

Final notes on Software Construction

Tijs van der Storm

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.

Programming

Theory of programming

- Values
 - Communication, simplicity, flexibility
- Principles
 - Local consequences, minimize repetition, logic together with data, symmetry, declarative expression, rate of change



Understanding

- Good names
- Small modules
- Consistent layout
- Document assumptions
- Tests



Intention revealing
names

```
private boolean isIgnored(Word word) {  
    return isStopWord(word) || isTooShort(word);  
}
```

```
private boolean isTooShort(Word word) {  
    return word.length() < MINIMUM_LENGTH;  
}
```

```
private boolean isStopWord(Word word) {  
    return stopWords.isStopWord(word);  
}
```

Modules

- Packages
- Classes
- Methods

Information hiding
Encapsulation
Interfaces

On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a “modularization” is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

Key Words and Phrases: software, modules, modularity, software engineering, KWIC index, software design

CR Categories: 4.0

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:¹

A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.

Hide the
implementation
decision


```
public class Word {  
    private final String value;  
  
    public Word(String value) {  
        assert !value.isEmpty();  
        this.value = value;  
    }  
}
```

Expose interfaces
internally

```
public class Distribution {  
    private Map<Word, Frequency> table;  
  
    public Distribution() {  
        this.table = new HashMap<Word, Frequency>();  
    }  
}
```

Hide the
implementation
decision

```
public class Source implements Iterable<Word> {  
    private final List<Word> words;  
  
    @Override  
    public Iterator<Word> iterator() {  
        return words.iterator();  
    }  
}
```



Don't leak the List
implementation
details

Contracts Protocols

Applying “Design by Contract”

Bertrand Meyer

Interactive Software Engineering

As object-oriented techniques steadily gain ground in the world of software development, users and prospective users of these techniques are clamoring more and more loudly for a “methodology” of object-oriented software construction — or at least for some methodological guidelines. This article presents such guidelines, whose main goal is to help improve the reliability of software systems. *Reliability* is here defined as the combination of correctness and robustness or, more prosaically, as the absence of bugs.

Everyone developing software systems, or just using them, knows how pressing this question of reliability is in the current state of software engineering. Yet the rapidly growing literature on object-oriented analysis, design, and programming includes remarkably few contributions on how to make object-oriented software more reliable. This is surprising and regrettable, since at least three reasons justify devoting particular attention to reliability in the context of object-oriented development:

If you promise value
is not empty, I
promise length() > 0

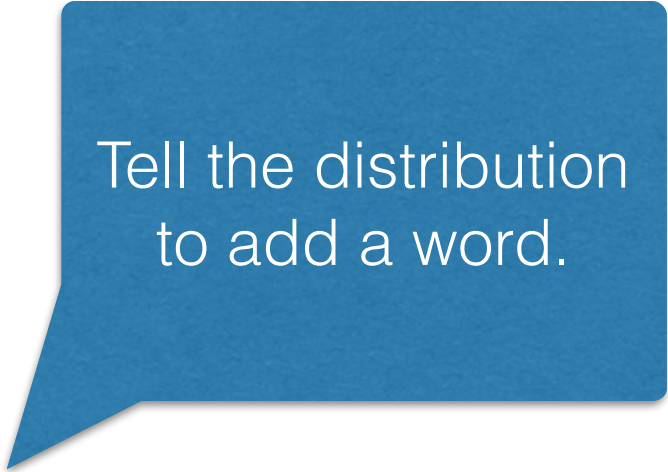
```
public class Word {  
    private final String value;  
  
    public Word(String value) {  
        assert !value.isEmpty();  
        this.value = value;  
    }  
  
    public int length() {  
        int len = value.length();  
        assert len > 0;  
        return len;  
    }  
}
```

Document the
contract

Dependencies Parameterization


```
public class TermFrequency {  
    private Source source;  
    private StopWords stopWords;  
  
    public TermFrequency(Source source, StopWords stopWords) {  
        this.source = source;  
        this.stopWords = stopWords;  
    }  
}
```

Make dependencies explicit
(dependency inversion)



Tell the distribution
to add a word.

```
public void add(Word word) {  
    initializeWordFrequencyIfNeeded(word);  
    incrementWordFrequency(word);  
}
```

```
private void incrementWordFrequency(Word word) {  
    table.get(word).increment();  
}
```



Tell the frequency to
increment itself

Assuring Good Style for Object-Oriented Programs

Karl J. Lieberherr and Ian M. Holland, Northeastern University

The language-independent Law of Demeter encodes the ideas of encapsulation and modularity in an easy-to-follow form for object-oriented programmers.

When is an object-oriented program written in good style? Is there some formula or rule that you can follow to write good object-oriented programs? What metrics can you apply to an object-oriented program to determine if it is good? What are the characteristics of good object-oriented programs?

In this article, we put forward a simple law, called the Law of Demeter, that we believe answers these questions and helps formalize the ideas in the literature.^{1,2} There are two kinds of style rules for ob-

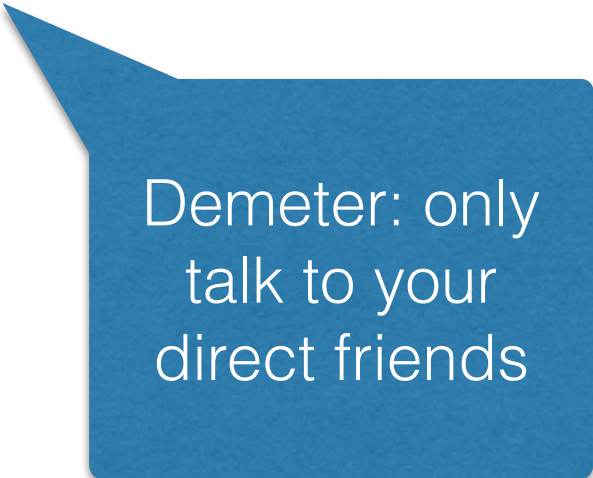
ject-oriented programs. Informally, the law says that each method can send messages to only a limited set of objects: to argument objects, to the self pseudovisible, and to the immediate subparts of self. (The self construct in Smalltalk and Flavors is called "this" in C++ and "Current" in Eiffel.) In other words, each method depends on a limited set of objects.

The goal of the Law of Demeter is to organize and reduce dependencies between classes. Informally, one class depends on another class when it calls a function defined in the other class. We be-

```
public TermFrequency(Source source, StopWords stopWords) {  
    this.source = source;  
    this.stopWords = stopWords;  
}
```

```
public void termFrequency(Distribution distribution) {  
    for (Word word: source) {  
        recordWord(distribution, word);  
    }  
}
```

```
private boolean isStopWord(Word word) {  
    return stopWords.isStopWord(word);  
}
```



Demeter: only
talk to your
direct friends

Parameterize over what kind of distribution TF operates on.

```
public class TermFrequency {  
    public void termFrequency(Distribution distribution) {  
        for (Word word: source) {  
            recordWord(distribution, word);  
        }  
    }  
}
```

Why Functional Programming Matters

John Hughes
The University, Glasgow

Abstract

As software becomes more and more complex, it is more and more important to structure it well. Well-structured software is easy to write and to debug, and provides a collection of modules that can be reused to reduce future programming costs. In this paper we show that two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute significantly to modularity. As examples, we manipulate lists and trees, program several numerical algorithms, and implement the alpha-beta heuristic (an algorithm from Artificial Intelligence used in game-playing programs). We conclude that since modularity is the key to successful programming, functional programming offers important advantages for software development.

The Power of Interoperability: Why Objects Are Inevitable

Jonathan Aldrich

Carnegie Mellon University

Pittsburgh, PA, USA

aldrich@cs.cmu.edu

Aim, Fire

Kent Beck

“Never write a line of functional code without a broken test case.”
—*Kent Beck*

“Test-first coding is not a testing technique.”
—*Ward Cunningham*

Calisthenics?

The Rules

1. One level of indentation per method
2. Don't use the ELSE keyword
3. Wrap all primitives and Strings
4. First class collections
5. One dot per line
6. Don't abbreviate
7. Keep all entities small
8. No classes with more than two instance variables
9. No getters/setters/properties

Classes for
dispatch

Simplicity

Information hiding
& encapsulation

Demeter

Communication

Minimize
dependencies

Simplicity

Tell don't ask

Summary

- Make small modules
- Minimize dependencies
- Make dependencies small and explicit
- Depend on abstractions
- Encapsulate implementation decisions
- Document your assumptions
- Don't repeat yourself



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

It Is Possible to Do Object-Oriented Programming in Java

Kevlin Henney
kevlin@curbralan.com
@KevlinHenney

evlin Henney
vlin@curbralan.com
@KevlinHenney

