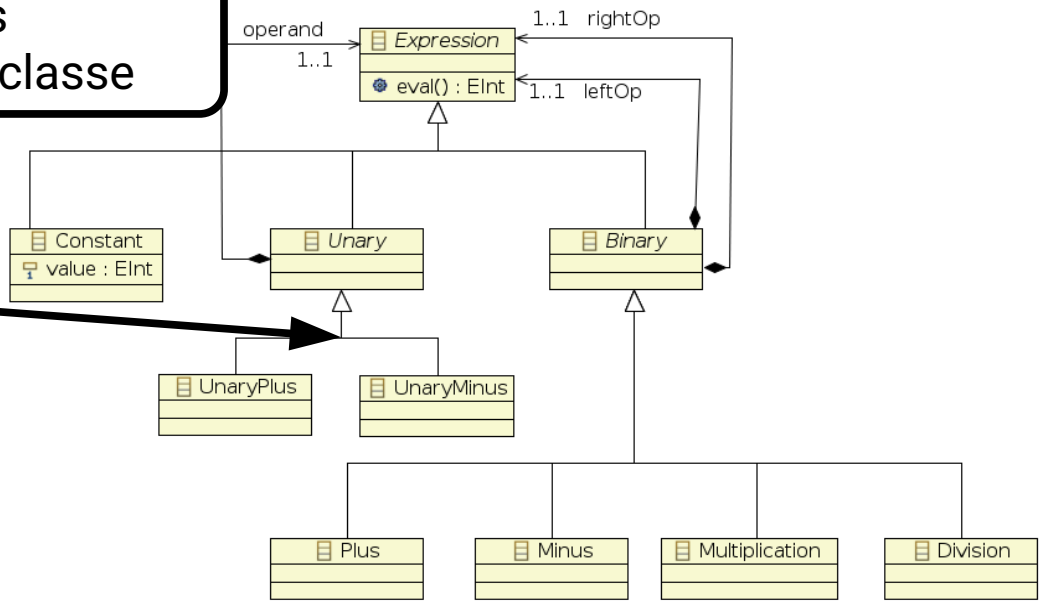
```
class UnaryPlus : public Unary
{
protected:
    Expr* operand;
public:
    Unary_Plus(Expr& pe);
    virtual int eval() const;
};
```



Object-Oriented concepts

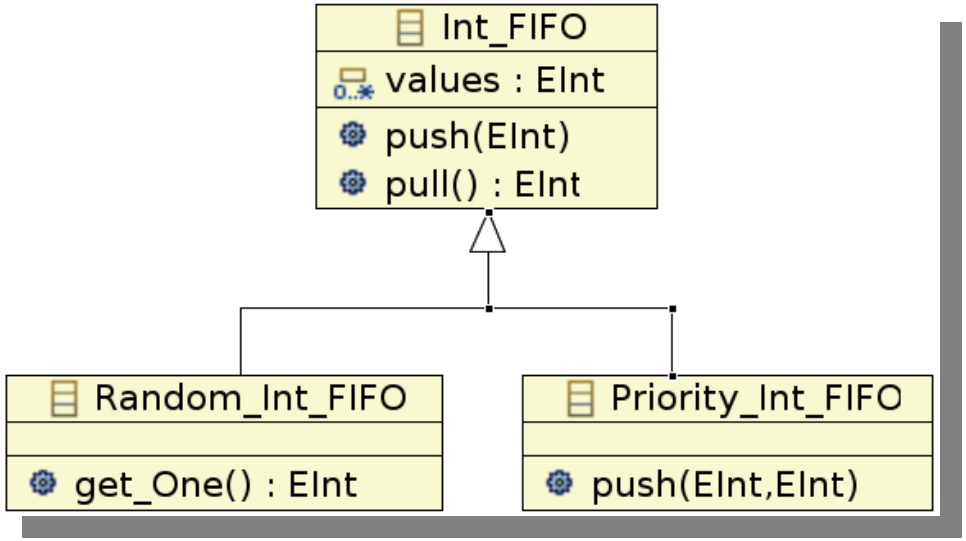
- A class diagram gives
  - structural aspects
  - relational aspects

Différents types d'héritages
pour « modifier » l'interface de la classe




```
class UnaryPlus : public Unary
{
protected:
    Expr* operand;
public:
    Unary_Plus(Expr& pe);
    virtual int eval() const;
};
```

Derivation public / private



- What if we declare Random_Int_FIFO like that ?

```
class Random_Int_FIFO : private Int_FIFO
{
 public:
   int get_One();
}
```

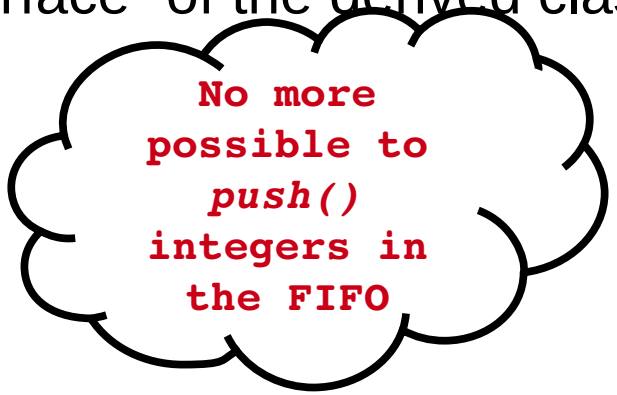
Derivation public / private

- The private derivation
  - All members of derived class become private
  - The “interface” of the derived class is lost...

```
class Random_Int_FIFO : private Int_FIFO
{
 public:
   int get_One();
}
```

Derivation public / private

- The private derivation
  - All members of derived class become private
  - The “interface” of the derived class is lost...




```
class Random_Int_FIFO : private Int_FIFO
{
 public:
   int get_One();
}
```

Derivation public / private

- The private derivation
  - All members of derived class become private
  - The “interface” of the derived class is lost...
  - But some parts of the interface can be set public again

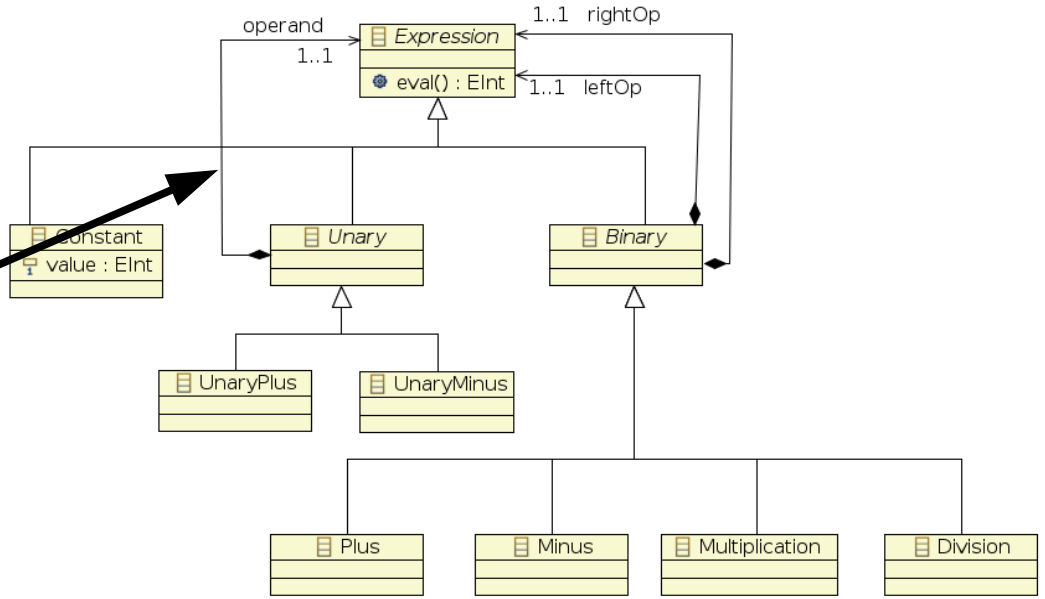
```
class Random_Int_FIFO : private Int_FIFO
{
 public:
   int get_One();
   using Int_FIFO::push;
}
```

Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects

No description of the behaviours

```
class Unary : public Expression
{
protected:
    Expr* operand;
public:
    Unary(Expr& pe);
    virtual int eval() const;
};
```

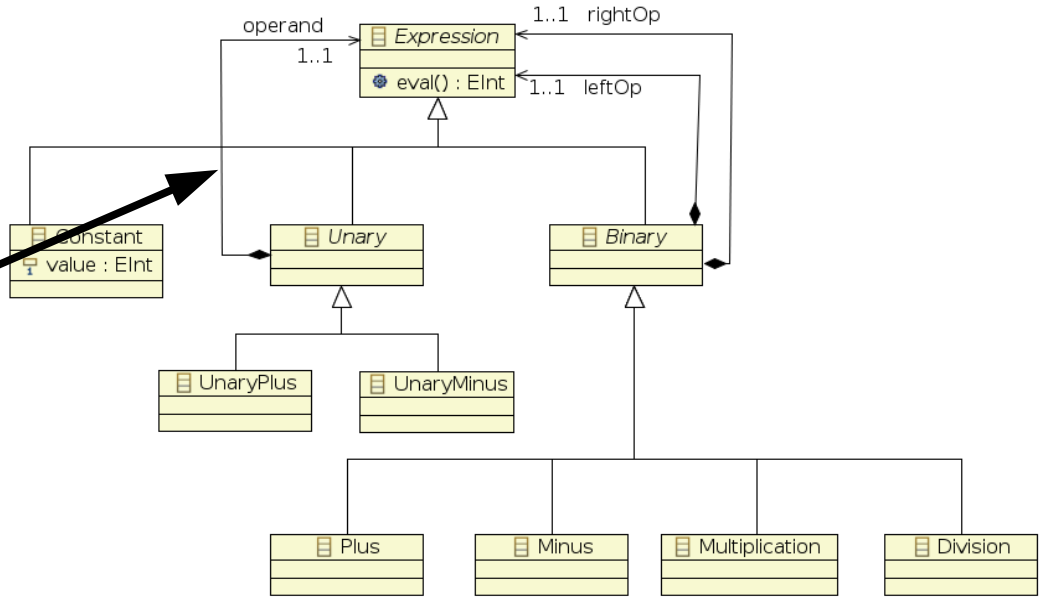


Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects

No description of the behaviours

```
class Unary : public Expression
{
protected:
    Expr* operand;
public:
    Unary(Expr& pe);
    virtual int eval() const;
};
```

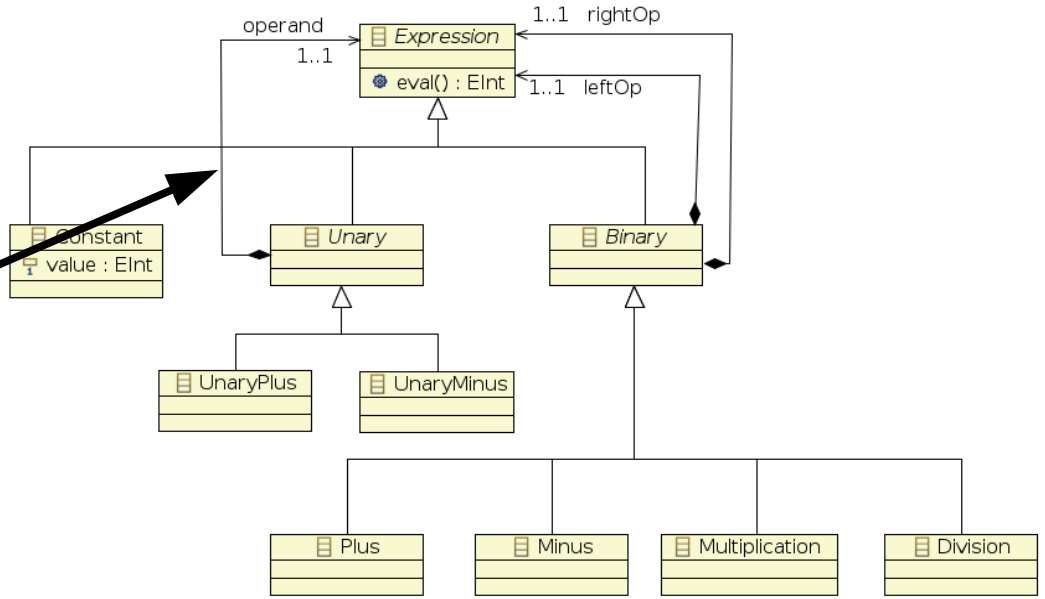


Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects

No description of the behaviours

```
class Unary : public Expression
{
protected:
    Expr* operand;
public:
    Unary(Expr& pe);
    virtual int eval() const;
};
```



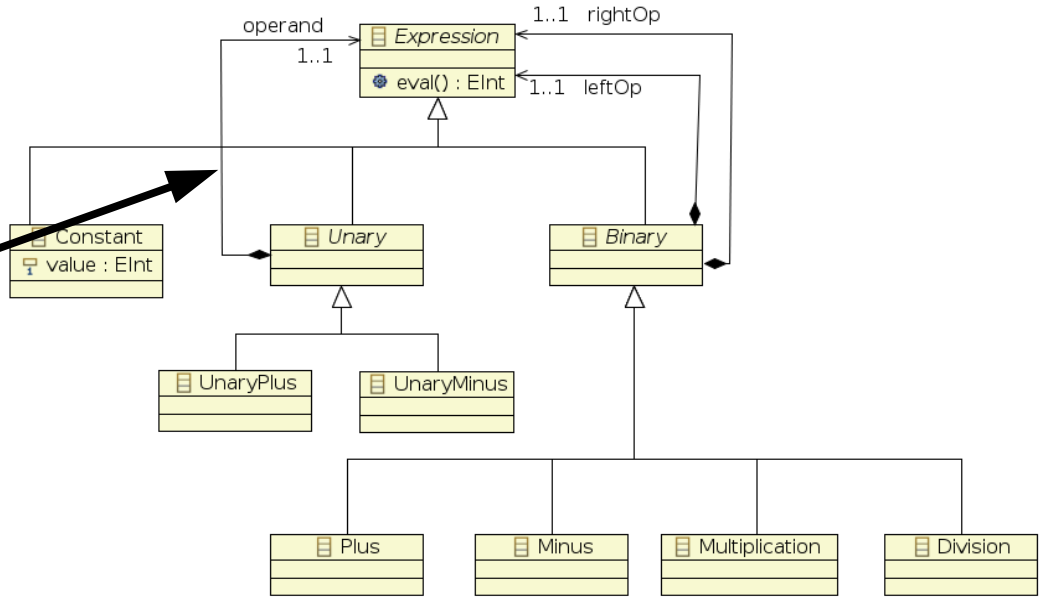
Réflexe associé ?

Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects

No description of the behaviours

```
class Unary : public Expression
{
protected:
    Expr* operand;
public:
    Unary(Expr& pe);
    virtual int eval() const;
    virtual ~Unary();
};
```



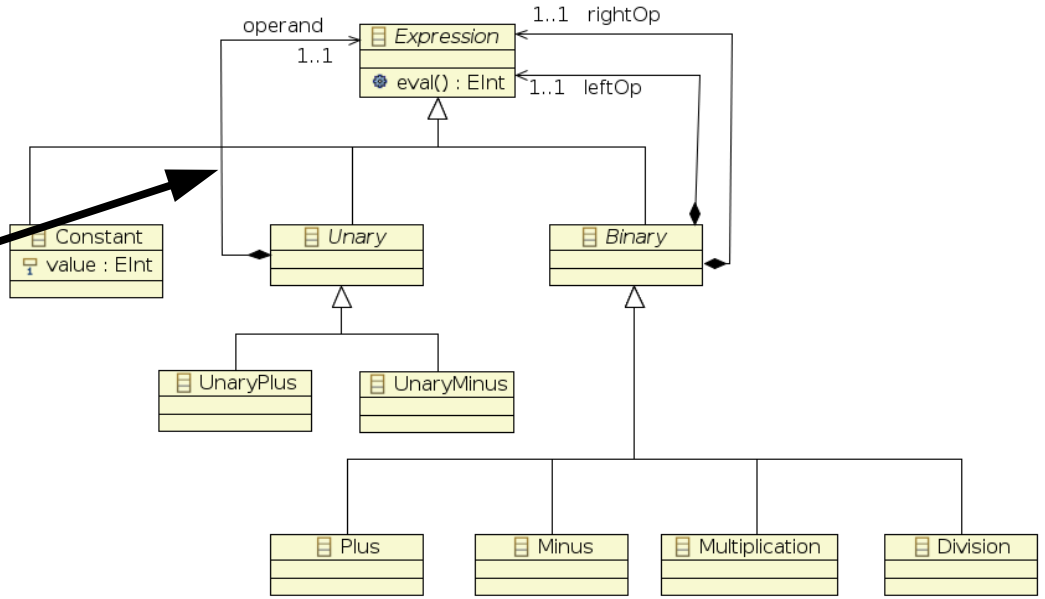
Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects

Remember the impact of the containment

No description of the behaviours

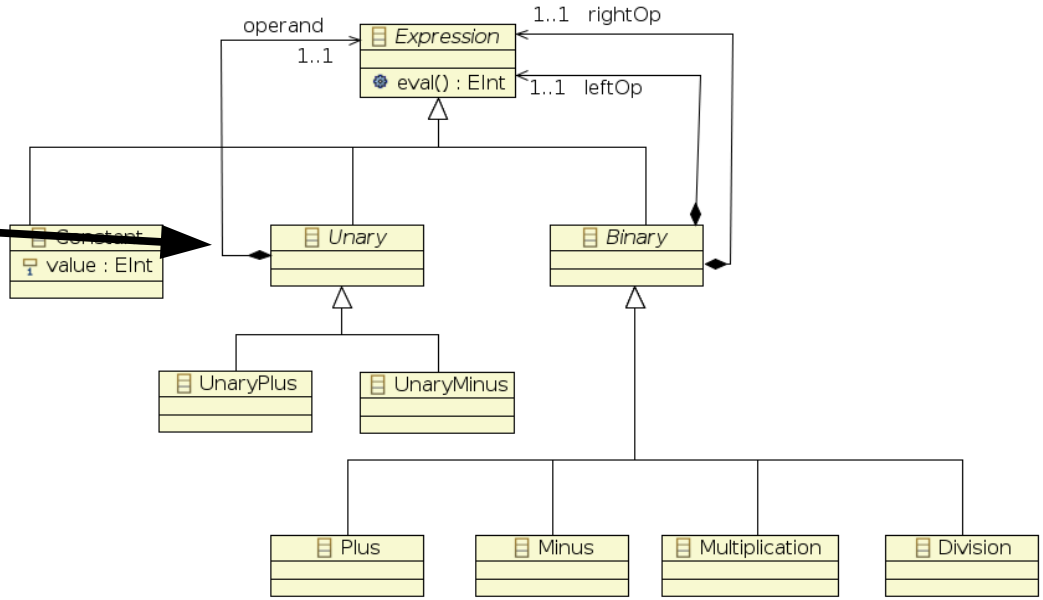
```
class Unary : public Expression
{
protected:
    Expr* operand;
public:
    Unary(Expr& pe);
    virtual int eval() const;
    virtual ~Unary();
};
```



Object-Oriented concepts

- Impact of containment:
  - The life of contained object(s) is your responsibility

```
Unary::Unary(Expression& pe)
{
  operand = pe.clone();
}


Expression* UnaryPlus::clone()
{
  return new UnaryPlus(*this);
}
```

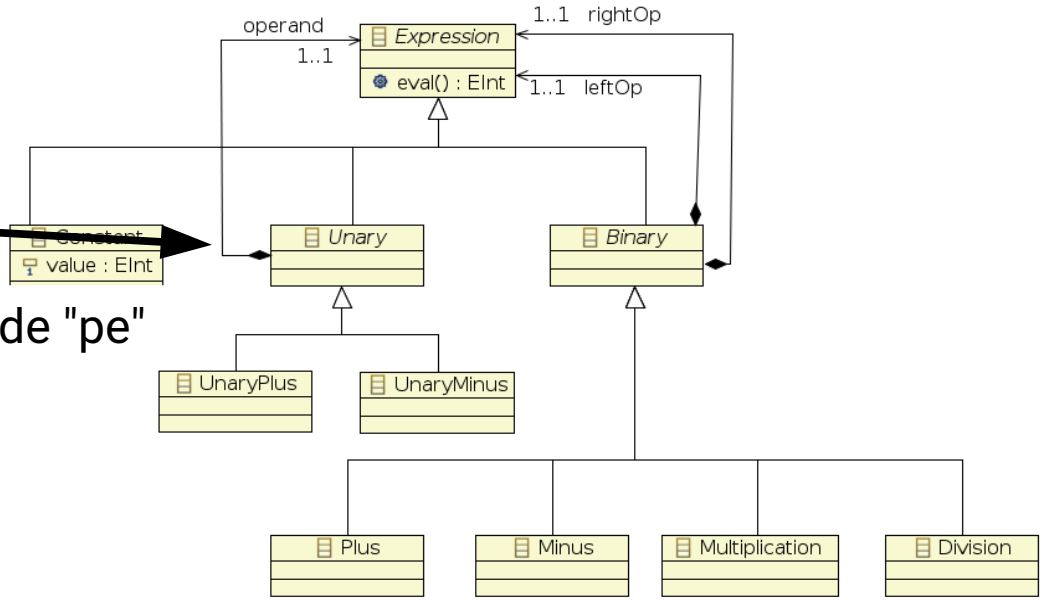


Object-Oriented concepts

- Impact of containment:
  - The life of contained object(s) is your responsibility

```
Unary::Unary(Expression& pe)
{
  operand = pe.clone();
}

Expression* UnaryPlus::clone()
{
  return new UnaryPlus(*this);
}
```

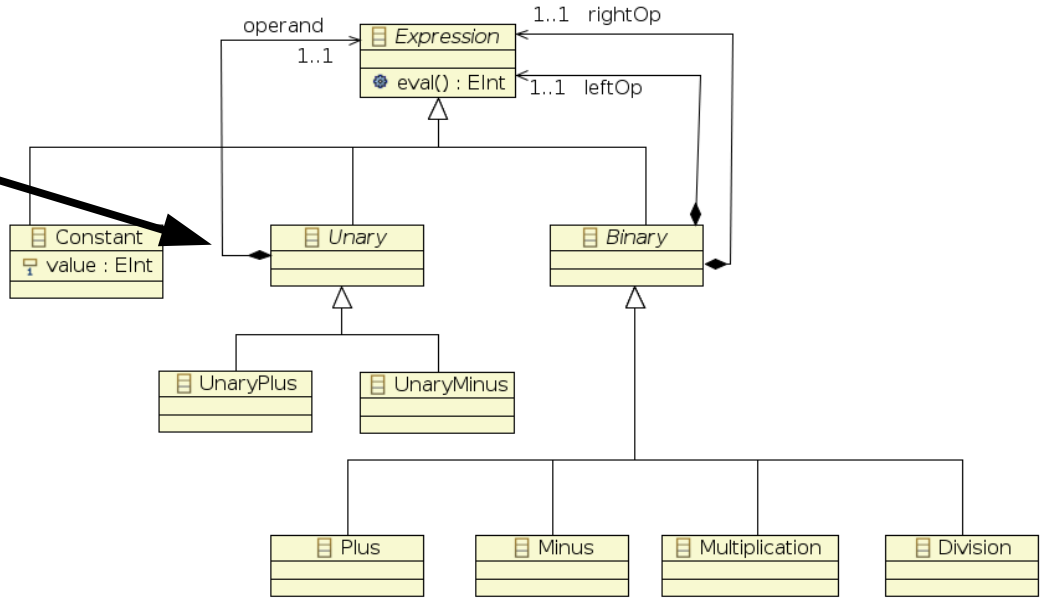
Du type dynamique de "pe"



Object-Oriented concepts

- Impact of containment:
  - The life of contained object(s) is your responsibility

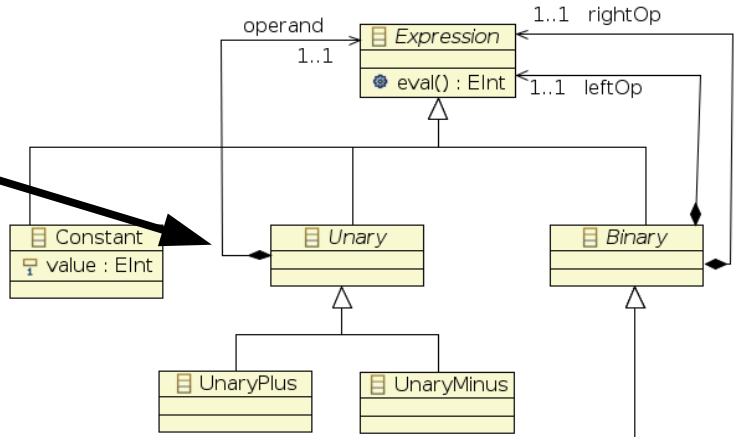
```
Unary::Unary(Expression& pe)
{
  operand = pe.clone();
}


Expression* UnaryPlus::clone()
{
  return new UnaryPlus(*this);
}

Unary::~Unary()
{
  delete operand;
}
```



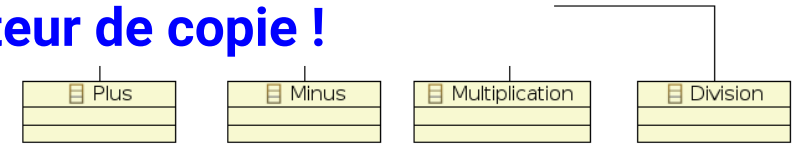
Object-Oriented concepts

- Impact of containment:
  - The life of contained object(s) is your responsibility

```
Unary::Unary(Expression& pe)
{
  operand = pe.clone();
}


Expression* UnaryPlus::clone()
{
  return new UnaryPlus(*this);
}

Unary::~Unary()
{
  delete operand;
}
```



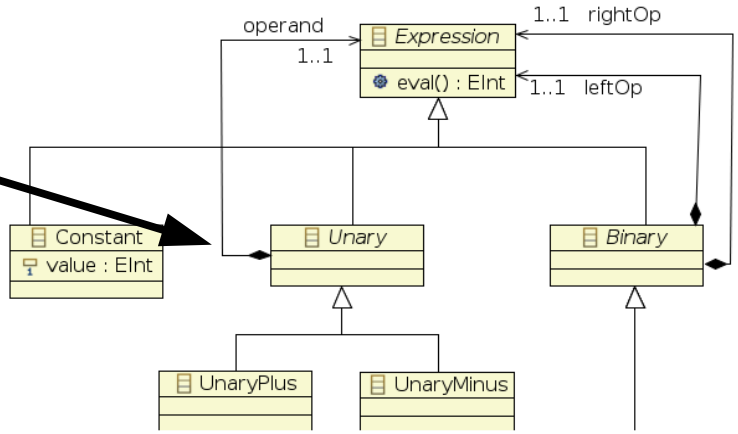
Appel au constructeur de copie !



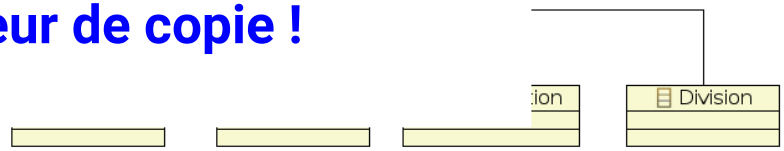
Object-Oriented concepts

- Impact of containment:
  - The life of contained object(s) is your responsibility

```
Unary::Unary(Expression& pe)
{
  operand = pe.clone();
}


Expression* UnaryPlus::clone()
{
  return new UnaryPlus(*this);
}

Unary::~Unary()
{
  delete operand;
}
```



Appel au constructeur de copie !
À définir ?

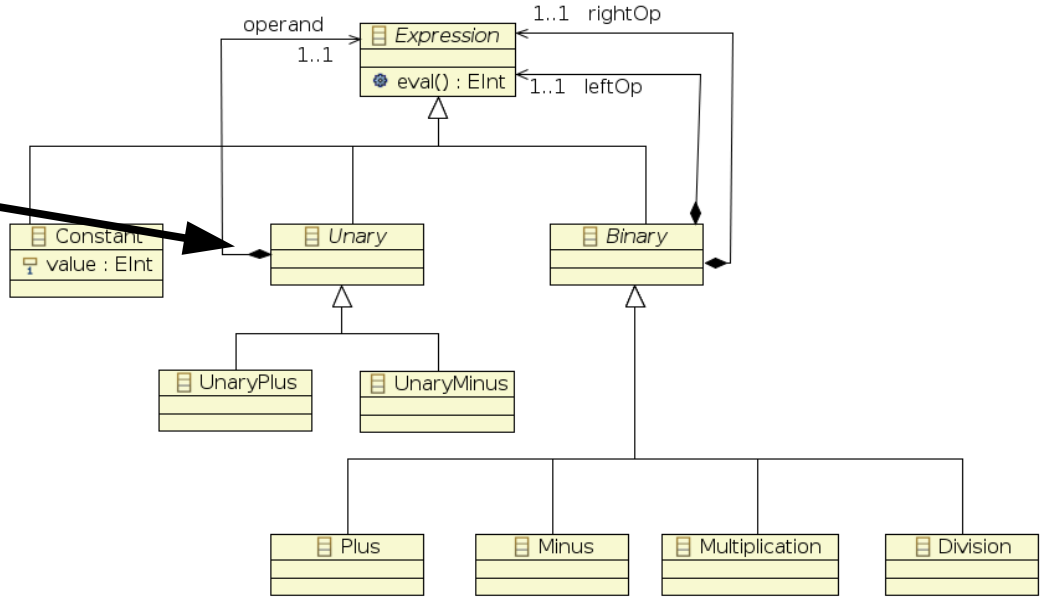


Object-Oriented concepts

- Impact of containment:
  - The life of contained object(s) is your responsibility

```
Unary::Unary(const Unary& un)
{
  operand = un.op->clone();
}
Unary::Unary(Expression& pe)
{
  operand = pe.clone();
}


Expression* UnaryPlus::clone()
{
  return new UnaryPlus(*this);
}

Unary::~Unary()
{
  delete operand;
}
```

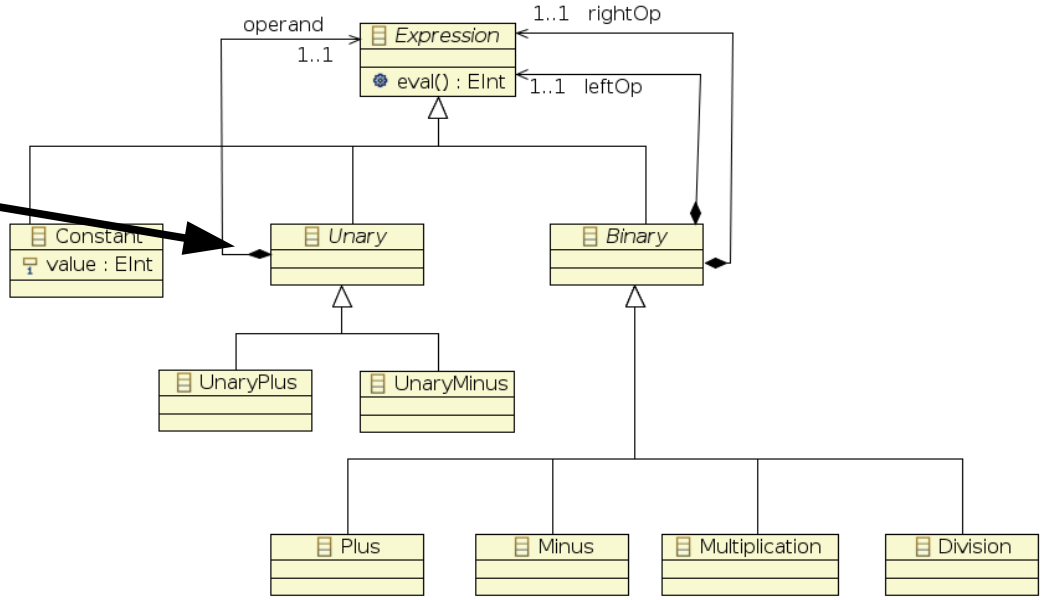
Suffisant ou à redéfinir dans UnaryPlus ?



Object-Oriented concepts

- Impact of containment:
  - The life of contained object(s) is your responsibility

```
Unary::Unary(const Unary& un)
{
  operand = un.op->clone();
}
Unary::Unary(Expression& pe)
{
  operand = pe.clone();
}


Expression* UnaryPlus::clone()
{
  return new UnaryPlus(*this);
}

Unary::~Unary()
{
  delete operand;
}
```

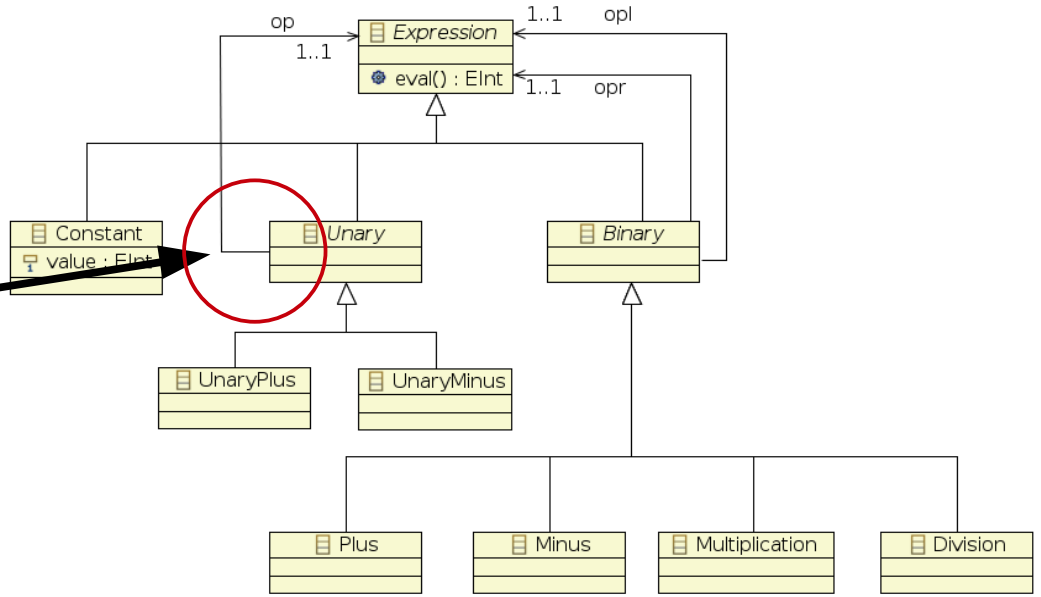


Et bien sûr ça impose la définition de l'opérateur d'affectation...

Object-Oriented concepts

- Impact of containment:
  - The life of contained object(s) is your responsibility

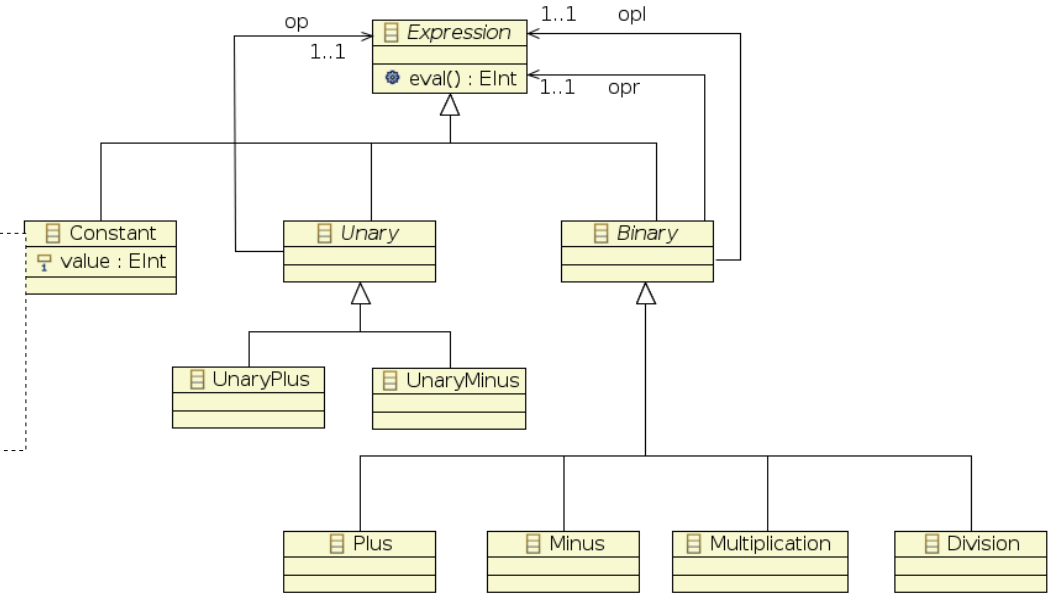
```
UnaryPlus::UnaryPlus(Expression& pe)
{
  operand = &pe
}
```



Object-Oriented concepts

- Take care of object life duration

```
UnaryPlus::UnaryPlus(Expression& pe)
{
  operand = &pe
}
```

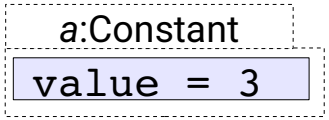


Here, no copied parameters can be accepted

Object-Oriented concepts

- Take care of object life duration

```
main(){
Constant a(3);

}
```



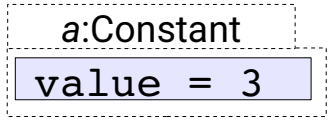

```
UnaryPlus::UnaryPlus(Expression& pe)
{
  operand = &pe
}
```

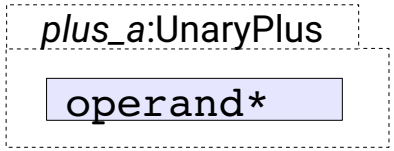
Object-Oriented concepts

- Take care of object life duration

```
main(){
Constant a(3);
UnaryPlus plus_a(a);
}
```

```
UnaryPlus::UnaryPlus(Expression& pe)
{
  operand = &pe
}
```



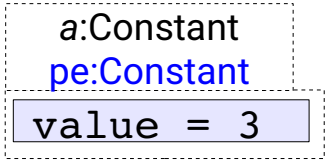


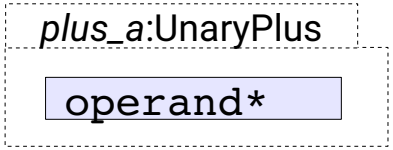
Object-Oriented concepts

- Take care of object life duration

```
main(){
Constant a(3);
UnaryPlus plus_a(a);
}
```

```
UnaryPlus::UnaryPlus(Expression& pe)
{
  operand = &pe
}
```



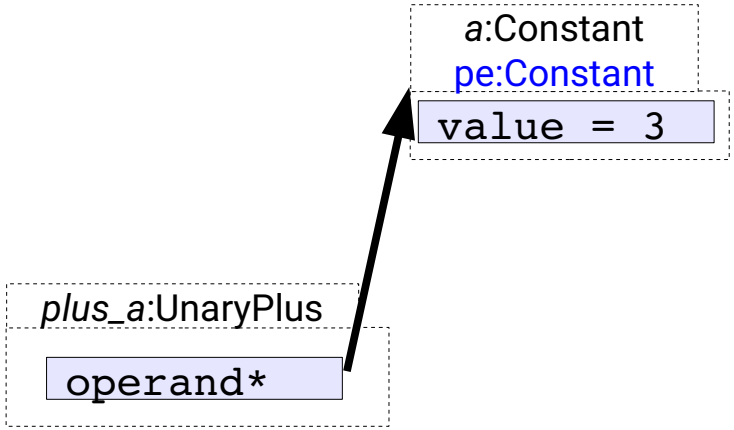


Object-Oriented concepts

- Take care of object life duration

```
main(){
Constant a(3);
UnaryPlus plus_a(a);
}
```

```
UnaryPlus::UnaryPlus(Expression& pe)
{
  operand = &pe
}
```

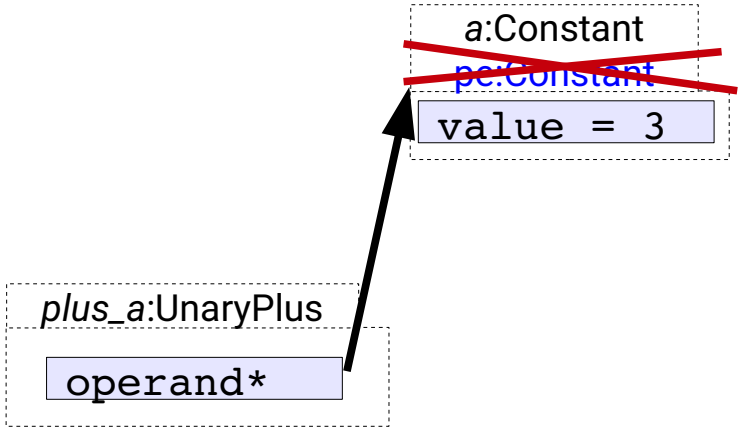


Object-Oriented concepts

- Take care of object life duration

```
main(){
Constant a(3);
UnaryPlus plus_a(a);
}
```

```
UnaryPlus::UnaryPlus(Expression& pe)
{
  operand = &pe
}
```

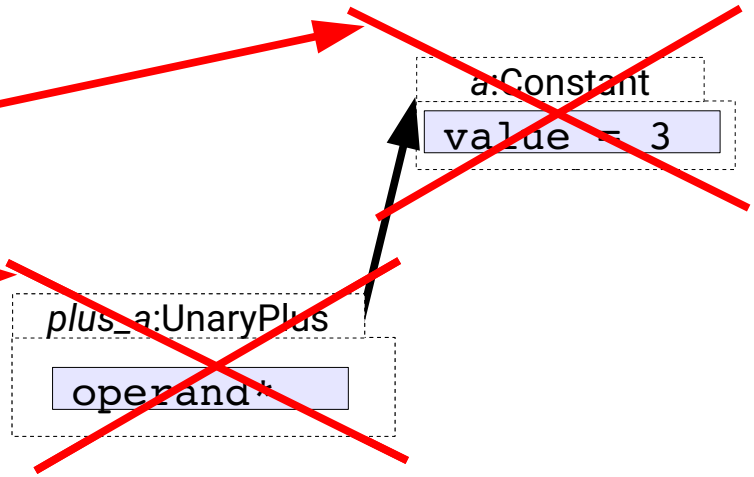


Object-Oriented concepts

- Take care of object life duration

```
main(){
Constant a(3);
UnaryPlus plus_a(a);
}
```

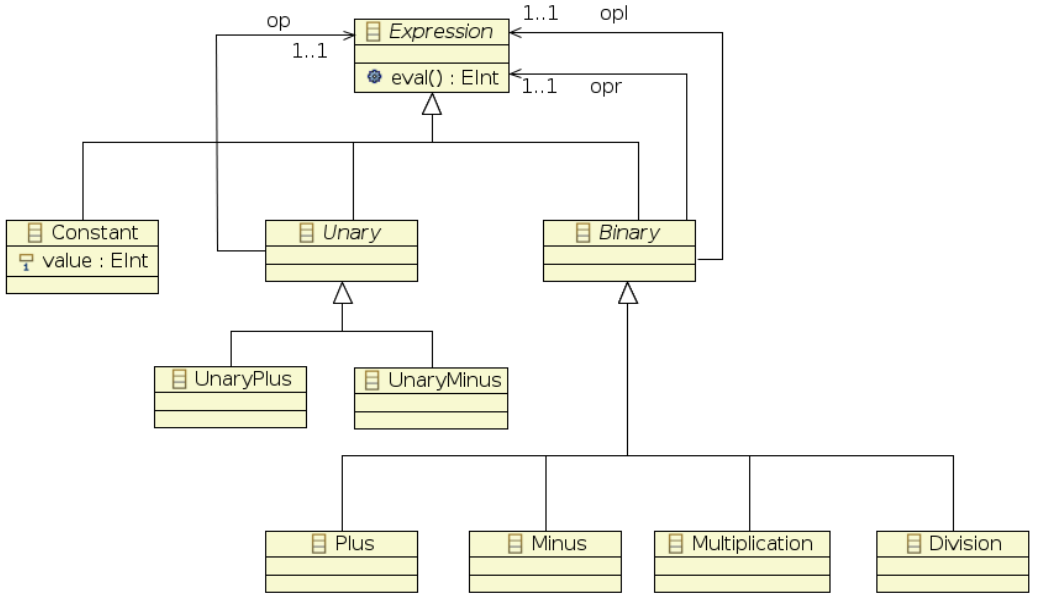
```
UnaryPlus::UnaryPlus(Expression& pe)
{
  operand = &pe
}
```



Object-Oriented concepts

- Take care of object life duration

```
UnaryPlus::UnaryPlus(Expression& pe)
{
  operand = &pe
}
```

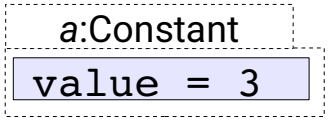


Here, no copied parameters can be accepted

Object-Oriented concepts

- Take care of object life duration

```
main(){
Constant a(3);

}
```



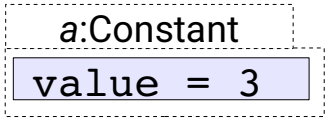

```
UnaryPlus::UnaryPlus(Expression pe)
{
  operand = &pe
}
```

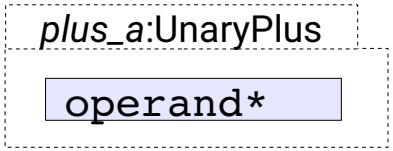
Object-Oriented concepts

- Take care of object life duration

```
main(){
Constant a(3);
UnaryPlus plus_a(a);
}
```

```
UnaryPlus::UnaryPlus(Expression pe)
{
  operand = &pe
}
```



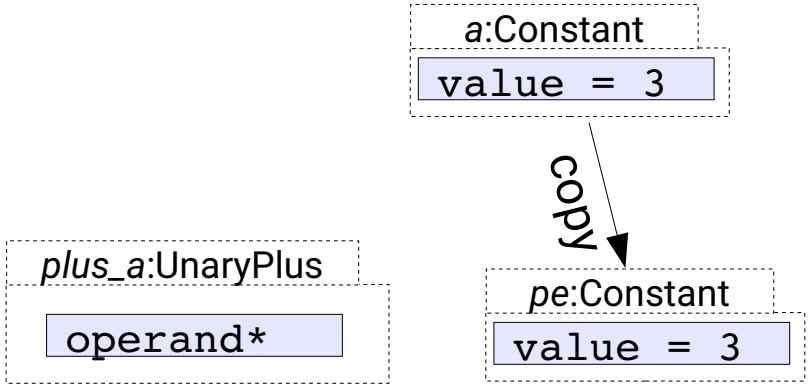


Object-Oriented concepts

- Take care of object life duration

```
main(){
Constant a(3);
UnaryPlus plus_a(a);
}
```

```
UnaryPlus::UnaryPlus(Expression pe)
{
  operand = &pe
}
```

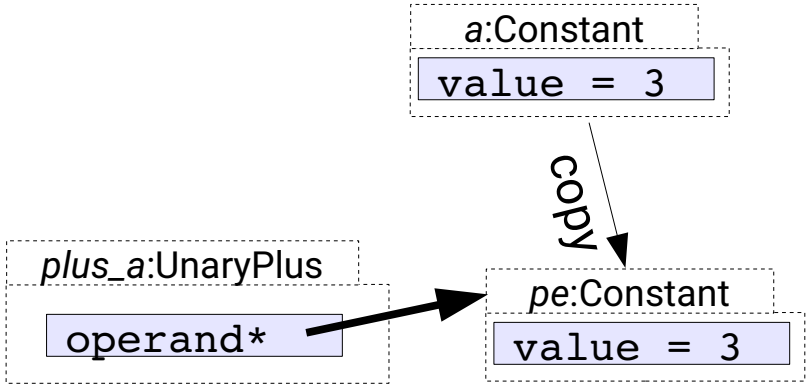


Object-Oriented concepts

- Take care of object life duration

```
main(){
Constant a(3);
UnaryPlus plus_a(a);
}
```

```
UnaryPlus::UnaryPlus(Expression pe)
{
  operand = &pe
}
```

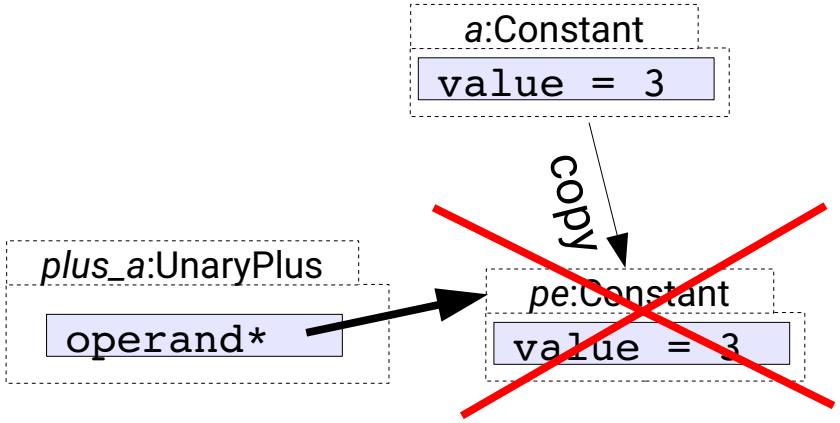


Object-Oriented concepts

- Take care of object life duration

```
main(){
Constant a(3);
UnaryPlus plus_a(a);
}
```

```
UnaryPlus::UnaryPlus(Expression pe)
{
  operand = &pe
}
```

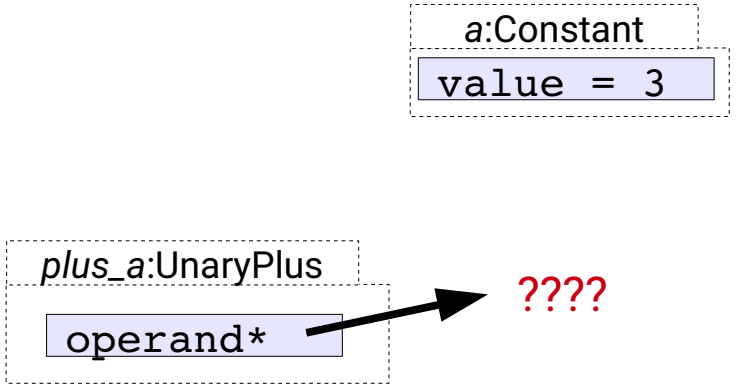


Object-Oriented concepts

- Take care of object life duration

```
main(){
Constant a(3);
UnaryPlus plus_a(a);
plus_a.eval()     // problem
                  (segmentation fault)
}
```

```
UnaryPlus::UnaryPlus(Expression pe)
{
  operand = &pe
}
```

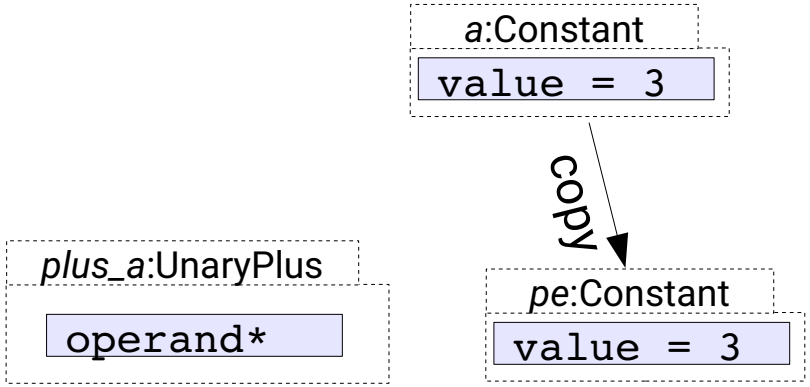


Object-Oriented concepts

- Take care to truncation..

```
main(){
Constant a(3);
UnaryPlus plus_a(a);
}
```

```
UnaryPlus::UnaryPlus(Expression pe)
{
  operand = &pe
}
```



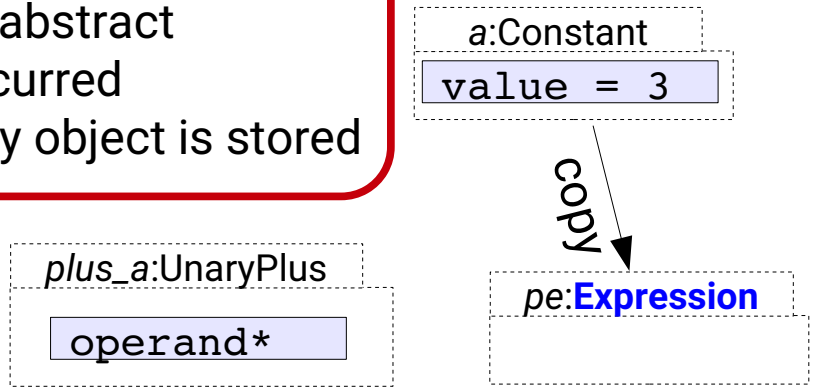
Object-Oriented concepts

- Take care to truncation..

Triple wrong because here:
1. Expression is abstract
2. truncation occurred
3.@ of temporary object is stored

```
main(){
Constant a(3);
UnaryPlus plus_a(a);
}
```

```
UnaryPlus::UnaryPlus(Expression pe)
{
  operand = &pe
}
```

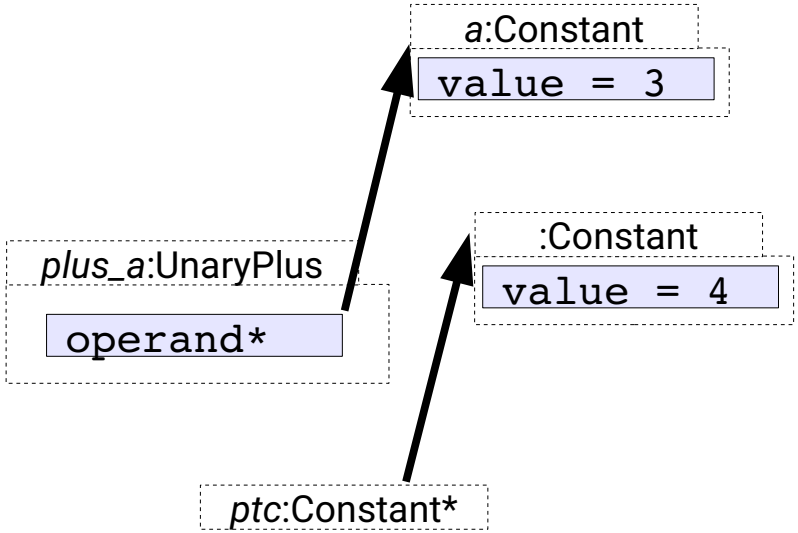


Object-Oriented concepts

- Take care of object life duration

```
Main(){
Constant* ptc = new Constant(4);

Constant a(3);
UnaryPlus plus_a(a);
}
```

```
UnaryPlus::UnaryPlus(Expression& pe)
{
  operand = &pe
}
```

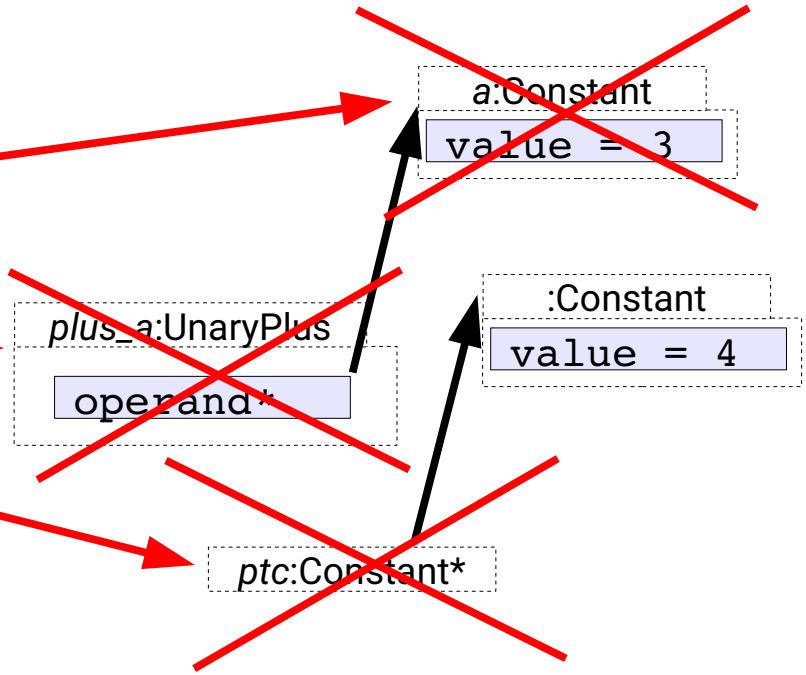


Object-Oriented concepts

- Take care of object life duration

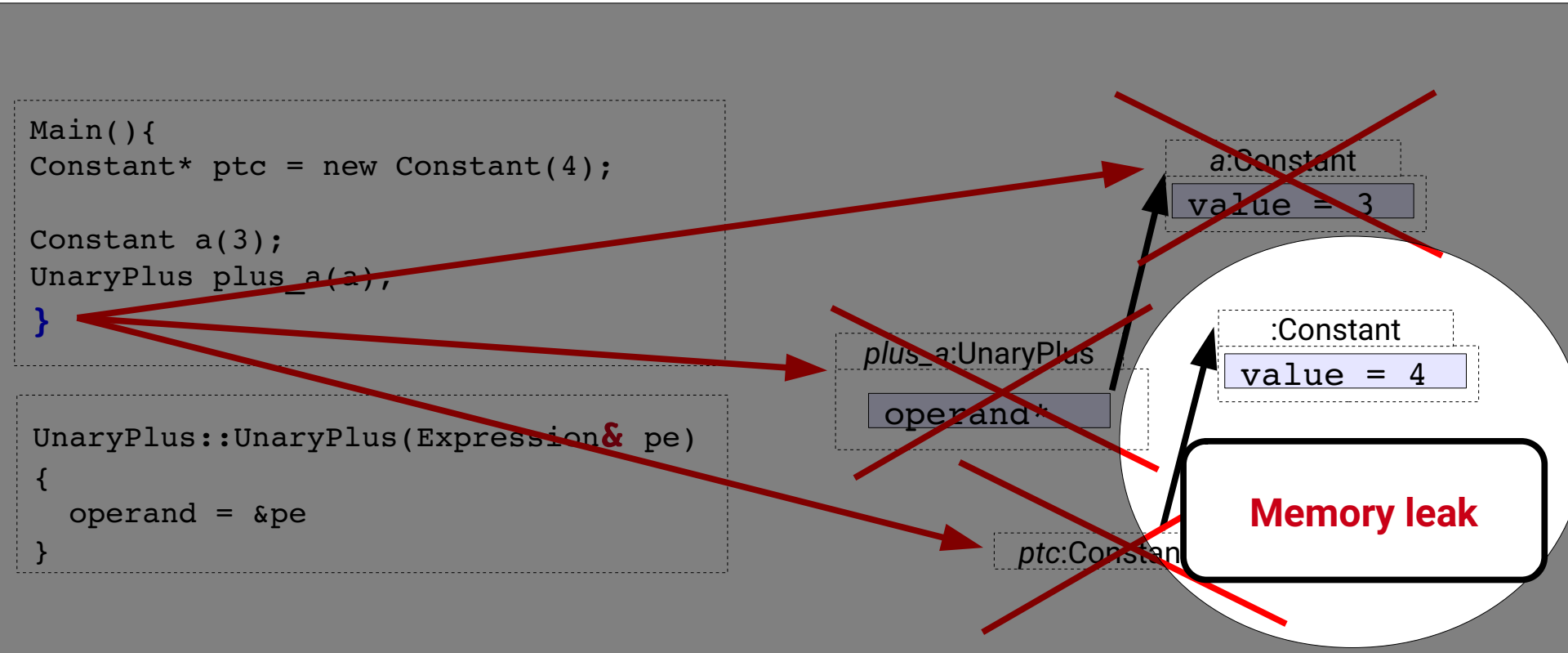
```
Main(){
Constant* ptc = new Constant(4);

Constant a(3);
UnaryPlus plus_a(a);
}
```

```
UnaryPlus::UnaryPlus(Expression& pe)
{
  operand = &pe
}
```



Object-Oriented concepts

- Take care of object life duration

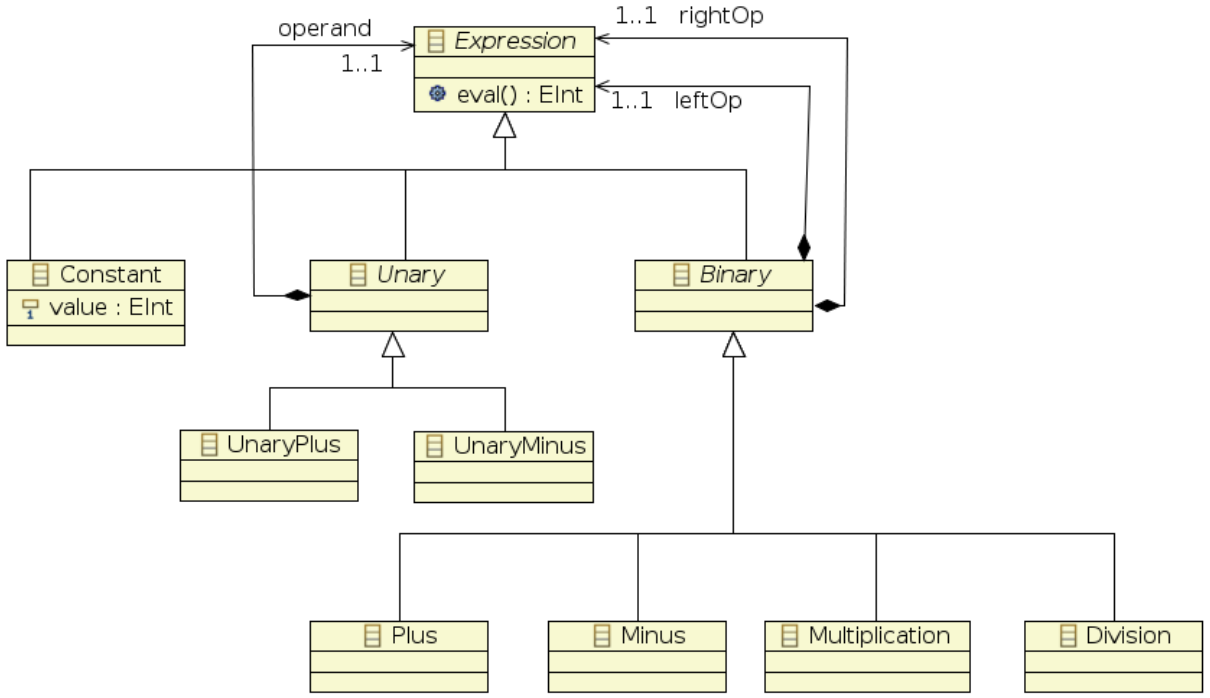


Object-Oriented concepts

- A class diagram gives
  - structural aspects
  - relational aspects

Almost equivalent to a set of .h files:

Expression.h
Unary.h
UnaryPlus.h
...

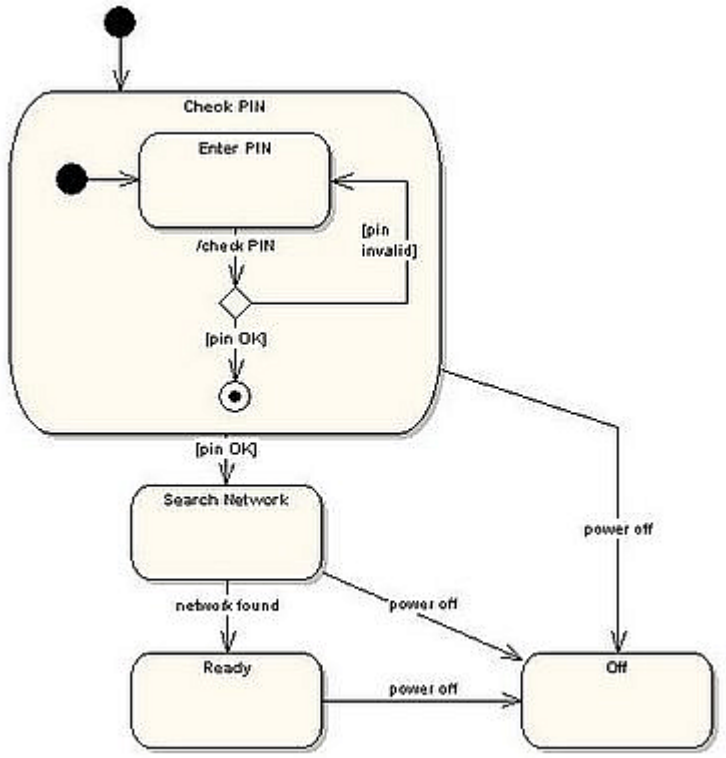


Be aware of previous comments !!

Object-Oriented concepts

- UML can give much more (caricature)
  - Behavioral aspects
- Textual specification

`myFunction()`
should
behave like that



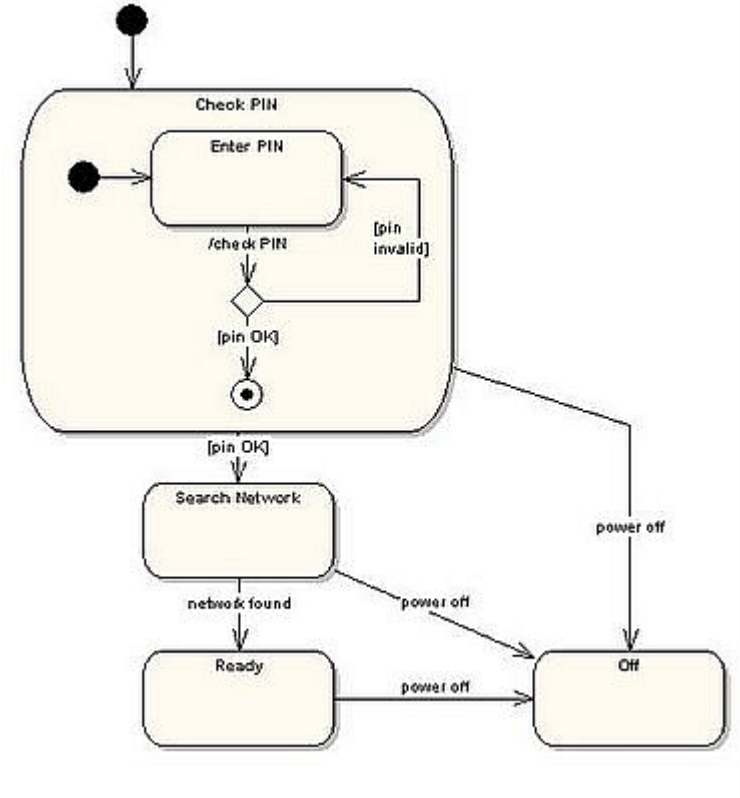
Object-Oriented concepts

- UML can give much more (caricature)
  - Behavioral aspects
- Textual specification

Almost equivalent to a set of .cpp files:

Expression.cpp
Unary.cpp
UnaryPlus.cpp
...

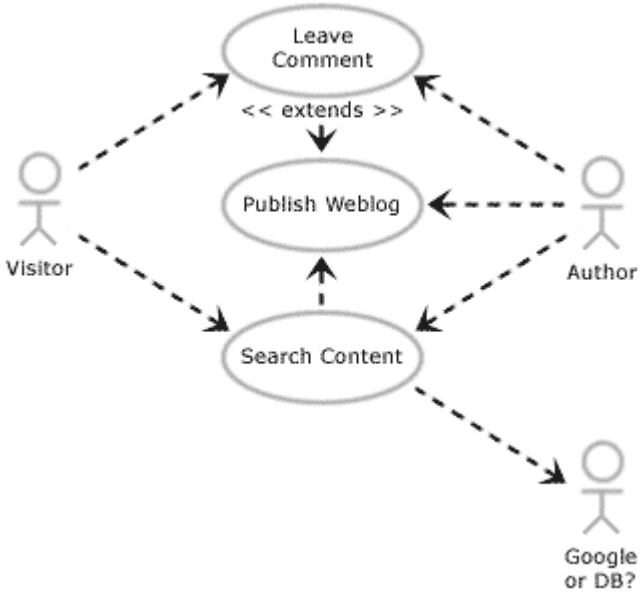
should



Object-Oriented concepts

- UML can give much more (caricature)
  - Behavioral aspects
  - …

Almost equivalent to
how the *main* uses objects:
system.h
system.cpp
main.cpp



The system behave like that

In a nutshell...

- Implementing a system:
  1. **Classes realization**
     - Declaration / definition of the class(es) --> .h
       - ≃ reflects the class diagram
     - Implementing the class(es) --> .cpp
       - ≃ implements the (member) function(s) (activity diagram, stateCharts, sequence diagram, ...)
  2. **`system` realization**
     - Make use of class(es)
       - Predefined classes (for instance the STL ones)
       - The one previously defined (at step 1.)
       - Definition / implementation / uses of a « system » class

TEST !!

In a nutshell...

- Implementing a system:
  1. **Classes realization**
     - Declaration / definition of the class(es) --> .h
       - ≃ reflects the class diagram
     - Implementing the class(es) --> .cpp
       - ≃ implements the (member) function(s) (activity diagram, stateCharts, sequence diagram, ...)
  2. **`system` realization**
     - Make use of class(es)
       - Predefined classes (**for instance the STL ones**)
       - The one previously defined (at step 1.)
       - Definition / implementation / uses of a « system » class

TEST !!

In a nutshell...

- Implementing a system:
  1. **Classes realization**
     - Declaration / definition of the class(es) --> .h
       - ≃ reflects the class diagram
     - Implementing the class(es) --> .cpp
       - ≃ implements the (member) function(s) (activity diagram, stateCharts, sequence diagram, ...)
  2. **`system` realization**
     - Make use of class(es)
       - Predefined classes (for instance the STL ones)
       - The one previously defined (at step 1.)
       - Definition / implementation / **uses of a « system » class**

TEST !!

In a nutshell...

- Implementing a system:
  1. **Classes realization**
     - Declaration / definition of the class(es) --> .h
       - ≃ reflects the class diagram
     - Implementing the class(es) --> .cpp
       - ≃ implements the (member) function(s) (activity diagram, stateCharts, sequence diagram, ...)
  2. **`system` realization**
     - Make use of class(es)
       - Predefined classes (for instance the STL ones)
       - The one previously defined (at step 1.)
       - Definition / implementation / **uses of a « system » class**