

Aspects of code quality



Introduction

- Let's talk about
 - quality code
 - and how to achieve it
- Learning goals:
 - How to recognize good/bad code
 - Knowledge of key problems/solutions
 - Awareness of the trade-offs involved

Topics of today

- Code quality
- Duplication
- Tangling & scattering
- Bad code example (Dancing Bears)
- Abstraction
- Design Patterns
- Trade-off example Rascal code (CWI)
- Conclusion

Code quality

Code quality

Background

- Fact 41: Maintenance typically consumes 40 to 80 percent of software costs. It is probably the most important life cycle phase of software.
- Fact 44: Understanding the existing product is the most difficult task of maintenance.

Source: Facts and Fallacies of Software Engineering (Robert L. Glass)

It's all about change

- Evolution is important from the start
 - Software is read much more than it is written
 - Understanding is a crucial component
 - Programming is an act of communication
- We can generalize this:
 - As soon as the first line of code is written, programming becomes maintenance
- Quality code is
 - easy to understand and change

How to characterize quality code?

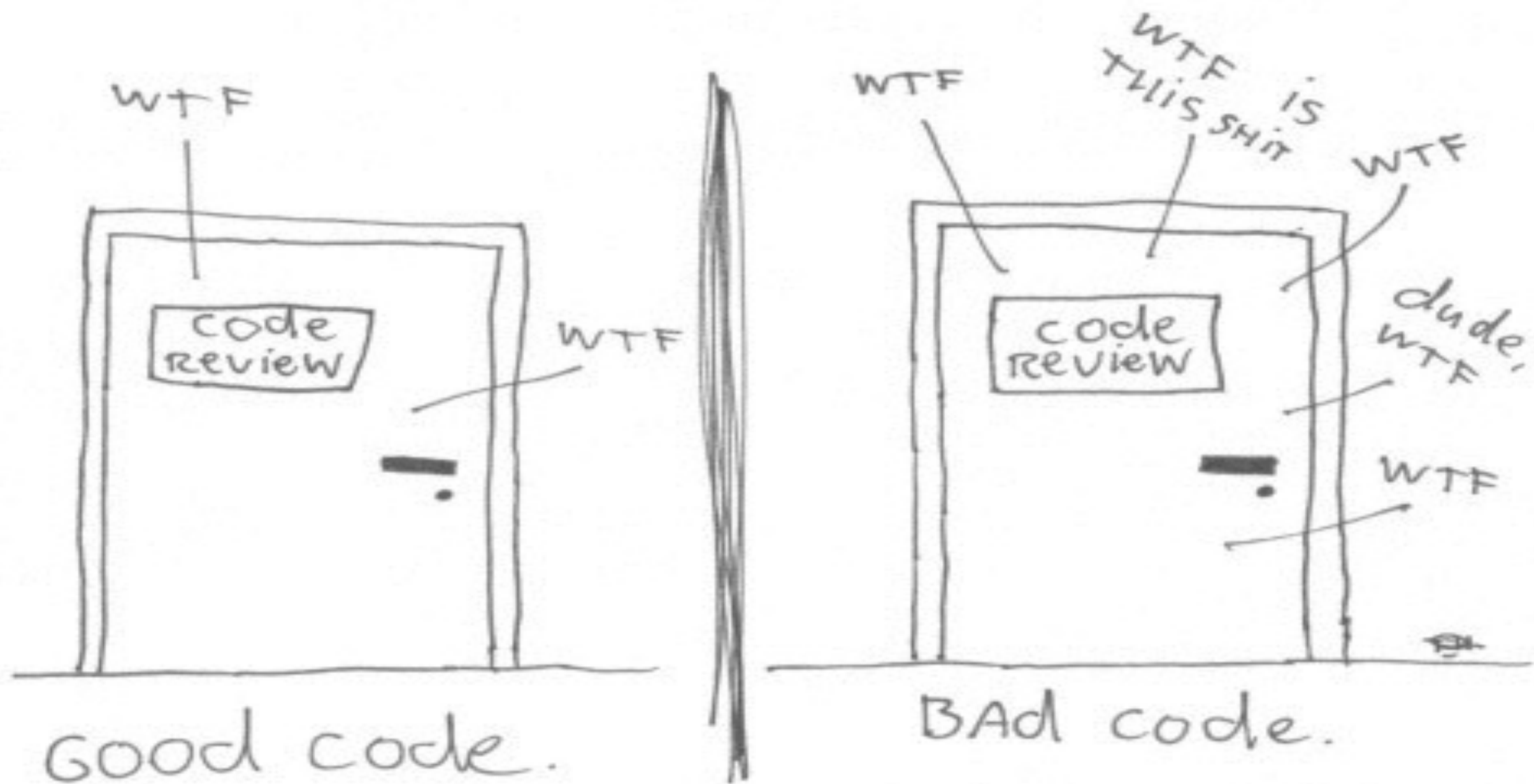
- DRY, OAOO, KISS, QWON



How to characterize quality code?

- DRY, OAOO, KISS, QWON
- DRY: Don't Repeat Yourself
- OAOO: Once And Only Once
- KISS: Keep It Simple Stupid
- QWON: Quality Without a Name

The ONLY valid measurement
of code quality: WTFs/minute



Some suggestions

- No duplication
- Well-factored
- Conceptual integrity
- Intentional naming
- Low complexity
- Strong cohesion
- Weak coupling
- Lean and mean
- Simple
- Testable
- Readable
- Not too smart
- Consistent
- Uniform
- Imperfect
- Flat
- ...

Duplication

Duplication

Duplication

- Two program elements or structures that are similar in a certain way
- Creates multiple sources of knowledge
- One change requires other changes
 - duplicate maintenance obligation
 - fault prone

Duplication ctd.

- Duplication related to coupling
 - implicit dependencies between distant elements
 - worse than normal coupling (which is explicit)
- Also counts for data
 - normal forms in databases
 - example later

Examples of duplication

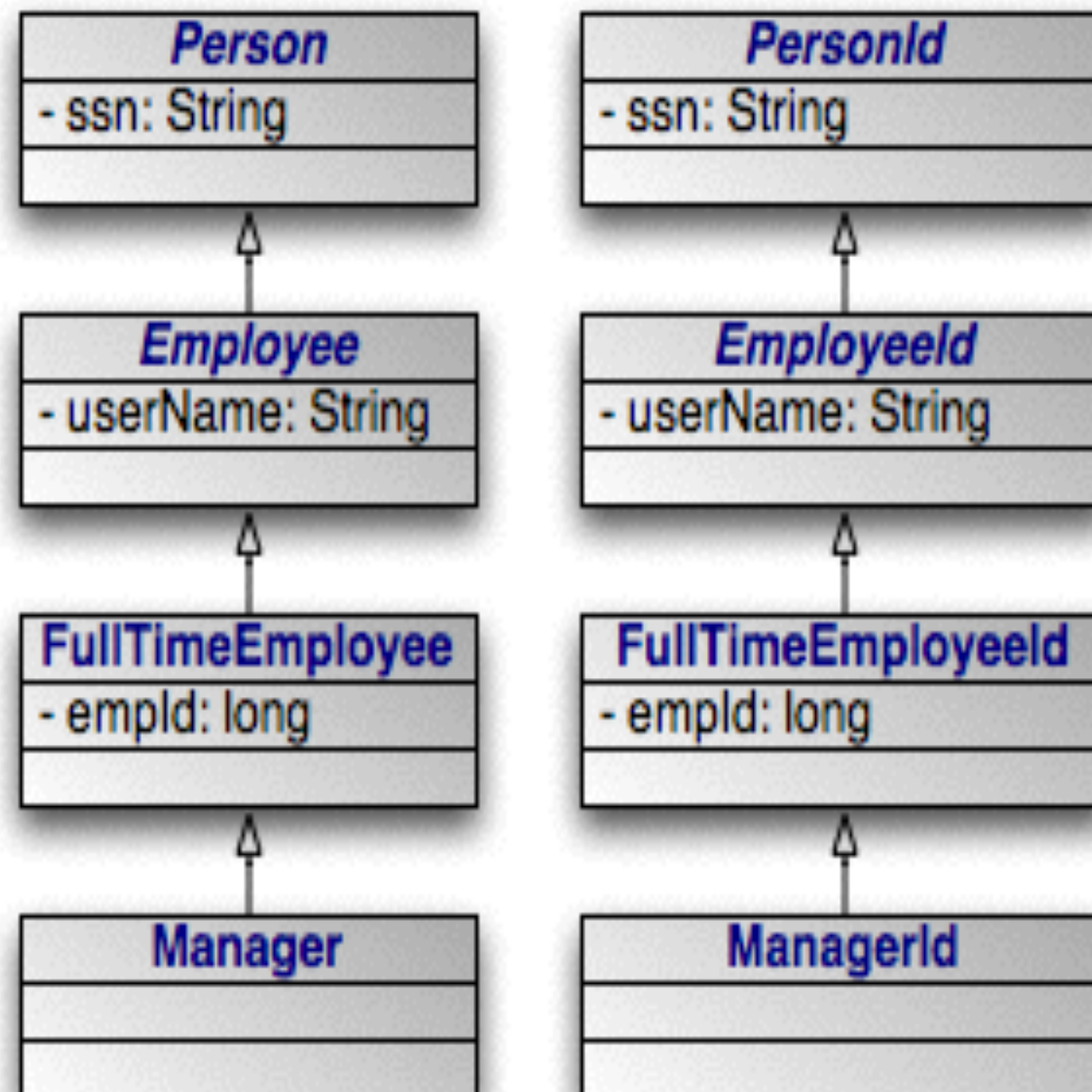
- copy-pasted code (clones)
- parallel inheritance hierarchies
- Implicit coupling
- sequence of calls where order matters
- similar parameter lists
- conventions
- idioms

Copy-pasted code

```
String error = "";
if (!containsNode(source)) {
    if (source == null) {
        error += "source is null";
    } else {
        error += "source : " + source.getId() + " ," + source.getName();
    }
    error += "\n";
}
if (!containsNode(target)) {
    if (source == null) {
        error += "target is null";
    } else {
        error += "target : " + target.getId() + " ," + target.getName();
    }
    error += "\n";
}
if (!error.equals("")) {
    throw new NoSuchElementException(error);
}
```

**DIRECT
CONSEQUENCE
OF COPY
PASTING!!!**

Parallel inheritance hierarchies



Implicit coupling and Order dependence

- Implicit coupling

- `edge.getSource().firePropertyChange(...);`
 `edge.getTarget().firePropertyChange(...);`

- Order dependence:

```
class Circle {  
    public void setRadius(double r) { ... }  
    public double getCircumference() { ... } }  
Circle c = new Circle();  
print(c.getCircumference());
```

- Also for parameter lists

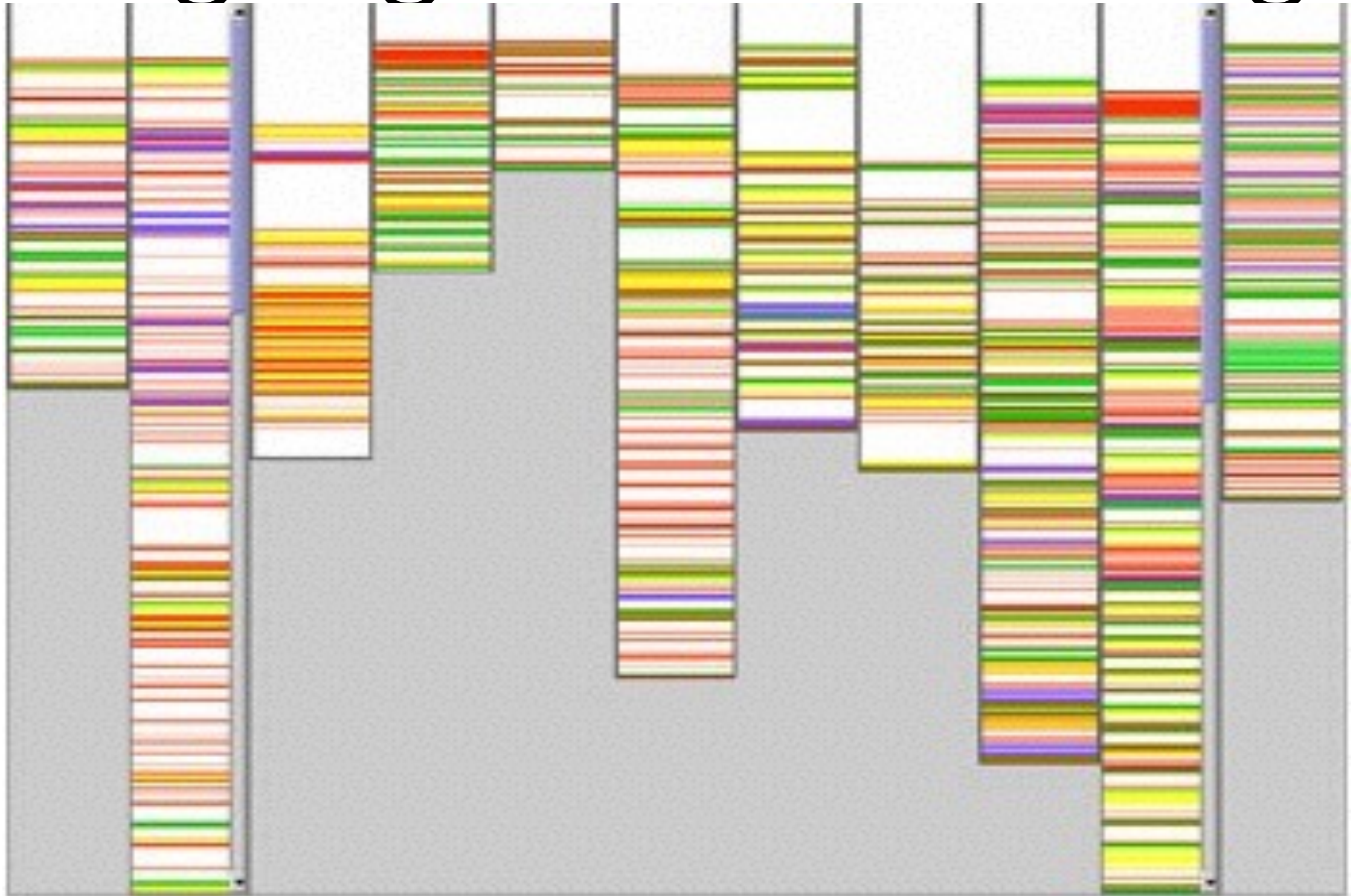
Tangling

Tangling

Tangling and scattering

- Tangling: one piece of code has multiple responsibilities
- Scattering: one responsibility is distributed over different pieces of code
- Common with cross cutting concerns
- Problems:
 - Scattering is duplication
 - Tangling makes code hard to understand
 - Change is error-prone

Tangling and scattering



Problems duplication & tangling

- Duplication: "the same in different places"
 - duplicate maintenance
 - change is fault-prone
- Tangling: "different things in one place"
 - complicates understanding
 - fault-prone

Show me some (bad) code!

Introduction

- Code from project some years ago
- Debugger for Toolbus Middleware
- Some examples to illustrate the topics of duplication and tangling

```

public SourceFileTableModel(String source) {
    String sourceFile = source;
    m_lineNumbers = getTextLineNumbers(sourceFile);
    m_sourceLines = new Object [m_lineNumbers][COLUMN_COUNT];
    int index = 0;
    final int breakPointPosition = 0;
    final int sourceCodeLineNumber = 1;
    final int text = 2;
    int firstEndLine = 0;
    int nextLine = 0;
    while (sourceFile.length() != 0) {
        if (sourceFile.contains("\n")) {
            firstEndLine = sourceFile.indexOf("\n");
            nextLine = firstEndLine + 1;
        } else {
            firstEndLine = sourceFile.length();
            nextLine = firstEndLine;
        }
        String firstLine = sourceFile.substring(0, firstEndLine);
        m_sourceLines[index][breakPointPosition] = false;
        m_sourceLines[index][sourceCodeLineNumber] = index + 1;
        firstLine = firstLine.replaceAll("\t", "    ");
        m_sourceLines[index][text] = firstLine;
        sourceFile = sourceFile.substring(nextLine, sourceFile.length());
        index++;
    }
}

```

m_lineNumbers is dependent on source as is m_sourceLines yet they are maintained separately

One loop that serves multiple purposes:

- initialization
- line counting
- splitting
- normalization

By the way...

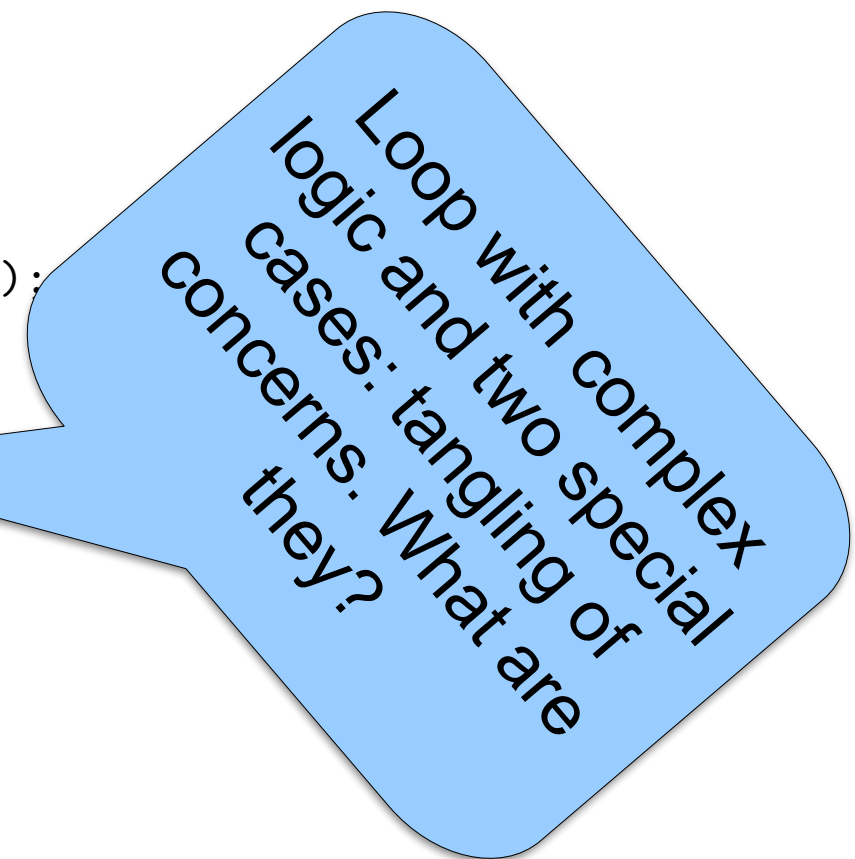
```
public int getTextLineNumbers(String source) {  
  
    String sourceFile = source;  
    int lineNumbers = 0;  
    int firstEndLine = 0;  
    int nextLine = 0;  
    while (sourceFile.length() != 0) {  
        if (sourceFile.contains("\n")) {  
            firstEndLine = sourceFile.indexOf("\n");  
            nextLine = firstEndLine + 1;  
        } else {  
            firstEndLine = sourceFile.length();  
            nextLine = firstEndLine;  
        }  
        sourceFile = sourceFile.substring(nextLine, sourceFile.length());  
        lineNumbers++;  
    }  
    return lineNumbers;  
}
```

This loop
looks
familiar...

Only here it's
just counting
lines, not
storing them

Another example of tangling

```
protected String createMultilineString(String text) {  
    if (text.length() <= CHARACTERS_PER_LINE) {  
        return text;  
    }  
  
    StringBuilder multilineText = new StringBuilder();  
    int linesCount = text.length() / CHARACTERS_PER_LINE;  
    for (int i = 0; i < linesCount; i++) {  
        int beginIndex = i * CHARACTERS_PER_LINE;  
        int endIndex = beginIndex + CHARACTERS_PER_LINE;  
  
        String line = text.substring(beginIndex, endIndex);  
        multilineText.append(line);  
        multilineText.append("<br>");  
    }  
  
    int beginIndex = CHARACTERS_PER_LINE * linesCount;  
    int endIndex = text.length();  
    String lastLine = text.substring(beginIndex, endIndex);  
    multilineText.append(lastLine);  
  
    return multilineText.toString();  
}
```



Refactoring opportunities

- The two responsibilities:
 - splitting and joining
- Separate the two concerns
 - using helper method and/or library class
- Result could be:

```
protected String createMultilineHTMLString(String text) {  
    return join(split(text, CHARACTERS_PER_LINE), "<br>");  
}
```

 - join contains the loop complexity just for joining

Summary

- Duplication: similar loops for splitting, counting, joining
- Tangling: joining and splitting in one loop
- Coevolving state: `m_lineNumbers` vs `m_sourceLines` (implicit coupling)
- Logic encoded using (local!) literals: 0, 1, 2 in table
 - missing abstraction?

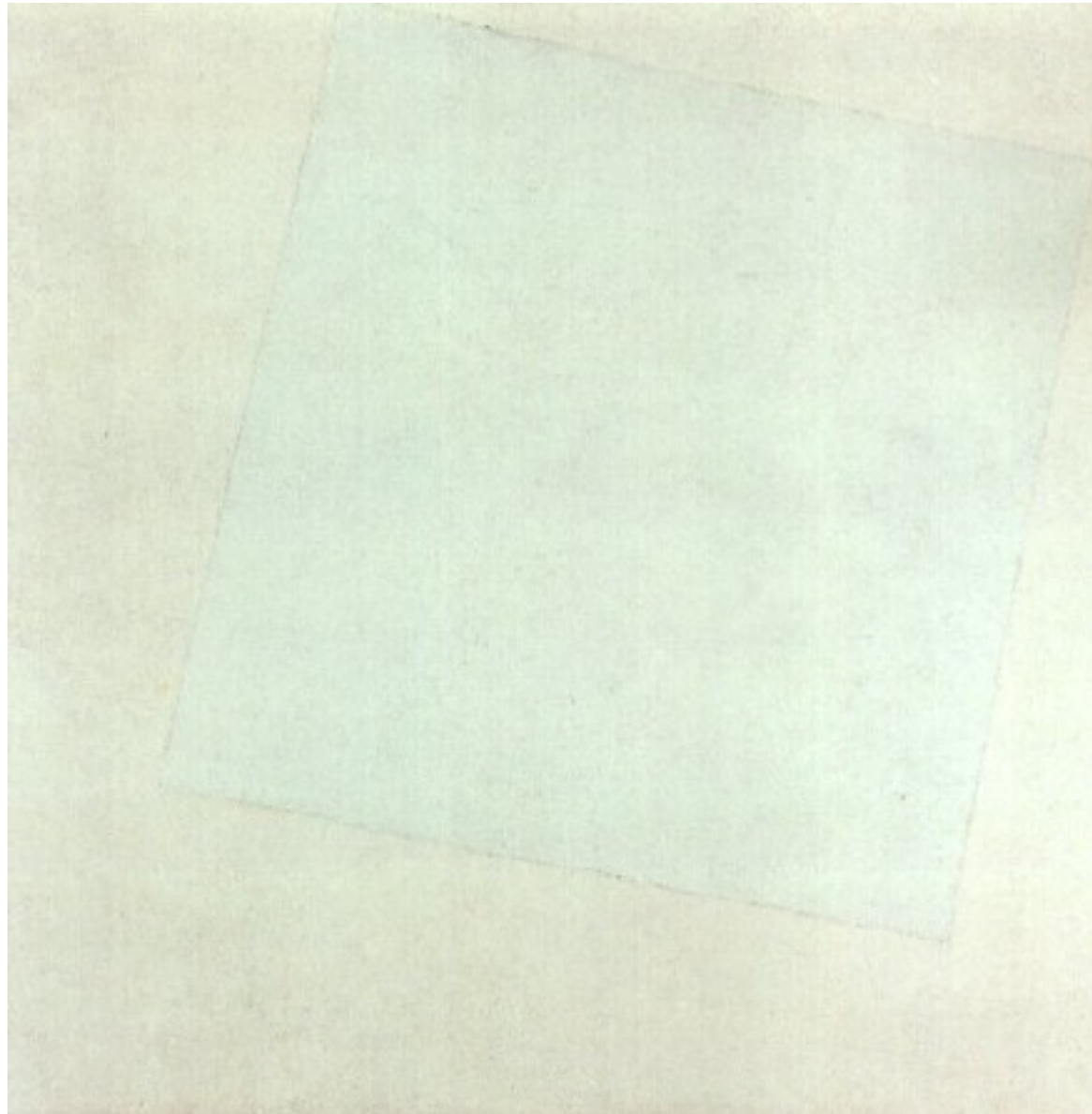
Problems duplication & tangling

- Duplication: "the same in different places"
 - duplicate maintenance
 - change is fault-prone
- Tangling: "different things in one place"
 - complicates understanding
 - fault-prone
- How to avoid these problems?

Abstraction

Abstraction

Abstraction



Malevich: "White on White" , 1918

Abstraction

- Leaving out details
 - information hiding (implementation and data)
- Allowing systematic variation
 - parameterization
- Giving a name to something
 - use the name of something instead of the thing itself
- Decreasing degrees of freedom
 - freedom also means freedom to make mistakes

Benefits of abstraction

- Benefits
 - single point of change
 - understanding through intention revealing names
 - reuse
 - variation
 - lower complexity through factoring
 - compression (i.e. less code)

Exercise: Abstraction in action

- Simple program
- Spot the duplication/coupling/degrees of freedom
- Resolve using abstraction

Abstraction in action

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

Abstraction in action

```
i = 0;  
next:  
    i++;  
    print i*i  
    if i <= 5  
        goto next;
```

Abstraction in action

```
i = 1;  
while i <= 5 do  
    print i*i  
    i++;  
end
```

Abstraction in action

```
for i = 1 to 5 do  
  print i*i  
end
```

Abstraction in action

```
def squares
  for i = 1 to 5 do
    print i*i
  end
end
```

Abstraction in action

```
def squares(n)
  for i = 1 to n do
    print i*i
  end
end
```


Abstraction in action

- Fewer degrees of freedom with the first steps
 - sequence to gotos to while loop to for loop
 - less risk of error
- Last two steps allow reuse and enable single point of change

- naming

- parameterization

```
def squares(n)
  for i = 1 to n do
    print i*i
  end
end
```

Abstraction in short

- Abstraction enables
 - well-factored code
 - problem decomposition
 - separation of concerns
 - reuse
 - variation
 - single point of change (DRY, OA00)
- But there are problems of abstraction as well...

Abstraction trade-offs

- Dependencies create distance
 - use site vs definition site
- Condensed/compressed code can be harder to understand
 - Good naming is essential

Trade-offs ctd.

- May introduce performance penalty
 - method call overhead
 - object allocation
- Law of leaky abstractions
 - the hidden implementation seeps through
 - client code must have knowledge of the implementation
- Risk of overdesign: making things too generic
 - YAGNI: You Ain't Gonna Need It

Abstraction and understanding

- Understanding crucial during maintenance
- Using abstraction: you need more context
 - Context can be far away
- Copy-pasted code: you can see what is happening
 - Distance is zero, as it were

Understanding compressed code

- Try to understand the following (Lisp) code:

```
(mismatch sequence list :from-end t  
                :start1 20 :start2 40  
                :end1 120 :end2 140 :test #'baz)
```

- (From Richard P. Gabriel, "Patterns of Software")

Why is this hard to understand?

- Meaning of "mismatch"
- Meaning of "baz"
- Meaning of the keyword parameters
- Mismatch is a control abstraction

Now try to understand this:

```
(let ((subseq1 (reverse (subseq sequence 20 120)))  
      (subseq2 (reverse (subseq list 40 140))))  
  (flet ((the-same (x y) (baz x y)))  
    (loop for index upfrom 0  
          as item1 in subseq1  
          as item2 in subseq2  
          finally (return t) do  
            (unless (the-same item1 item2)  
              (return index))))))
```


Patterns

Patterns

Contra extreme abstraction

- Alternative viewpoint:
 - extreme abstraction makes programs hard to read
 - patterns, conventions, idioms achieve better quality
- Patterns of program design:
 - documented, recognizable ways of solving certain problem
- Examples:
 - Design patterns (Gang of Four, GOF, book)
 - Implementation patterns (new book by Kent Beck)

Patterns

- Patterns are not abstractions
- Descriptions of solutions to common problems that can be tailored to certain context
- Examples:
 - Abstract factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
 - Visitor: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Patterns and code literacy

- Patterns can be
 - learned
 - recognized
 - discussed
 - taught
- Use of patterns thus can help understanding
- Programming as an act of communication but not though abstraction

Refactoring trade-off example

- Refactoring trade-off example

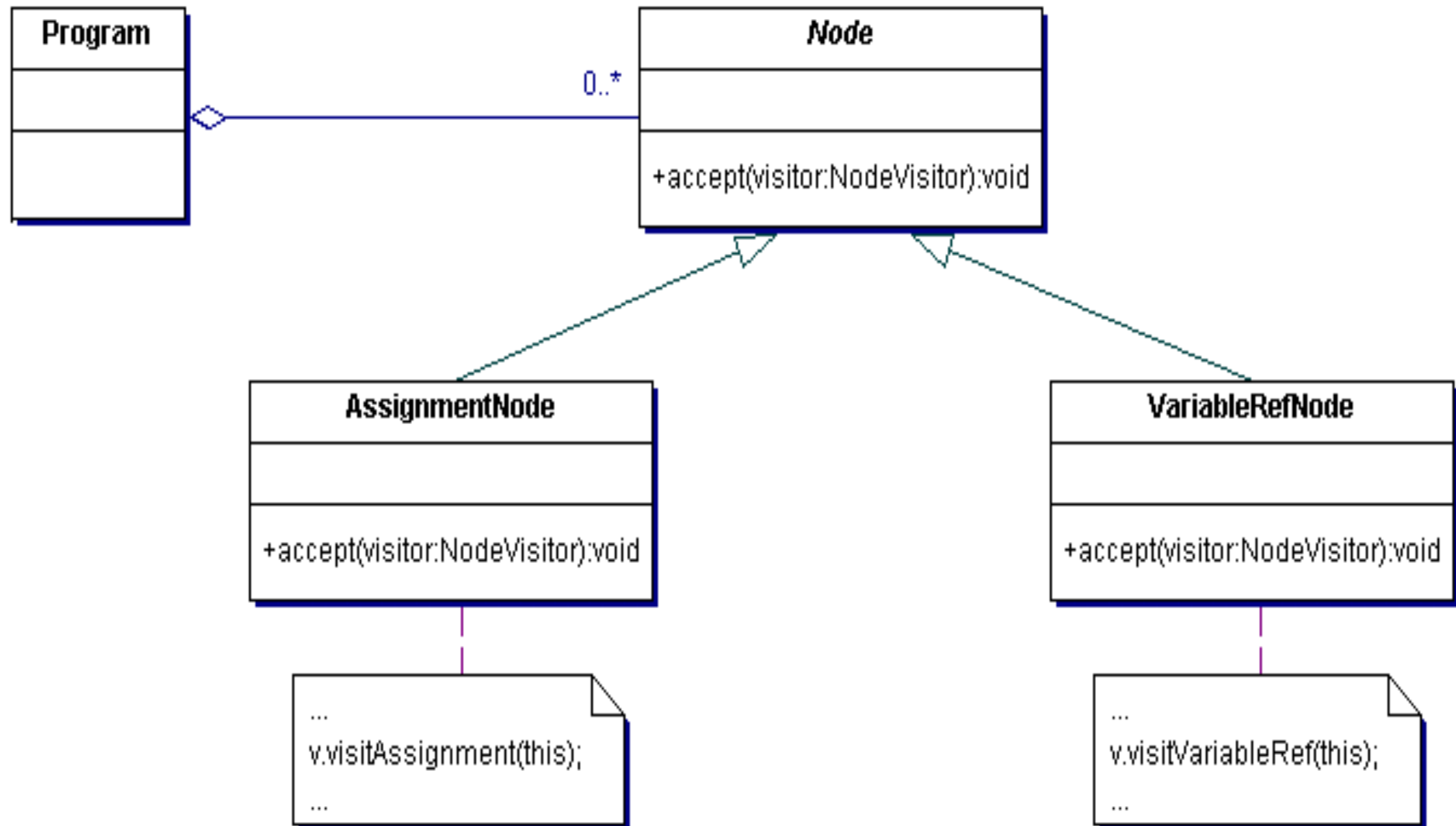
Example of refactoring trade-off

- Duplication vs factoring
- Use of design patterns
 - Visitor
 - Double dispatch
- Real case in Rascal implementation
 - By Paul, Jurgen and me
- Interpreter of Rascal contained "bad" code

Rascal

- Rascal is a typed functional languages for software analysis and transformation
- Many operators are overloaded
- Interpreter checks whether operations are allowed:
 - $1 + 2$: 3
 - $[1] + [2]$: [1, 2]
 - "1" + "2": "12"
 - $1 + \text{"x"}$: error
- How to implement this?

Visitor pattern



Visitor pattern ctd.

- Can be used for implementing interpreters
- Abstract Syntax Trees have accept()
- Interpreter has visitExp, visitStat, visitDecl etc.
- During visiting
 - Expressions get evaluated
 - Variables get assigned
 - Etc. Etc.
- If statements do the typechecking on overloading.

Addition...

(integers, doubles, strings)

@Override

```
public Result visitExpressionAddition(Addition x) {
    Result left = x.getLhs().accept(this);
    Result right = x.getRhs().accept(this);
    widenArgs(left, right);
    Type resultType = left.getType().lub(right.getType());
    if (left.getType().isIntegerType() && right.getType().isIntegerType()) {
        return result(((Integer) left.getValue()).add(((Integer) right.getValue())));
    }
    if (left.getType().isDoubleType() && right.getType().isDoubleType()) {
        return result(((Double) left.getValue()).add(((Double) right.getValue())));
    }
    if (left.getType().isStringType() && right.getType().isStringType()) {
        return result(vf.string(((String) left.getValue()).getValue()
            + ((String) right.getValue()).getValue()));
    }
}
```

```

if (left.getType().isListType()){
    if(right.getType().isListType()) {
        return result(resultType, ((IList) left.getValue())
            .concat((IList) right.getValue()));
    }
    if(right.getType().isSubtypeOf(left.getType().getElementType())){
        return result(left.getType(),
            (IList)left.getValue()).append(right.getValue()));
    }
}

if (right.getType().isListType()){
    if(left.getType().isSubtypeOf(right.getType().getElementType())){
        return result(right.getType(),
            (IList)right.getValue()).insert(left.getValue()));
    }
}

if (left.getType().isRelationType() && right.getType().isRelationType()) {
    return result(resultType, ((ISet) left.getValue())
        .union((ISet) right.getValue()));
}

```

```
if (left.getType().isSetType()){
    if(right.getType().isSetType()) {
        return result(resultType, ((ISet) left.getValue())
            .union((ISet) right.getValue()));
    }
    if(right.getType().isSubtypeOf(left.getType().getElementType())){
        return result(left.getType(),
            ((ISet)left.getValue()).insert(right.getValue()));
    }
}
if (right.getType().isSetType()){
    if(left.getType().isSubtypeOf(right.getType().getElementType())){
        return result(right.getType(),
            ((ISet)right.getValue()).insert(left.getValue()));
    }
}
if (left.getType().isMapType() && right.getType().isMapType()) {
    return result(resultType, ((IMap) left.getValue())
        .join((IMap) right.getValue()));
}
```

```

if(left.getType().isTupleType() && right.getType().isTupleType()) {
    Type leftType = left.getType();
    Type rightType = right.getType();
    int leftArity = leftType.getArity();
    int rightArity = rightType.getArity();
    int newArity = leftArity + rightArity;
    Type fieldTypes[] = new Type[newArity];
    String fieldNames[] = new String[newArity];
    IValue fieldValues[] = new IValue[newArity];
    for(int i = 0; i < leftArity; i++){
        fieldTypes[i] = leftType.getFieldType(i);
        fieldNames[i] = leftType.getFieldName(i);
        fieldValues[i] = ((ITuple) left.getValue()).get(i);
    }
    for(int i = 0; i < rightArity; i++){
        fieldTypes[leftArity + i] = rightType.getFieldType(i);
        fieldNames[leftArity + i] = rightType.getFieldName(i);
        fieldValues[leftArity + i] = ((ITuple) right.getValue()).get(i);
    }
    for(int i = 0; i < newArity; i++){
        if(fieldNames[i] == null){
            fieldNames[i] = "f" + String.valueOf(i);
        }
    }
    Type newTupleType = tf.tupleType(fieldTypes, fieldNames);
    return result(newTupleType, vf.tuple(fieldValues));
}

```

Otherwise: error

```
throw new TypeError(  
    "Operands of + have illegal types: "  
    + left.getType() + ", "  
    + right.getType(), x);  
}
```

Not so nice

- A lot of duplication:
 - Similar if statements
- Hard to understand
 - Control flow sometimes depends on earlier side-effects
- This was just +;
 - -, *, /, >, <, >=, <=, ., o, join
 -

Double dispatch to the rescue

- Take inspiration from numbers in Smalltalk
- $1 + 2$ is method call: `1.+(2)`
- In Integer class
 - `+ X`
 - `x addInt: self`
 - `addInt: x`
 - `// do the addition`

In Java

```
public interface Value {  
    public Value add(Value arg);  
    public Value addInt(Int arg);  
    public Value addReal(Real arg);  
}
```

```
public class Int implements Value  
{  
    int n;  
    public Int(int n) { this.n = n; }  
    public Value add(Value arg) {  
        return arg.addInt(this);  
    }  
    public Value addInt(Int arg) {  
        return new Int(arg.n + n);  
    }  
    public Value addReal(Real arg) {  
        return new Real(arg.f + n);  
    }  
}
```

```
public class Real implements Value {  
    float f;  
    public Real(float f) { this.f = f; }  
    public Value add(Value arg) {  
        return arg.addReal(this);  
    }  
    public Value addInt(Int arg) {  
        return new Real(arg.n + f);  
    }  
    public Value addReal(Real arg) {  
        return new Real(arg.f + f);  
    }  
}
```

Example

- $(1).\text{add}(2) \rightarrow (2).\text{addInt}(1) \rightarrow 3$
- $(1).\text{add}(2.0) \rightarrow (2.0).\text{addInt}(1) \rightarrow 3.0$
- $(1.0).\text{add}(2.0) \rightarrow (2.0).\text{addReal}(1.0) \rightarrow 3.0$

Result in Rascal

```
public Result<IValue> visitExpressionAddition(Addition x) {  
    Result<IValue> left = x.getLhs().accept(this);  
    Result<IValue> right = x.getRhs().accept(this);  
  
    return left.add(right,  
        new EvaluatorContext(this, x));  
}
```

Typechecking is dispatch

- "Correspondence"
 - If and case statements
 - Dynamic dispatch of Java
- Complex logic has disappeared
- Dispatch handles typechecking
 - Default "add" method in superclass Value throws TypeError

But...

- Pros
 - Adding a new overloaded implementation is as easy as adding the necessary methods
 - For each data type all relevant operator implementations are in a single class
- Cons
 - First all typechecking related to a single operator was at one place
 - Now it's scattered over each data class
- Trade-off!

Multiple perspectives

- First
 - Tangling from the perspective of data class
 - To understand how `+` works on integers you have to understand complex control-flow
- Now
 - Scattering from the perspective of the operations
 - To understand the evaluation of an operation you have to look at many classes

Conclusion

Conclusion

Concluding

- Duplication and tangling detrimental to quality
- Abstraction is the key mechanism
- Overuse of abstraction has its own problems
- Patterns may provide a useful middle ground
- However, trade-offs remain.



Kent Beck

@KentBeck



Following

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

Reply Retweet Favorite More