UNIVERSITEIT VAN AMSTERDAM

# Software Design

Software Construction 2018

Dr. Vadim Zaytsev aka @grammarware

raincode LABS compiler experts

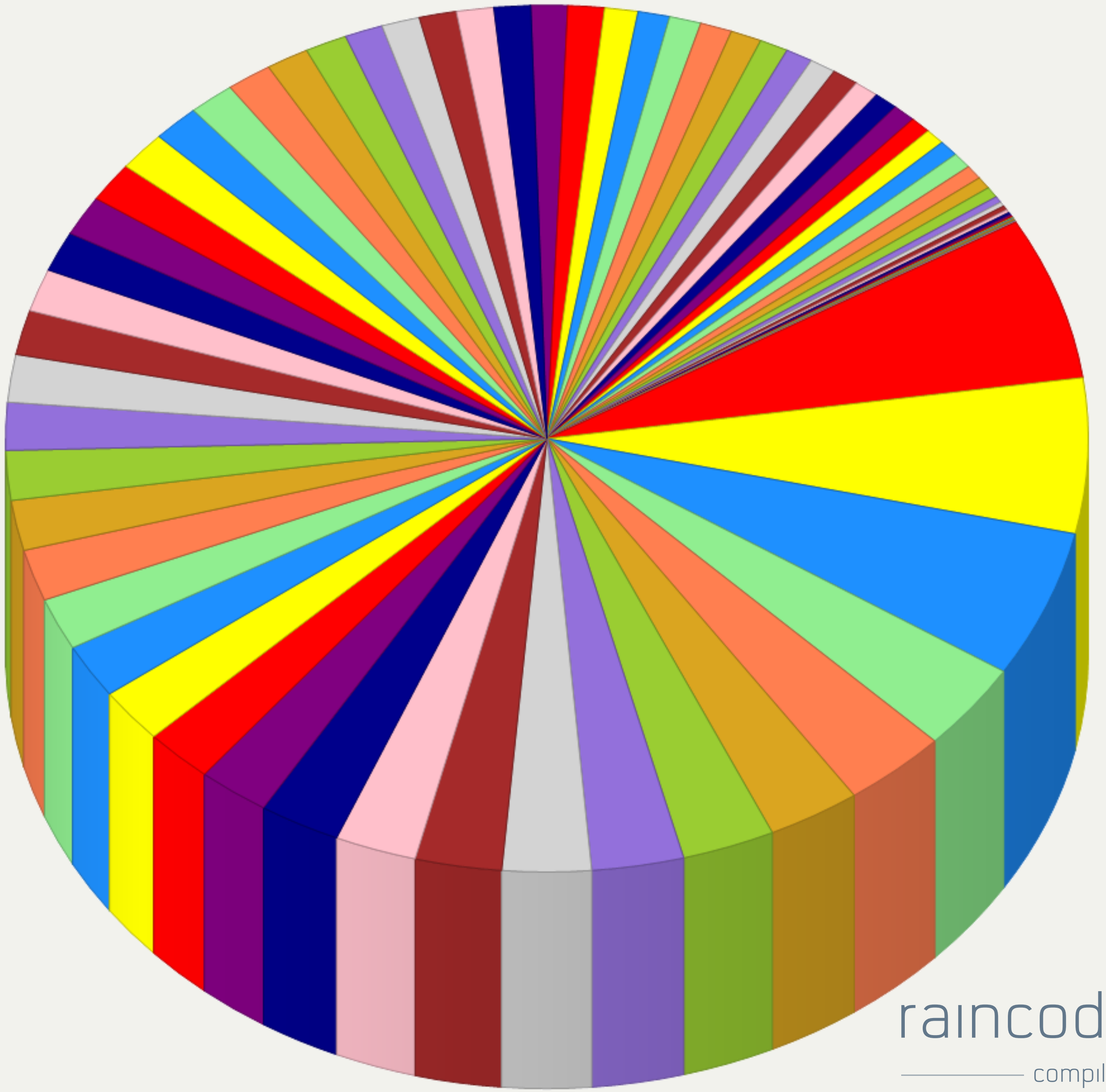
# Reminder: screencast [DL: 18.04.01]

- Technically any platform
- Identification (accounts + photos?)
- Language + frameworks
- Parsing (syntactic analysis)
- AST (design)
- Static analysis (type check)
- Interpretation (rendering)
- Styling (QLS)

# 200+: Nico, Rocco, Laurence; 100+: 9 people

**4223 Commits**

| | |
|---|---|
| 257 | Nico Tromp |
| 244 | rmathijn |
| 232 | lauwdude |
| 138 | Remi van Veen |
| 131 | Peter Takacs |
| 117 | Mihai Onofrei |
| 117 | hasan |
| 116 | bicker |
| 112 | Tim Nederveen |
| 109 | Simon Schneider |
| 103 | jewelEarthDeveloper |
| 102 | Edwin |
| 90 | piotrkosytorz |
| 86 | GrimGerbil |
| 86 | Jordy Bottelier |
| 85 | Unknown |
| 82 | rashadaoud |
| 81 | Niels Boerkamp |
| 79 | DennisvanderWerf |
| 78 | Laurens de Gilde |
| 75 | Dennis Kruidenberg |
| 75 | ighmelene |
| 70 | Hector Stenger |
| 67 | Metchu |
| 63 | Dylan Bartels |
| 63 | Meess |
| 62 | Cornelius Ries |
| 62 | carlyhill6895 |
| 59 | Joanna Roczniak |
| 58 | MichielBoswijk |
| 57 | Elias |
| 54 | porke |
| 53 | Sara Oonk |
| 48 | Tim |
| 48 | olimoli9160 |
| 47 | Thijs Klaver |
| 47 | evanscharrenburg |
| 46 | AHerczeg |
| 46 | TerryvanWalen |
| 46 | Toine Khonraad |
| 42 | Michael de Lang |
| 40 | Jaap Koetsier |
| 40 | Nick |
| 40 | tdobber |
| 39 | bramo |
| 36 | V-Jong |
| 34 | Jorick van Rhenen |
| 33 | George Vletsas |
| 33 | Scoudem |
| 31 | Leó Gunnar vidisson |
| 30 | Joana Correia Magalhães Sousa |
| 29 | Jouke |
| 26 | Quinten |
| 25 | Deepa Karra |
| 25 | Lars Lokhoff |



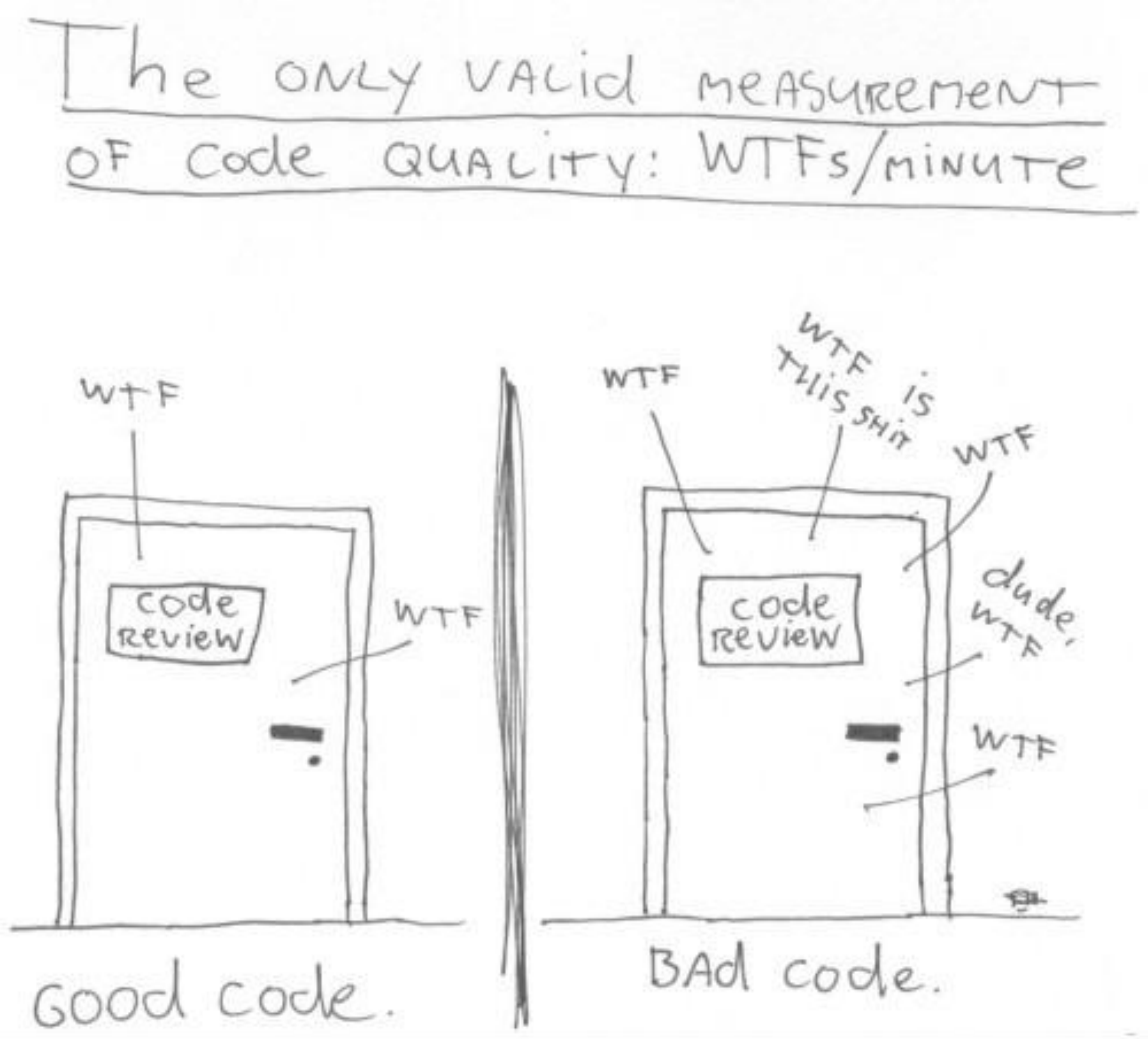
raincode LABS
compiler experts

# Grading of QL/QLS

- Functionality
- Testing
- Simplicity
- Modularity
- Layout and style
- Separation of concerns
- Advice: try one another
- Expected: grade is less important than personal improvement

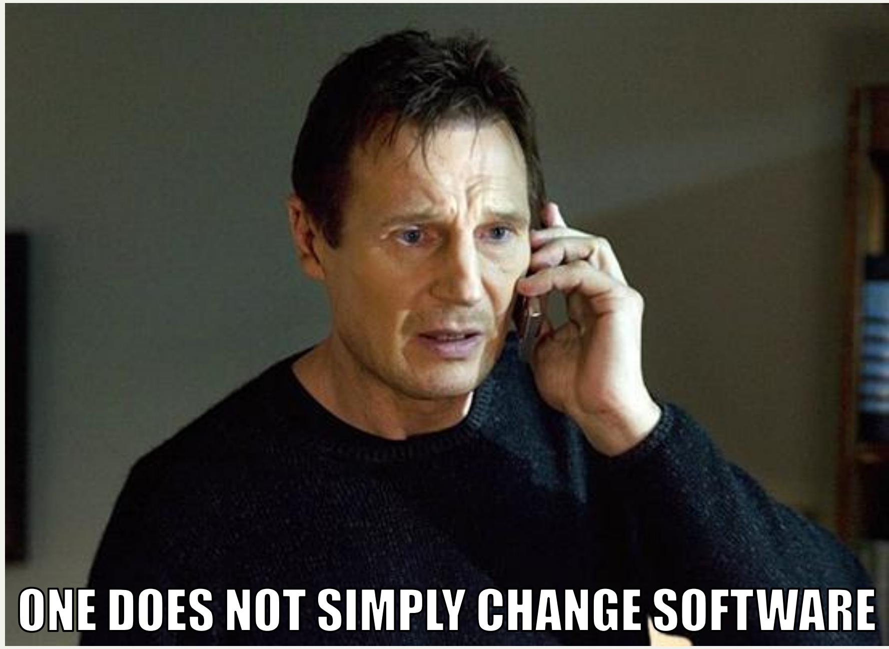
# Software construction

- Drivers
    - features, change, understandability, readability, testability, reliability, etc
- Symptoms and alarm bells
    - complexity, duplication, coupling, smells, tangling, scattering, etc
- Tools, techniques, methods
    - abstraction, encapsulation, patterns, dependency inversion, Demeter, information hiding, contracts, etc



(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

# Rotting design: rigidity



- Software is difficult to change
  - even in simple ways
- Every change causes a cascade
  - multi-week marathon across many modules
- Eventually consistency is gone
  - everything is in-between
- Managers do not allow to fix non-critical problems
  - every fix makes it worse

Robert C. Martin, *Design Principles and Design Patterns*, 2000

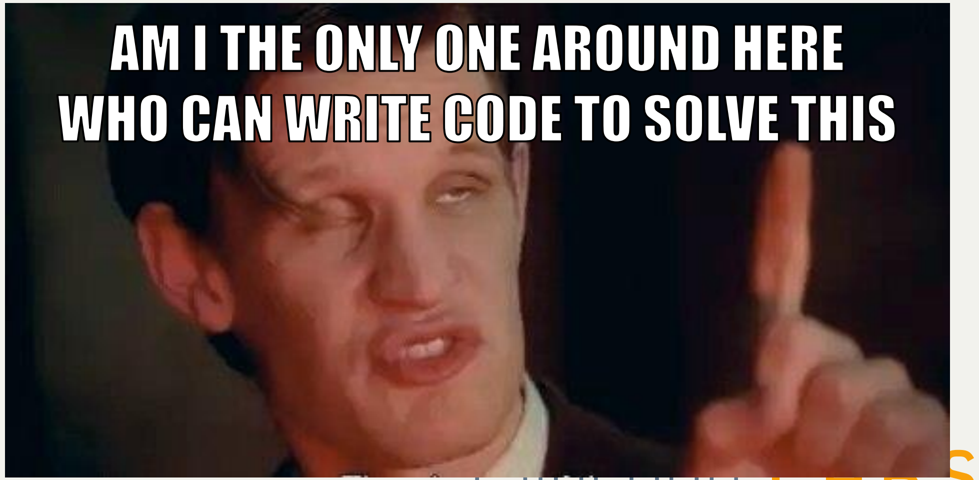
# Rotting design: fragility

- Software breaks in response to change
  - in many places
- No conceptual relationship
  - unexpected effects
- Fragile software gradually decreases in quality
  - every fix introduces more problems
- You lost control

Robert C. Martin, *Design Principles and Design Patterns*, 2000

# Rotting design: immobility

- Inability to reuse
  - within the project, across projects
- Each piece of code comes with baggage
  - looks similar => time lost to rev.engineer
- Easier to rewrite than to reuse
  - often after a failed reuse attempt


AM I THE ONLY ONE AROUND HERE
WHO CAN WRITE CODE TO SOLVE THIS

Robert C. Martin, *Design Principles and Design Patterns*, 2000

# Rotting design: viscosity

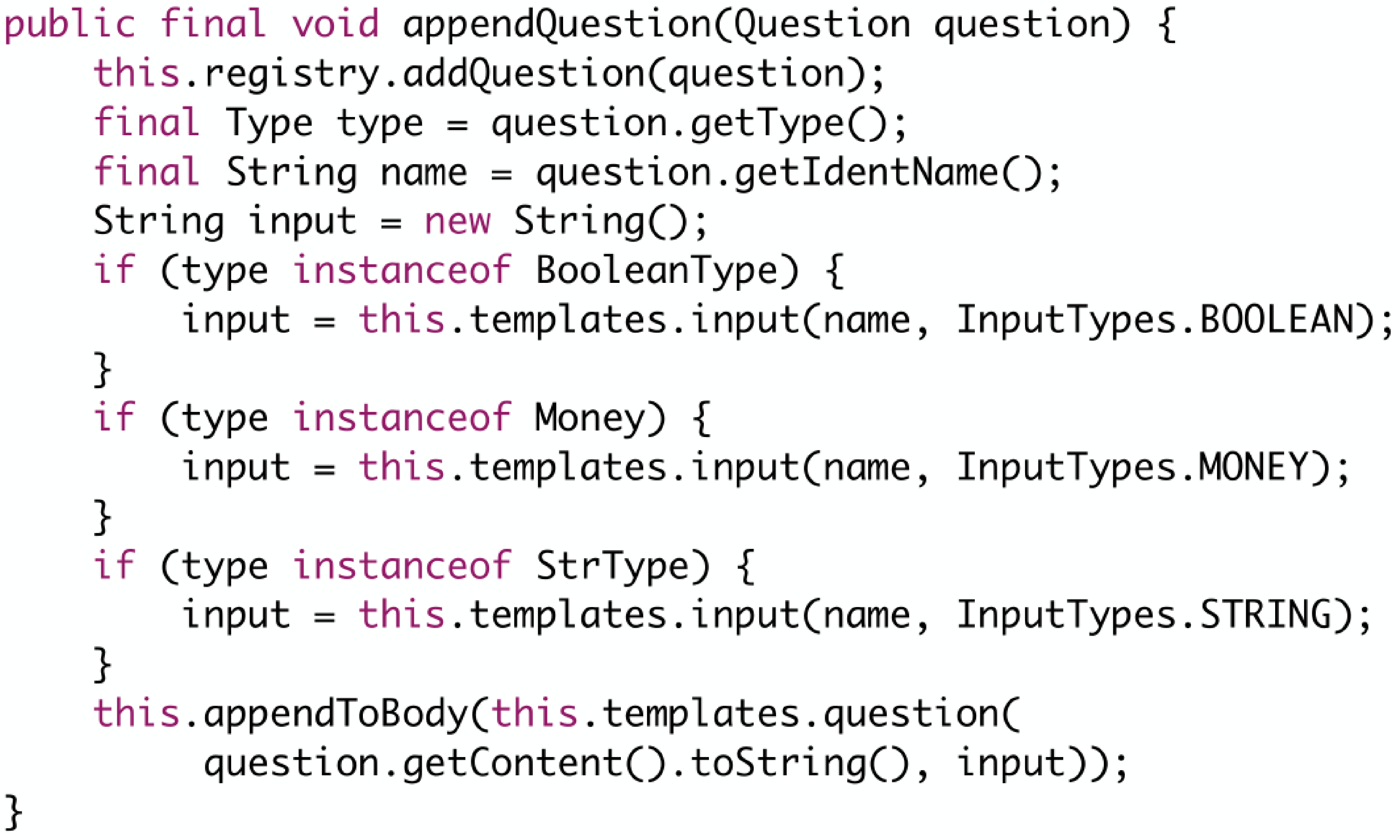
- Easy to do the wrong thing, hard to do the right thing
  - change choices
- Changes can preserve or contradict the design
  - preserve: these are the good ones
  - contradict: these are hacks
- IDE viscosity
  - e.g., long compile times => no large recompiles

Robert C. Martin, *Design Principles and Design Patterns*, 2000

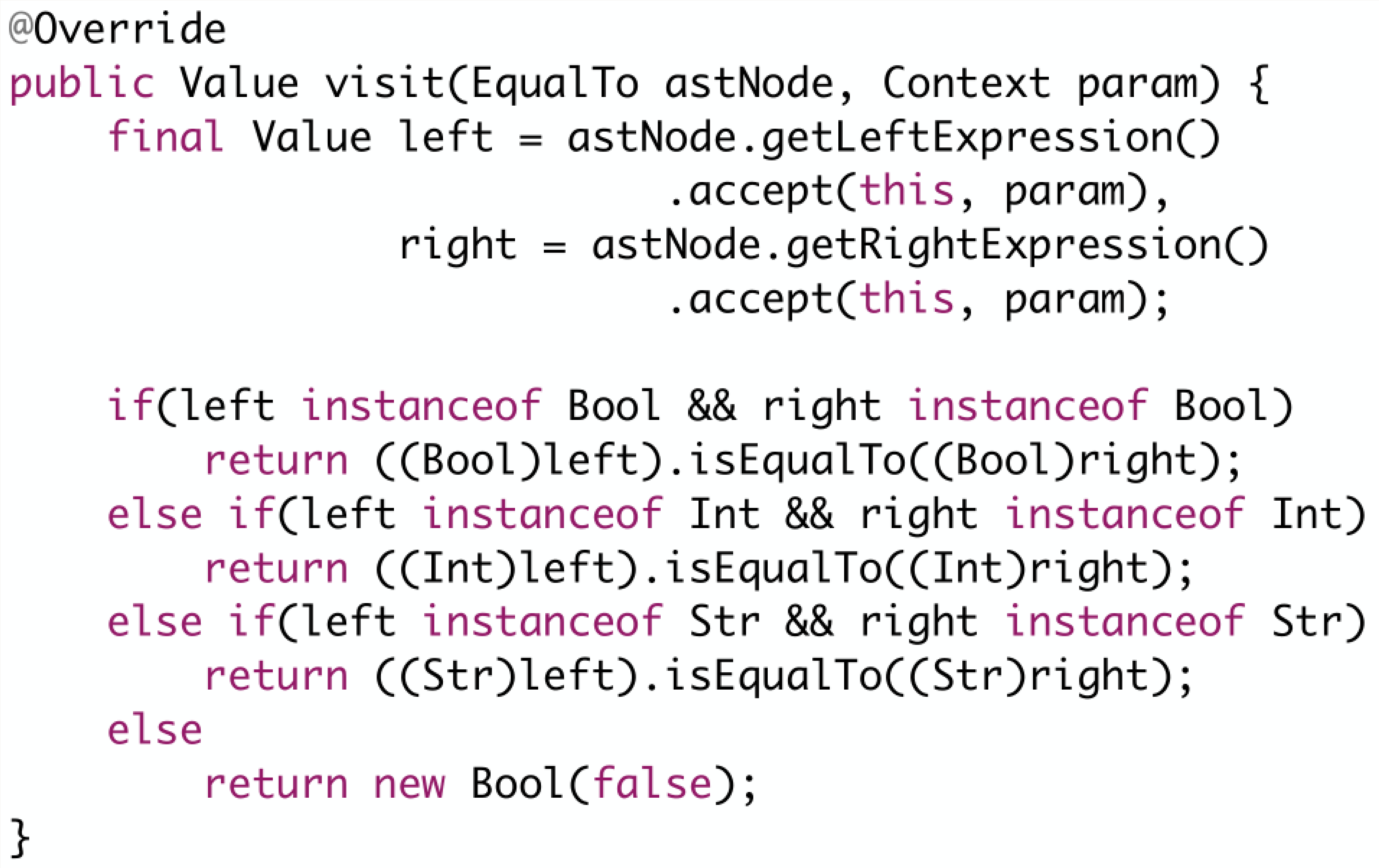
# Code smell example

- instanceof / typeof / GetType / …
- switch case statements
- hand coding dispatch

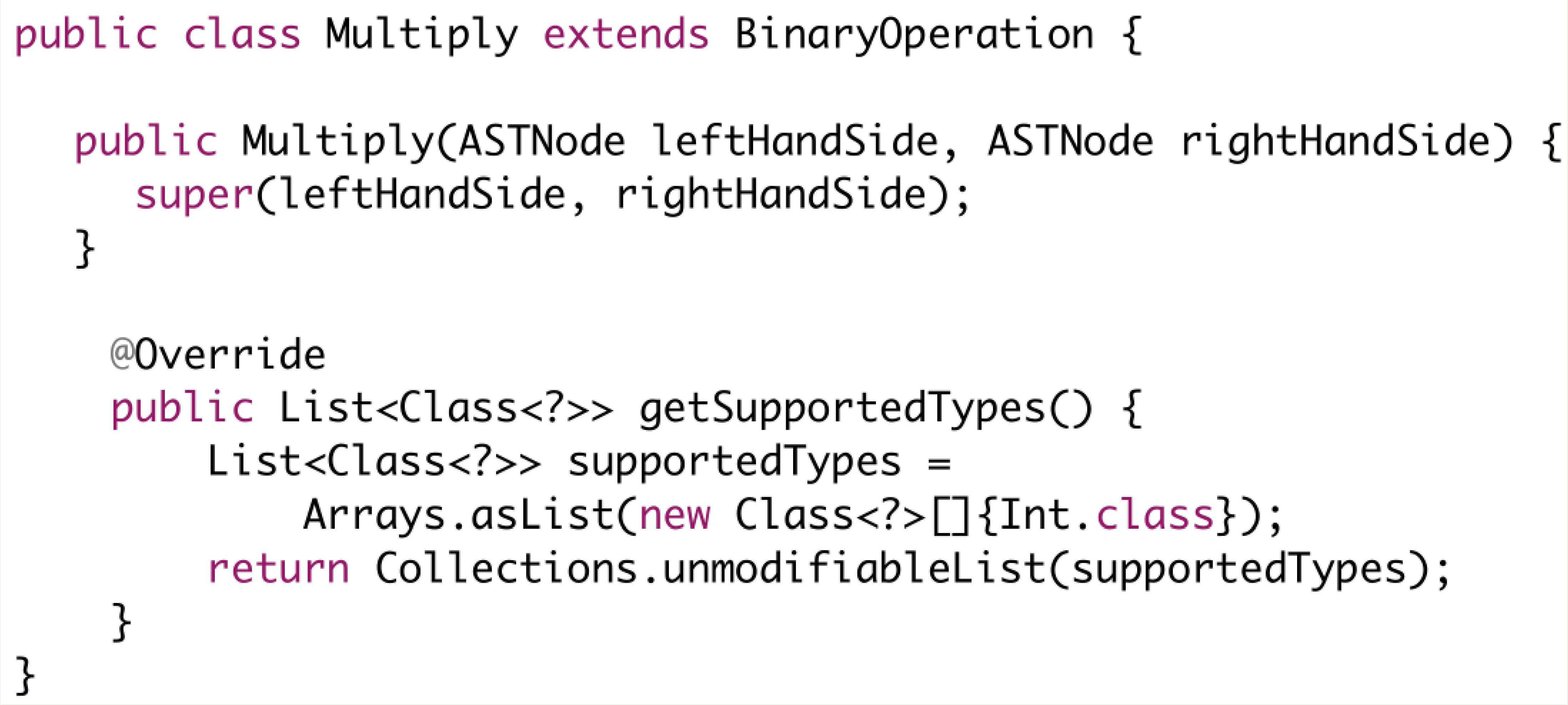
- objects know about themselves
- you get feedback from the compiler



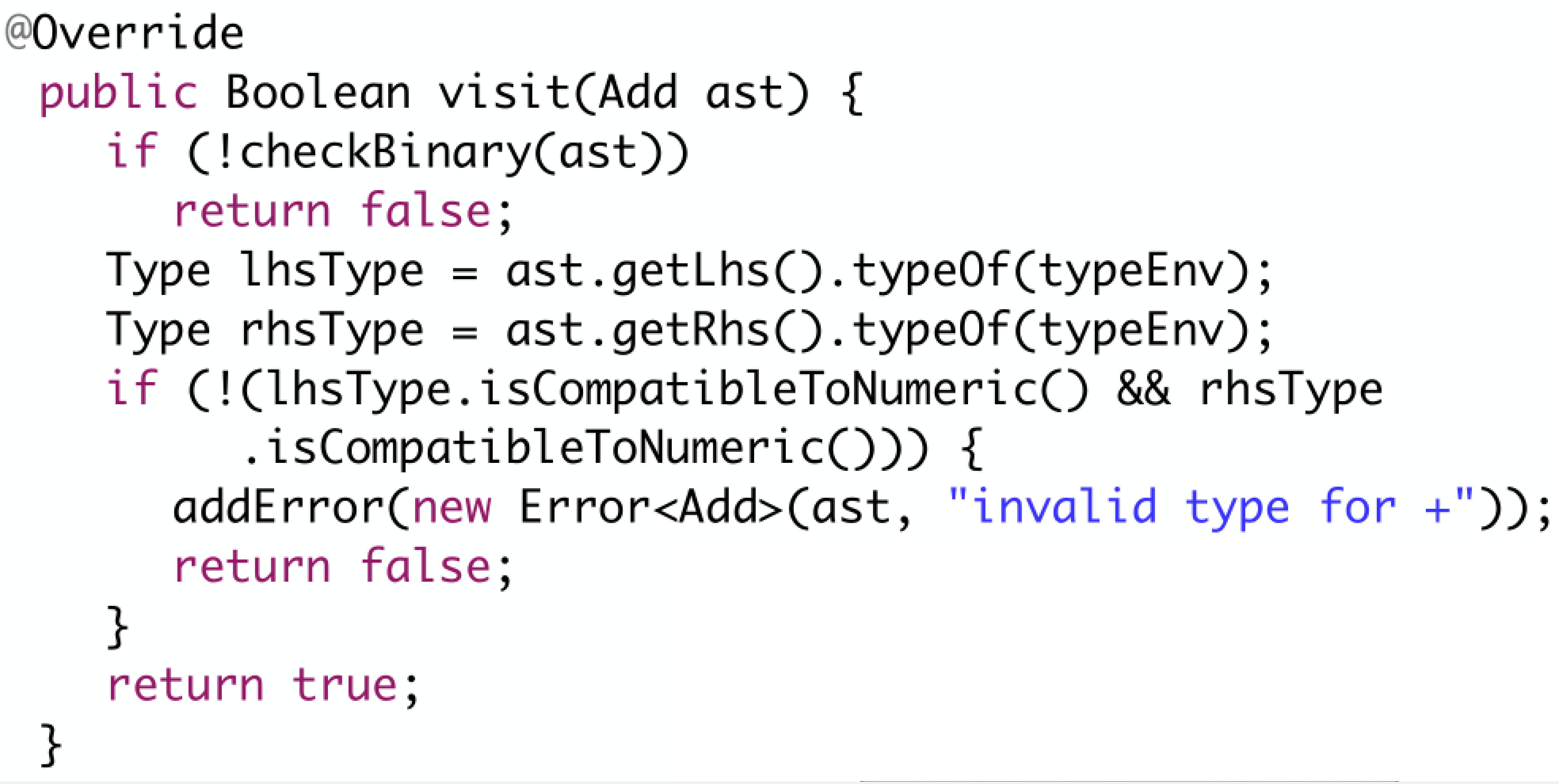
```java
public final void appendQuestion(Question question) {
    this.registry.addQuestion(question);
    final Type type = question.getType();
    final String name = question.getIdentName();
    String input = new String();
    if (type instanceof BooleanType) {
        input = this.templates.input(name, InputTypes.BOOLEAN);
    }
    if (type instanceof Money) {
        input = this.templates.input(name, InputTypes.MONEY);
    }
    if (type instanceof StrType) {
        input = this.templates.input(name, InputTypes.STRING);
    }
    this.appendToBody(this.templates.question(
         question.getContent().toString(), input));
}
```



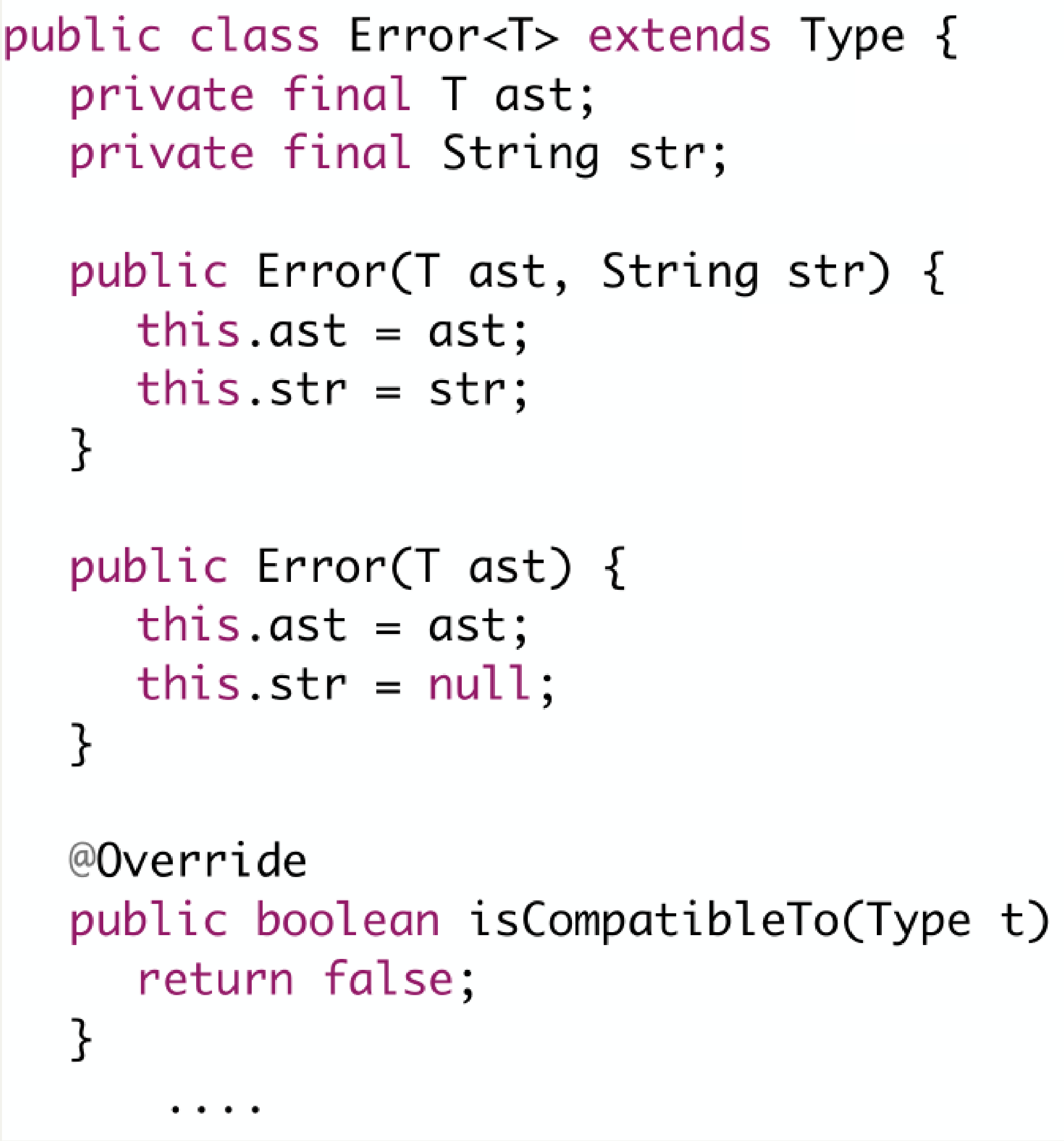
```
@Override
public Value visit(EqualTo astNode, Context param) {
    final Value left = astNode.getLeftExpression()
                           .accept(this, param),
                right = astNode.getRightExpression()
                           .accept(this, param);

    if(left instanceof Bool && right instanceof Bool)
        return ((Bool)left).isEqualTo((Bool)right);
    else if(left instanceof Int && right instanceof Int)
        return ((Int)left).isEqualTo((Int)right);
    else if(left instanceof Str && right instanceof Str)
        return ((Str)left).isEqualTo((Str)right);
    else
        return new Bool(false);
}
```



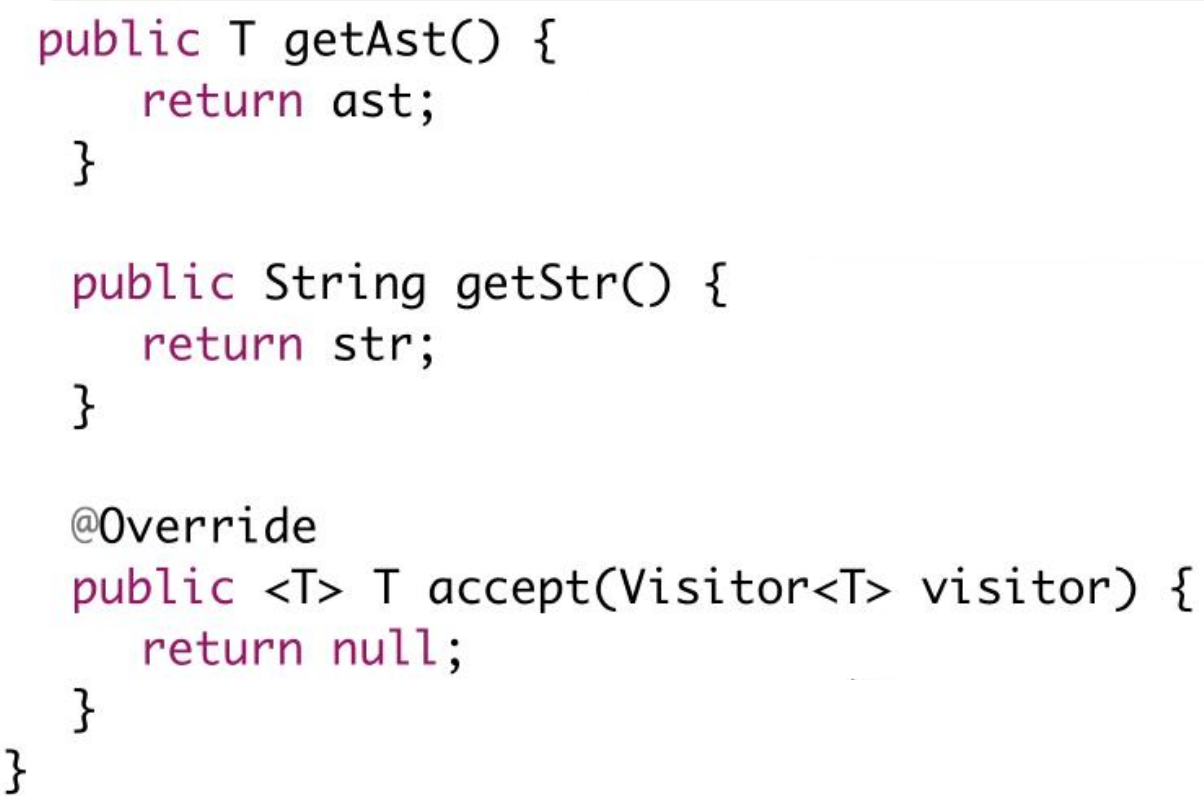
```
public class Multiply extends BinaryOperation {

  public Multiply(ASTNode leftHandSide, ASTNode rightHandSide) {
     super(leftHandSide, rightHandSide);
  }

    @Override
    public List<Class<?>> getSupportedTypes() {
        List<Class<?>> supportedTypes =
            Arrays.asList(new Class<?>[]{Int.class});
        return Collections.unmodifiableList(supportedTypes);
    }
}
```



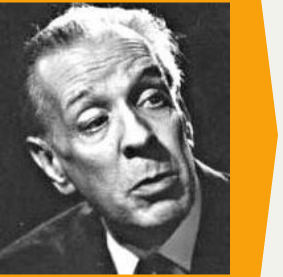
```
@Override
 public Boolean visit(Add ast) {
   if (!checkBinary(ast))
     return false;
   Type lhsType = ast.getLhs().typeOf(typeEnv);
   Type rhsType = ast.getRhs().typeOf(typeEnv);
   if (!(lhsType.isCompatibleToNumeric() && rhsType
       .isCompatibleToNumeric())) {
     addError(new Error<Add>(ast, "invalid type for +"));
     return false;
   }
   return true;
 }
```

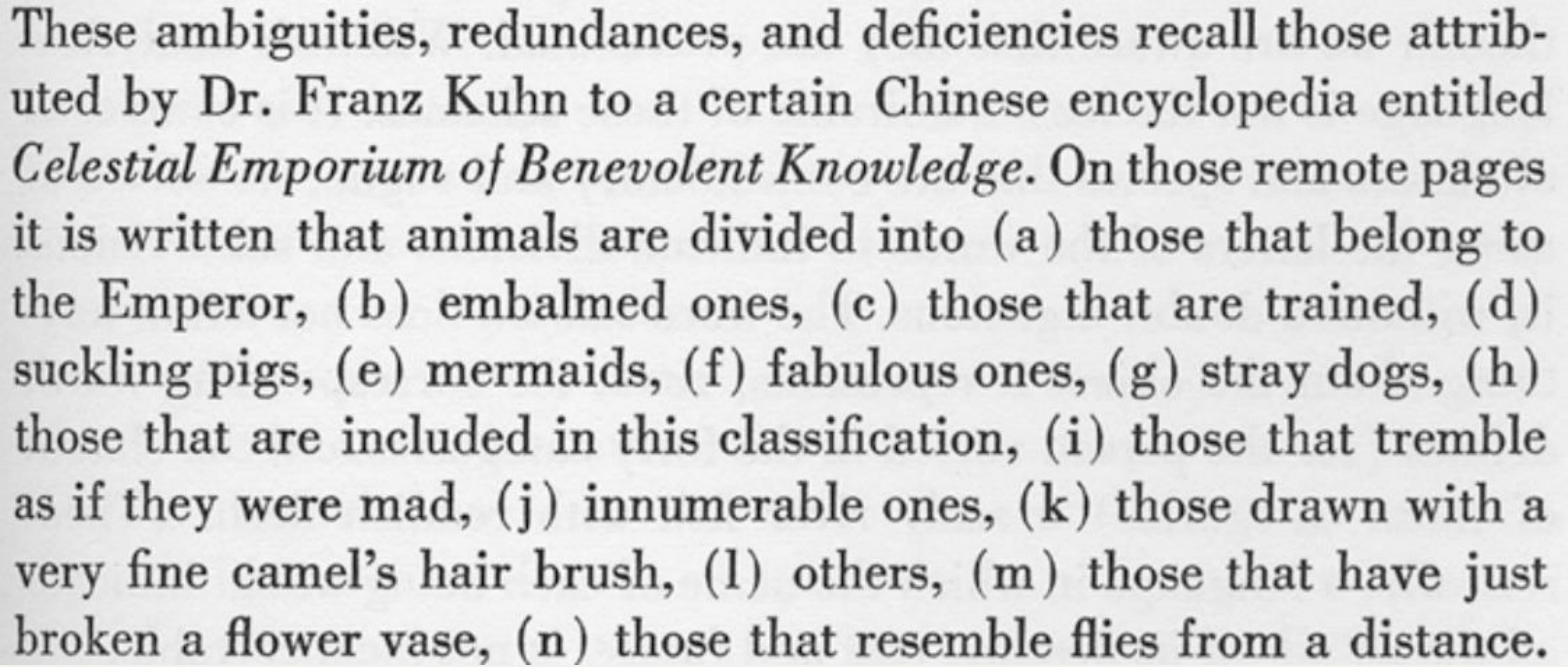


```
public class Error<T> extends Type {
  private final T ast;
  private final String str;

  public Error(T ast, String str) {
    this.ast = ast;
    this.str = str;
  }

  public Error(T ast) {
    this.ast = ast;
    this.str = null;
  }

  @Override
  public boolean isCompatibleTo(Type t)
    return false;
  }
     ....
```



```
  public T getAst() {
     return ast;
  }

  public String getStr() {
     return str;
  }

  @Override
  public <T> T accept(Visitor<T> visitor) {
     return null;
  }
}
```





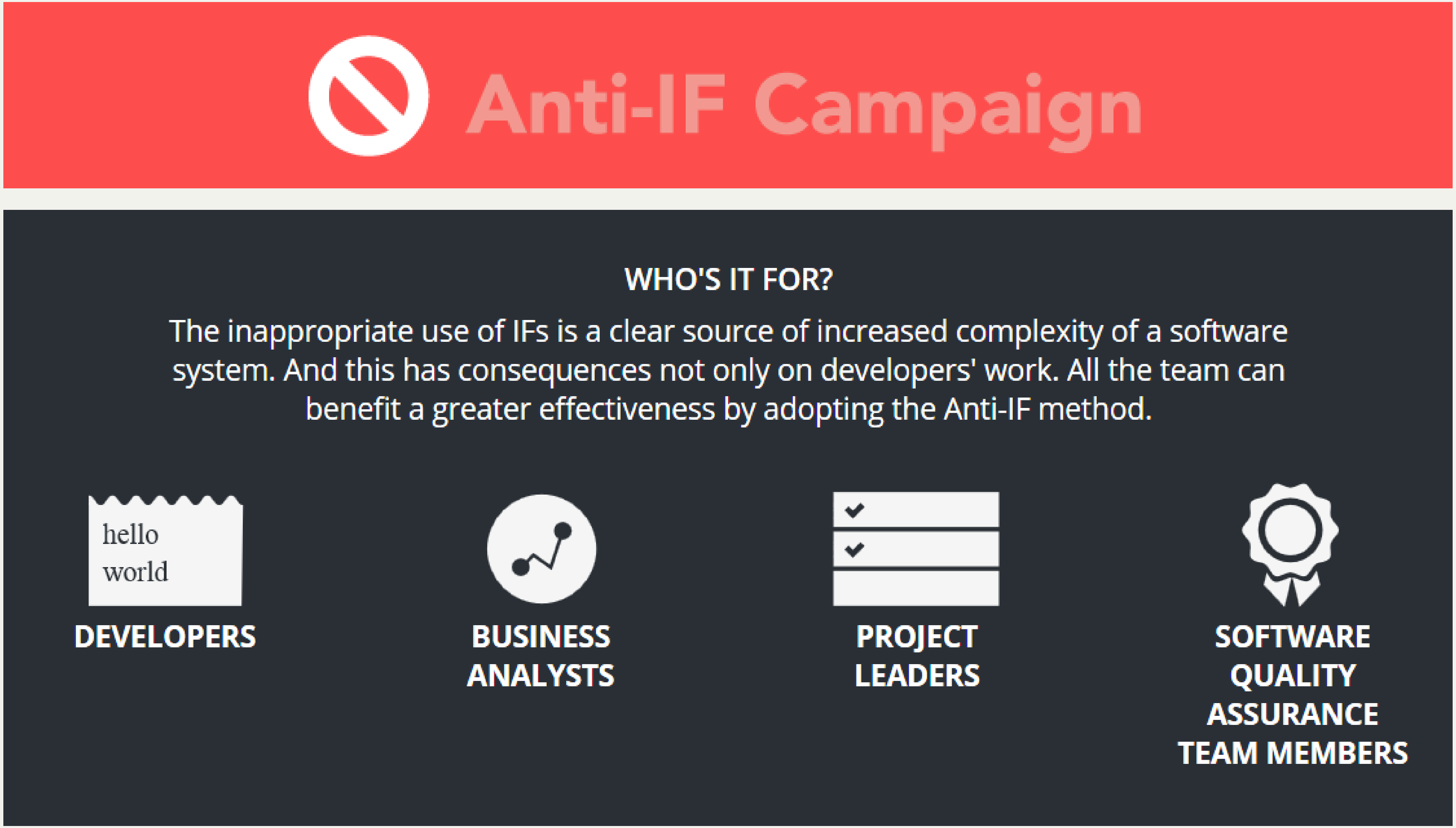
These ambiguities, redundances, and deficiencies recall those attributed by Dr. Franz Kuhn to a certain Chinese encyclopedia entitled *Celestial Emporium of Benevolent Knowledge*. On those remote pages it is written that animals are divided into (a) those that belong to the Emperor, (b) embalmed ones, (c) those that are trained, (d) suckling pigs, (e) mermaids, (f) fabulous ones, (g) stray dogs, (h) those that are included in this classification, (i) those that tremble as if they were mad, (j) innumerable ones, (k) those drawn with a very fine camel's hair brush, (l) others, (m) those that have just broken a flower vase, (n) those that resemble flies from a distance.

# Mind the gap: meta-level / object-level

- Classes in Java = types of Java
- Types of QL can be objects
- Cleaner to deal with
- Identity is natural
- Do not abuse base level’s type system to fit yours

# Different types for different things

- Separate hierarchies
  - Statements / Questions
  - Expressions
  - Types
  - Forms
- Inheritance is “IS A”
  - Error is a not a type, question is not an expression, …



# Anti-IF Campaign

## WHO'S IT FOR?

The inappropriate use of IFs is a clear source of increased complexity of a software system. And this has consequences not only on developers' work. All the team can benefit a greater effectiveness by adopting the Anti-IF method.

DEVELOPERS

BUSINESS ANALYSTS

PROJECT LEADERS

SOFTWARE QUALITY ASSURANCE TEAM MEMBERS

http://www.antiifcampaign.com/ — by Francesco Cirillo

# Abstraction

# Abstraction, stepwise

```
print 1
print 4
print 9
print 16
print 25
```

# Abstraction, stepwise

```
map 0 to i
next:
      set i +:= 1
      print i*i
      if i <= 5
                goto next
      endif
```

# Abstraction, stepwise

```
map 1 to i
do while i <= 5
      print i*i
      set i +:= 1
enddo
```

# Abstraction, stepwise

```
do from 1 to 5 index i
      print i*i
enddo
```

# Abstraction, stepwise

```
proc squares
      do index i from 1 to 5
            print i*i
      enddo
endproc
```

# Abstraction, stepwise

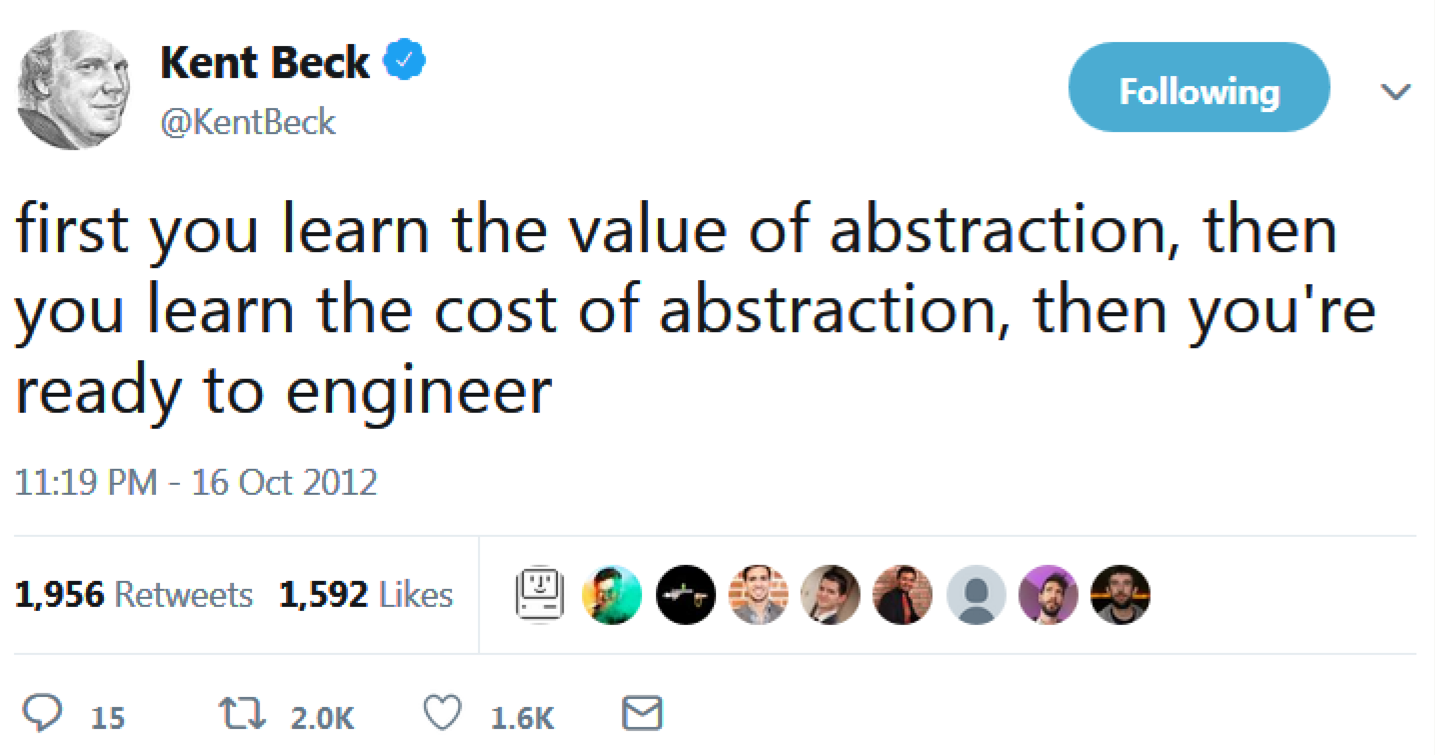
```
proc squares(n integer)
      do index i from 1 to n
            print i*i
      enddo
endproc
```

# Abstraction

- Well-factored code
- Problem decomposition
- Separation of concerns
- Reuse
- Variation
- Single point of change

- Creates distance
- Condensed code harder
- Good naming is essential
- Overhead may impact perf
- Leaky abstractions
- Overdesign risks

# Value and cost of abstraction



Kent Beck

@KentBeck

first you learn the value of abstraction, then you learn the cost of abstraction, then you're ready to engineer

11:19 PM - 16 Oct 2012

1,956 Retweets 1,592 Likes

Kent Beck, Twitter, 2012

# SOLID

# SOLID Principles: SRP

- Single responsibility principle
- Responsibility is a reason to change
- Several responsibilities entangle code
  - you change for one thing, another one breaks
- Examples
  - a grammar that creates an AST
  - QL + QLS

# SOLID Principles: OCP

- Open-closed principle
- Open for extension, closed for modification
- Open for extension
  - can inherit, add fields, methods
- Close for modification
  - can be compiled, stored
- Abstraction is the key!

# SOLID Principles: LSP

- Liskov substitution principle
- Subclasses should be valid substitutes
  - contravariance of method arguments
  - covariance of return types
  - no new exceptions thrown
  - preconditions can only be weaker
  - postconditions can only be stronger
  - invariants must be preserved

# SOLID Principles: ISP

- Interface segregation principle
- Do not depend on things you do not use
- One general purpose interface is worse than many client specific ones
  - role interfaces

# SOLID Principles: DIP

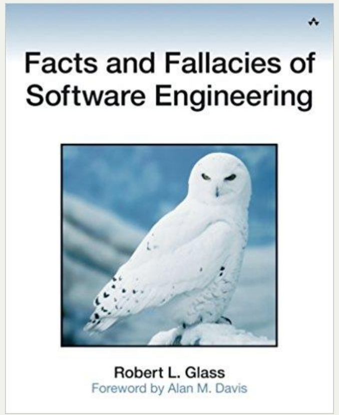
- Dependency inversion principle
- Depend on abstractions
    - Do not depend on concretions
- Abstractions should not depend on details
- High-level modules should not depend on low-level ones
- Modules should depend on abstractions

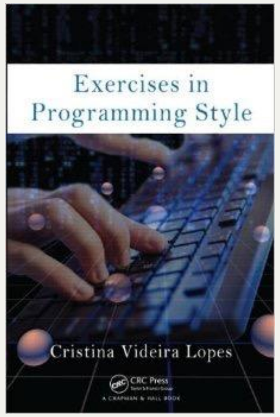
# Summary

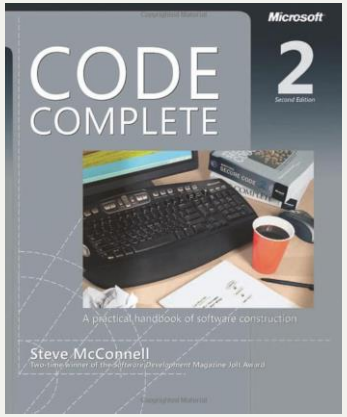
- Make small modules
- Minimise dependencies
- Make dependencies small and explicit
- Depend on abstractions
- Encapsulate implementation decisions
- Document your assumptions
- Use patterns to communicate
- Do not repeat yourself

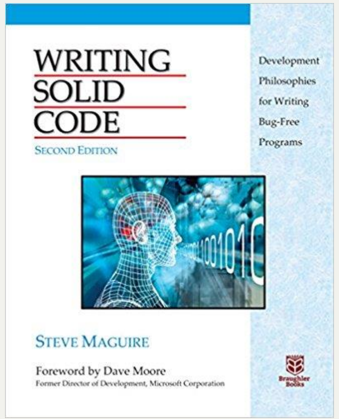
raincode LABS
compiler experts

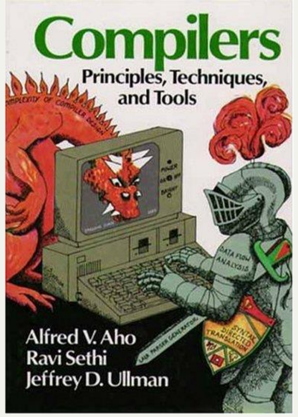
# Course summary

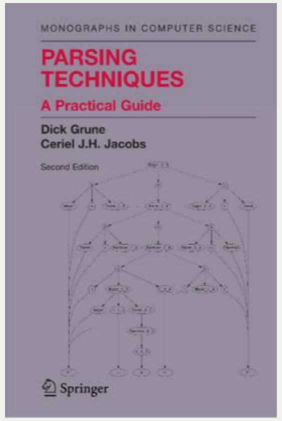

Facts and Fallacies of Software Engineering
Robert L. Glass
Foreword by Alan M. Davis

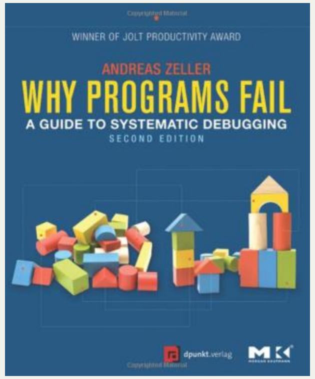


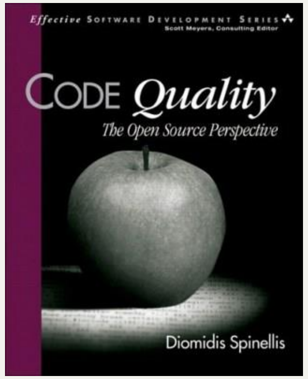



WRITING SOLID CODE
SECOND EDITION
Development Philosophies for Writing Bug-Free Programs
STEVE MAGUIRE
Foreword by Dave Moore
Former Director of Development, Microsoft Corporation



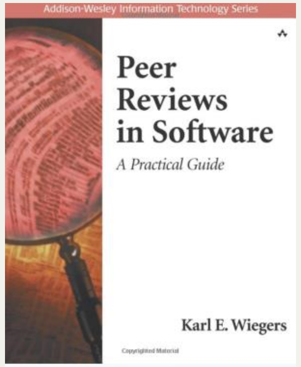



WINNER OF JOLT PRODUCTIVITY AWARD
ANDREAS ZELLER
WHY PROGRAMS FAIL
A GUIDE TO SYSTEMATIC DEBUGGING
SECOND EDITION
dpunkt.verlag
MK


Effective Software Development Series
CODE Quality
The Open Source Perspective
Diomidis Spinellis

# Conclusion

- Practices can always be improved
- Code can always be improved
- There are tangible thresholds of criticality
- Not covered [enough]
  - git (messages, pull-merge-push, stash-rebase-pop-push)
  - code review
  - automation
  - (mega)modelling
  - clean code
  - code reading & comprehension
  - deployment, shipping & versioning
  - …
- All memes in this lecture were made wrong, and only one student noticed



THAT'S ALL FOLKS!
SO LONG, AND THANKS FOR ALL THE FISH!