OBJECT ALGEBRAS

SOFTWARE DESIGN UND PROGRAMMIERTECHNIKEN

EARLIER IN THIS LECTURE

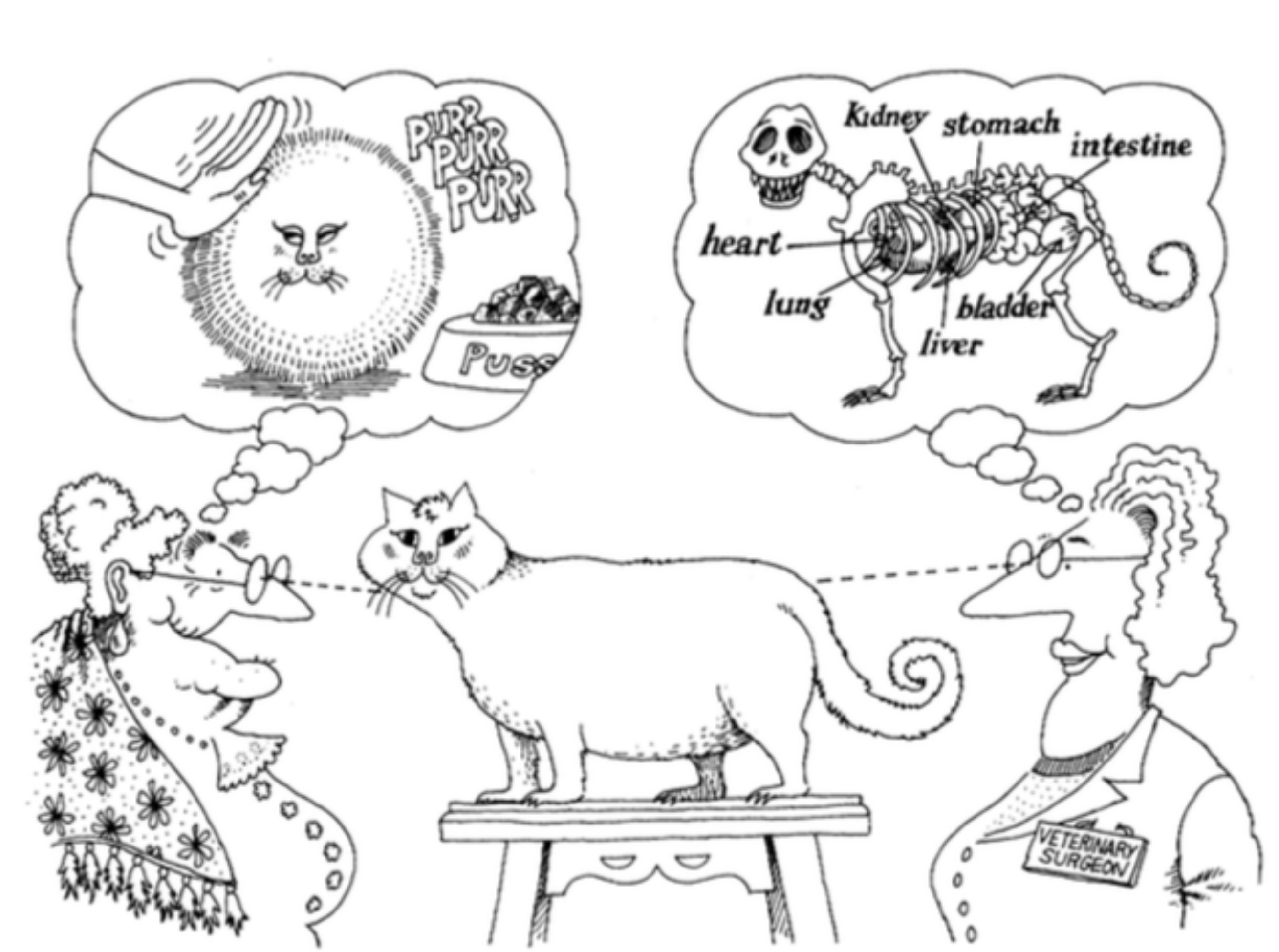
- The Open Closed Principle
- The Visitor Pattern
- The Abstract Factory Pattern

OPEN CLOSED PRINCIPLE

"An entity should be open for extend, but closed for modifications."

- In Object Oriented languages the main feature to support variations is subtyping.
- We have to decide which decomposition to choose.

OPEN CLOSED PRINCIPLE



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Recall:

"The tyranny of the dominant decomposition"

DECOMPOSITION BY EXAMPLE

- Let us assume we want to model a very simple language of arithmetic expressions with only
  - Literals
  - Addition
- Natural choice: represent the language by its abstract syntax tree.

```
interface Exp {
  int eval();
}
class Lit implements Exp {
  int value;
  int eval() { return value; }
}
class Add implements Exp {
  Exp lhs, rhs;
  int eval() { return lhs.eval() + rhs.eval(); }
}
```

USAGE EXAMPLE

To create the term "1 + 2" we create the syntax tree:

```
Exp e = new Add(new Lit(1), new Lit(2));
```

USAGE EXAMPLE

To create the term "1 + 2" we create the syntax tree:

```
Exp e = new Add(new Lit(1), new Lit(2));
```

We can evaluate the term by calling "eval".

```
println(e.eval());    // prints 3
```

EVALUATE THE DESIGN

What do you think of the design?

- Is it possible to add new variants (e.g. multiplication) according to OCP?
- Is it possible to add new operations (e.g. pretty printing)?

NEW VARIANTS

Adding **new variants** of expressions is easy since we structured our class hierarchy according to the structure of arithmetic expressions.

```
class Mul implements Exp {
  Exp lhs, rhs;
  int eval() { … }
}
```

NEW OPERATIONS

Adding **new operations** on the other hand requires changes of the existing code and thus does not comply to OCP.

```
interface Exp {
  int eval();
  String pretty();
}
...
```

```
...
class Lit implements Exp {
  int value;
  int eval() { return value; }
  String pretty() { return value.toString(); }
}
class Add implements Exp {
  Exp lhs, rhs;
  int eval() { return lhs.eval() + rhs.eval(); }
  String pretty() {
    return lhs.pretty() + "+" + rhs.pretty();
  }
}
```

AN ALTERNATIVE DESIGN

```
interface ExpOp⟨R⟩ {
  R Lit(int n);
  R Add(R lhs, R rhs);
}
```

Structure the class hierarchy according to the operations.

AN ALTERNATIVE DESIGN

```
interface ExpOp⟨R⟩ {
  R Lit(int n);
  R Add(R lhs, R rhs);
}
```

Structure the class hierarchy according to the operations.

```
class Eval implements ExpOp⟨Integer⟩ {
  Integer Lit(int value) { return value; }
  Integer Add(Integer lhs, Integer rhs) { return lhs + rhs; }
}
```

AN ALTERNATIVE DESIGN

```
interface ExpOp⟨R⟩ {
  R Lit(int n);
  R Add(R lhs, R rhs);
}
```

Does this remind you of something?

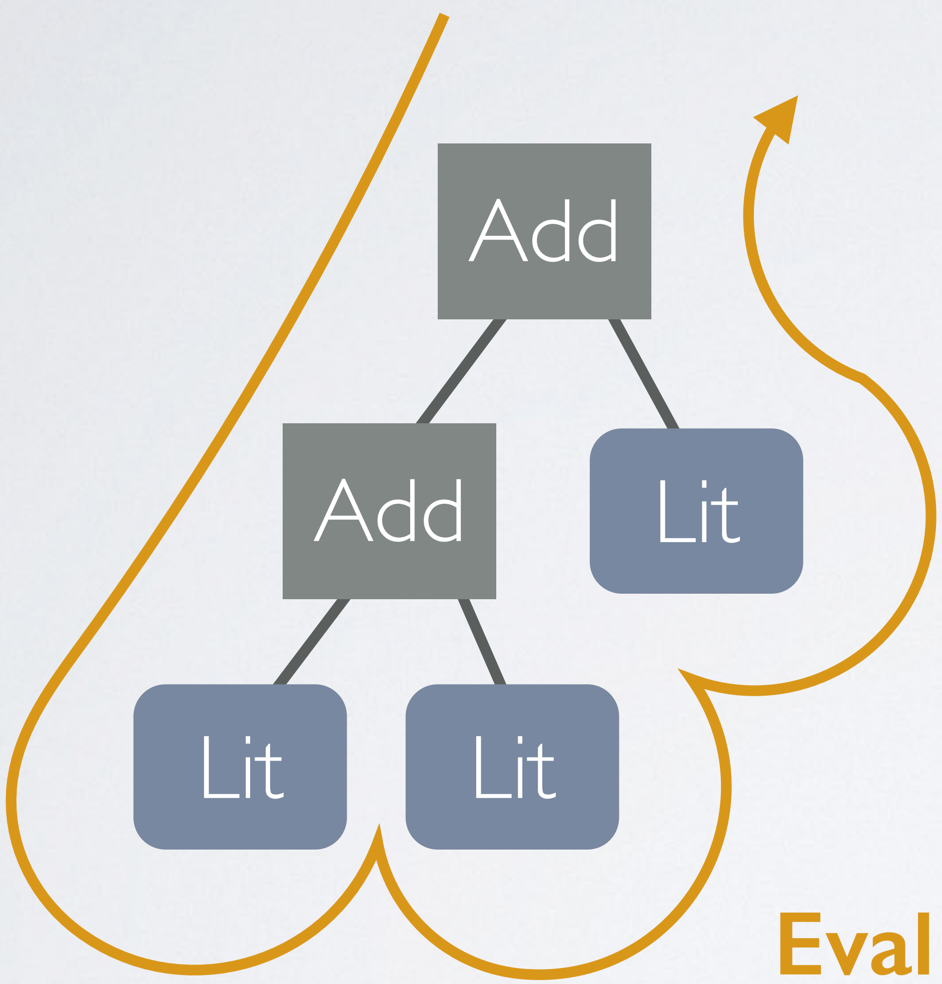
AN ALTERNATIVE DESIGN

```
interface ExpOp⟨R⟩ {
  R Lit(int n);
  R Add(R lhs, R rhs);
}
```

Does this remind you of something?

This is exactly the interface for the **internal visitor** we have seen in the exercise session.

THE VISITOR PATTERN




```
class Eval implements ExpOp⟨Integer⟩ {
  Lit(int value) { return value; }
  Add(Integer lhs, Integer rhs) { return lhs + rhs; }
}
```

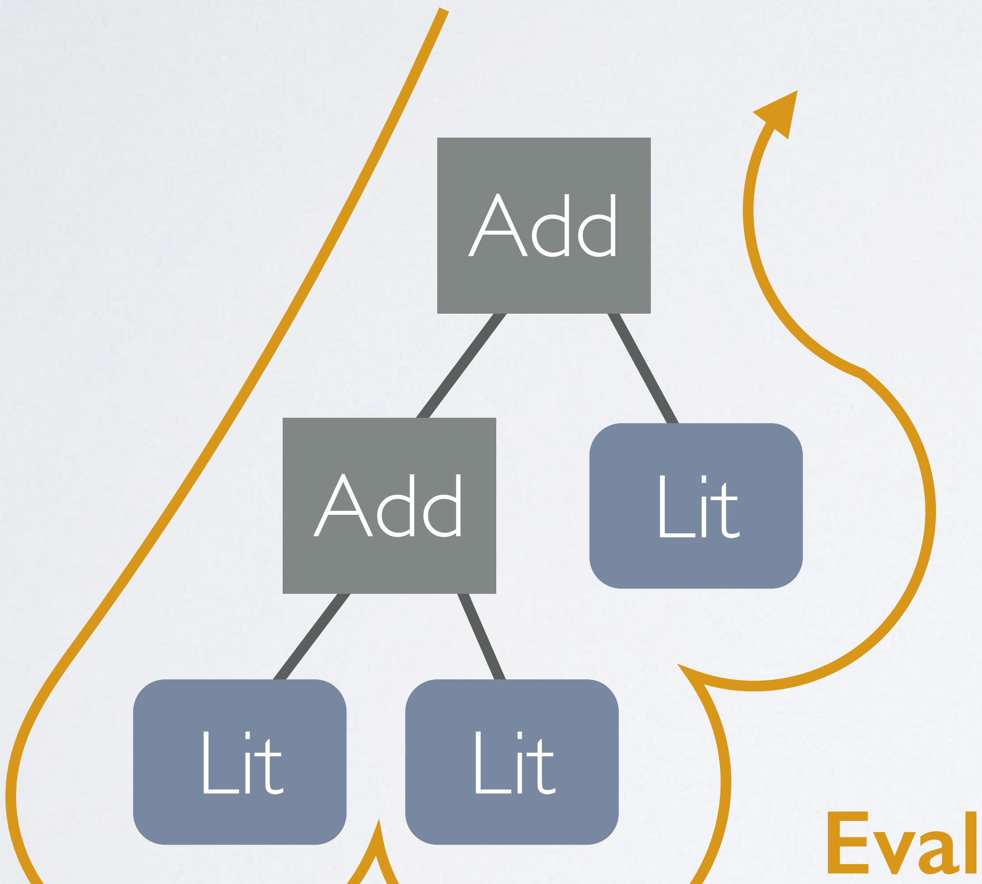
Implements operations on fixed class hierarchies as traversals providing one method for every variant we might encounter during the traversal.

THE VISITOR PATTERN

- The interface of the visitor can be derived from the class hierarchy:

THE VISITOR PATTERN

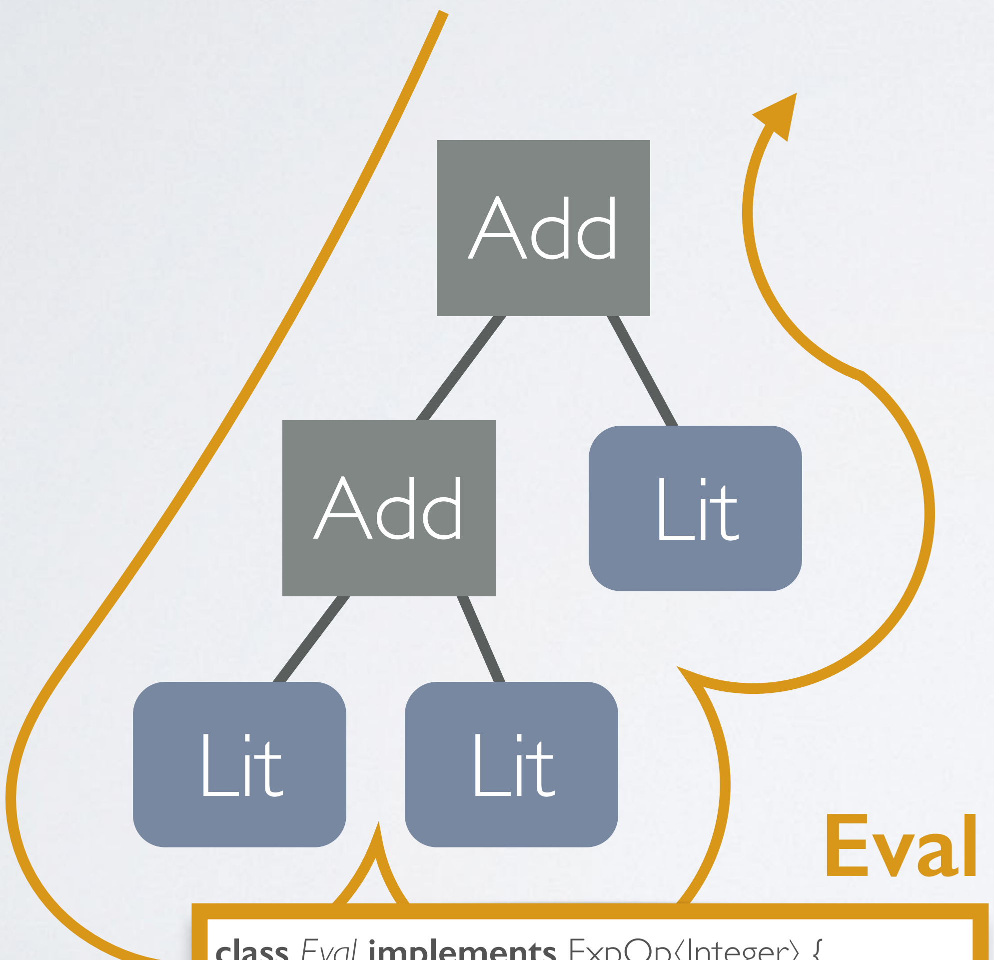
- As a result: Adding new operations is simple, since it corresponds to adding a new visitor implementation.



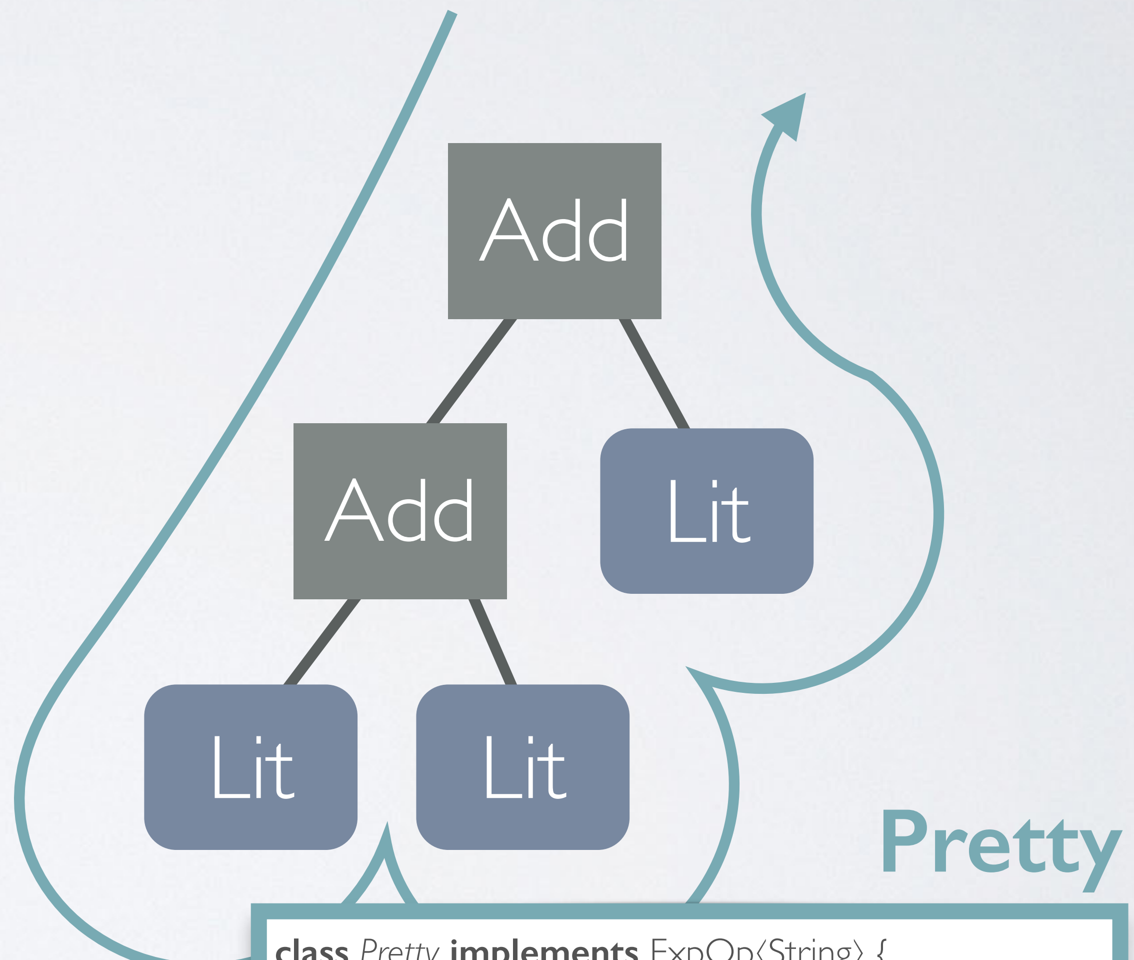

```
class Eval implements ExpOp⟨Integer⟩ {
  Lit(int value) { return value; }
  Add(Integer lhs, Integer rhs) { return lhs + rhs; }
}
```

THE VISITOR PATTERN

- As a result: Adding new operations is simple, since it corresponds to adding a new visitor implementation.




```
class Eval implements ExpOp⟨Integer⟩ {
  Lit(int value) { return value; }
  Add(Integer lhs, Integer rhs) { return lhs + rhs; }
}
```




```
class Pretty implements ExpOp⟨String⟩ {
  Lit(int value) { return value.toString; }
  Add(String lhs, String rhs) { return lhs + "+" + rhs; }
}
```

PROBLEMS WITH VISITORS

- Adds the "conceptual complexity" of double dispatch
- Only convenient to use as internal visitor or with "accept methods" implemented in the class hierarchy.
- Now it is **difficult to add new variants** to the class hierarchy (e.g. multiplication)

THE EXPRESSION PROBLEM

```
                    The Expression Problem
               Philip Wadler, 12 November 1998

The Expression Problem is a new name for an old problem. The goal is
to define a datatype by cases, where one can add new cases to the
datatype and new functions over the datatype, without recompiling
existing code, and while retaining static type safety (e.g., no
casts).
…
```

THE EXPRESSION PROBLEM (2)

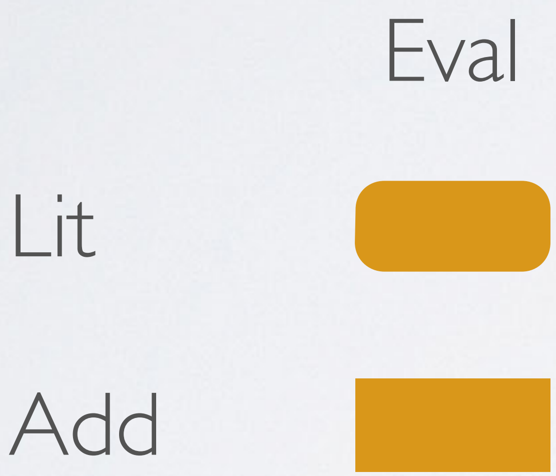
Zenger and Odersky [1] added an additional criterion to the expression problem:

- Extensibility in the dimension of variants as well as operations
- Static type safety
- No modification or duplication
- **Independent extensibility**

[1] Zenger, M., Odersky, M.: *Independently extensible solutions to the expression problem*. FOOL 2005.

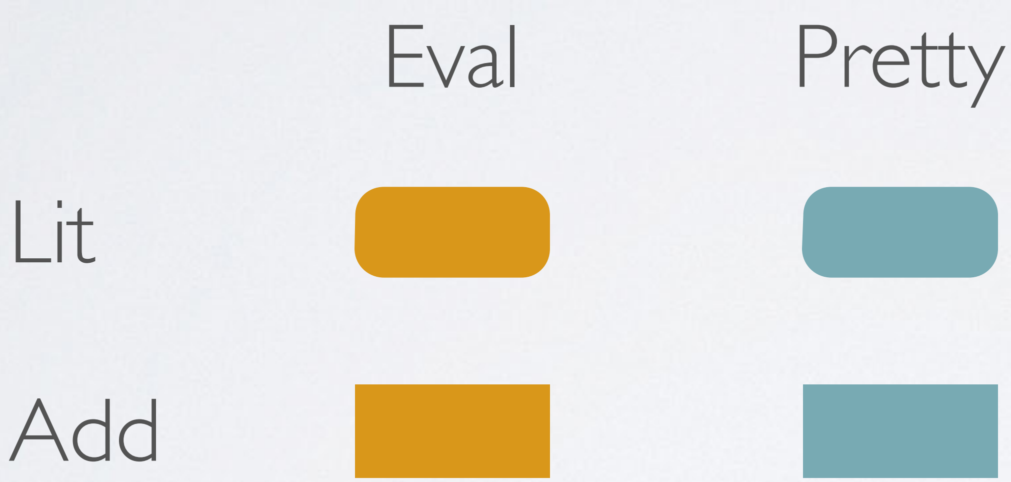
INDEPENDENT EXTENSIBILITY

It should be possible to define extensions in separate modules and later combine them.



INDEPENDENT EXTENSIBILITY

It should be possible to define extensions in separate modules and later combine them.



INDEPENDENT EXTENSIBILITY

It should be possible to define extensions in separate modules and later combine them.

INDEPENDENT EXTENSIBILITY

It should be possible to define extensions in separate modules and later combine them.

INDEPENDENT EXTENSIBILITY

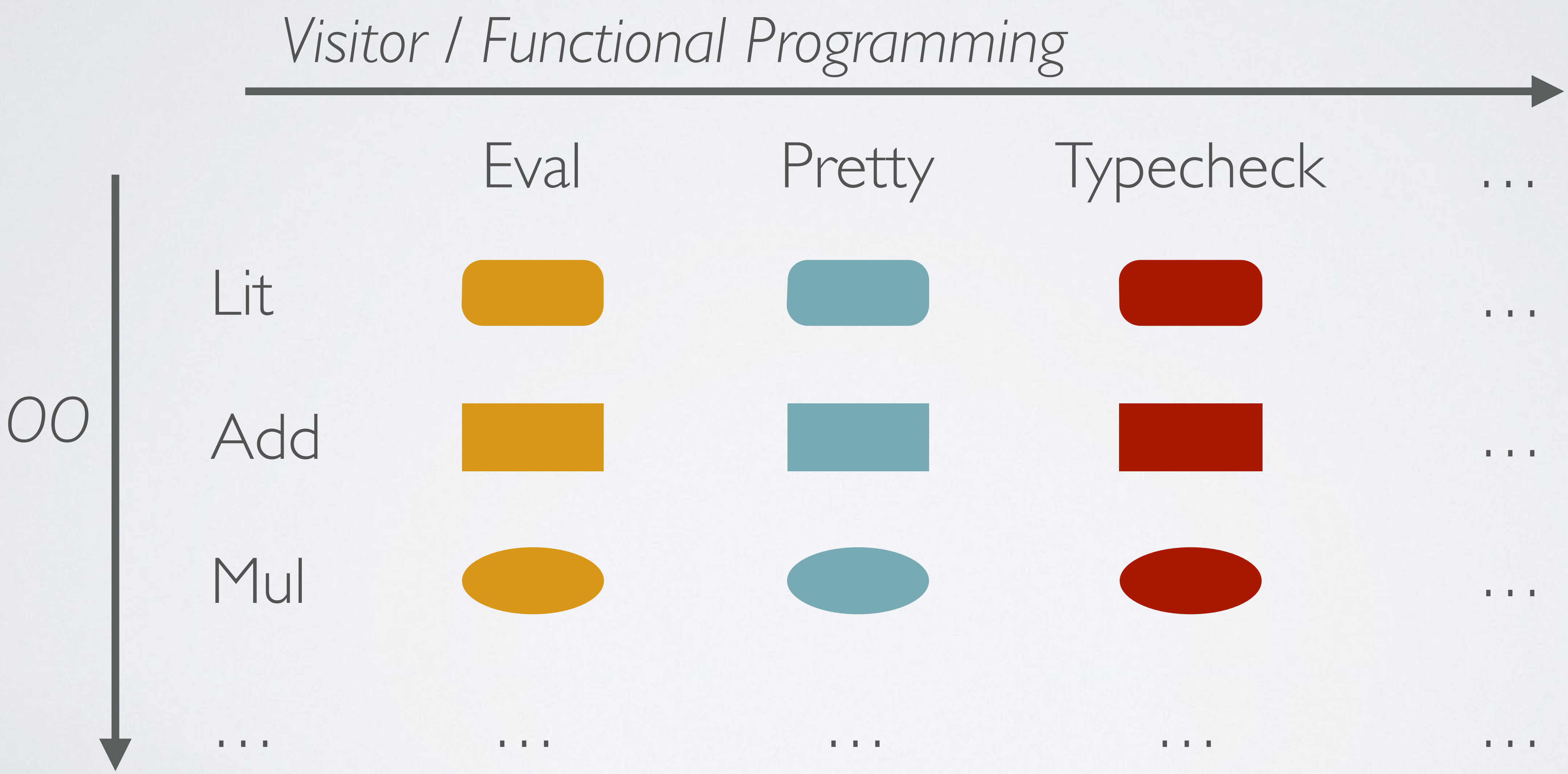
It should be possible to define extensions in separate modules and later combine them.

INDEPENDENT EXTENSIBILITY

It should be possible to define extensions in separate modules and later combine them.

INDEPENDENT EXTENSIBILITY

It should be possible to define extensions in separate modules and later combine them.



OBJECT ALGEBRAS

A Solution to the Expression Problem

REMINDER: THE ABSTRACT FACTORY PATTERN

Instead of explicitly creating an instance of a class and binding to a specific implementation, we abstract over the creation by calling a factory method.

REMINDER: THE ABSTRACT FACTORY PATTERN

Instead of explicitly creating an instance of a class and binding to a specific implementation, we abstract over the creation by calling a factory method.

```
new Foo();
```

```
factory.Foo();
```

REMINDER: THE ABSTRACT FACTORY PATTERN

Instead of explicitly creating an instance of a class and binding to a specific implementation, we abstract over the creation by calling a factory method.

new Foo(); → factory.Foo();

This allows selecting the actual representation at runtime by passing the corresponding abstract factory.

EXAMPLE FACTORY

The abstract factory for creating expressions has the following interface:

```
interface ExpFactory {
  Exp Lit(int value);
  Exp Add(Exp lhs, Exp rhs);
}
```

USAGE EXAMPLE

To construct the term “1 + 2” we now call the factory methods:

```
Exp term(ExpFactory factory) {
  return factory.Add(factory.Lit(1), factory.Lit(2))
}
```

GENERALIZED FACTORY

```
interface ExpFactory {
  Exp Lit(int value);
  Exp Add(Exp lhs, Exp rhs);
}
```

The factory is however “limited” to only produce expressions.

GENERALIZED FACTORY

```
interface ExpFactory {
  Exp Lit(int value);
  Exp Add(Exp lhs, Exp rhs);
}
```

The factory is however "limited" to only produce expressions.

```
interface ExpFactoryG⟨E⟩ {
  E Lit(int value);
  E Add(E lhs, E rhs);
}
```

We can generalize to a factory that can produce arbitrary values.

GENERALIZED FACTORY

We can recover the abstract factory by instantiating E with Exp:

interface *ExpFactory* **extends** ExpFactoryG⟨Exp⟩ {}

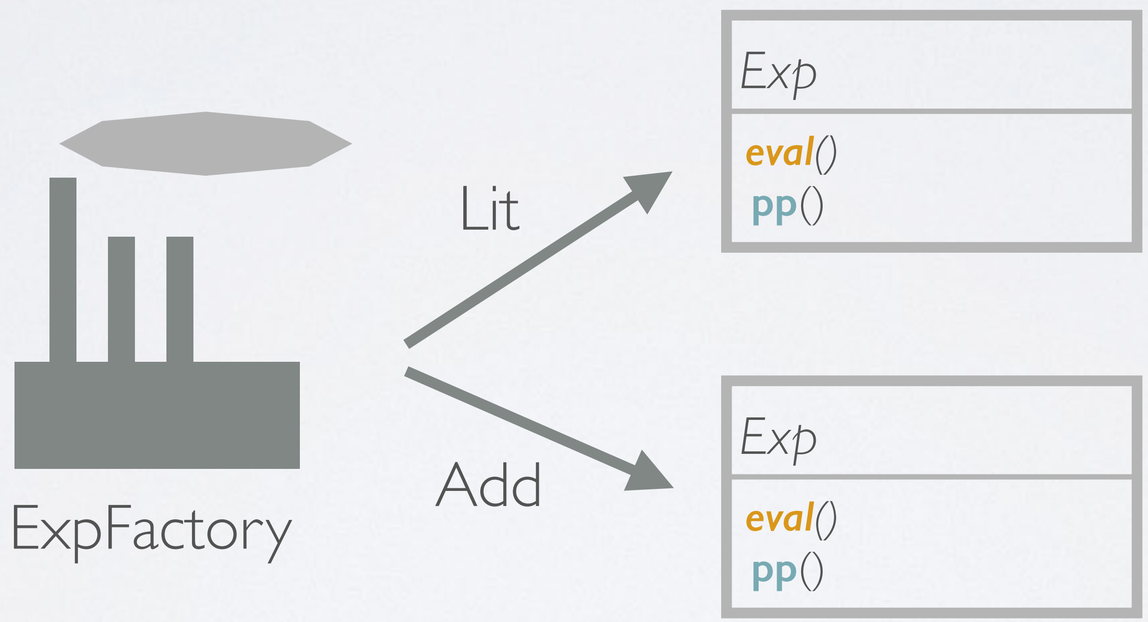
MODULARIZING FACTORIES

interface *ExpFactory* **extends** ExpFactoryG⟨Exp⟩ {}

MODULARIZING FACTORIES

interface *ExpFactory* **extends** ExpFactoryG⟨Exp⟩ {}

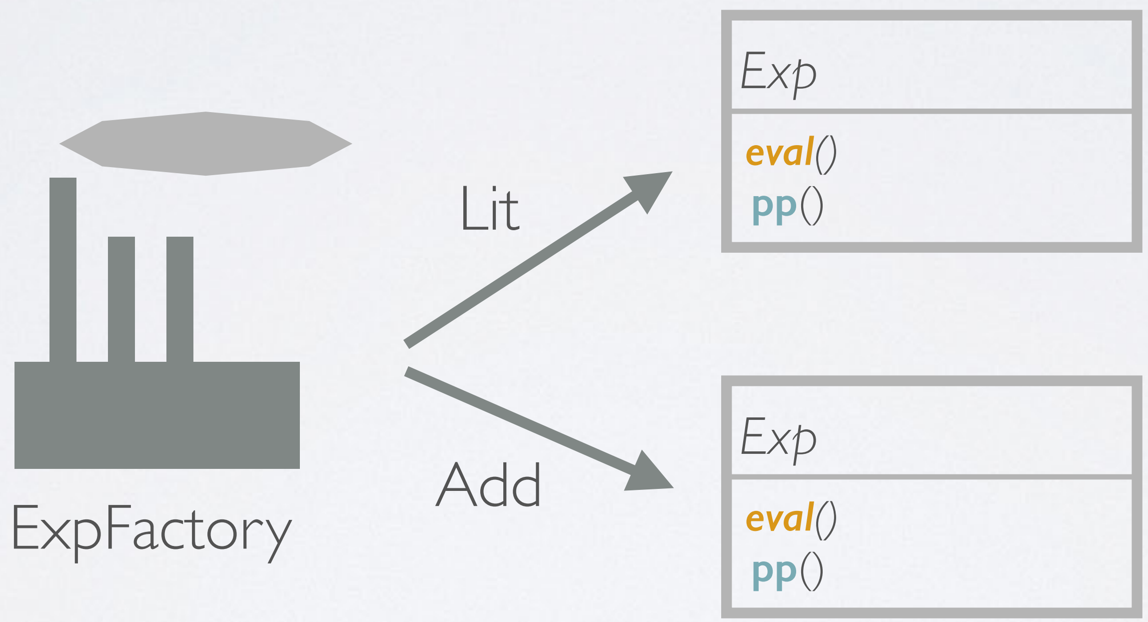
ExpFactory is a factory that produces objects of type Exp.



MODULARIZING FACTORIES

interface *ExpFactory* **extends** ExpFactoryG⟨Exp⟩ {}

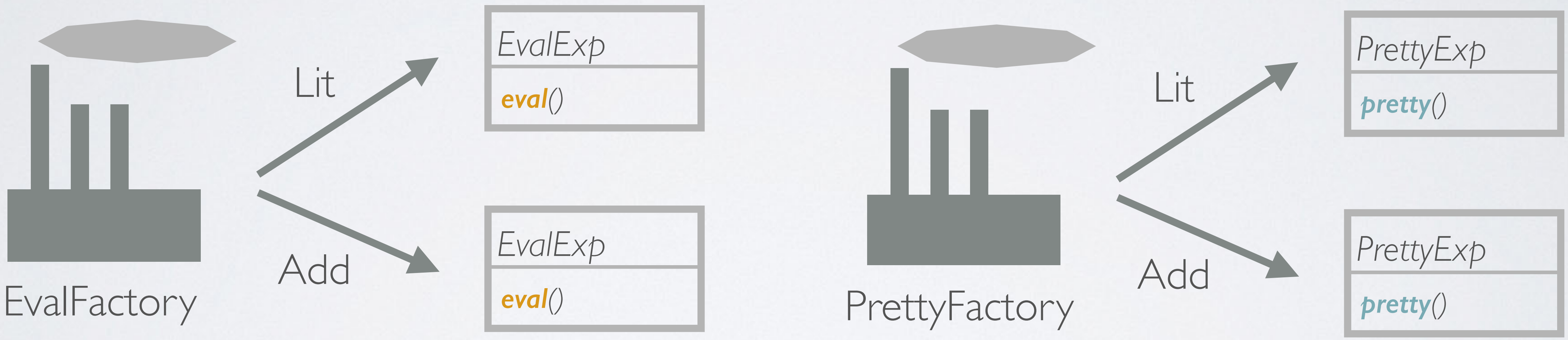
ExpFactory is a factory that produces objects of type Exp.



It monolithically implements all operations on expressions.

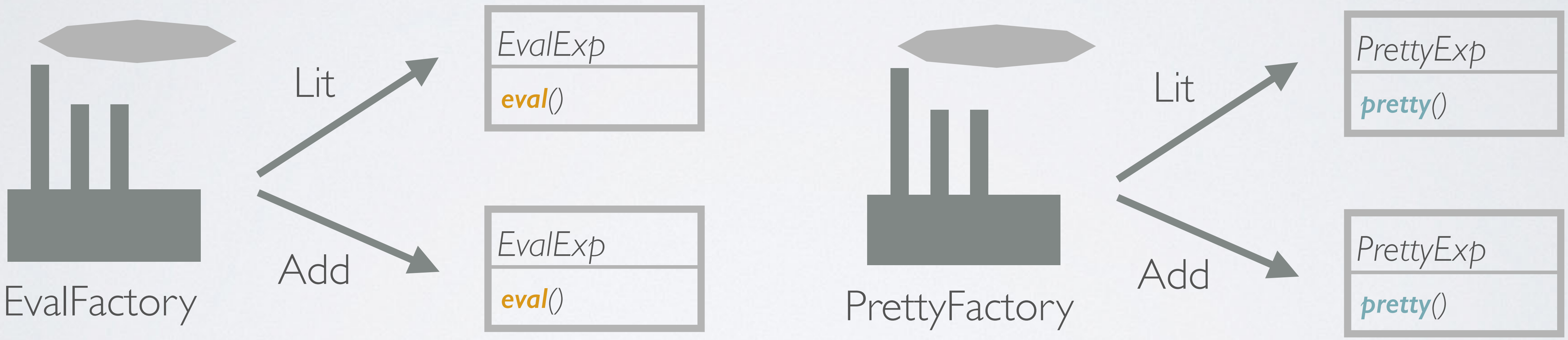
MODULARIZING FACTORIES

Let's split it into multiple factories where each one only produces a slice of the interface.



MODULARIZING FACTORIES

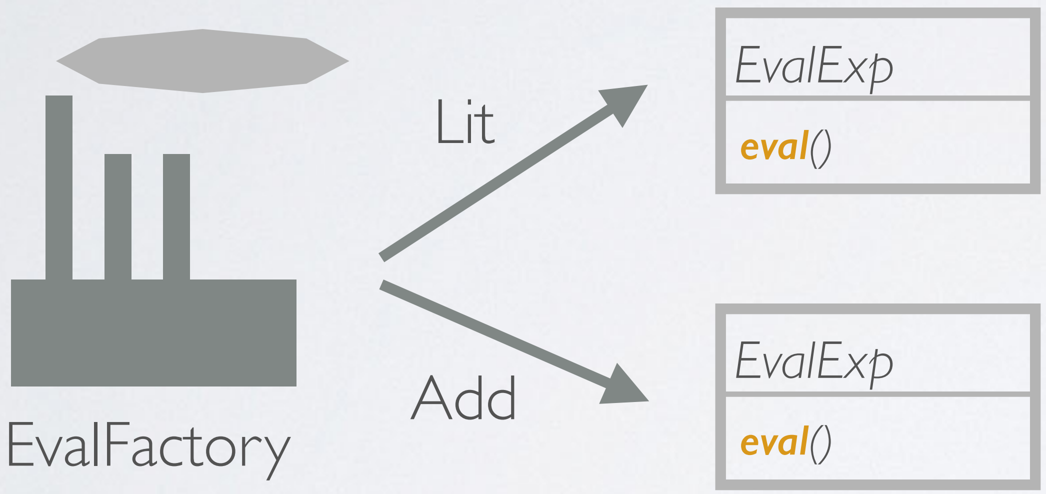
Let's split it into multiple factories where each one only produces a slice of the interface.

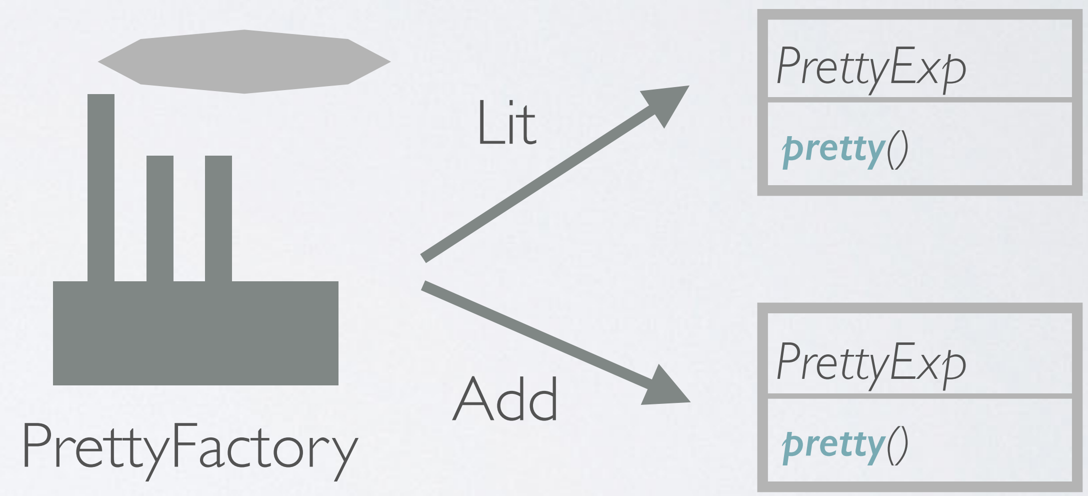



```
interface EvalExp {  int eval();  }
```

MODULARIZING FACTORIES

Let's split it into multiple factories where each one only produces a slice of the interface.





interface *EvalExp* { int eval(); }

interface *PrettyExp* { String pretty(); }

THE EVAL-FACTORY

Let us define the factory that builds expressions that can be evaluated.

1.

```
class EvalFactory implements ExpFactoryG⟨EvalExp⟩ {
  EvalExp Lit(int value) { return new EvalLit(int); }
  EvalExp Add(EvalExp lhs, EvalExp rhs) {
    return new EvalAdd(lhs, rhs);
  }
}
```

THE EVAL-FACTORY

2.

```
class EvalLit implements EvalExp {
  int value;
  int eval() { return value; }
}
```

THE EVAL-FACTORY

2.

```
class EvalLit implements EvalExp {
  int value;
  int eval() { return value; }
}
```

3.

```
class EvalAdd implements EvalExp {
  EvalExp lhs, rhs;
  int eval() { return lhs.eval() + rhs.eval(); }
}
```

USING THE EVAL FACTORY

A term now also needs to be parameterized in "what kind of specialized Exp-object" the factory will construct:

```
⟨E⟩ E term(ExpFactoryG⟨E⟩ factory) {
  return factory.Add(factory.Lit(1), factory.Lit(2))
}
```

USING THE EVAL FACTORY

A term now also needs to be parameterized in "what kind of specialized Exp-object" the factory will construct:

```
⟨E⟩ E term(ExpFactoryG⟨E⟩ factory) {
  return factory.Add(factory.Lit(1), factory.Lit(2))
}
```

```
println(term(new EvalFactory()).eval())   // prints 3
```

ADDING NEW OPERATIONS

New **operations** can now be added by implementing another factory:

```
class PrettyFactory implements ExpFactoryG<PrettyExp> {
  PrettyExp Lit(int value) { return new PrettyLit(int);}
  PrettyExp Add(PrettyExp lhs, PrettyExp rhs) {
    return new PrettyAdd(lhs, rhs);
  }
}
```

ADDING NEW VARIANTS

New **variants** of expressions can be added by extending the factory interface and adding a factory method for the variant:

```
interface MulExpFactoryG⟨E⟩ extends ExpFactoryG⟨E⟩ {
  E Mul(E lhs, E rhs);
}
```

ADDING NEW VARIANTS

The missing case for "Mul" can be added without duplication or modification of code:

```
class EvalMulFactory extends EvalFactory
    implements MulExpFactoryG⟨EvalExp⟩ {
  EvalExp Mul(EvalExp lhs, EvalExp rhs) {
    return new EvalMul(lhs, rhs);
  }
}
```

INDEPENDENT EXTENSIBILITY, REVISITED

Every combination of operation and variant can be defined in its own module (e.g. EvalAdd, PrettyAdd, EvalLit, …).

	Eval
Lit	EvalLit
Add	EvalAdd

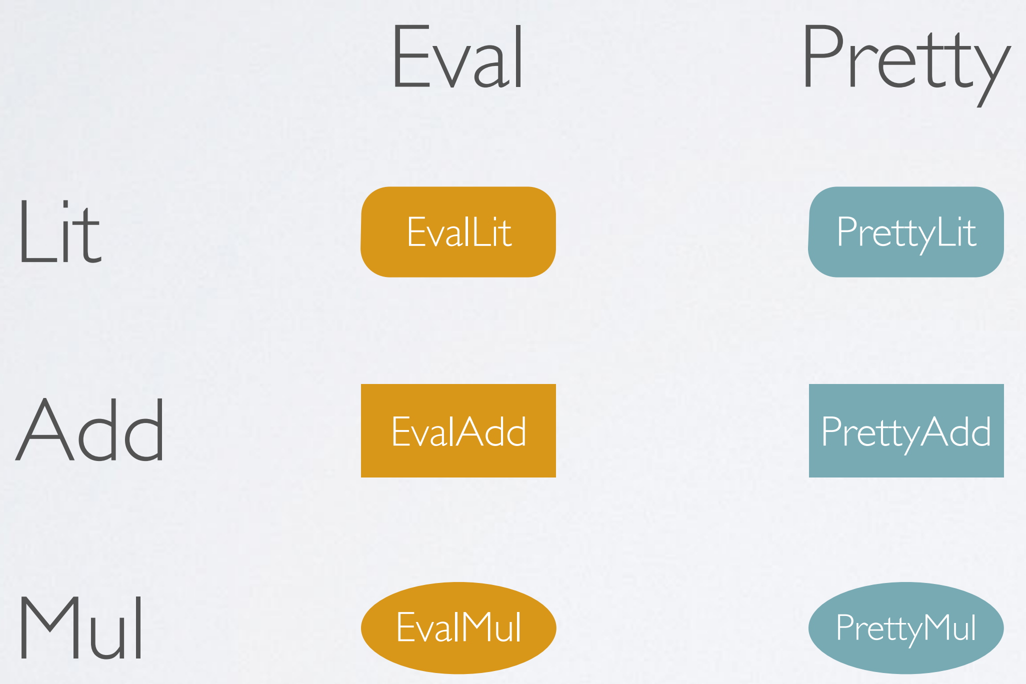
INDEPENDENT EXTENSIBILITY, REVISITED

Every combination of operation and variant can be defined in its own module (e.g. EvalAdd, PrettyAdd, EvalLit, …).

	Eval	Pretty
Lit	EvalLit	PrettyLit
Add	EvalAdd	PrettyAdd

INDEPENDENT EXTENSIBILITY, REVISITED

Every combination of operation and variant can be defined in its own module (e.g. EvalAdd, PrettyAdd, EvalLit, …).



INDEPENDENT EXTENSIBILITY, REVISITED

Every combination of operation and variant can be defined in its own module (e.g. EvalAdd, PrettyAdd, EvalLit, …).

	Eval	Pretty	Typecheck
Lit	EvalLit	PrettyLit	TCLit
Add	EvalAdd	PrettyAdd	TCAdd
Mul	EvalMul	PrettyMul	TCMul

INDEPENDENT EXTENSIBILITY, REVISITED

Every combination of operation and variant can be defined in its own module (e.g. EvalAdd, PrettyAdd, EvalLit, …).

	Eval	Pretty	Typecheck	…
Lit	EvalLit	PrettyLit	TCLit	…
Add	EvalAdd	PrettyAdd	TCAdd	…
Mul	EvalMul	PrettyMul	TCMul	…
…	…	…	…	…

INDEPENDENT EXTENSIBILITY, REVISITED

Every combination of operation and variant can be defined in its own module (e.g. EvalAdd, PrettyAdd, EvalLit, …).

		Eval	Pretty	Typecheck	…
	Lit	EvalLit	PrettyLit	TCLit	…
OO	Add	EvalAdd	PrettyAdd	TCAdd	…
	Mul	EvalMul	PrettyMul	TCMul	…
	…	…	…	…	…

INDEPENDENT EXTENSIBILITY, REVISITED

Every combination of operation and variant can be defined in its own module (e.g. EvalAdd, PrettyAdd, EvalLit, …).

Visitor / Functional Programming →

OO ↓

	Eval	Pretty	Typecheck	…
Lit	EvalLit	PrettyLit	TCLit	…
Add	EvalAdd	PrettyAdd	TCAdd	…
Mul	EvalMul	PrettyMul	TCMul	…
…	…	…	…	…

INTERNAL VISITORS VS. GENERALIZED FACTORIES

Compare the interfaces of internal visitors and generalized factories:

```
interface ExpOp⟨R⟩ {
  R Lit(int value);
  R Add(R lhs, R rhs);
}
```

```
interface ExpFactoryG⟨E⟩ {
  E Lit(int value);
  E Add(E lhs, E rhs);
}
```

INTERNAL VISITORS VS. GENERALIZED FACTORIES

Compare the interfaces of internal visitors and generalized factories:

```
interface ExpOp⟨R⟩ {
  R Lit(int value);
  R Add(R lhs, R rhs);
}
```

```
interface ExpFactoryG⟨E⟩ {
  E Lit(int value);
  E Add(E lhs, E rhs);
}
```

➡ It is the same! This is what we call the **signature** of an object algebra.

CHURCH ENCODING

For the FP-inclined: A program parameterized by an object algebra has the type:

```
⟨E⟩ E term(ExpFactoryG⟨E⟩ f) {
  return f.Add(f.Lit(1), f.Lit(2))
}
```

CHURCH ENCODING

For the FP-inclined: A program parameterized by an object algebra has the type:

```
⟨E⟩ E term(ExpFactoryG⟨E⟩ f) {
  return f.Add(f.Lit(1), f.Lit(2))
}
```

:

$\forall$ E. ExpFactoryG⟨E⟩ → E

CHURCH ENCODING

For the FP-inclined: A program parameterized by an object algebra has the type:

```
⟨E⟩ E term(ExpFactoryG⟨E⟩ f) {
  return f.Add(f.Lit(1), f.Lit(2))
}
```

:

$\forall$ E. ExpFactoryG⟨E⟩ $\rightarrow$ E

```
interface ExpFactoryG⟨E⟩ {
  E Lit(int value);
  E Add(E lhs, E rhs);
}
```

CHURCH ENCODING

For the FP-inclined: A program parameterized by an object algebra has the type:

```
⟨E⟩ E term(ExpFactoryG⟨E⟩ f) {
  return f.Add(f.Lit(1), f.Lit(2))
}
```

:

$\forall$ E. ExpFactoryG⟨E⟩ → E

```
interface ExpAlg⟨E⟩ {
  Lit: int → E
  Add: (E, E) → E
}
```

≅

```
interface ExpFactoryG⟨E⟩ {
  E Lit(int value);
  E Add(E lhs, E rhs);
}
```

CHURCH ENCODING

For the FP-inclined: A program parameterized by an object algebra has the type:

$\forall$ E. ExpFactoryG⟨E⟩ ➞ E

```
interface ExpAlg⟨E⟩ {
  Lit: int ➞ E
  Add: (E, E) ➞ E
}
```

≅

```
interface ExpFactoryG⟨E⟩ {
  E Lit(int value);
  E Add(E lhs, E rhs);
}
```

CHURCH ENCODING

For the FP-inclined: A program parameterized by an object algebra has the type:

$$\forall\, E.\ (\text{int} \rightarrow E,\ (E, E) \rightarrow E) \rightarrow E \quad \cong \quad \forall\, E.\ \underline{\text{ExpFactoryG}}\langle E\rangle \rightarrow E$$

```
interface ExpAlg⟨E⟩ {
  Lit: int → E
  Add: (E, E) → E
}
```

≅

```
interface ExpFactoryG⟨E⟩ {
  E Lit(int value);
  E Add(E lhs, E rhs);
}
```

CHURCH ENCODING

For the FP-inclined: A program parameterized by an object algebra has the type:

$\forall$ E. (int → E, (E, E) → E) → E $\cong$ $\forall$ E. ExpFactoryG⟨E⟩ → E

Church Encoding of Exp!

```
interface ExpAlg⟨E⟩ {
  Lit: int → E
  Add: (E, E) → E
}
```

$\cong$

```
interface ExpFactoryG⟨E⟩ {
  E Lit(int value);
  E Add(E lhs, E rhs);
}
```

OBJECT ALGEBRAS SUMMARIZED

- Replace instantiation of classes by calls to a **generic factory interface** — this allows selecting the factory later that implements the desired set of operations.
- New variants are added by defining a new interface that **extends the interface** of the generalized abstract factory.
- New operations are added by adding new classes that **implement the abstract factory**.
- Every variant-operation-combination can be defined in a **separate module**.

REFERENCES

Bruno C. d. S. Oliveira and William R. Cook. *Extensibility for the Masses — Practical Extensibility with Object Algebras*. ECOOP 2012

Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh and William R. Cook. *Feature-Oriented Programming with Object Algebras*. ECOOP 2013

Tijs van der Storm. *Who's afraid of Object Algebras*. Joy of Coding (2014). Talk available at http://www.infoq.com/presentations/object-algebras