SOLID Deconstruction

Kevlin Henney
kevlin@curbralan.com
@KevlinHenney

TURING 图灵程序设计丛书
WILEY
Pattern-Oriented Software Architecture Volume 4
A Pattern Language for Distributed Computing
面向模式的软件架构
分布式计算的模式语言
卷4
[德] Frank Buschmann
[英] Kevlin Henney 著
[美] Douglas C.Schmidt
肖鹏 陈立 译
人民邮电出版社
POSTS & TELECOM PRESS
TURING 图灵程序设计丛书
WILEY
Pattern-Oriented Software Architecture Volume 5
On Patterns and Pattern Languages
面向模式的软件架构
模式与模式语言
卷5
[德] Frank Buschmann
[英] Kevlin Henney 著
[美] Douglas C. Schmidt
肖鹏 等译
人民邮电出版社
POSTS & TELECOM PRESS
97
Collective Wisdom
from the Experts
程序员
应该知道的
97件事
O'REILLY®
Kevlin Henney 编
李军 译 吕骏 审校
電子工業出版社

S
O
L
I
D

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

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

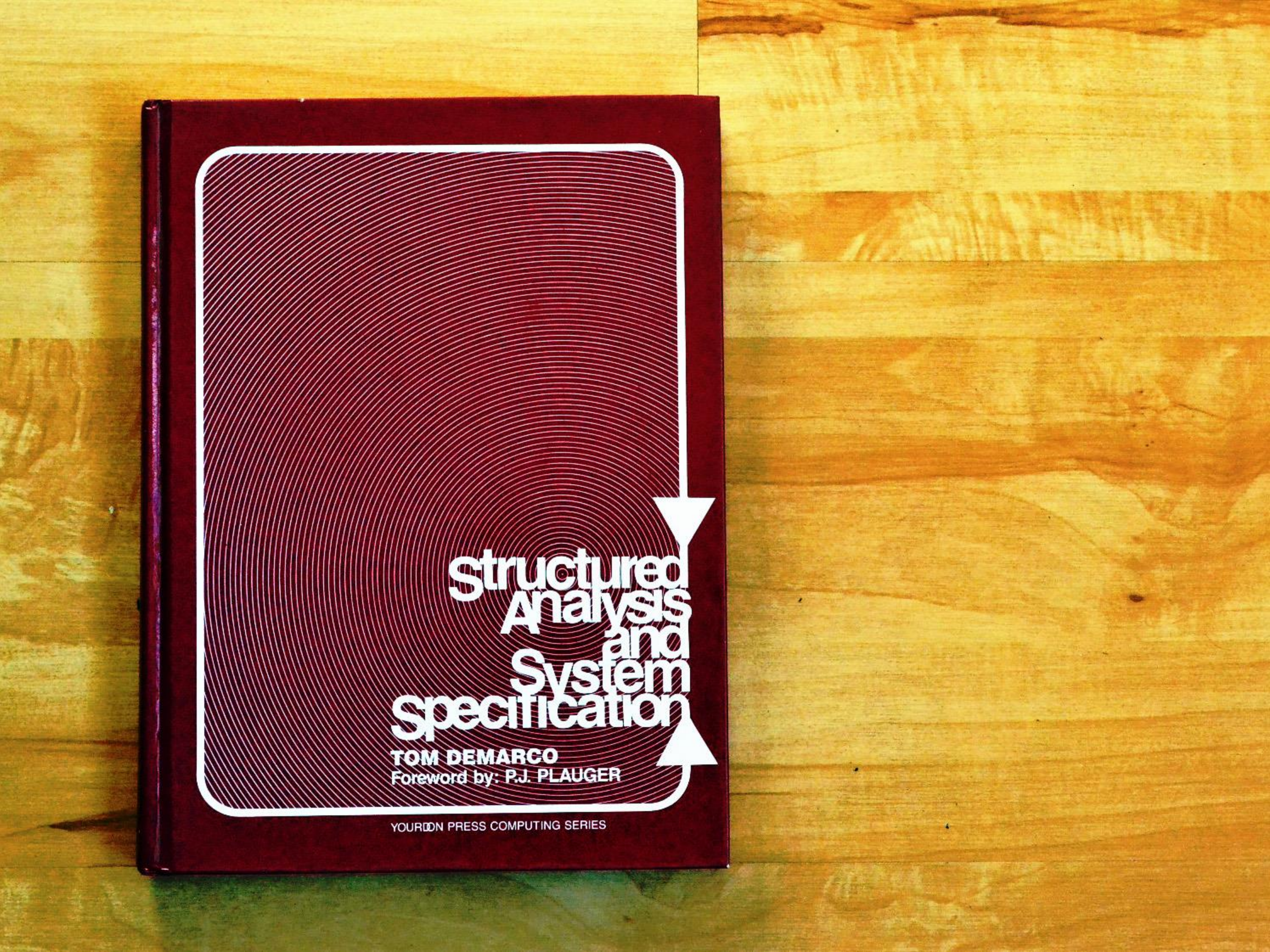
Dependency Inversion

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. ("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.

Glenn Vanderburg: Blog

1 of 1 article • info • syndicate

Cohesion

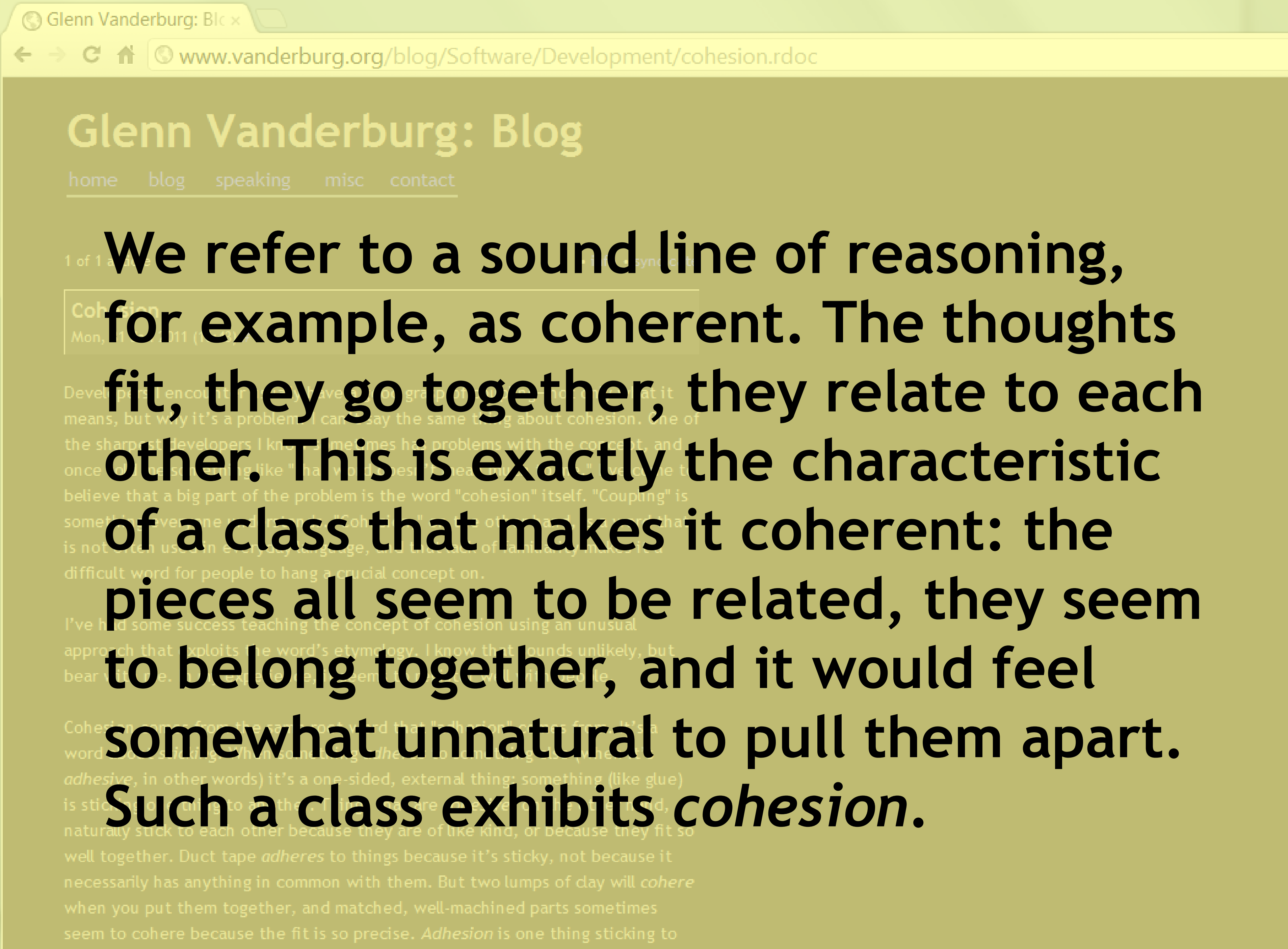
Mon, 31 Jan 2011 (16:43) #

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

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*.



believe that a big part of the problem is the word "cohesion" itself. "Coupling" is

difficult word for people to hang a crucial concept on.

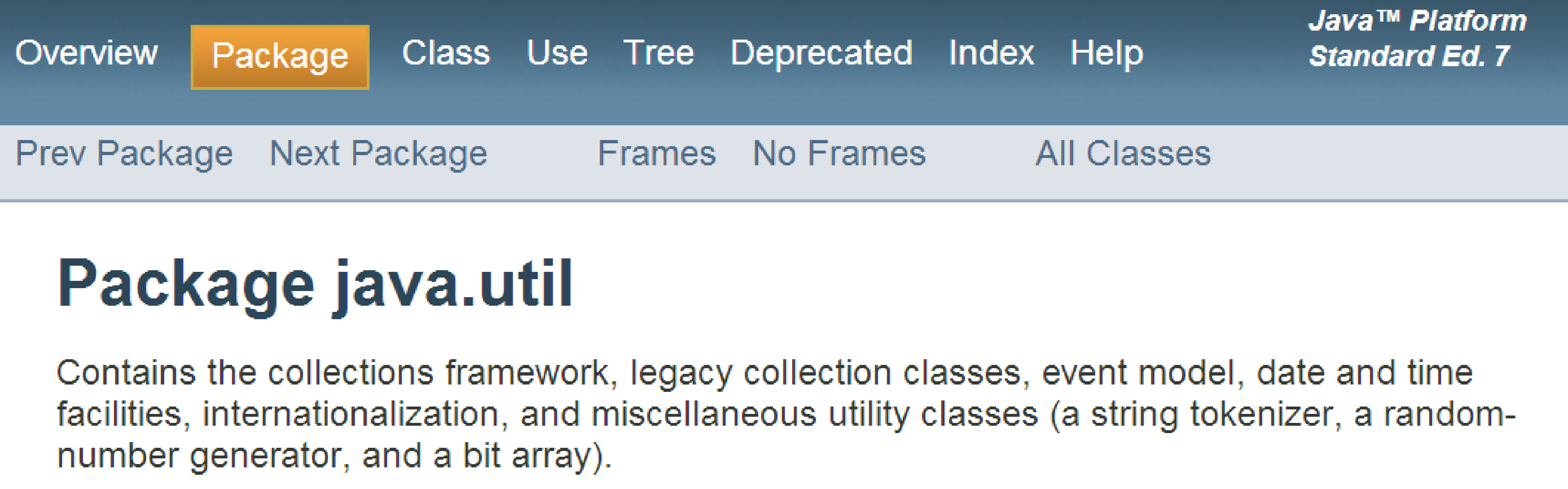
I've had some success teaching the concept of cohesion using an unusual

adhesive, in other words) it's a one-sided, external thing: something (like glue)

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

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

Doug McIlroy



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

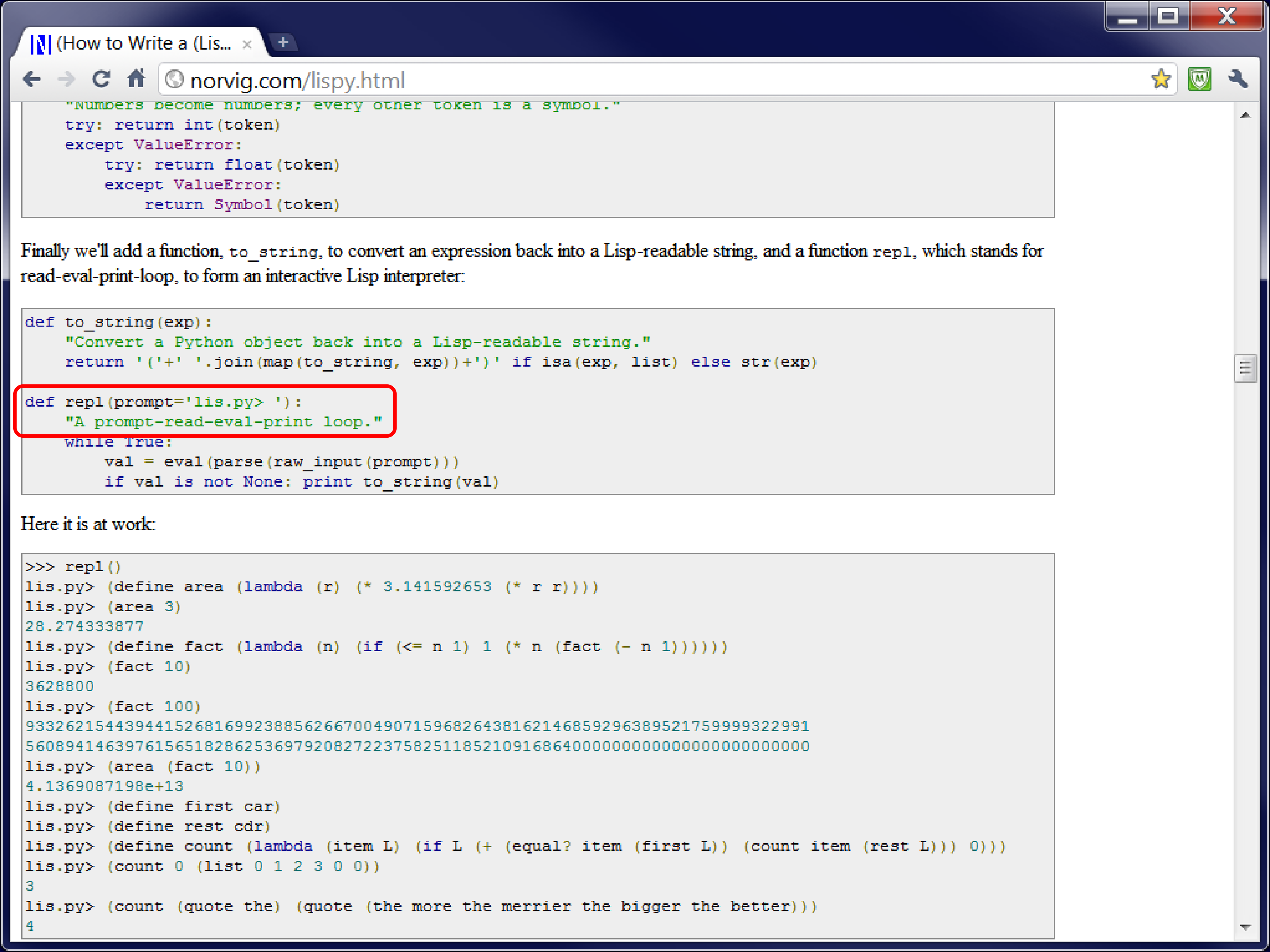
- the state of being useful, profitable or beneficial
- useful, especially through having several functions
- functional rather than attractive

Concise Oxford English Dictionary

```
#include <stdlib.h>
```

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

Grady Booch, *Object Solutions*



```
    "Numbers become numbers; every other token is a symbol."
    try: return int(token)
    except ValueError:
        try: return float(token)
        except ValueError:
            return Symbol(token)
```

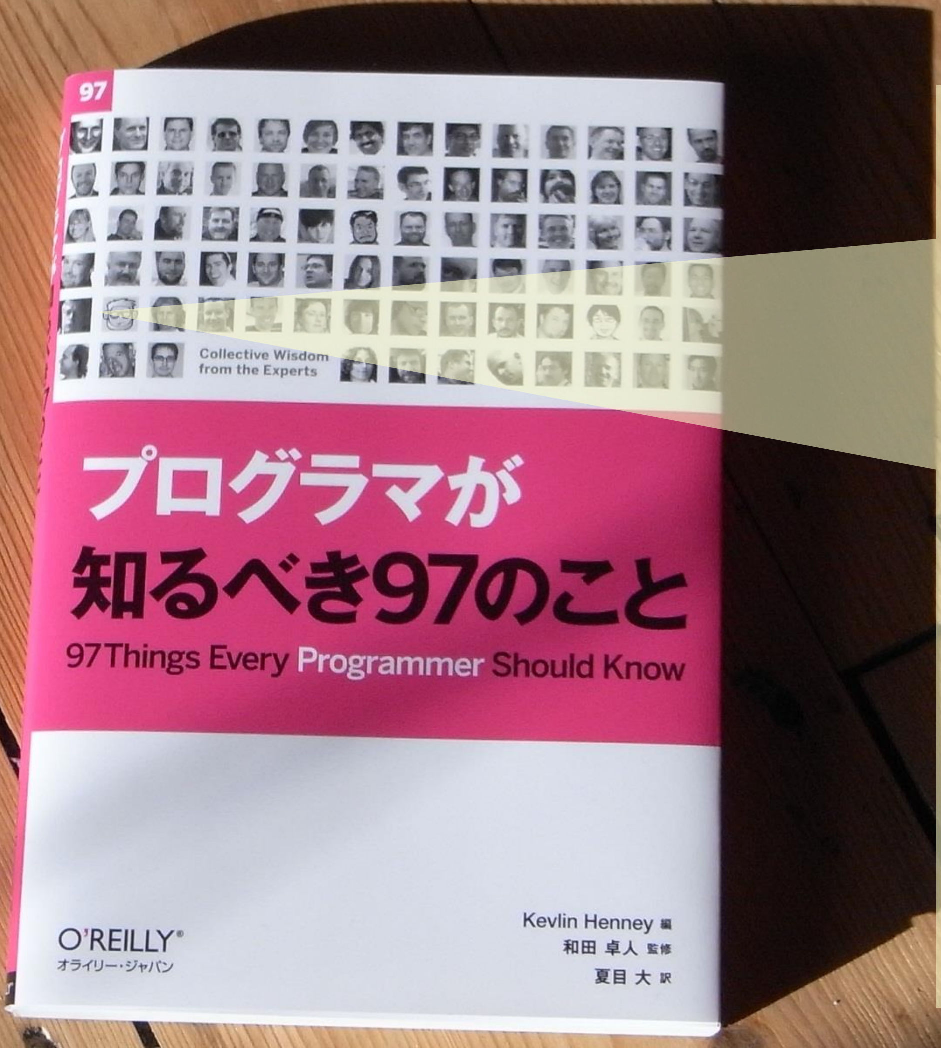
Finally we'll add a function, `to_string`, to convert an expression back into a Lisp-readable string, and a function `repl`, which stands for read-eval-print-loop, to form an interactive Lisp interpreter:

```
def to_string(exp):
    "Convert a Python object back into a Lisp-readable string."
    return '('+' '.join(map(to_string, exp))+')' if isa(exp, list) else str(exp)

def repl(prompt='lis.py> '):
    "A prompt-read-eval-print loop."
    while True:
        val = eval(parse(raw_input(prompt)))
        if val is not None: print to_string(val)
```

Here it is at work:

```
>>> repl()
lis.py> (define area (lambda (r) (* 3.141592653 (* r r))))
lis.py> (area 3)
28.274333877
lis.py> (define fact (lambda (n) (if (<= n 1) 1 (* n (fact (- n 1))))))
lis.py> (fact 10)
3628800
lis.py> (fact 100)
9332621544394415268169923885626670049071596826438162146859296389521759999322991
5608941463976156518286253697920827223758251185210916864000000000000000000000000
lis.py> (area (fact 10))
4.1369087198e+13
lis.py> (define first car)
lis.py> (define rest cdr)
lis.py> (define count (lambda (item L) (if L (+ (equal? item (first L)) (count item (rest L))) 0)))
lis.py> (count 0 (list 0 1 2 3 0 0))
3
lis.py> (count (quote the) (quote (the more the merrier the bigger the better)))
4
```


97
Collective Wisdom from the Experts
プログラマが知るべき97のこと
97 Things Every Programmer Should Know
O'REILLY
オライリー・ジャパン
Kevlin Henney 編
和田 卓人 監修
夏目 大 訳

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

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: Blog

home blog speaking misc contact

1 of 1 article • info • syndicate

Cohesion

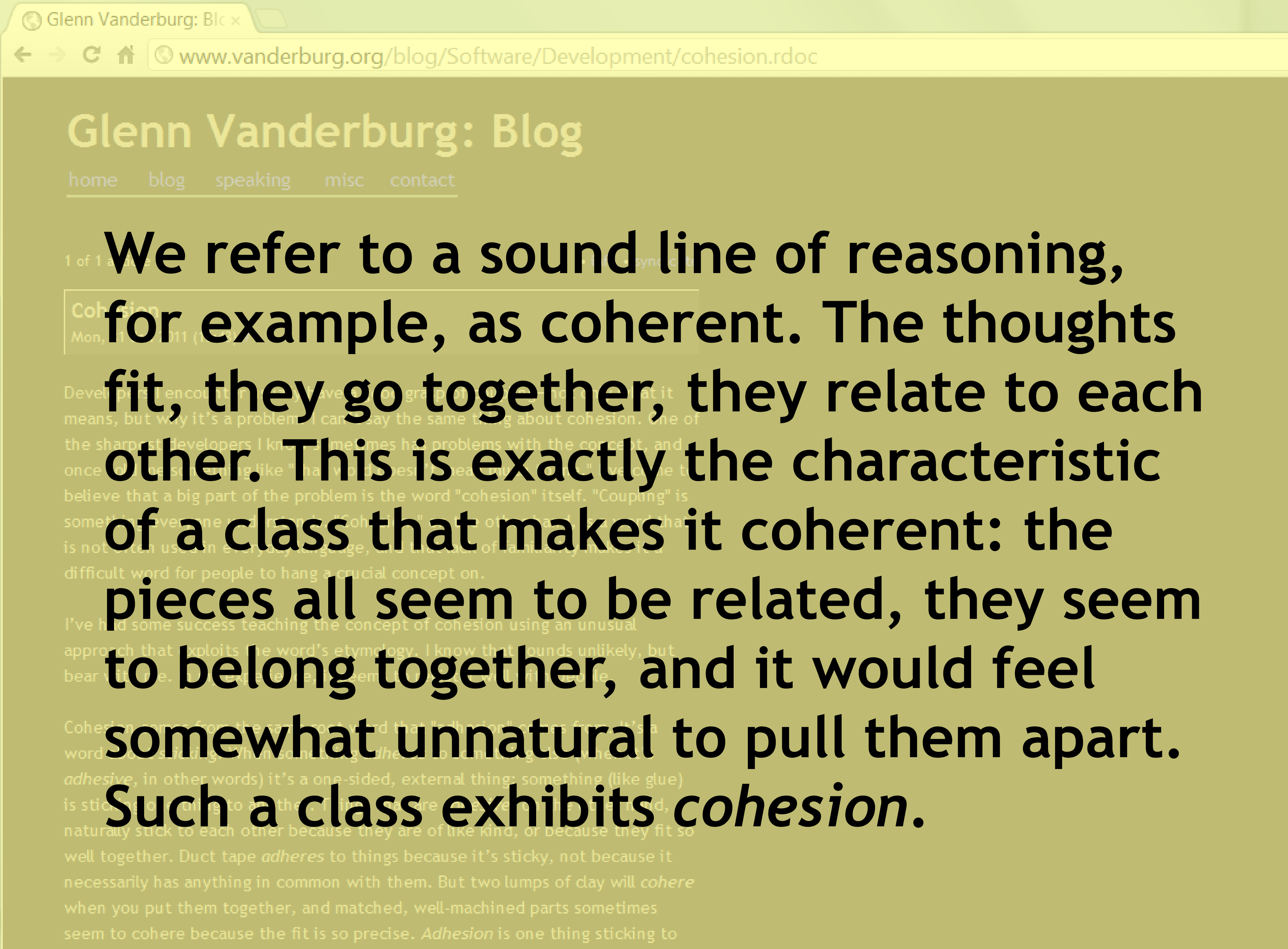
Mon, 31 Jan 2011 (16:43) #

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

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*.



Glenn Vanderburg: Blc ×

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

Glenn Vanderburg: Blog

home blog speaking misc contact

1 of 1 article [illegible] • info • syndicate

Cohesion

Mon, [illegible] 2011 ([illegible]) #

Developers I encounter [illegible] 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 resonate 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

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*.

```java
public interface LineIO
{
    String read();
    void write(String toWrite);
}
```

```
public interface LineReader
{
    String read();
}

public interface LineWriter
{
    void write(String toWrite);
}
```

Single Responsibility

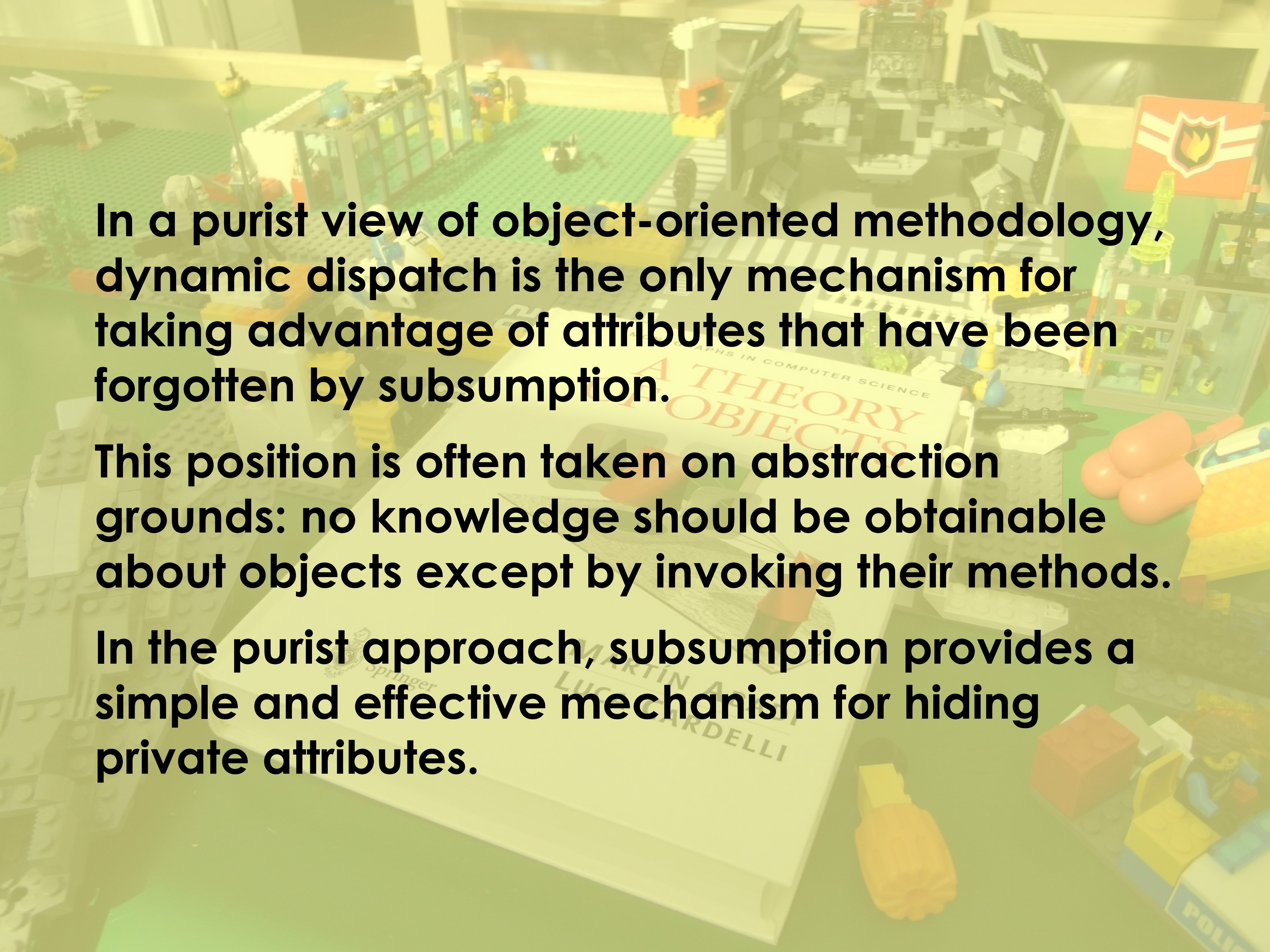
Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

MONOGRAPHS IN COMPUTER SCIENCE
A THEORY OF OBJECTS
Martín Abadi
Luca Cardelli
Springer



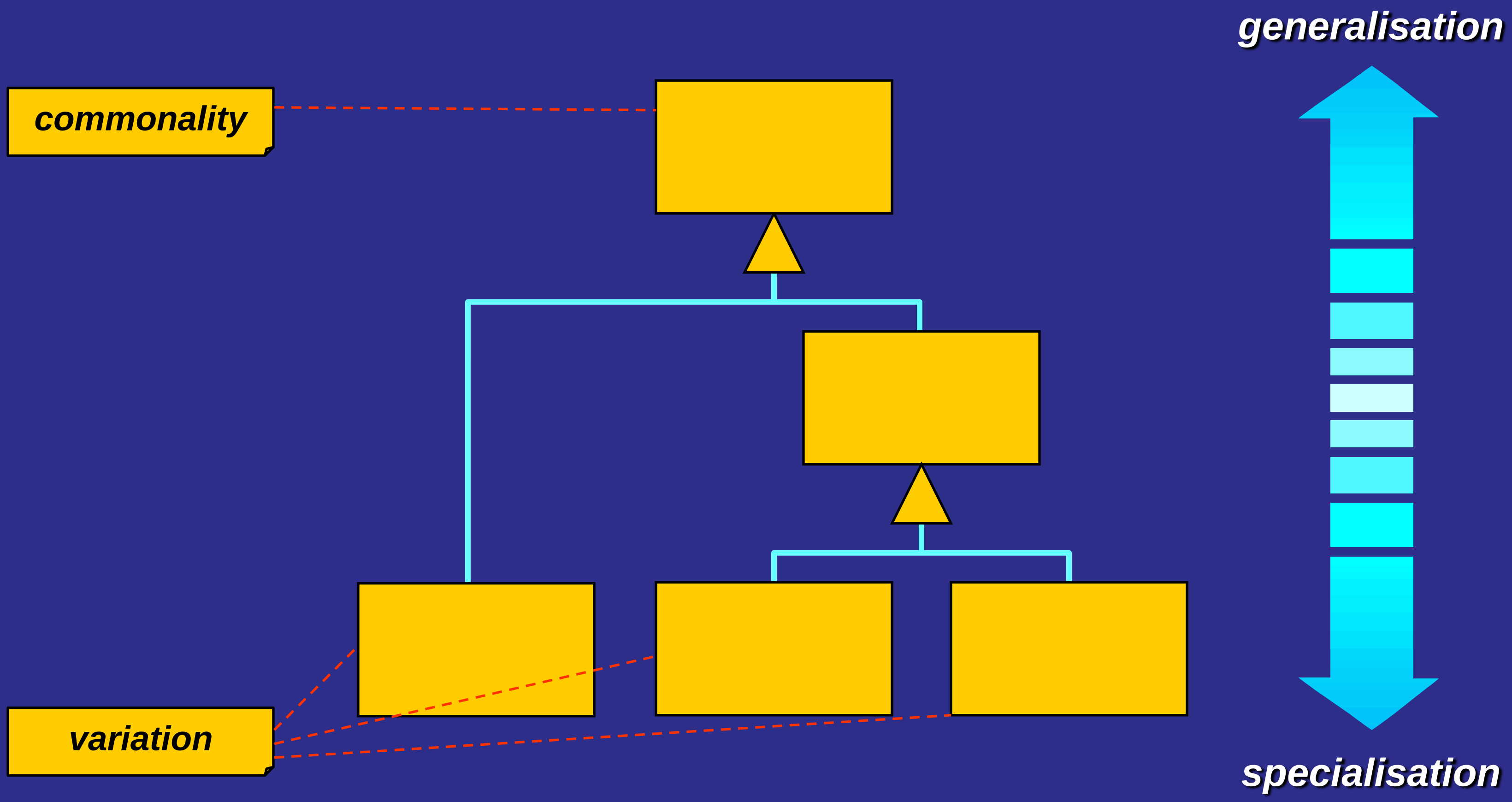
In a purist view of object-oriented methodology, dynamic dispatch is the only mechanism for taking advantage of attributes that have been forgotten by subsumption.

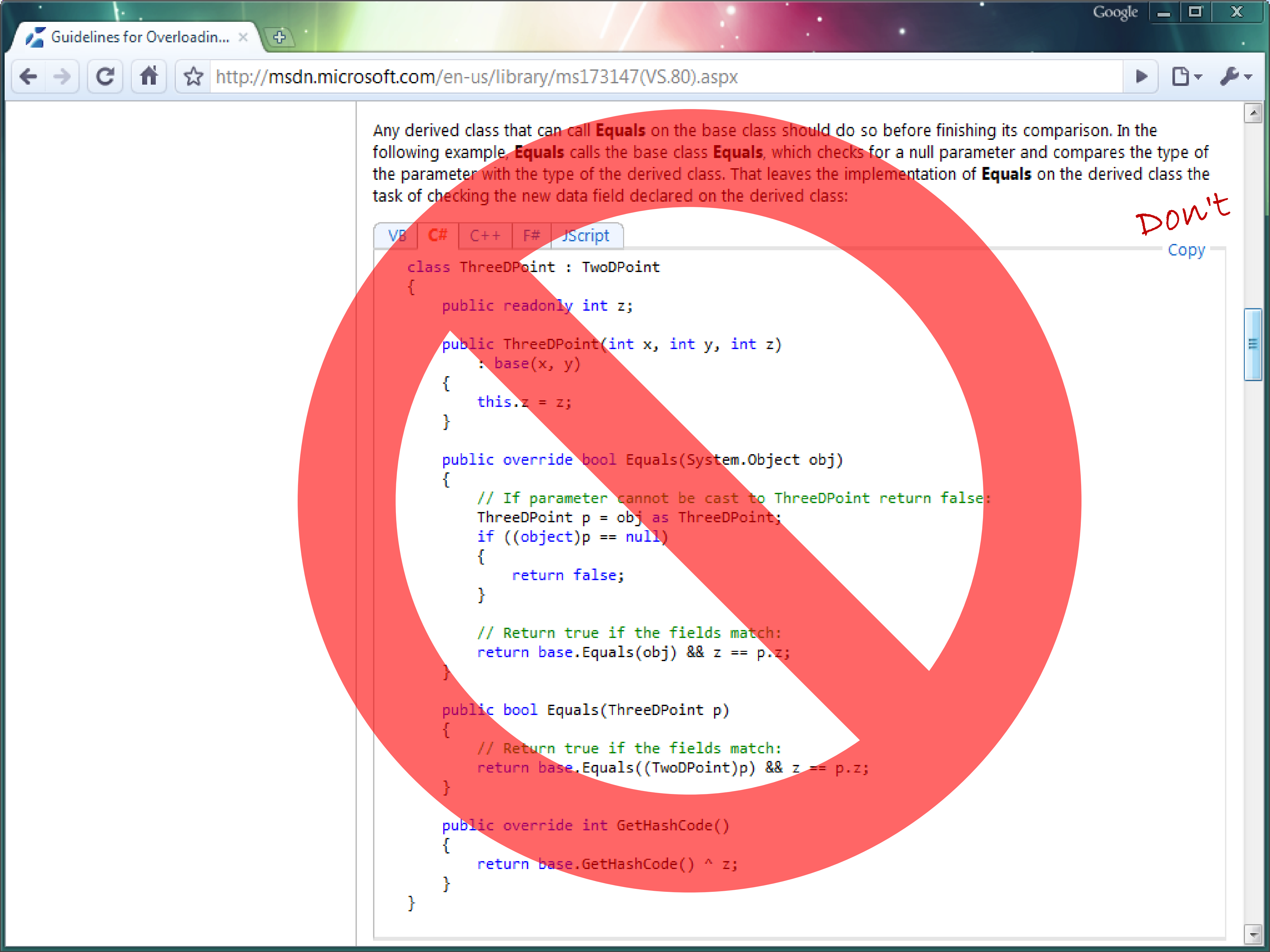
This position is often taken on abstraction grounds: no knowledge should be obtainable about objects except by invoking their methods.

In the purist approach, subsumption provides a simple and effective mechanism for hiding private attributes.

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"


generalisation
commonality
variation
specialisation


Guidelines for Overloadin...
http://msdn.microsoft.com/en-us/library/ms173147(VS.80).aspx
Google
Don't
VB
C#
C++
F#
JScript
Copy

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:

```
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;
    }
}
```

```
public class RecentlyUsedList
{
    ...
    public int Count
    {
        get ...
    }
    public string this[int index]
    {
        get ...
    }
    public void Add(string newItem) ...
    ...
}
```

```
public class RecentlyUsedList
{
    private IList<string> items = new List<string>();
    public int Count
    {
        get
        {
            return items.Count;
        }
    }
    public string this[int index]
    {
        get
        {
            return items[index];
        }
    }
    public void Add(string newItem)
    {
        if(newItem == null)
            throw new ArgumentNullException();
        items.Remove(newItem);
        items.Insert(0, newItem);
    }
    ...
}
```

```
public class RecentlyUsedList : List<string>
{
    public override void Add(string newItem)
    {
        if(newItem == null)
            throw new ArgumentNullException();
        items.Remove(newItem);
        items.Insert(0, newItem);
    }
    ...
}
```

```
namespace List_spec
{
    ...
    [TestFixture]
    public class Addition
    {
        private List<string> list;
        [Setup]
        public void List_is_initially_empty()
        {
            list = ...
        }
        ...
        [Test]
        public void Addition_of_non_null_item_is_appended() ...
        [Test]
        public void Addition_of_null_is_permitted() ...
        [Test]
        public void Addition_of_duplicate_item_is_appended() ...
        ...
    }
    ...
}
```

```
namespace List_spec
{
    ...
    [TestFixture]
    public class Addition
    {
        private List<string> list;
        [Setup]
        public void List_is_initially_empty()
        {
            list = new List<string>();
        }
        ...
        [Test]
        public void Addition_of_non_null_item_is_appended() ...
        [Test]
        public void Addition_of_null_is_permitted() ...
        [Test]
        public void Addition_of_duplicate_item_is_appended() ...
        ...
    }
    ...
}
```

```
namespace List_spec
{
    ...
    [TestFixture]
    public class Addition
    {
        private List<string> list;
        [Setup]
        public void List_is_initially_empty()
        {
            list = new RecentlyUsedList();
        }
        ...
        [Test]
        public void Addition_of_non_null_item_is_appended() ...
        [Test]
        public void Addition_of_null_is_permitted() ...
        [Test]
        public void Addition_of_duplicate_item_is_appended() ...
        ...
    }
    ...
}
```

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 good module structure should be [...] 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.

Bertrand Meyer
Object-Oriented Software Construction

[...] A good module structure should be [...] 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

The principle states:

A. ***High-level modules should not depend on low-level modules. Both should depend on abstractions.***

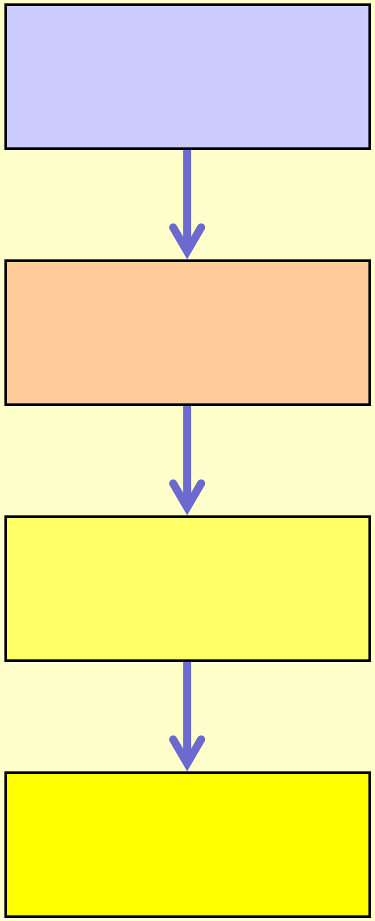
B. ***Abstractions should not depend upon details. Details should depend upon abstractions.***

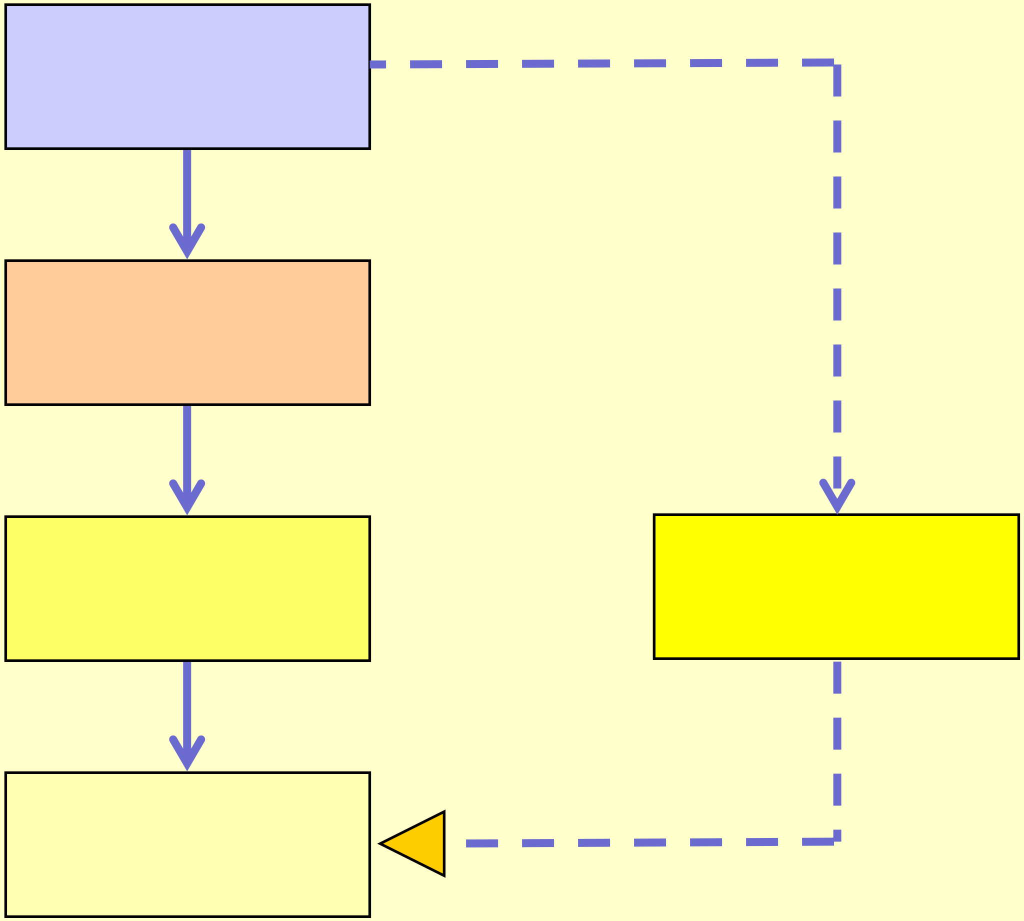
http://en.wikipedia.org/wiki/Dependency_inversion_principle

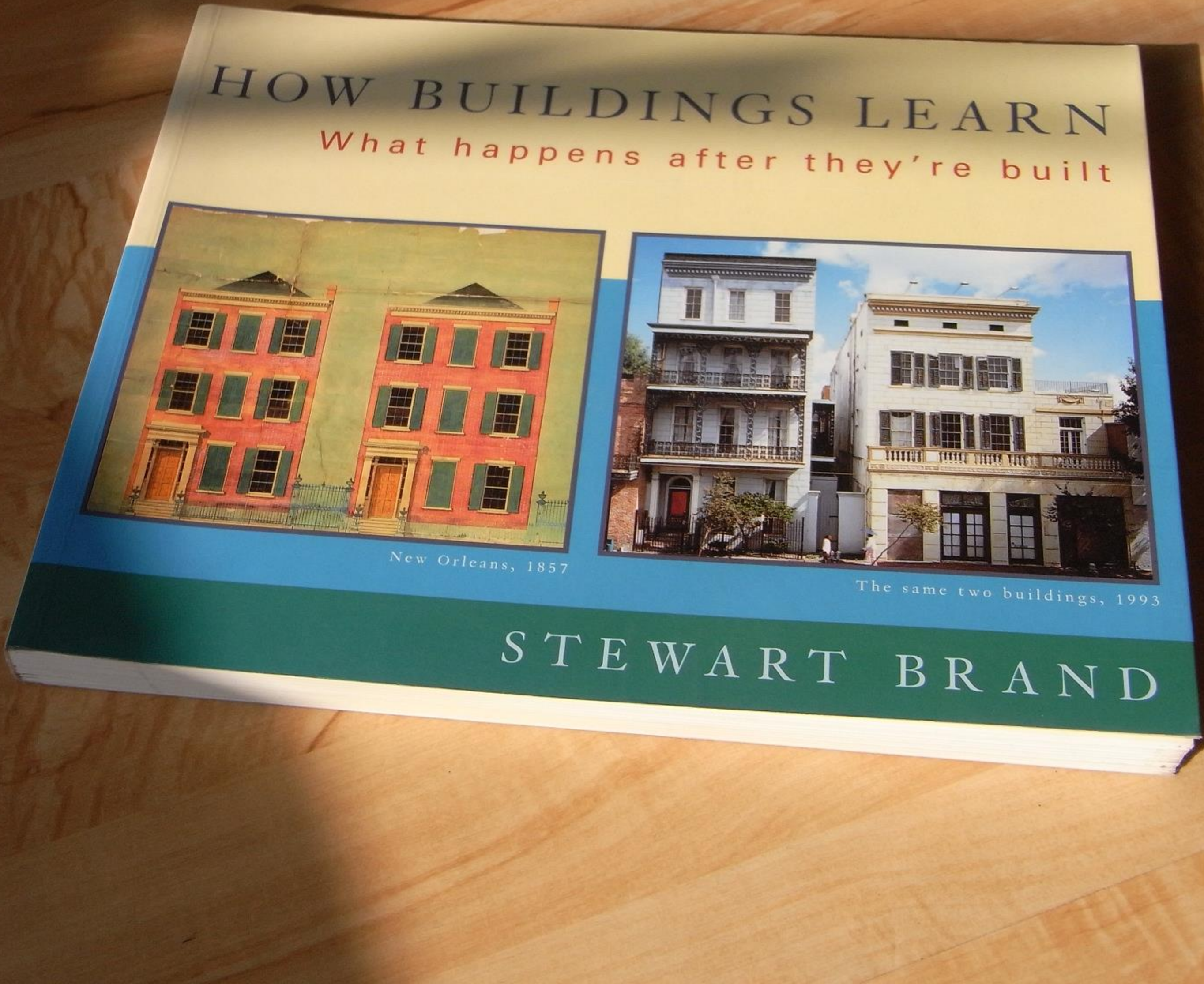
inversion, *noun*

- the action of inverting or the state of being inverted
- reversal of the normal order of words, normally for rhetorical effect
- an inverted interval, chord, or phrase
- a reversal of the normal decrease of air temperature with altitude, or of water temperature with depth

Concise Oxford English Dictionary







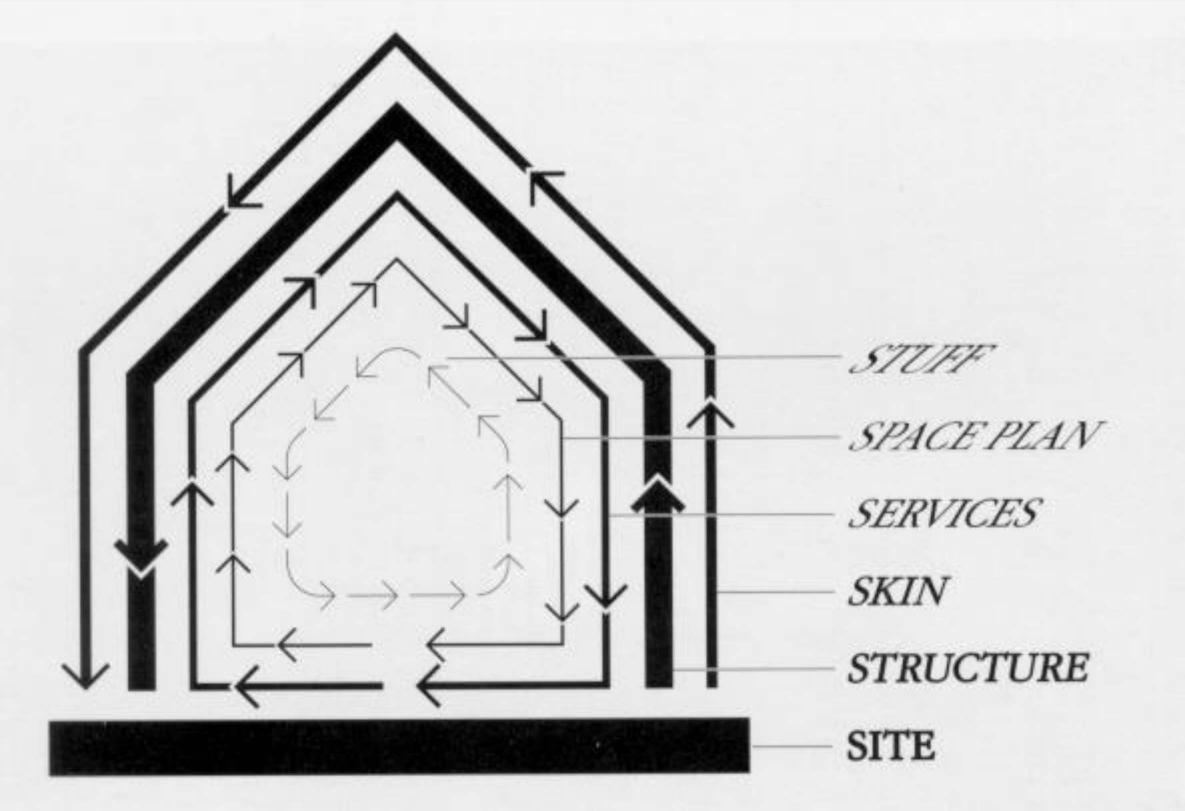
HOW BUILDINGS LEARN

What happens after they're built

New Orleans, 1857

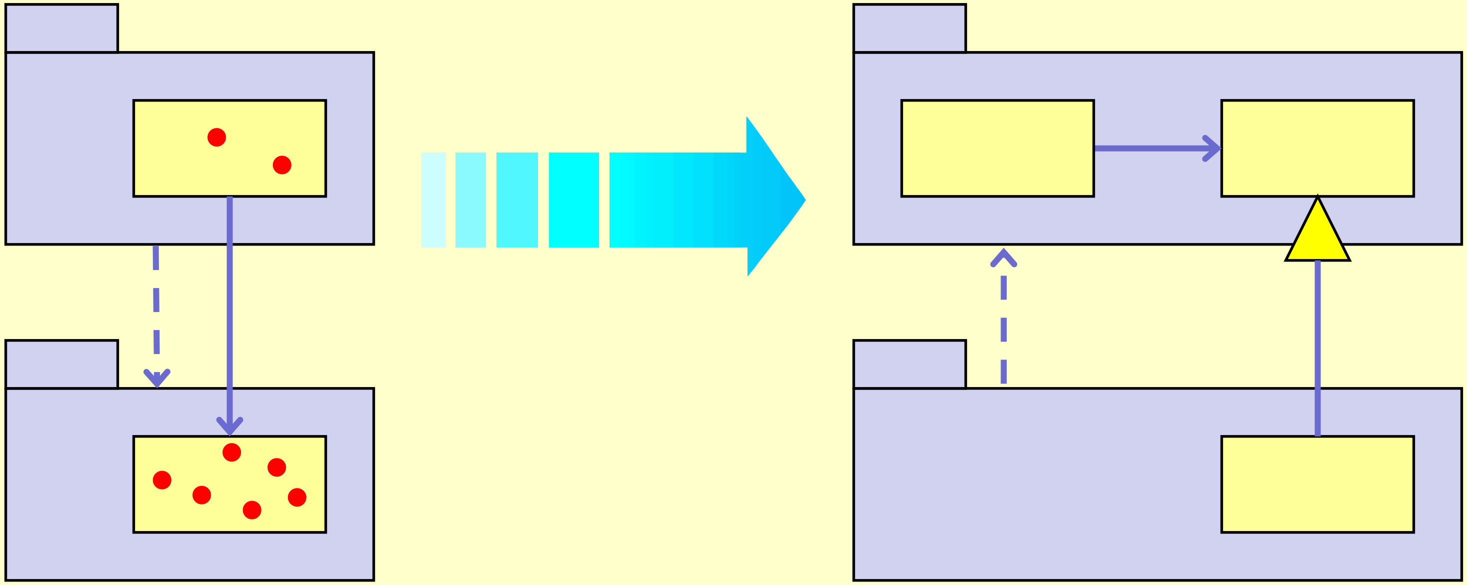
The same two buildings, 1993

STEWART BRAND

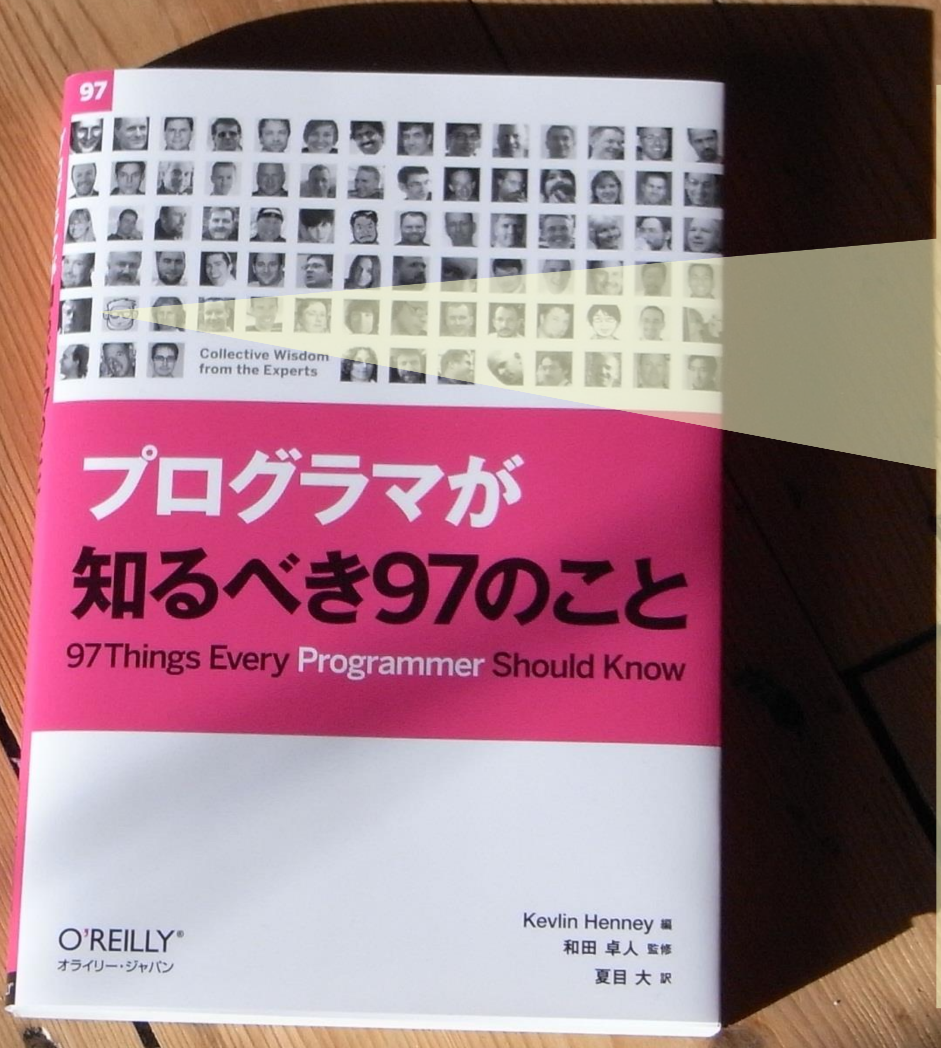


Stewart Brand, *How Buildings Learn*
See also *http://www.laputan.org/mud/*

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



Scenario buffering by dot-voting possible changes and invert dependencies as needed


97
Collective Wisdom from the Experts
プログラマが知るべき97のこと
97 Things Every Programmer Should Know
Kevlin Henney 編
和田 卓人 監修
夏目 大 訳
O'REILLY
オライリー・ジャパン

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.

S

O

L

I

D

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

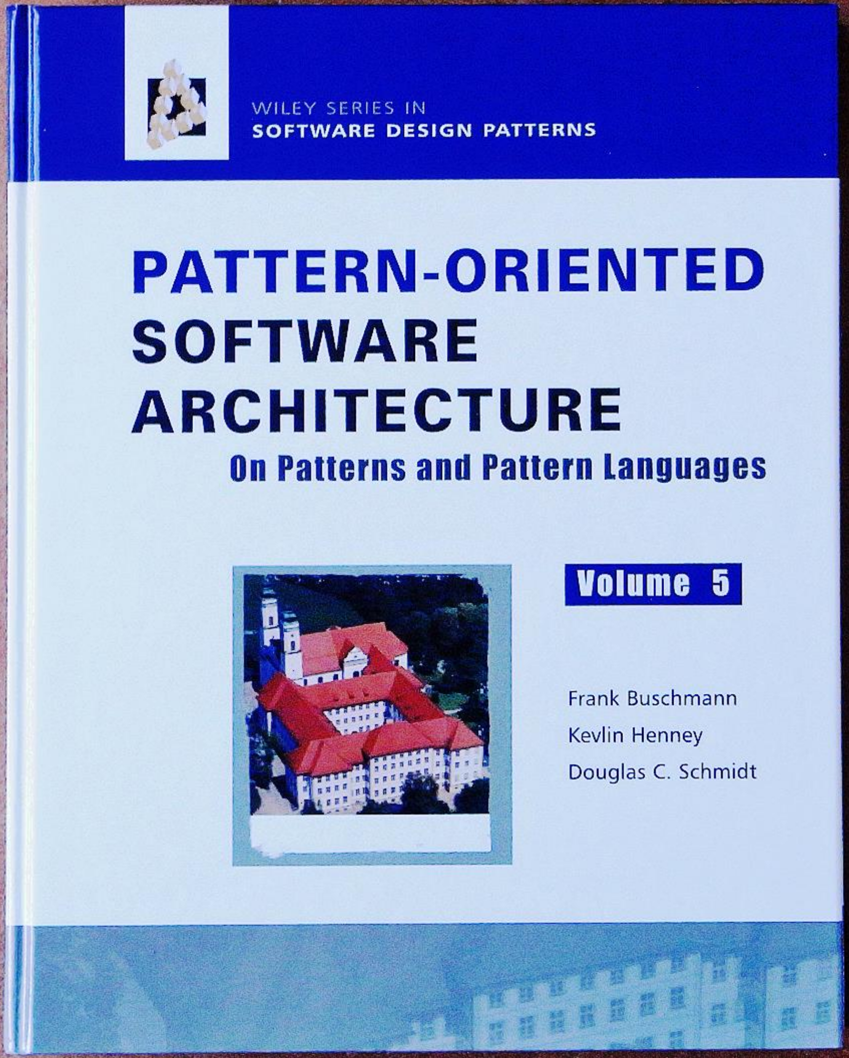
Expert

Proficient

Competent

Advanced Beginner

Novice



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED
SOFTWARE
ARCHITECTURE
On Patterns and Pattern Languages

Volume 5

Frank Buschmann
Kevlin Henney
Douglas C. Schmidt

At some level the style becomes the substance.