Design principles

(some of which are in the syllabus)

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.

Design

Symptoms of Rotting Design

There are four primary symptoms that tell us that our designs are rotting. They are not orthogonal, but are related to each other in ways that will become obvious. they are: rigidity, fragility, immobility, and viscosity.

Rigidity. Rigidity is the tendency for software to be difficult to change, even in simple ways. Every change causes a cascade of subsequent changes in dependent modules. What begins as a simple two day change to one module grows into a multi-week marathon of change in module after module as the engineers chase the thread of the change through the application.

Fragility. Closely related to rigidity is fragility. Fragility is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed. Such errors fill the hearts of managers with foreboding. Every time they authorize a fix, they fear that the software will break in some unexpected way.

Immobility. Immobility is the inability to reuse software from other projects or from parts of the same project. It often happens that one engineer will discover that he needs a module that is similar to one that another engineer wrote. However, it also often happens that the module in question has too much baggage that it depends upon. After much work, the engineers discover that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate. And so the software is simply rewritten instead of reused.

The Open Closed Principle (OCP)[1]

A module should be open for extension but closed for modification.

The Liskov Substitution Principle (LSP)[1]

Subclasses should be substitutable for their base classes.

The Dependency Inversion Principle (DIP)[1]

Depend upon Abstractions. Do not depend upon concretions.

The Interface Segregation Principle (ISP)[2]

Many client specific interfaces are better than one general purpose interface

Design Patterns

		Characterization		
		Creational	**Structural**	**Behavioral**
Jurisdiction	**Class**	Factory Method	Adapter (class) Bridge (class)	Template Method
	Object	Abstract Factory Prototype Solitaire	Adapter (object) Bridge (object) Flyweight Glue Proxy	Chain of Responsibility Command Iterator (object) Mediator Memento Observer State Strategy
	Compound	Builder	Composite Wrapper	Interpreter Iterator (compound) Walker

Intent

What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Motivation

A scenario in which the pattern is applicable, the particular design problem or issue the pattern addresses, and the class and object structures that address this issue. This information will help the reader understand the more abstract description of the pattern that follows.

Applicability

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can one recognize these situations?

Participants

Describe the classes and/or objects participating in the design pattern and their responsibilities using CRC conventions [5].

Collaborations

Describe how the participants collaborate to carry out their responsibilities.

Diagram

A graphical representation of the pattern using a notation based on the Object Modeling Technique (OMT) [25], to which we have added method pseudo-code.

Consequences

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What does the design pattern objectify? What aspect of system structure does it allow to be varied independently?

Implementation

What pitfalls, hints, or techniques should one be aware of when implementing the pattern? Are there language-specific issues?

Examples

This section presents examples from real systems. We try to include at least two examples from different domains.

See Also

What design patterns have closely related intent? What are the important differences? With which other patterns should this one be used?

The
CHECKS
Pattern Language of Information Integrity

Any program that accepts user input will need to separate good input from bad, and to make sure little of the latter gets recorded. This pattern language tells how to make these checks without complicating the program and compromising future flexibility.

The language has eleven patterns presented in three sections. The first section describes values as they should be captured by the user-interface and used within the domain model. The second and third sections discuss detecting and correcting mistakes, first during data entry and then after posting or publication. The patterns draw from the author's experience developing financial software in Smalltalk. They are written as if part of a larger language and therefor may seem sketchy or incomplete. This paper is as much an experiment in the selection and linking of patterns as an attempt to communicate practical knowledge.

Section1. First consider quantities used by the domain model. Your domain code must express the "logic" of the business in its richest and often illogical detail. Every clause of every statement should be motivated by some business fact of life. Other concerns will be pushed into the specialized values of this section or pulled out into objects described later. The patterns are:

Arguments & results

Pattern	Problem	Solution
Arguments Object	How can you simplify a complex protocol that has a regular argument structure?	Make an Arguments Object to capture the common parts of the protocol.
Selector Object	How can you simplify a protocol where several messages differ mainly in their names?	Make a single message taking an object representing the message selector as an extra argument.
Curried Object	How can you simplify an extremely complicated protocol?	Send simpler messages to an intermediary which elaborates them within its context.
Result Object	How can you manage a difficult answer to a difficult question?	Make a Result Object the whole answer to the question.
Future Object	How can you answer a question while you think about something else?	Make a Future Object which computers the answer in parallel.
Lazy Object	How can you answer a question that is easy to answer now, but that may never be asked?	Make a Lazy Object which can answer the question later, if necessary.

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 Encapsulation Interfaces

Information hiding
Replace with different implementation interfaces

Programming Techniques
R. Morris Editor

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

Hide the implementation decision

```
public class Distribution  {
   private Map<Word,Frequency> table;

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

}
```

```
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"

NB: not in the syllabus

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:

```
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

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

Dependencies

Parameterization

Demeter
Single responsibility
Dependency injection
Segregation of interfaces

```
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)

move configuration outwards

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

Parameterize over what kind of distribution TF operates on.

```
public class TermFrequency {

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

}
```

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-

mally, the law says that each method can send messages to only a limited set of objects: to argument objects, to the self pseudovariable, 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

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.

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*

Object-Oriented Design:
A Responsibility-Driven Approach

Rebecca Wirfs-Brock
rebeccaw@orca.wv.tek.com
(503) 685-2561

P.O. Box 1000, Mail Station 61-028
Tektronix, Inc.
Wilsonville, OR 97070

Brian Wilkerson
(503) 242-0725

21 S.W. Washington, Suite 312
Instantiations, Inc.
Portland, OR 97205

RESPONSIBILITY
TOWARDS SOMEONE/
SOMETHING

Add is a responsibility of a Distribution

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

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

Simplicity

Classes for dispatch

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
- Use patterns to communicate
- Don’t repeat yourself