Northeast Scala Symposium
Boston, January 2015

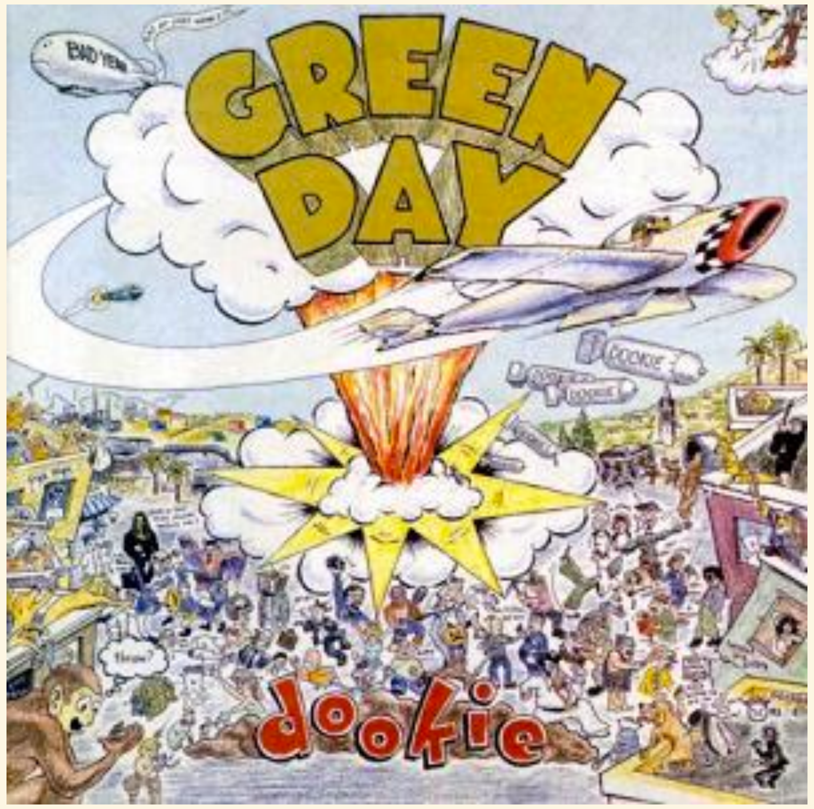
The Interpreter Pattern Revisited

Rúnar Óli Bjarnason
@runarorama

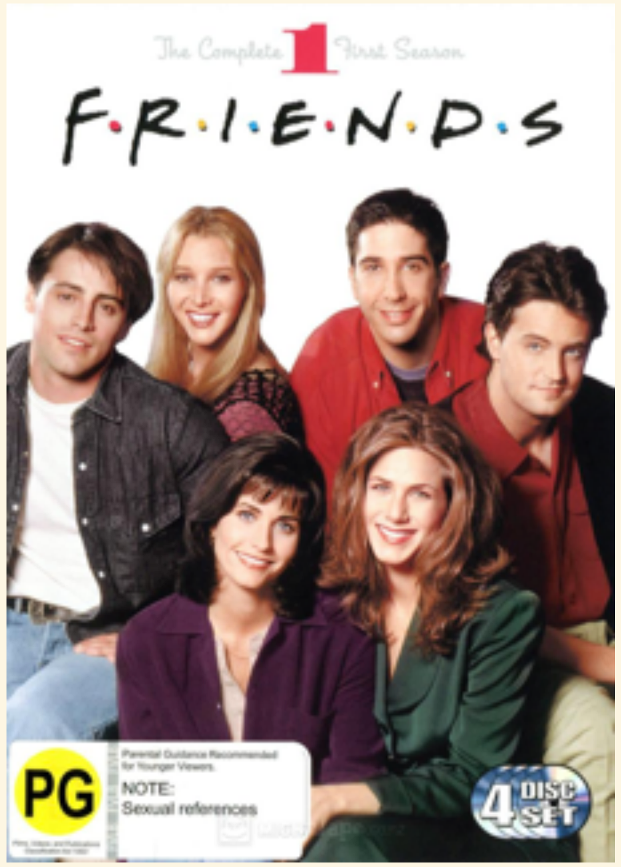
1994

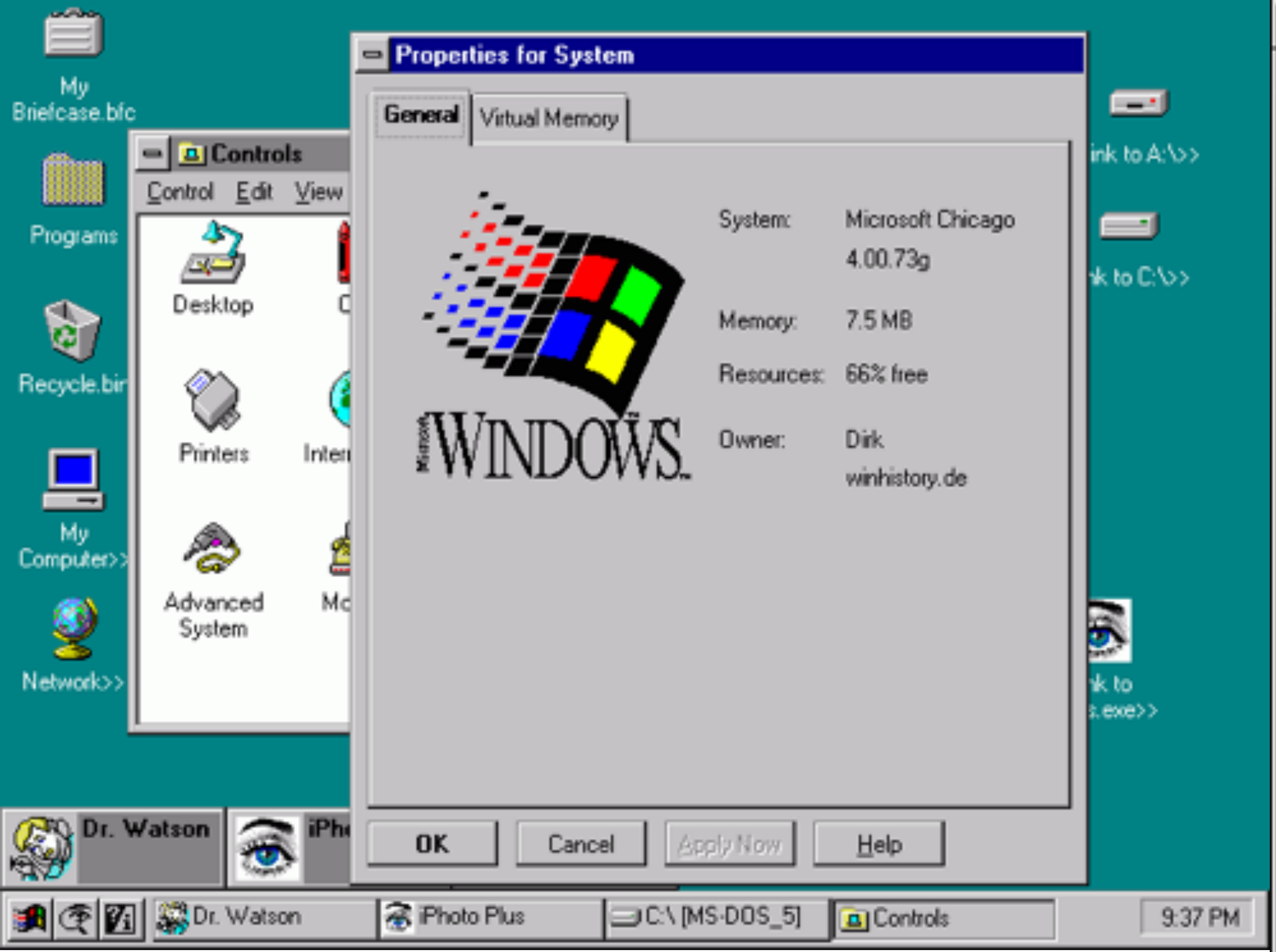
Did not yet exist:

- eBay
- Craigslist
- CGI movies
- Internet Explorer
- Java
- JavaScript
- Justin Bieber
- Netscape Navigator
- Nintendo 64
- PalmPilot
- Yahoo!



"Green Day - Dookie cover" by Source. Licensed under Fair use via Wikipedia - http://en.wikipedia.org/wiki/File:Green_Day_-_Dookie_cover.jpg#mediaviewer/File:Green_Day_-_Dookie_cover.jpg





AIR Mosaic
File Edit Options Navigate Help
Open Hotlist Back Forward Reload Home Find Kiosk Stop Add
NUM

Document Title: WebCrawler Searching

Document URL: http://www.biotech.washington.edu/WebCrawler/WebQuery.html

Search the Web

To search the WebCrawler database, type in your search keywords here. This database is indexed by content. That means that the contents of documents are indexed, not just their titles and URLs. Type as many relevant keywords as possible; it will help to uniquely identify what you're looking for.

[✓ AND words together] Search

If you're having trouble using the WebCrawler, here are some helpful hints for searching. Answers to frequently asked questions might help too. If the WebCrawler doesn't find what you want, try looking at some other Web Indexes. For more information on the WebCrawler, see the WebCrawler Home Page.

UNIFIED
MODELING
LANGUAGE

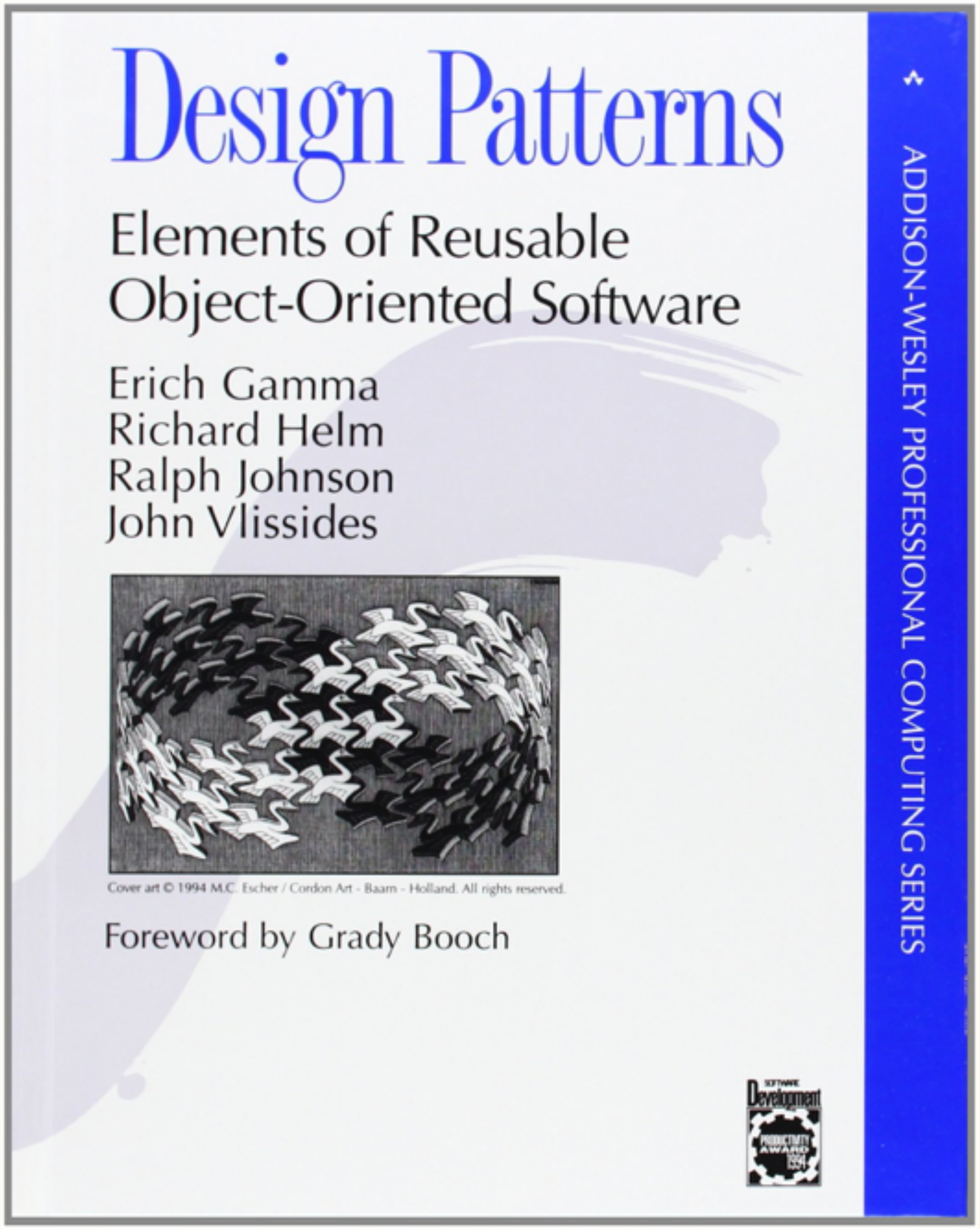

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Image: http://www.amazon.co.uk/Design-patterns-elements-reusable-object-oriented/dp/0201633612

Hits such as…

- Abstract Factory
- Prototype
- Singleton
- Proxy
- Iterator
- Strategy
- Template Method
- Visitor

Rare groove

Interpreter (243)

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

“Interpreter is one of the Design Patterns published in the GoF which is not really used.”

oodesign.com

“I'm not sure this pattern has wide applicability”

Dr. Jon Pearce
CS Department Chair
San Jose State University

“Of course I loved all of ***Design Patterns****, except for pages 243 to 256, which had the magic property of inducing a coma-like trance whenever I tried to skim through them. I could put on a black ninja suit and sneak through the building, and presuming I didn't get arrested, I could tear those pages out of every single copy of Design Patterns at Amazon, and almost nobody would notice.”*

Steve Yegge

“Interpreter is the only useful pattern in the GoF book.”

Rúnar

Motivation

- Take a problem that occurs often
- Express instances of it as sentences in a simple language
- Solve the problem by interpreting these sentences

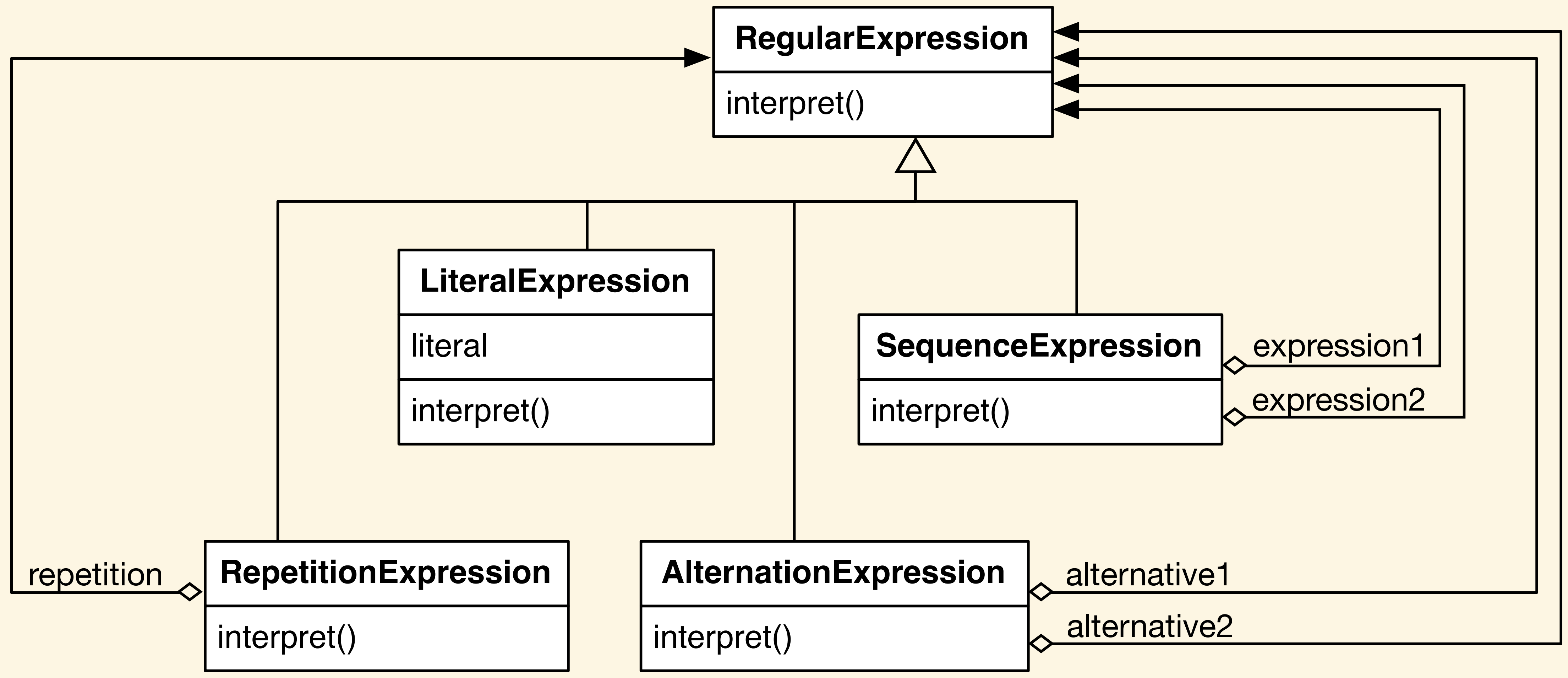
Example

- Searching for strings that match a pattern.
- Regular expressions are a standard language for specifying such patterns.
- Match strings by interpreting regular expressions.

Example

```
expression ::= literal | alternation | sequence | repetition |
               '(' expression ')'
alternation ::= expression '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ...}*
```

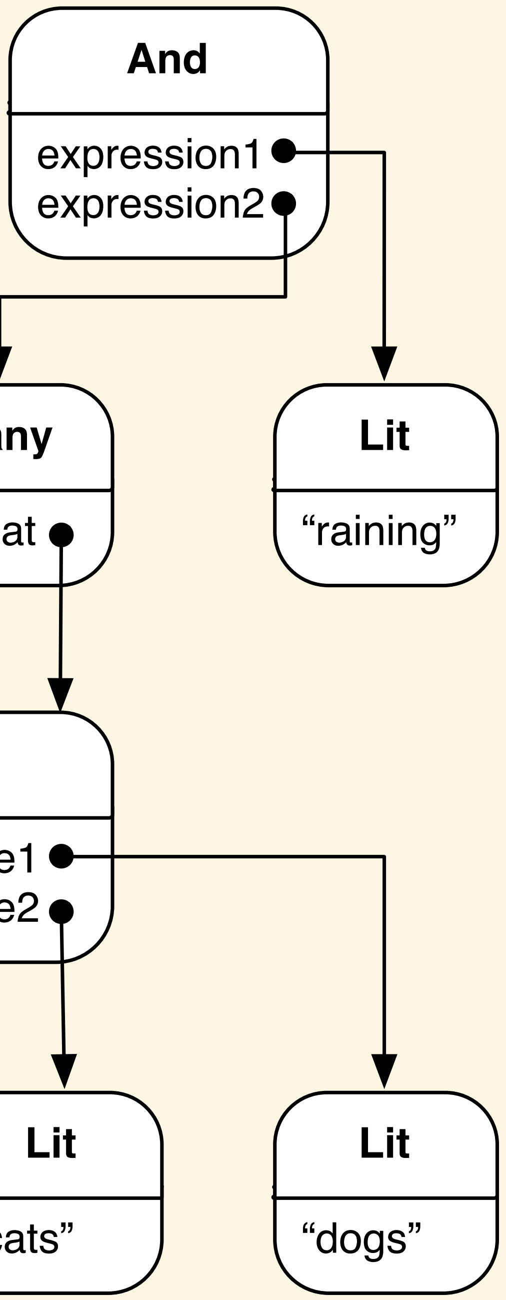
Example


RegularExpression
interpret()
LiteralExpression
literal
interpret()
SequenceExpression
interpret()
expression1
expression2
RepetitionExpression
interpret()
repetition
AlternationExpression
interpret()
alternative1
alternative2

```scala
sealed trait RegularExpression

case class LiteralExpression(
  literal: String) extends RegularExpression

case class SequenceExpression(
  expression1: RegularExpression,
  expression2: RegularExpression) extends RegularExpression

case class RepetitionExpression(
  repetition: RegularExpression) extends RegularExpression

case class AlternationExpression(
  alternative1: RegularExpression,
  alternative2: RegularExpression) extends RegularExpression
```

```scala
sealed trait Exp
case class Lit(s: String) extends Exp
case class And(a: Exp, b: Exp) extends Exp
case class Many(e: Exp) extends Exp
case class Or(a: Exp, b: Exp) extends Exp
```

```
raining & (dogs | cats) *
```



```
And(Lit("raining"),
    Many(Or(Lit("dogs"),
            Lit("cats"))))
```

Interpretation

- `Lit` checks if the input matches a literal
- `Or` checks if the input matches any of its subexpressions
- `And` checks if the input matches all of its subexpressions in sequence
- `Many` checks if the input matches its subexpression zero or more times

```scala
sealed trait Exp {
  def interpret(s: String): (Boolean, String) =
    this match {
      case Lit(l) if (s startsWith l) =>
        (true, s drop l.length)
      case And(a,b) =>
        val (p, ns) = a interpret s
        if (p) b interpret ns else (false, s)
      case Or(a,b) =>
        val (p, ns) = a interpret s
        if (p) (true, ns) else b interpret s
      case Many(e) =>
        val (p, ns) = e interpret s
        if (p) Many(e) interpret ns else (true, s)
      case _ => (false, s)
    }
}
```

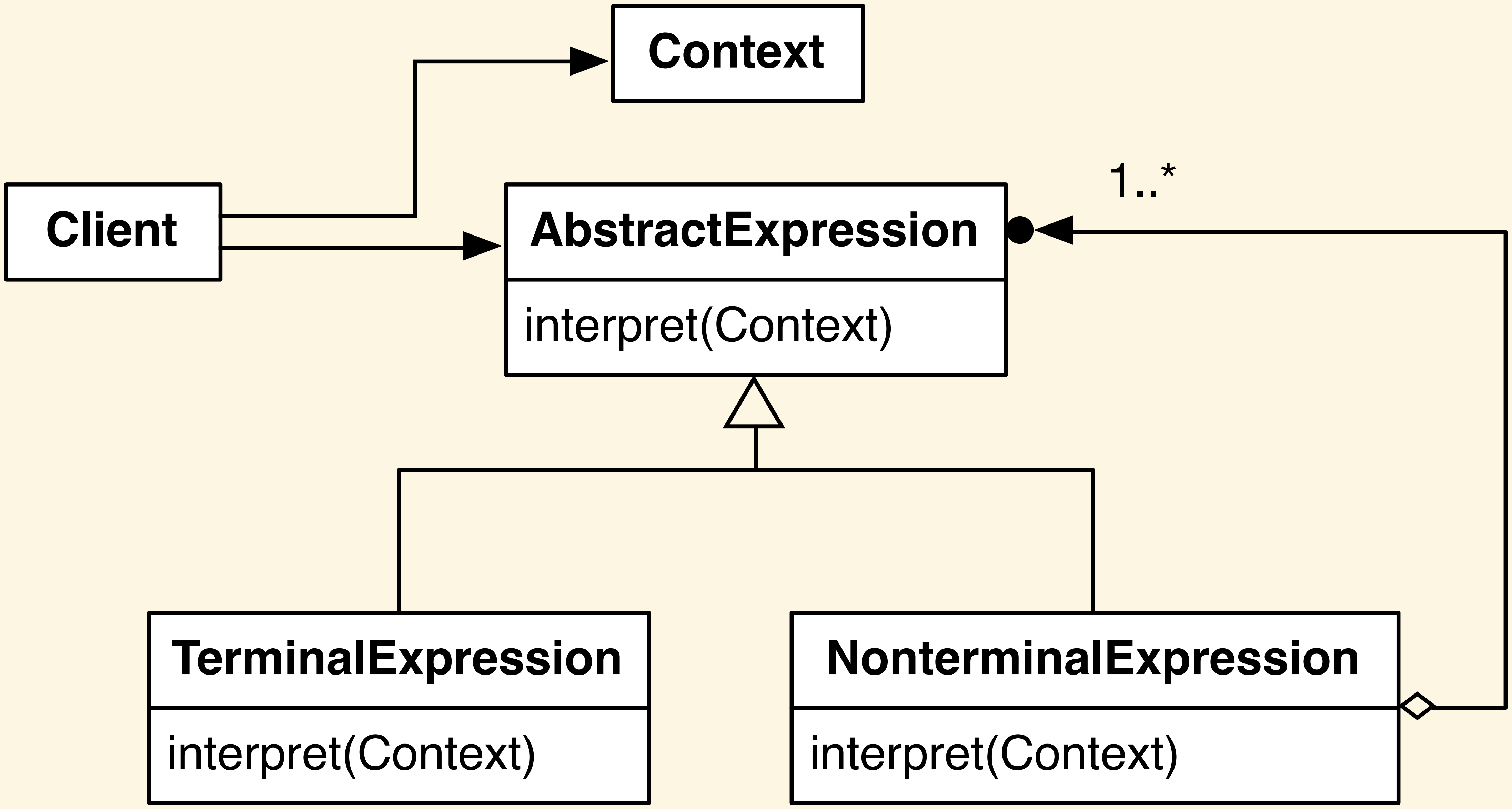
```scala
import scalaz.State

def toStateMachine(e: Exp): State[String, Boolean] =
  State(e.interpret)
```

"…interpreters are usually *not* implemented by interpreting parse trees directly but by first translating them into another form. For example, regular expressions are often transformed into state machines."

—GoF page 245

- Expressions are ***purely syntactic***
- They have no ***intrinsic meaning***
- Semantics are given by ***interpretation*** in a ***context***
- For example by translation to a state machine

Structure


Context
Client
AbstractExpression
interpret(Context)
1..*
TerminalExpression
interpret(Context)
NonterminalExpression
interpret(Context)

Interpreter is the Algebraic Data Type pattern

Algebraic Data Types are little languages

```scala
sealed trait Option[A]
case class None[A]() extends Option[A]
case class Some[A](a: A) extends Option[A]
```

- Take a problem that occurs often
- Express instances of it as sentences in a simple language
- Solve the problem by interpreting these sentences

- Option is ***purely syntactic***
- `Some` and `None` have no ***intrinsic meaning***
- Semantics are given by ***interpretation*** in a ***context***

```scala
sealed trait Option[A] {
  def interpret(c: Context): Unit
}
```

```scala
sealed trait Option[A] {
  def interpretWithFoo(foo: Foo): Unit
  def interpretWithBar(foo: Bar): Unit
}
```

```scala
sealed trait Option[A] {
  def fold[B](z: B)(f: A => B): B =
    this match {
      case None() => z
      case Some(a) => f(a)
    }
}
```

Visitor!

```
sealed trait Option[A] {
  def fold[B](z: B)(f: A => B): B =
    this match {
      case None() => z
      case Some(a) => f(a)
    }
}
```

```scala
sealed trait Exp {
  def fold[A](lit: String => A,
              and: (A,A) => A,
              many: A => A,
              or: (A,A) => A): A = {
    def go(x: Exp): A = x match {
      case Lit(s) => lit(s)
      case And(a,b) => and(go(a), go(b))
      case Many(e) => many(go(e))
      case Or(a,b) => or(go(a), go(b))
    }
    go(this)
}
```

```scala
sealed trait Exp[T] {
  def fold[A](lit: T => A,
              and: (A,A) => A,
              many: A => A,
              or: (A,A) => A): A = {
    def go(x: Exp[T]): A = x match {
      case Lit(s)   => lit(s)
      case And(a,b) => and(go(a), go(b))
      case Many(e)  => many(go(e))
      case Or(a,b)  => or(go(a), go(b))
    }
    go(this)
}
```

Folding replaces all the instructions in the program with instructions in a *different* language.

That language admits the same structure.

```
x.fold(identity[Boolean], _&&_, !_, _||_)
```

```
boolexp ::= disjunction | conjunction | negation | variable |
            '(' boolexp ')'
disjunction ::= expression 'or' expression
conjunction ::= expression 'and' expression
negation ::= 'not' expression
constant ::= 'true' | 'false'
variable ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ...}*
```

```scala
sealed trait Exp
case class Lit(b: Boolean) extends Exp
case class And(a: Exp, b: Exp) extends Exp
case class Not(e: Exp) extends Exp
case class Or(a: Exp, b: Exp) extends Exp
case class Var(v: String) extends Exp
```

```scala
def replace(e: Exp, env: String => Exp): Exp
def evaluate(e: Exp, env: String => Boolean): Boolean
```

```scala
sealed trait Exp[A]
case class Lit(b: Boolean) extends Exp[A]
case class And(a: Exp[A], b: Exp[A]) extends Exp[A]
case class Not(e: Exp[A]) extends Exp[A]
case class Or(a: Exp[A], b: Exp[A]) extends Exp[A]
case class Var(v: A) extends Exp[A]
```

```scala
sealed trait Exp[V] {
  def fold[A](lit: Boolean => A,
              and: (A,A) => A,
              not: A => A,
              or: (A,A) => A
              lookup: V => A): A = {
    def go(x: Exp[V]): A = x match {
      case Lit(b)   => lit(b)
      case And(x,y) => and(go(x), go(y))
      case Not(e)   => not(go(e))
      case Or(x,y)  => or(go(x), go(y))
      case Var(v)   => lookup(v)
    }
    go(this)
}
```

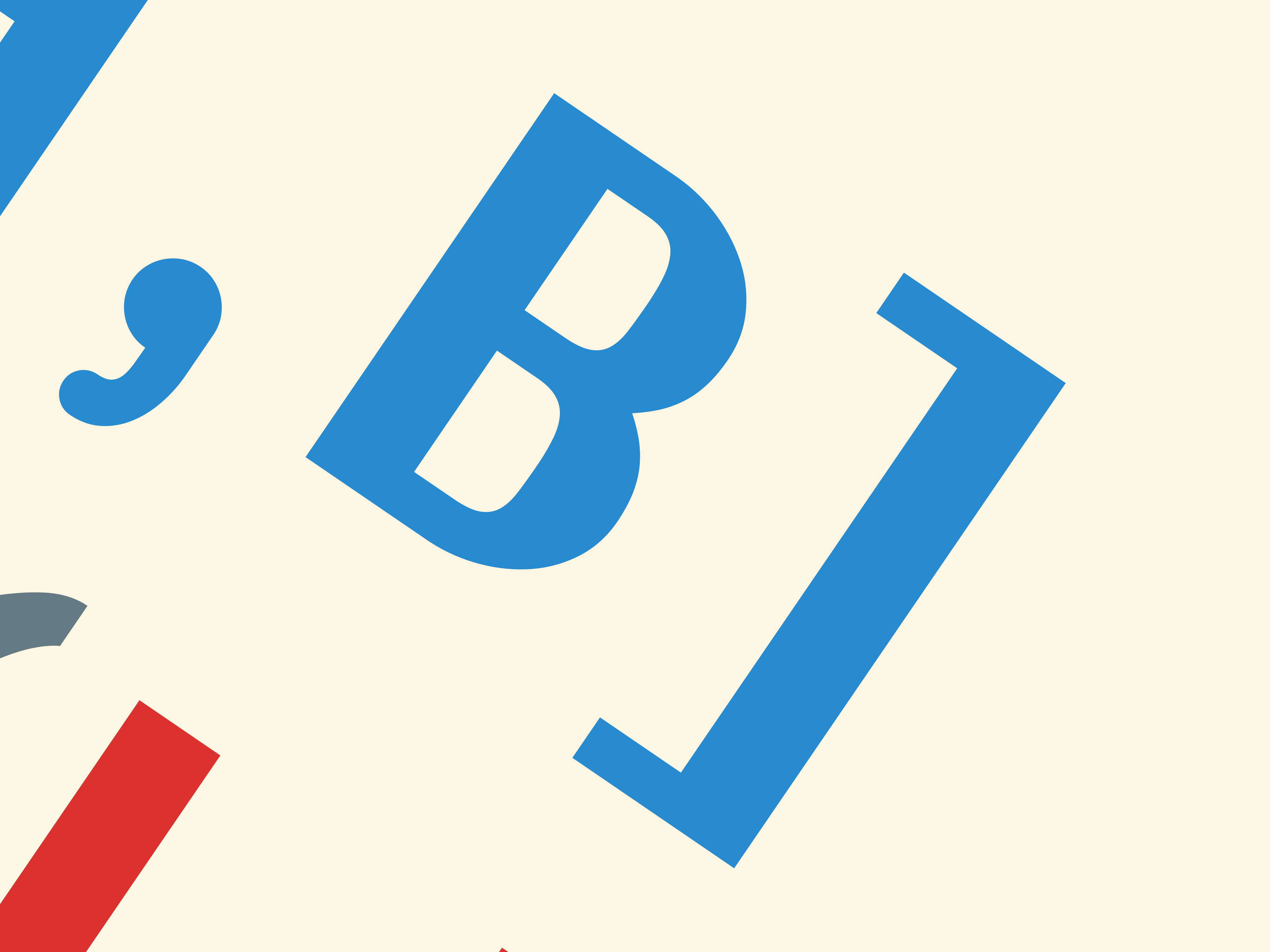
```
def evaluate[A](e: Exp[A], env: A => Boolean): Boolean =
  e.fold(identity, _&&_, !_, _||_, env)

def replace[A,B](e: Exp[A], env: A => Exp[B]): Exp[B] =
  e.fold(Lit(_), And(_,_), Not(_), Or(_,_), env)
```

```scala
def replace[A,B](e: Exp[A], env: A => Exp[B]): Exp[B] =
  e.fold(Lit(_), And(_,_), Not(_), Or(_,_), env)
```

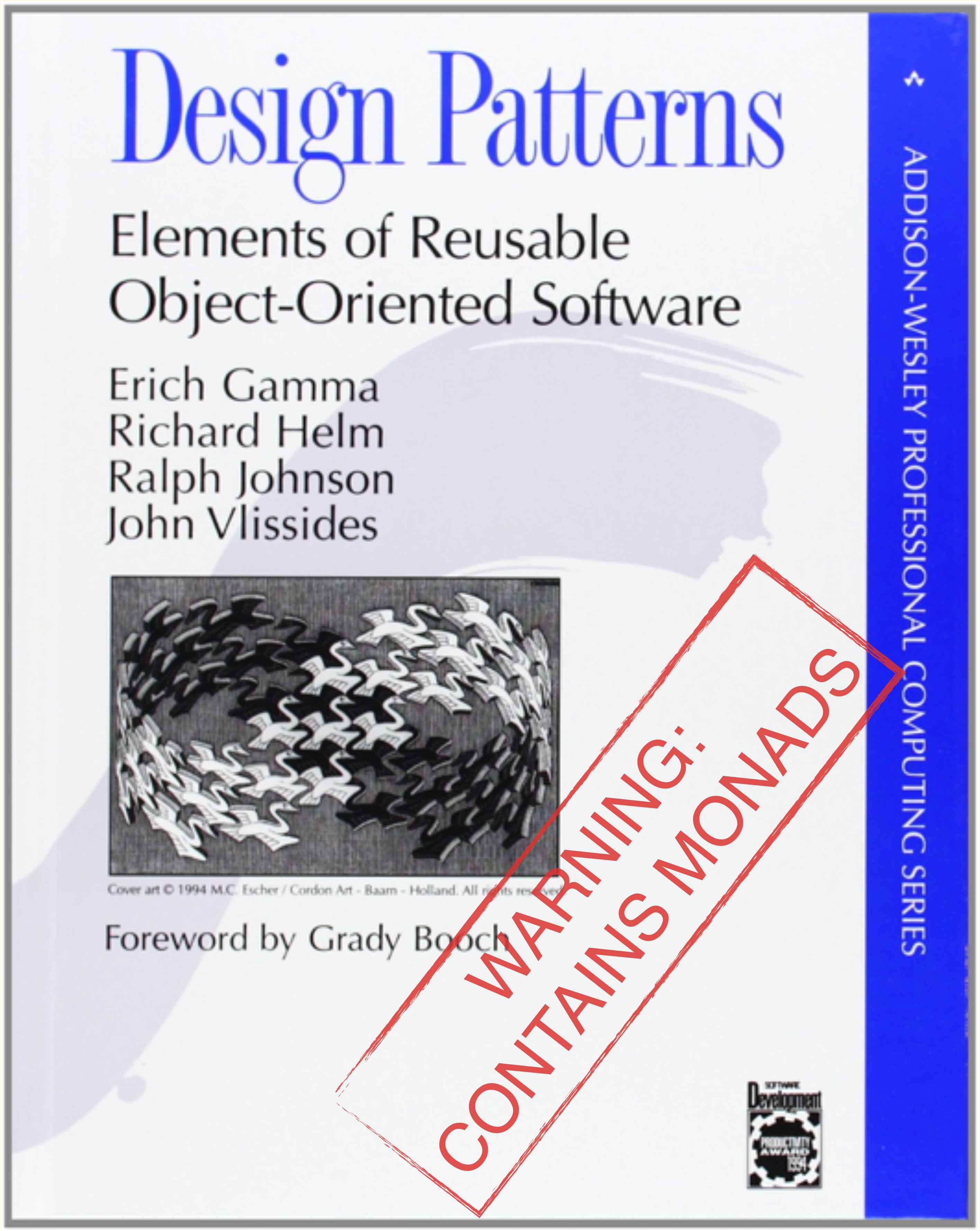
```scala
def flatMap[A,B](e: Exp[A], env: A => Exp[B]): Exp[B] =
  e.fold(Lit(_), And(_,_), Not(_), Or(_,_), env)
```

```
def flatMap[A,B](e: Exp[A], env: A => Exp[B]): E
  e.fold(Lit(_), And(_,_), Not(_), Or(_,_), env)
```

```
def flatMap[A,B](e: Exp[A], e
  e.fold(Lit(_), And(_,_), No
flatMap[A,B](e: Exp[
fold(Lit(_), And(_,_
tMap[A,B](e: Ex
d(Lit(_), And(_
[A,B](e:
t( ),
```


,B]

B]



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

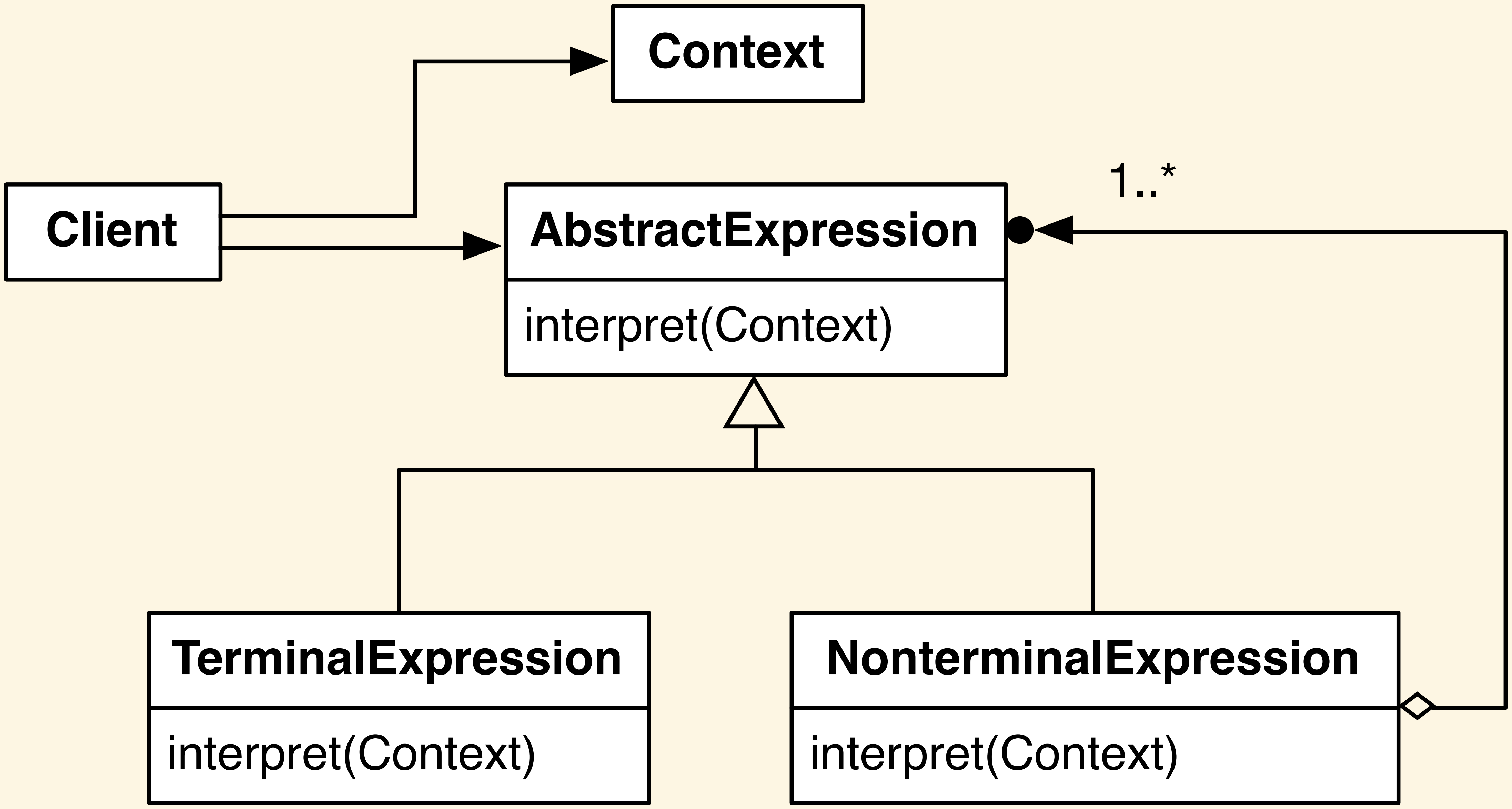
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

WARNING:
CONTAINS MONADS

Interpreter



```scala
sealed trait Exp[F[_],A]

case class TerminalExp[F[_],A](a: A)
  extends Exp[F,A]

case class NonTerminalExp[F[_],A](s: F[Exp[F,A]])
  extends Exp[F,A]
```

```
sealed trait Free[F[_],A]

case class Return[F[_],A](a: A)
  extends Free[F,A]

case class Suspend[F[_],A](s: F[Exp[F,A]])
  extends Free[F,A]
```

```scala
sealed trait BoolAlg[A]
case class Lit[A](b: Boolean) extends BoolAlg[A]
case class And[A](a: A, b: A) extends BoolAlg[A]
case class Or[A](a: A, b: A) extends BoolAlg[A]
case class Not(a: A) extends BoolAlg[A]

type BoolExp[A] = Free[BoolAlg,A]
```

```scala
sealed trait Free[F[_],A] {
  def foldMap[G[_]:Monad](f: F ~> G): G[A] =
    this match {
      case Return(a) =>
        Monad[G].pure(a)
      case Suspend(s) =>
        Monad[G].bind(f(s))(_.foldMap(f))
  }
}
```

```scala
trait Monad[M[_]] {
  def pure[A](a: A): M[A]
  def bind[A,B](a: M[A])(f: A => M[B]): M[B]
}

trait ~>[F[_],G[_]] {
  def apply[A](a: F[A]): G[A]
}

sealed trait Free[F[_],A] {
  def foldMap[G[_]:Monad](f: F ~> G): G[A] =
    this match {
      case Return(a) =>
        Monad[G].pure(a)
      case Suspend(s) =>
        Monad[G].bind(f(s))(_.foldMap(f))
  }
}
```

```scala
type Trivial[A] = Unit

type Option[A] = Free[Trivial,A]
```

https://github.com/MonsantoCo/stoop/

```scala
sealed trait CouchF[A]

case class CreateDB[A](name: String, a: A) extends CouchF[A]
case class DropDB[A](name: String, k: Boolean => A) extends CouchF[A]
case class GetAllDBs(k: List[DB] => A) extends CouchF[A]
case class Fail[A](e: Throwable) extends CouchF[A]
case class NewDoc[A](db: DB,
                     doc: Option[Doc],
                     json: JsValue,
                     params: Map[String, JsValue],
                     k: (String \/ (Doc, Rev, Boolean)) => A)
  extends CouchF[A]

// Etc…
```

```scala
type Couch[A] = Free[CouchF,A]

val runCouch: (CouchF ~> scalaz.concurrent.Task) = ???
```

Summary

False OOP tao: **Inheritance**
Modeling different *behaviour* with different concrete subclasses.

True OOP tao: **Free Monads**
Modeling different *meaning* with different interpreters.

Functional Programming in scala

Paul Chiusano
Rúnar Bjarnason

Foreword by Martin Odersky

MANNING