SOLID

Deconstruction

Kevlin Henney

kevlin@curbralan.com

@KevlinHenney

SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

A Pattern Language for Distributed Computing

Volume 4

Frank Buschmann
Kevlin Henney
Douglas C. Schmidt

SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

On Patterns and Pattern Languages

Volume 5

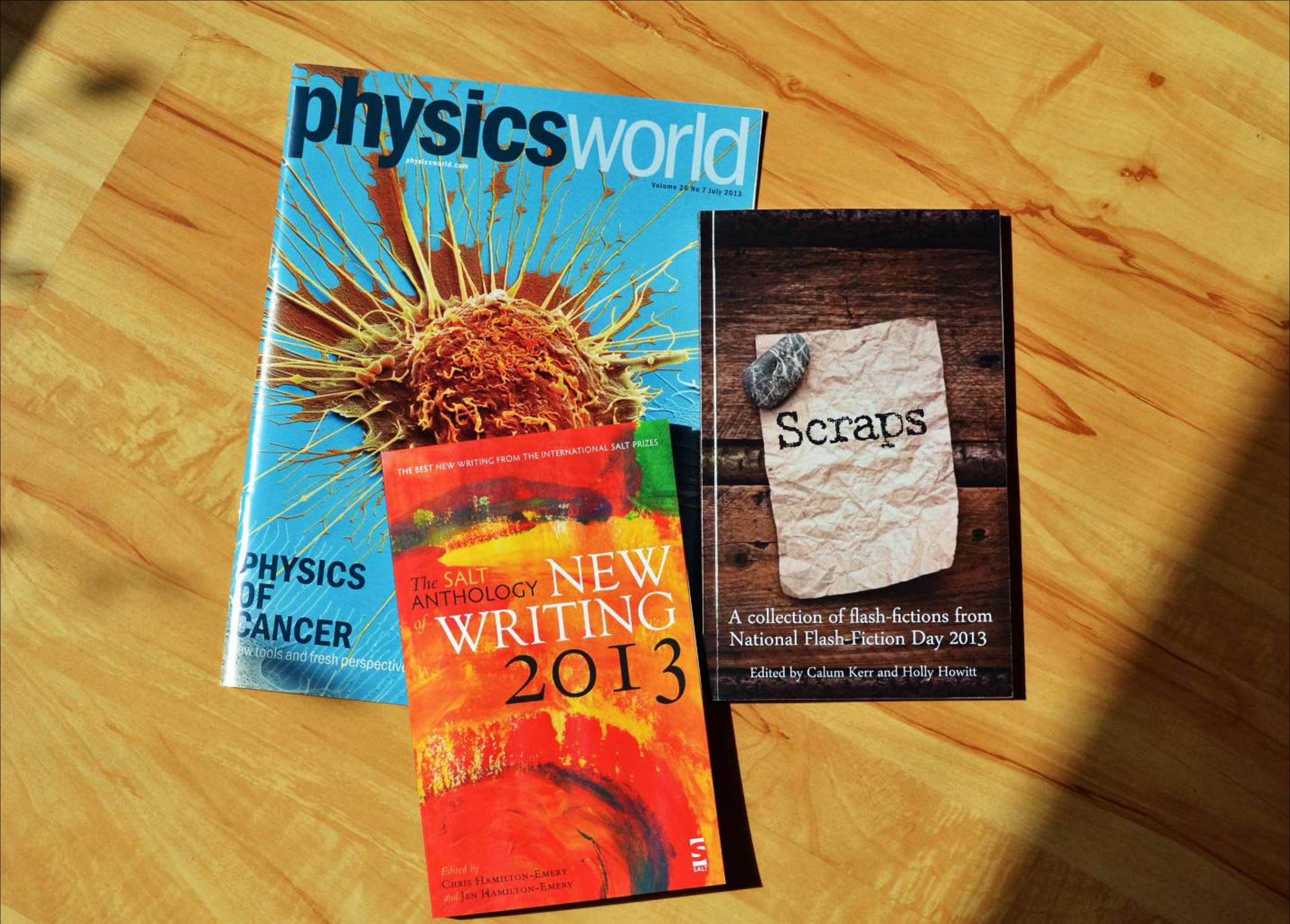
Frank Buschmann
Kevlin Henney
Douglas C. Schmidt

Collective Wisdom from the Experts

97 Things Every Programmer Should Know

O'REILLY

Edited by Kevlin Henney


physicsworld
physicsworld.com
Volume 26 No 7 July 2013
PHYSICS OF CANCER
ew tools and fresh perspective
THE BEST NEW WRITING FROM THE INTERNATIONAL SALT PRIZES
The SALT ANTHOLOGY NEW WRITING 2013
Edited by
Chris Hamilton-Emery
and Jen Hamilton-Emery
SALT
Scraps
A collection of flash-fictions from
National Flash-Fiction Day 2013
Edited by Calum Kerr and Holly Howitt

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

From: jim.f...@bytes.com (Jim Fleming)
Newsgroups: comp.object
Subject: The Ten Commandments of OO Programming
Date: 16 Mar 1995 00:26:14 GMT

We are trying to compose The Ten Commandments of OO Programming.

...

1. Class names should be nouns and method names should be verbs.
2. Comments are as important as the code and should be "classified".
3. Inheritance should help new developers "program the differences".
4. Encapsulation of everything, including source, is essential.
5. Verbs (methods) should be chosen carefully to support polymorphis
6. Reuse can only be acheived when standard classes are understandab
7. New developers should reuse the CASE tools of the experienced dev
8. Cross-platform portability should take priority over other design
9. A new class should be used for each new legacy library interface.
10. Multiple returns should be used for error handling and debugging

--
Jim Fleming   /|\   Unir Corporation   Unir Technolo
XTechno Cat I   / | \   One Naperville Plaza   184 Shuman Bl

From: rma...@rcmcon.com (Robert Martin)
Newsgroups: comp.object
Subject: Re: The Ten Commandments of OO Programming
Date: Thu, 16 Mar 1995 15:12:00 GMT

:::
1. Software entities (classes, modules, etc) should be open for
   extension, but closed for modification. (The open/closed
   principle -- Bertrand Meyer)

2. Derived classes must usable through the base class interface
   without the need for the user to know the difference. (The
   Liskov Substitution Principle)

3. Details should depend upon abstractions. Abstractions should
   not depend upon details. (Principle of Dependency Inversion)

:::

--
Robert Martin | Design Consulting | Training courses offered
Object Mentor Assoc.| rma...@rcmcon.com | Object Oriented Analysi
2080 Cranbrook Rd. | Tel: (708) 918-1004 | Object Oriented Design
Green Oaks IL 60048 | Fax: (708) 918-1023 | C++

From: rma...@rcmcon.com (Robert Martin)
Newsgroups: comp.object
Subject: Re: The Ten Commandments of OO Programming
Date: Thu, 16 Mar 1995 15:12:00 GMT

:::
4. The granule of reuse is the same as the granule of release.
   Only components that are released through a tracking system can
   be effectively reused.

5. Classes within a released component should share common closure.
   That is, if one needs to be changed, they all are likely to need
   to be changed. What affects one, affects all.

6. Classes within a releasedd componen should be reused together.
   That is, it is impossible to separate the components from each
   other in order to reuse less than the total.
:::

--
Robert Martin | Design Consulting | Training courses offered
Object Mentor Assoc.| rma...@rcmcon.com | Object Oriented Analysi
2080 Cranbrook Rd. | Tel: (708) 918-1004 | Object Oriented Design
Green Oaks IL 60048 | Fax: (708) 918-1023 | C++

principle

- *a fundamental truth or proposition that serves as the foundation for a system of belief or behaviour or for a chain of reasoning.*
- *morally correct behaviour and attitudes.*
- *a general scientific theorem or law that has numerous special applications across a wide field.*
- *a natural law forming the basis for the construction or working of a machine.*

Oxford Dictionary of English

pattern

- *a regular form or sequence discernible in the way in which something happens or is done.*
- *an example for others to follow.*
- *a particular recurring design problem that arises in specific design contexts and presents a well-proven solution for the problem. The solution is specified by describing the* roles *of its constituent participants, their responsibilities and* relationships*, and the ways in which they collaborate.*

Concise Oxford English Dictionary

Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages

Expert

Proficient

Competent

Advanced Beginner

Novice

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

In object-oriented programming, the single responsibility principle states that every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.

http://en.wikipedia.org/wiki/Single_responsibility_principle

The term was introduced by Robert C. Martin [...]. Martin described it as being based on the principle of cohesion, as described by Tom DeMarco in his book *Structured Analysis and Systems Specification*.

http://en.wikipedia.org/wiki/Single_responsibility_principle

Structured Analysis and System Specification

TOM DEMARCO
Foreword by: P.J. PLAUGER

YOURDON PRESS COMPUTING SERIES

25.2.4 Cohesion

Cohesion is a good quality exhibited by some design structures. Before I define it, look at Fig. 101, an alternate Structure Chart for the space vehicle guidance system we considered earlier. Fig. 101 is an abominable design. It is proof positive that one can design poorly even using a Structure Chart. ("Plowin' ain't potatoes.") What the design of Fig. 101 lacks is cohesion. Every module on the figure is weakly cohesive.

Fig. 99, on the other hand, is made up of strongly cohesive modules. By comparing the two figures, you can probably see exactly what cohesion is. It has to do with the integrity or "strength" of each module. The more valid a module's reason for existing as a module, the more cohesive it is.

Cohesion is a measure of the strength of association of the elements inside a module. A highly cohesive module is a collection of statements and data items that should be treated as a whole because they are so closely related. Any attempt to divide them up would only result in increased coupling and decreased readability.

25.2.4 Cohesion

Cohesion is a good quality exhibited by some design structures. Before I define it, look at Fig. 101, an alternate Structure Chart for the space vehicle guidance system we considered earlier. Fig. 101 is an abominable design. It is proof positive that one can design poorly even using a Structure Chart. ("Ptowie' ain't potatoes.") What the design of Fig. 101 lacks is cohesion. Every module on the figure is weakly cohesive.

Fig. 99, on the other hand, is made up of strongly cohesive modules. By comparing the two figures, you can probably see exactly what cohesion is. It has to do with the integrity or "strength" of each module. The more valid a module's reason for existing as a module, the more cohesive it is.

Cohesion is a measure of the strength of association of the elements inside a module. A highly cohesive module is a collection of statements and data items that should be treated as a whole because they are so closely related. Any attempt to divide them up would only result in increased coupling and decreased readability.

Glenn Vanderburg: Blog

Cohesion

Mon, 31 Jan 2011 (16:43) #

Developers I encounter usually have a good grasp of coupling—not only what it means, but why it's a problem. I can't say the same thing about cohesion. One of the sharpest developers I know sometimes has problems with the concept, and once told me something like "that word doesn't mean much to me." I've come to believe that a big part of the problem is the word "cohesion" itself. "Coupling" is something everyone understands. "Cohesion," on the other hand, is a word that is not often used in everyday language, and that lack of familiarity makes it a difficult word for people to hang a crucial concept on.

I've had some success teaching the concept of cohesion using an unusual approach that exploits the word's etymology. I know that sounds unlikely, but bear with me. In my experience, it seems to register well with people.

Cohesion comes from the same root word that "adhesion" comes from. It's a word about *sticking*. When something *adheres* to something else (when it's *adhesive*, in other words) it's a one-sided, external thing: something (like glue) is sticking one thing to another. Things that are *cohesive*, on the other hand, naturally stick to each other because they are of like kind, or because they fit so well together. Duct tape *adheres* to things because it's sticky, not because it necessarily has anything in common with them. But two lumps of clay will *cohere* when you put them together, and matched, well-machined parts sometimes seem to cohere because the fit is so precise. *Adhesion* is one thing sticking to

Glenn Vanderburg: Blo ×

www.vanderburg.org/blog/Software/Development/cohesion.rdoc

Glenn Vanderburg: Blog

home blog speaking misc contact

1 of 1 a

Cohesion

Mon,

We refer to a sound line of reasoning, for example, as coherent. The thoughts fit, they go together, they relate to each other. This is exactly the characteristic of a class that makes it coherent: the pieces all seem to be related, they seem to belong together, and it would feel somewhat unnatural to pull them apart. Such a class exhibits *cohesion*.

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together.

Doug McIlroy

The hard part isn't writing little programs that do one thing well. The hard part is combining little programs to solve bigger problems. In McIlroy's summary, the hard part is his second sentence: Write programs to work together.

John D Cook

http://www.johndcook.com/blog/2010/06/30/where-the-unix-philosophy-breaks-down/

The harder part isn't writing little programs that do one thing well. The harder part is combining little programs to solve bigger problems. In McIlroy's summary, the harder part is his second sentence: Write programs to work together.

Software applications do things they're not good at for the same reason companies do things they're not good at: to avoid transaction costs.

John D Cook

http://www.johndcook.com/blog/2010/06/30/where-the-unix-philosophy-breaks-down/

Package java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

utility

- the state of being useful, profitable or beneficial
- useful, especially through having several functions
- functional rather than attractive

Concise Oxford English Dictionary

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

Interface inheritance (subtyping) is used whenever one can imagine that client code should depend on less functionality than the full interface. Services are often partitioned into several unrelated interfaces when it is possible to partition the clients into different roles. For example, an administrative interface is often unrelated and distinct in the type system from the interface used by "normal" clients.

"General Design Principles"
CORBAservices

The dependency should be on the interface, the whole interface, and nothing but the interface.

Glenn Vanderburg: Blo ×

www.vanderburg.org/blog/Software/Development/cohesion.rdoc

Glenn Vanderburg: Blog

home blog speaking misc contact

1 of 1 a

Cohesion

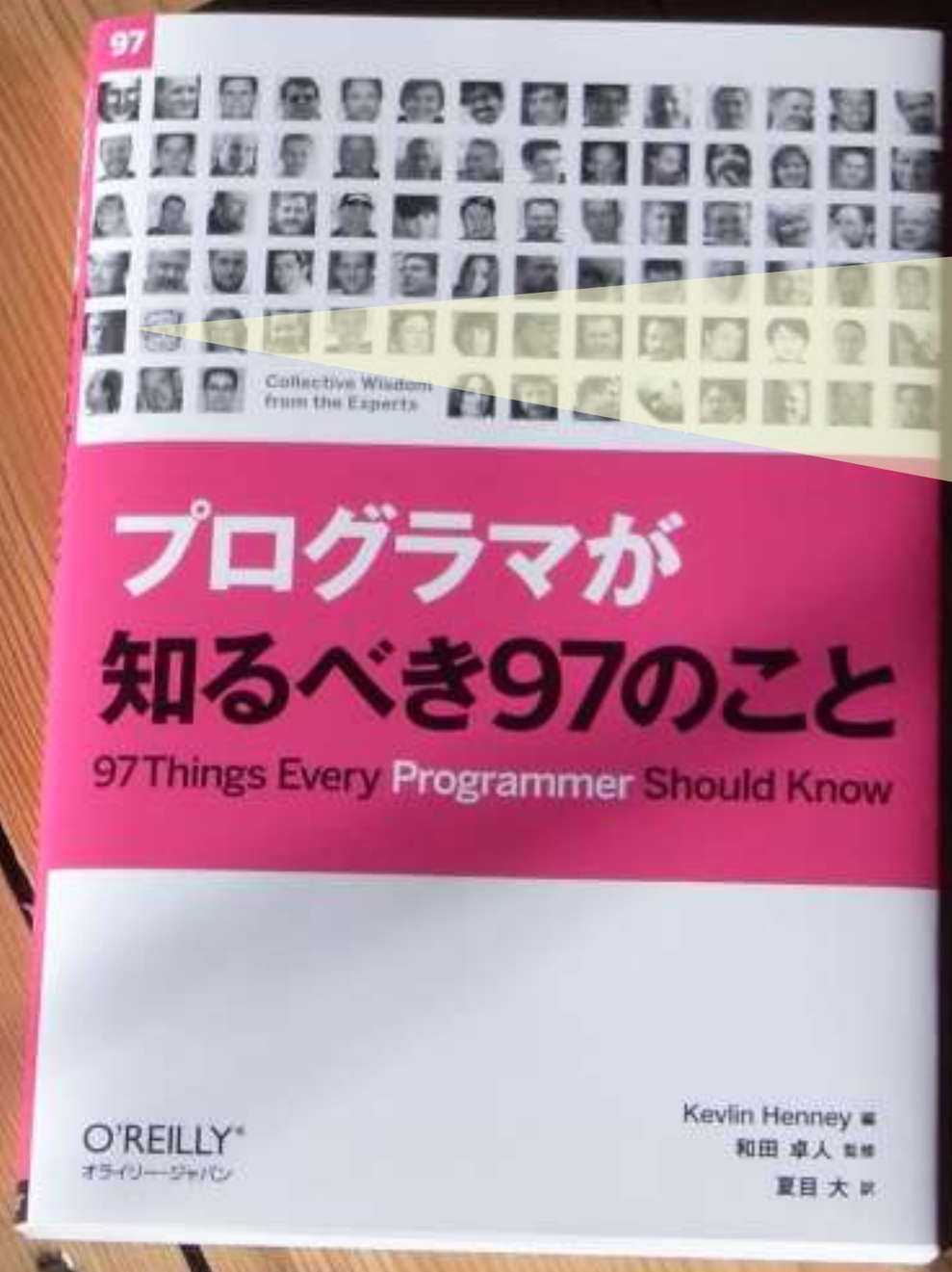
Mon,

We refer to a sound line of reasoning, for example, as coherent. The thoughts fit, they go together, they relate to each other. This is exactly the characteristic of a class that makes it coherent: the pieces all seem to be related, they seem to belong together, and it would feel somewhat unnatural to pull them apart. Such a class exhibits *cohesion*.

We refer to a sound line of reasoning, for example, as coherent. The thoughts fit, they go together, they relate to each other. This is exactly the characteristic of an interface that makes it coherent: the pieces all seem to be related, they seem to belong together, and it would feel somewhat unnatural to pull them apart. Such an interface exhibits *cohesion*.

Every class should embody only about 3–5 distinct responsibilities.

Grady Booch, *Object Solutions*



One of the most foundational principles of good design is:

> **Gather together those things that change for the same reason, and separate those things that change for different reasons.**

This principle is often known as the *single responsibility principle*, or SRP. In short, it says that a subsystem, module, class, or even a function, should not have more than one reason to change.

Single Responsibility

Open-Closed

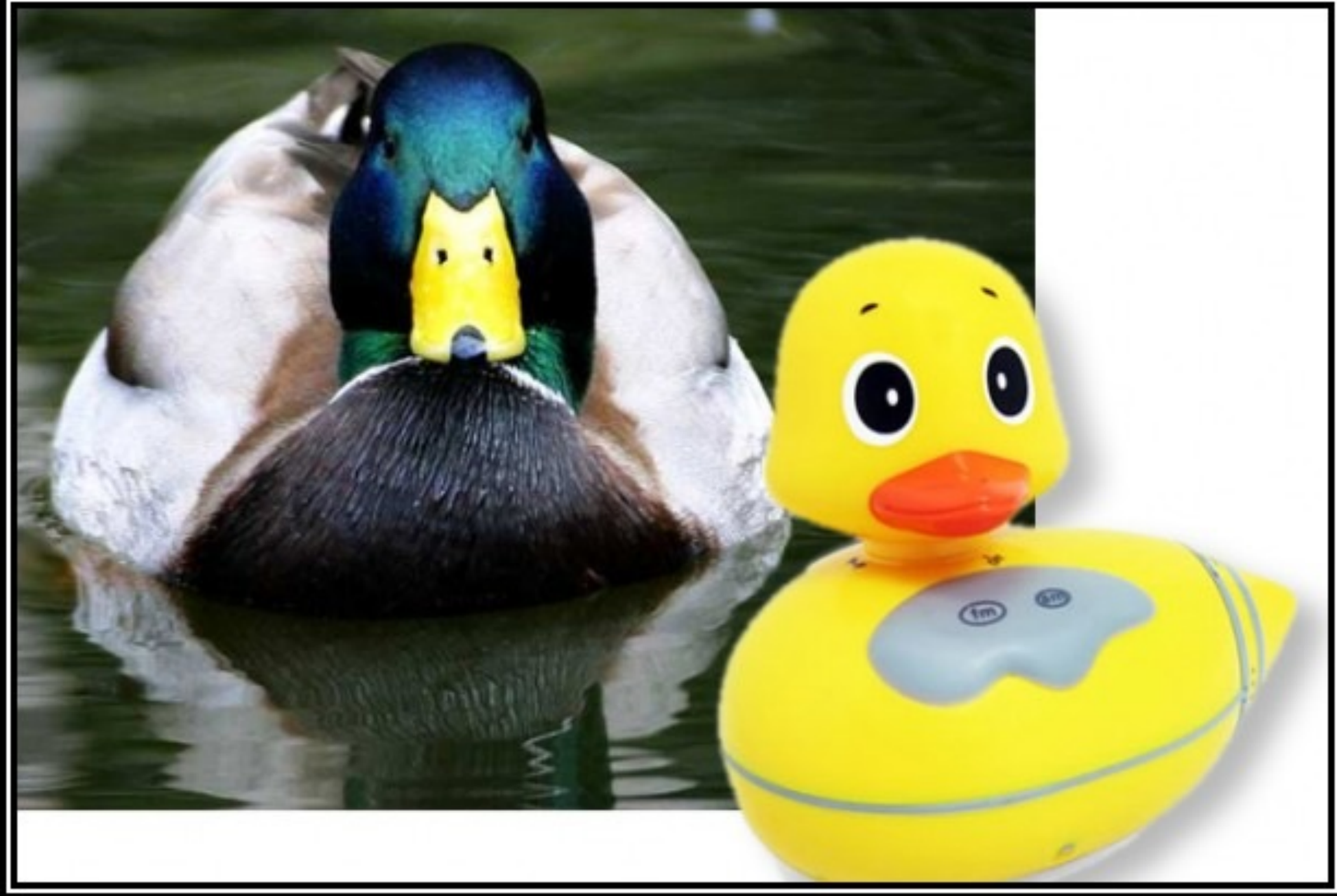
Liskov Substitution

Interface Segregation

Dependency Inversion

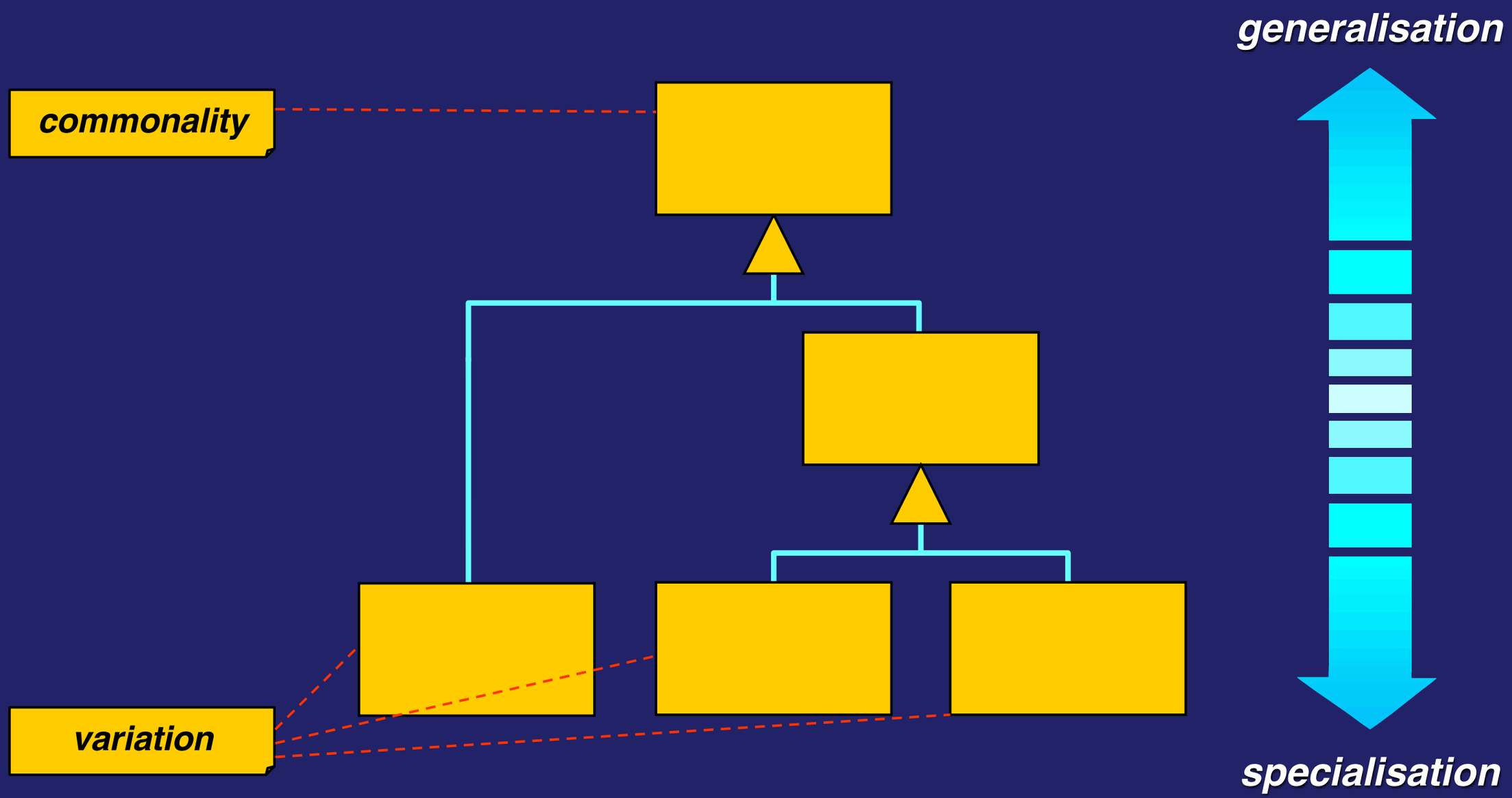
A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction


generalisation
commonality
variation
specialisation

Guidelines for Overloadin...

http://msdn.microsoft.com/en-us/library/ms173147(VS.80).aspx

Any derived class that can call **Equals** on the base class should do so before finishing its comparison. In the following example, **Equals** calls the base class **Equals**, which checks for a null parameter and compares the type of the parameter with the type of the derived class. That leaves the implementation of **Equals** on the derived class the task of checking the new data field declared on the derived class:

Don't

VB | C# | C++ | F# | JScript

Copy

```
class ThreeDPoint : TwoDPoint
{
    public readonly int z;

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        this.z = z;
    }

    public override bool Equals(System.Object obj)
    {
        // If parameter cannot be cast to ThreeDPoint return false:
        ThreeDPoint p = obj as ThreeDPoint;
        if ((object)p == null)
        {
            return false;
        }

        // Return true if the fields match:
        return base.Equals(obj) && z == p.z;
    }

    public bool Equals(ThreeDPoint p)
    {
        // Return true if the fields match:
        return base.Equals((TwoDPoint)p) && z == p.z;
    }

    public override int GetHashCode()
    {
        return base.GetHashCode() ^ z;
    }
}
```

```java
public interface SimpleList {
    int size();

    String get(int index);

    boolean add(String newItem);
}
```

```
public abstract class SimpleListTest {
    private SimpleList list;

    @Test public void additionOfNonNullIsAppended() {
        list.add("hello");
        list.add("there");
        assertThat(list.get(1), equalTo("there"));
    }

    @Test public void additionOfNullIsPermitted() {
        list.add(null);
    }

    @Test public void additionOfDuplicateIsOkay() {
        list.add("hello");
        list.add("hello");
        assertThat(list.size(), equalTo(2));
    }
}
```

```
public class SimpleListImpl extends ArrayList<String>
   implements SimpleList {
}
```

```
public abstract class SimpleListTest {
    private SimpleList list;

    @Before
    public void createList() {
        list = new SimpleListImpl();
    }

    @Test public void additionOfNonNullIsAppended() {
        ...
    }

    @Test public void additionOfNullIsPermitted() {
        ...
    }

    @Test public void additionOfDuplicateIsOkay() {
        ...
    }
}
```

```java
public class LeastRecentlyUsedList extends SimpleListImpl {

    @Override
    public boolean add(String newItem) {
        if (newItem == null) throw new IllegalArgumentException();
        this.remove(newItem);
        this.add(0, newItem);
        return true;
    }
}
```

```
public abstract class SimpleListTest {
    private SimpleList list;

    @Before
    public void createList() {
        list = new LeastRecentlyUsedList();
    }

    @Test public void additionOfNonNullIsAppended() {
        ...
    }

    @Test public void additionOfNullIsPermitted() {
        ...
    }

    @Test public void additionOfDuplicateIsOkay() {
        ...
    }
}
```

```java
public class LeastRecentlyUsedList implements SimpleLRUList {
    private List<String> list = new ArrayList<>();

    @Override
    public int size() {
        return list.size();
    }

    @Override
    public String get(int index) {
        return list.get(index);
    }

    @Override
    public boolean add(String newItem) {
        if (newItem == null) throw new IllegalArgumentException();
        list.remove(newItem);
        list.add(0, newItem);
        return true;
    }
}
```

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. **What is wanted here is something like the following substitution property**: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is **unchanged** when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

The principle stated that a good module structure should be both open and closed:

- Closed, because clients need the module's services to proceed with their own development, and once they have settled on a version of the module should not be affected by the introduction of new services they do not need.
- Open, because there is no guarantee that we will include right from the start every service potentially useful to some client.

Bertrand Meyer
Object-Oriented Software Construction

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is **unchanged** when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"

A myth in the object-oriented design community goes something like this:

> If you use object-oriented technology, you can take any class someone else wrote, and, by using it as a base class, refine it to do a similar task.

Robert B Murray
C++ Strategies and Tactics

Published Interface is a term I used (first in *Refactoring*) to refer to a class interface that's used outside the code base that it's defined in.

The distinction between published and public is actually more important than that between public and private.

The reason is that with a non-published interface you can change it and update the calling code since it is all within a single code base. [...] But anything published so you can't reach the calling code needs more complicated treatment.

Martin Fowler

http://martinfowler.com/bliki/PublishedInterface.html

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

In object-oriented programming, the dependency inversion principle refers to a specific form of decoupling where conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are inverted (i.e. reversed) for the purpose of rendering high-level modules independent of the low-level module implementation details.

http://en.wikipedia.org/wiki/Dependency_inversion_principle

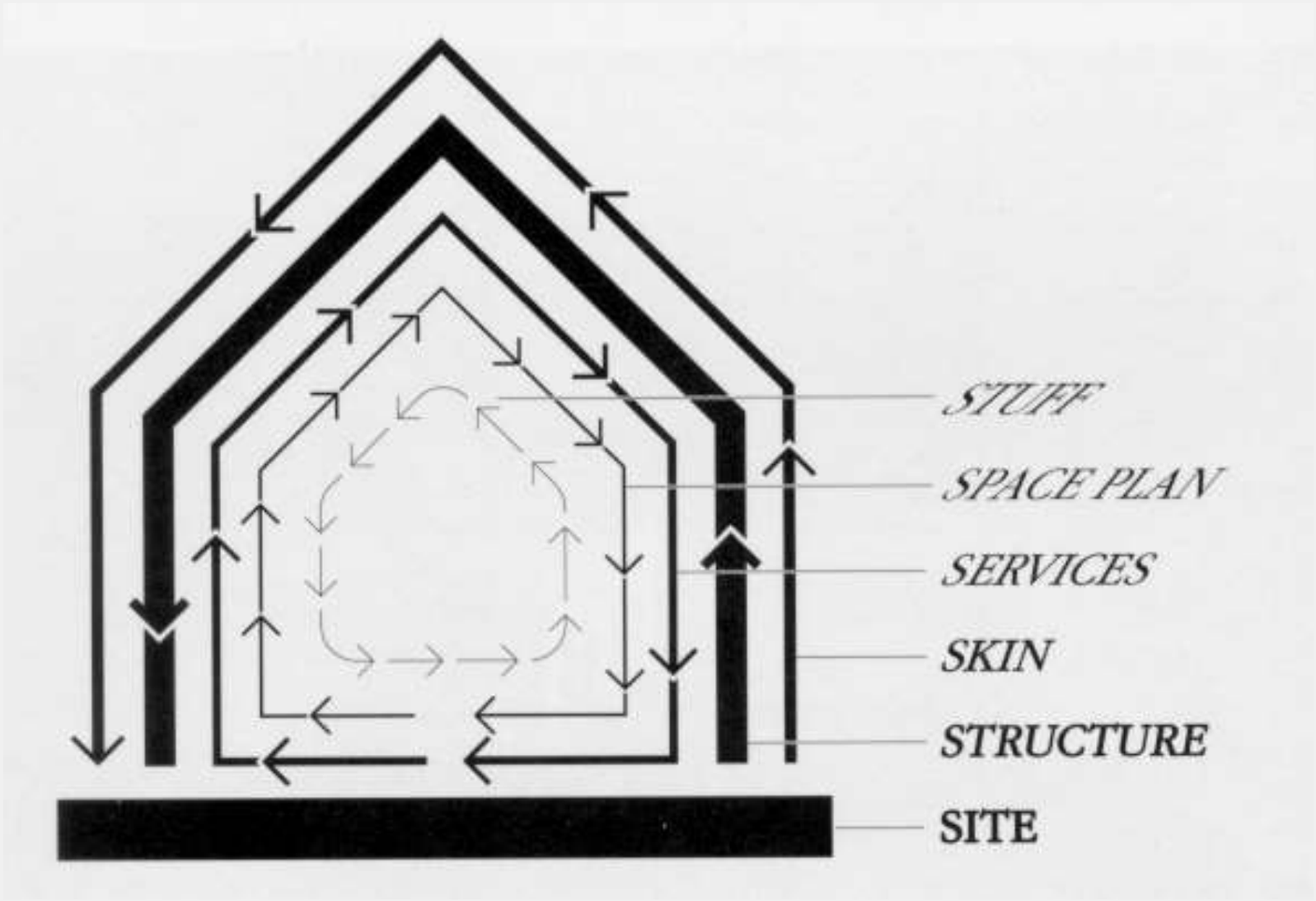
HOW BUILDINGS LEARN

What happens after they're built

New Orleans, 1852

The same two buildings, 1993

STEWART BRAND

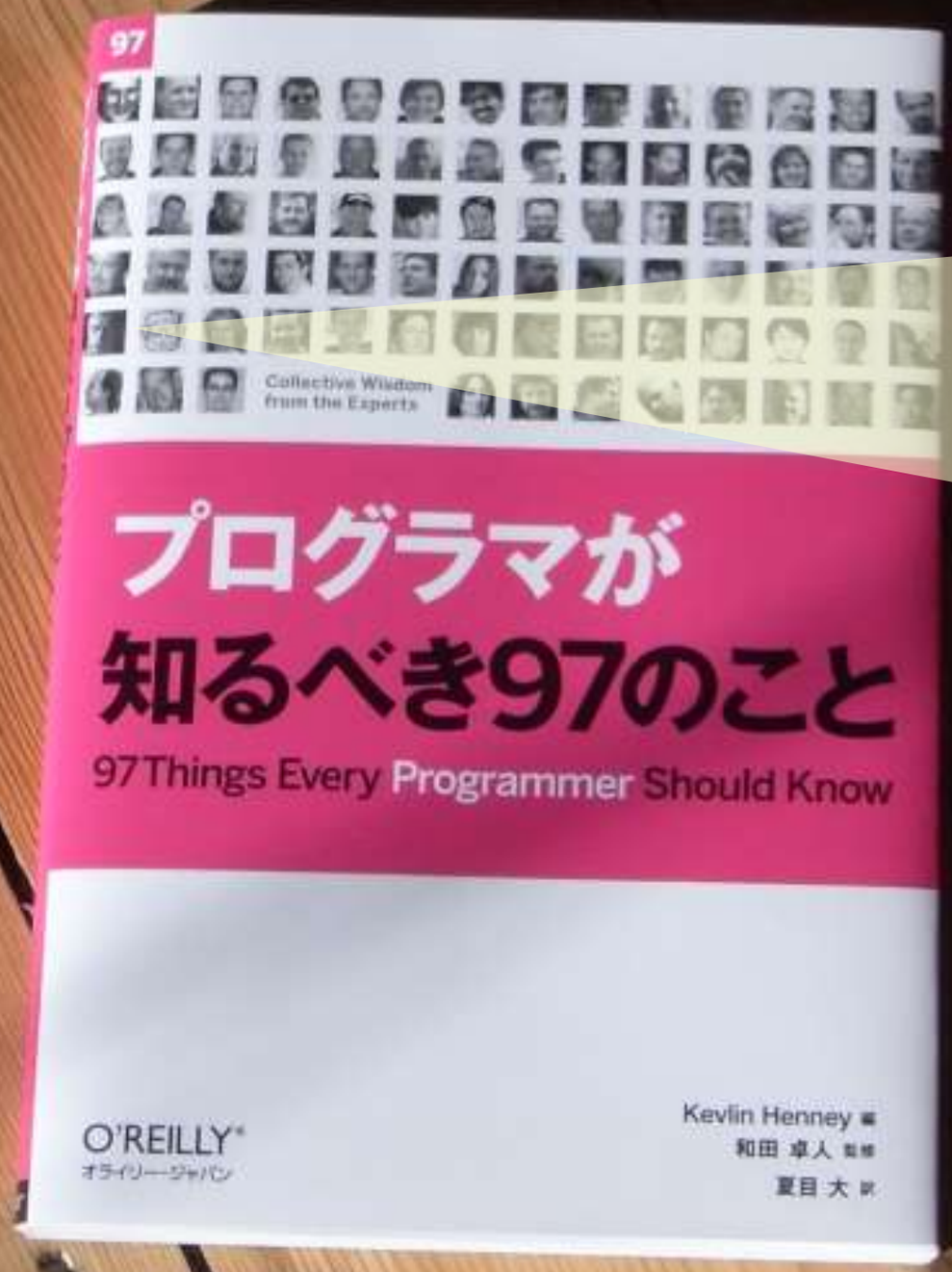


Stewart Brand, *How Buildings Learn*
See also *http://www.laputan.org/mud/*

```
public SomeClass {
...
    try {
        InitialContext ic = new InitialContext();
        String snName = "java:comp/env/mail/MyMailSession";
        Session session = (javax.mail.Session) ic.lookup(snName);
    } catch (NameNotFoundException e) {
        out("\nJNDI binding was not found");
    } catch (NamingException e) {
        out("\nJNDI binding error");
    }
...
}
```

```
public SomeClass {
    public SomeClass(Session session) {
        ...
    }
...
}
```

```
package com.sun...;
```



One of the most foundational principles of good design is:

> **Gather together those things that change for the same reason, and separate those things that change for different reasons.**

This principle is often known as the *single responsibility principle*, or SRP. In short, it says that a subsystem, module, class, or even a function, should not have more than one reason to change.

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

Cohesion by Usage

Cohesion by Change

Conformance

Consistency over Space

Consistency over Time

unctional

oose

nit Testable

ntrospective

dempotent

“OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP. There are possibly other systems in which this is possible, but I’m not aware of them.

•Allan Kay

Powershell

```
$inputsWithExpectations | ?{
   [String] $actual = GetNextFriday13th($_[0])
   [String] $expected = $_[1]
   $actual -ne $expected
}
```

scheme

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
  ((variable? exp) (lookup-variable-value exp env))
  ((quoted? exp) (text-of-quotation exp))
  ((assignment? exp) (eval-assignment exp env))
  ((definition? exp) (eval-definition exp env))
  ((if? exp) (eval-if exp env))
  ((lambda? exp))
    (make-procedure (lambda-parameters exp)
                    (lambda-body exp)
                    env))
((begin? exp)
  (eval-sequence (begin-actions exp) env))
((cond? exp) (eval (cond->if exp) env))
((application? exp)
  (apply (eval (operator exp) env)
         (list-of-values (operands exp) env)))
(else
  (error "Unknown expression type -EVAL" exp))))
```

“Asking a question should not change the answer, and nor should asking it twice!

•Betrand Meyer

Retweeted by @kevlinhenney