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

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

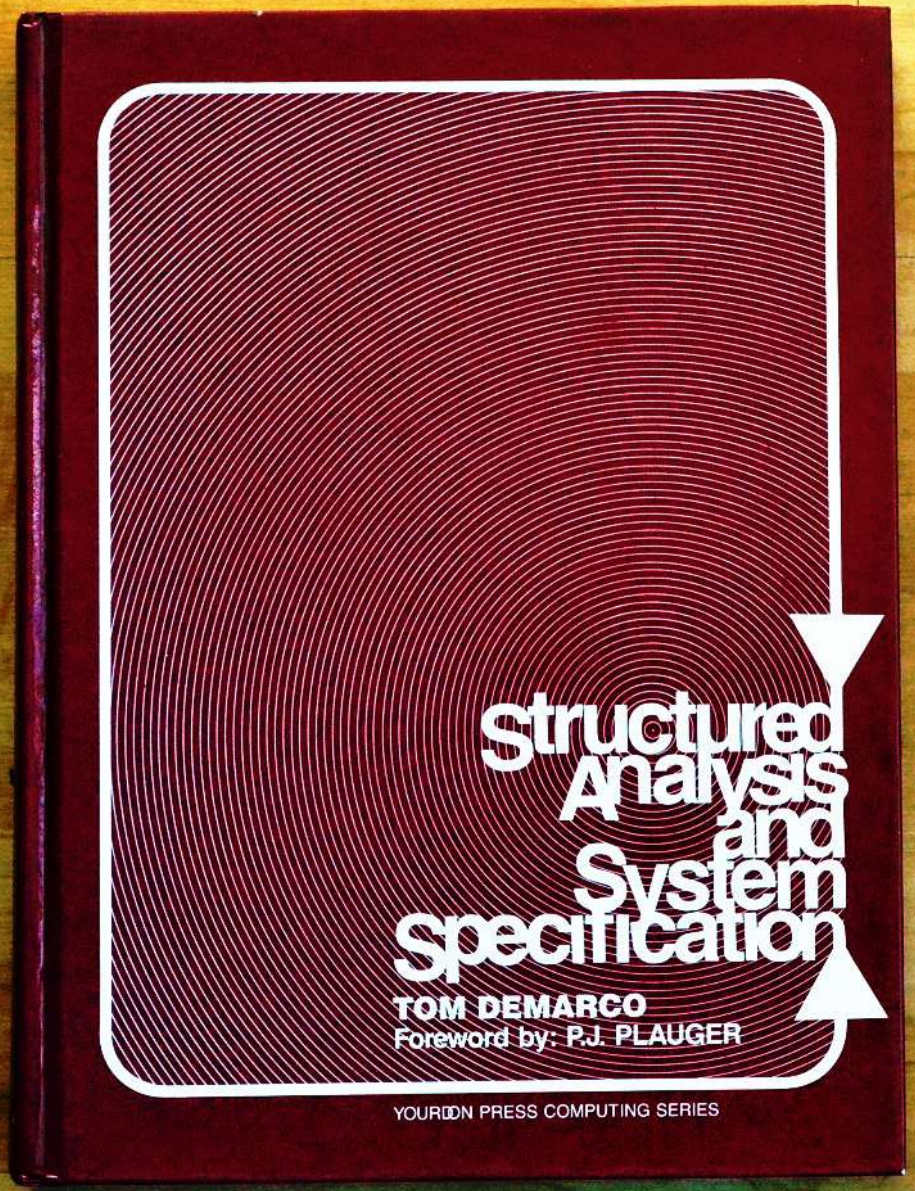
Dependency Inversion

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 in an article by the same name as part of his *Principles of Object Oriented Design*, made popular by his book *Agile Software Development, Principles, Patterns, and Practices*. 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. 98, 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

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*.

1 of 1 a

Cohesion

Mon,

means, but why it's a problem. I can say the same thing about cohesion. One of

the sharpest developers I know sometimes has problems with the concept, and

once

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

some

is not

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

Cohesion comes from the word that "adhesion" comes from. Unlike

word

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

is sti

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. Write programs to handle text streams, because that is a universal interface.

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/

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/


THOS. SMITH & SONS
OF SHEFFIELD LS
1902

The effect of portion size on how much people eat is something of a mystery – why don't they simply leave what they don't want, or alternatively, where possible, why not help themselves to more?

http://bps-research-digest.blogspot.com/2006/06/power-of-one-why-larger-portions-cause.html

The effect of portion size on how much people eat is something of a mystery – why don't they simply leave what they don't want, or alternatively, where possible, why not help themselves to more?

Andrew Geier and colleagues at the University of Pennsylvania think it has to do with 'Unit bias' – "…the sense that a single entity (within a reasonable range of sizes) is the appropriate amount to engage, consume or consider".

http://bps-research-digest.blogspot.com/2006/06/power-of-one-why-larger-portions-cause.html

The effect of portion size on how much people eat is something of a mystery – why don't they simply leave what they don't want, or alternatively, where possible, why not help themselves to more?

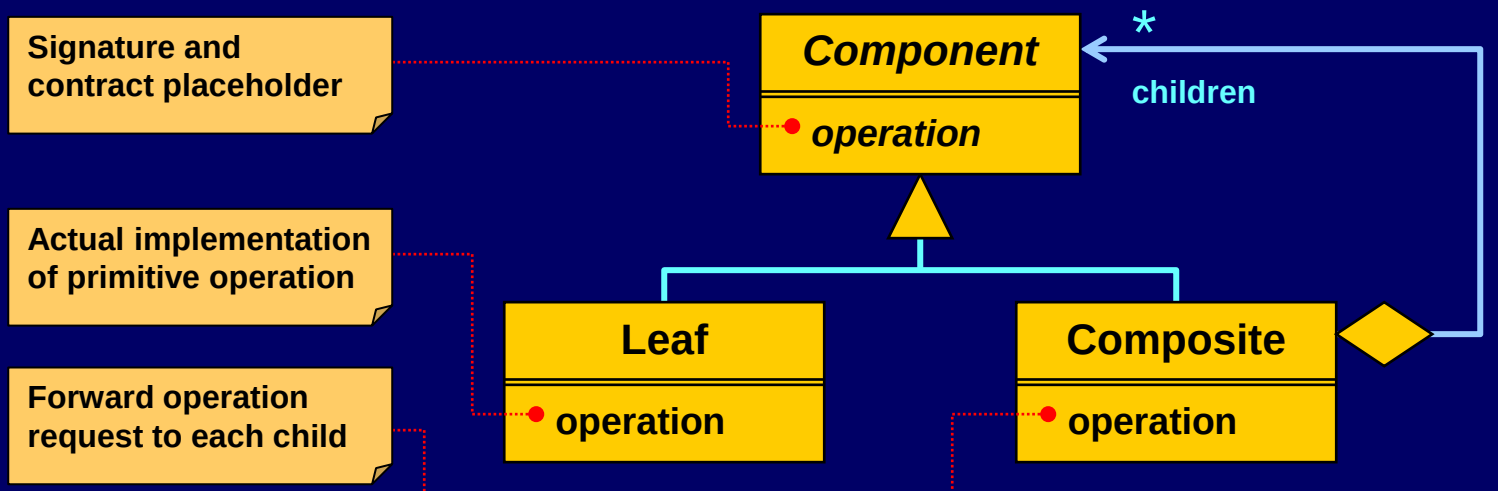
Andrew Geier and colleagues at the University of Pennsylvania think it has to do with 'Unit bias' – "…the sense that a single entity (within a reasonable range of sizes) is the appropriate amount to engage, consume or consider".

The researchers concluded that this 'unit bias' applies in other walks of life too – they cited the example of films: "double features are rare, but very long movies are not", and amusement-park rides: "one ride on a particular attraction is usually enough, whether it takes one or five minutes".

http://bps-research-digest.blogspot.com/2006/06/power-of-one-why-larger-portions-cause.html

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

Grady Booch, *Object Solutions*


Signature and contract placeholder
Component
operation
*
children
Actual implementation of primitive operation
Leaf
operation
Composite
operation
Forward operation request to each child

To hide the hierarchical nature of the Composite arrangement from clients, its component interface must accumulate all methods offered by its leaf and composite objects. The more diverse these functions are, the more the component interface becomes bloated with functions implemented only by few leaf and composite objects, making the interface useless for clients.

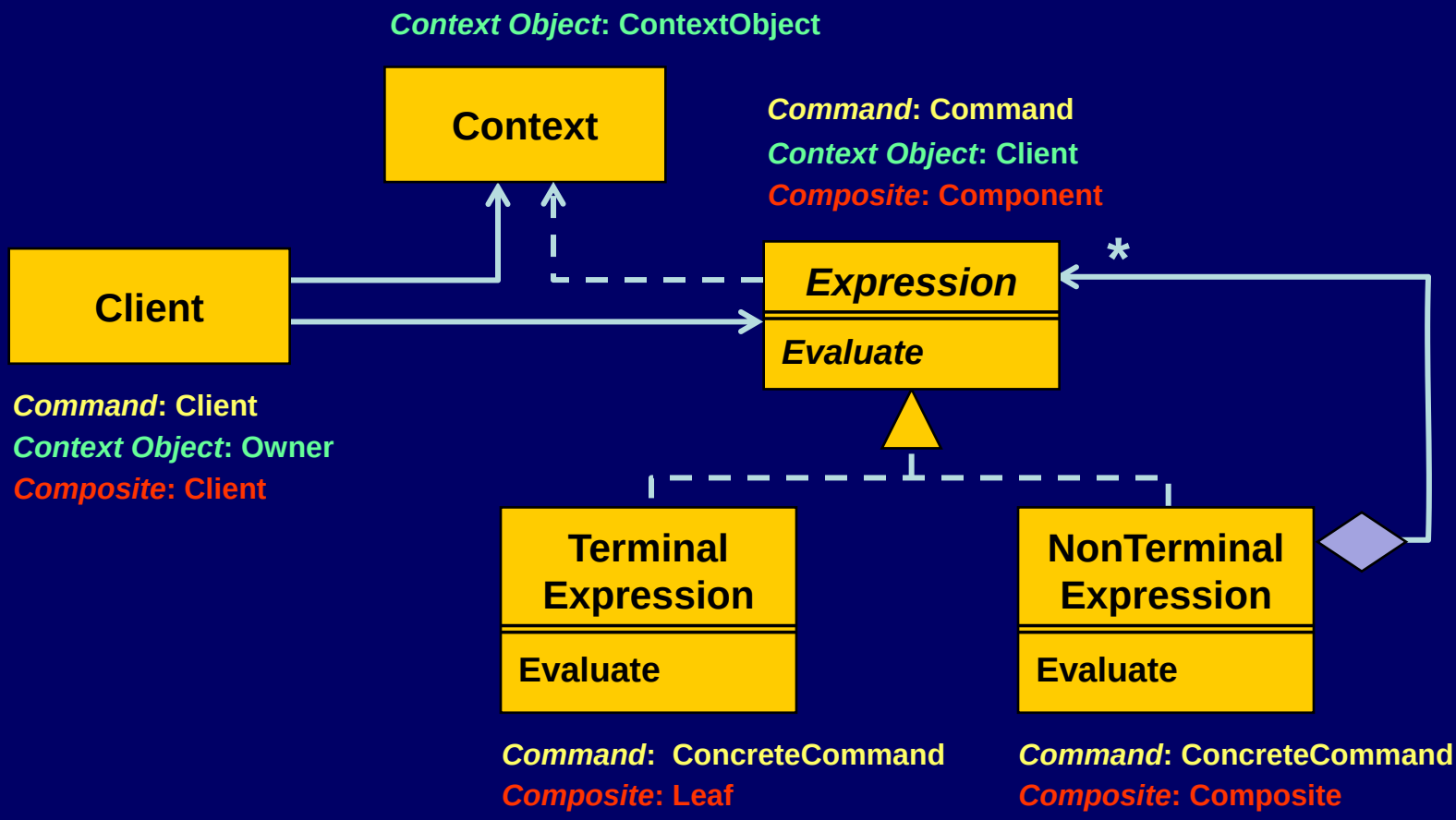
Frank Buschmann, Kevlin Henney & Douglas C Schmidt

Pattern-Oriented Software Architecture, Volume 4:
A Pattern Language for Distributed Computing

An Interpreter design defines a direct and convenient way to represent and interpret grammars for little languages, such as structured messages and scripts, and thus avoids the complexities of more sophisticated representation models.

Frank Buschmann, Kevlin Henney & Douglas C Schmidt

Pattern-Oriented Software Architecture, Volume 4:
A Pattern Language for Distributed Computing


Context Object: ContextObject
Context
Command: Command
Context Object: Client
Composite: Component
Client
Expression
Evaluate
*
Command: Client
Context Object: Owner
Composite: Client
Terminal Expression
Evaluate
NonTerminal Expression
Evaluate
Command: ConcreteCommand
Composite: Leaf
Command: ConcreteCommand
Composite: Composite

```
    "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)
933262154439441526816992388562667004907159682643816214685929638952175999932299
1
560894146397615651828625369792082722375825118521091686400000000000000000000000
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
```

Combined Method

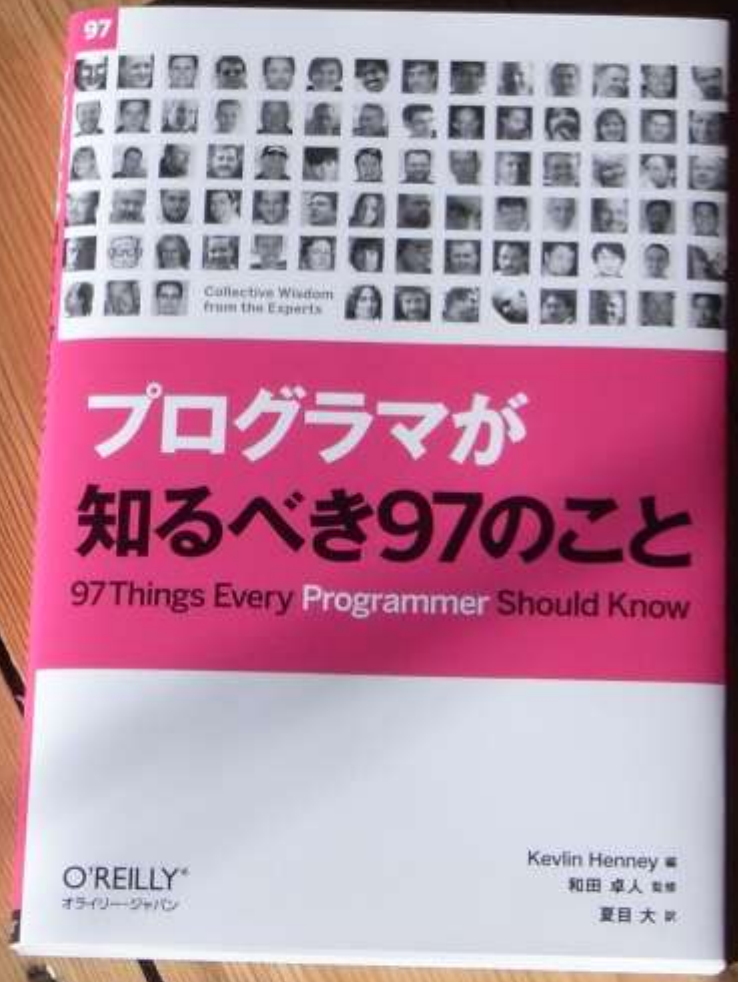
Clients often must invoke multiple methods on a component in the same order to perform a specific task. From a client's perspective, however, it is tedious and error-prone to call the method sequence explicitly each time it wants to execute the task on the component.

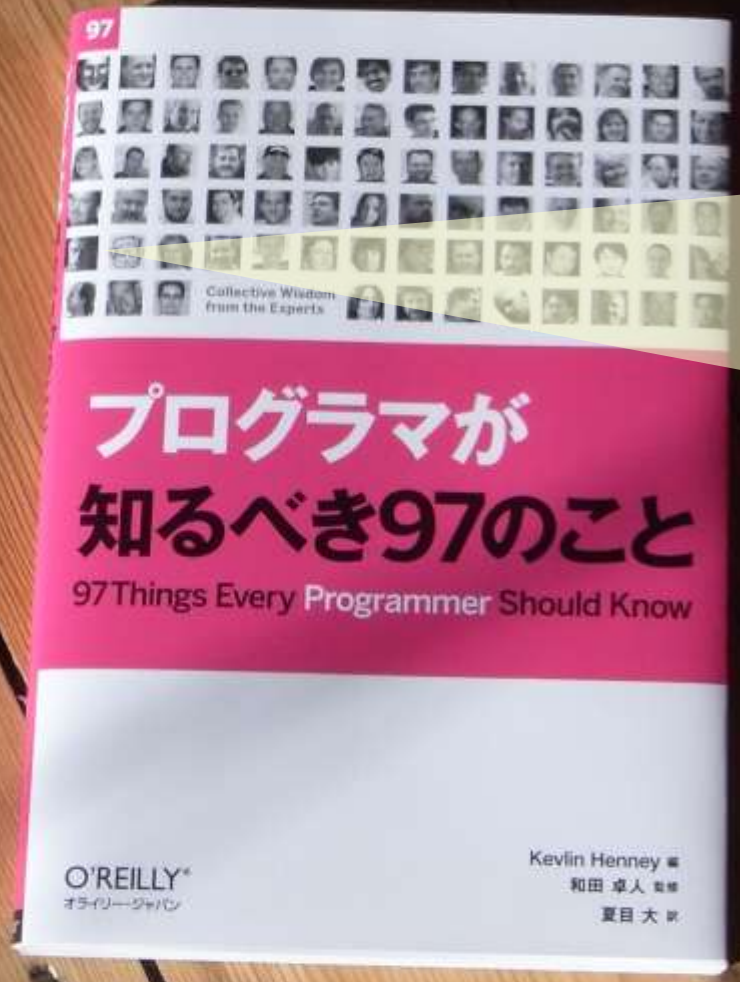
Therefore:

Combine methods that must be, or commonly are, executed together on a component into a single method.

Frank Buschmann, Kevlin Henney & Douglas C Schmidt

Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing


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

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

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*.

1 of 1 a

Cohesion

Mon,

means, but why it's a problem. I can say the same thing about cohesion. One of

the sharpest developers I know sometimes has problems with the concept, and

once

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

some

is not

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

Cohesion comes from the word that "adhesion" comes from. Unlike

word

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

is sti

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*.

Single Responsibility

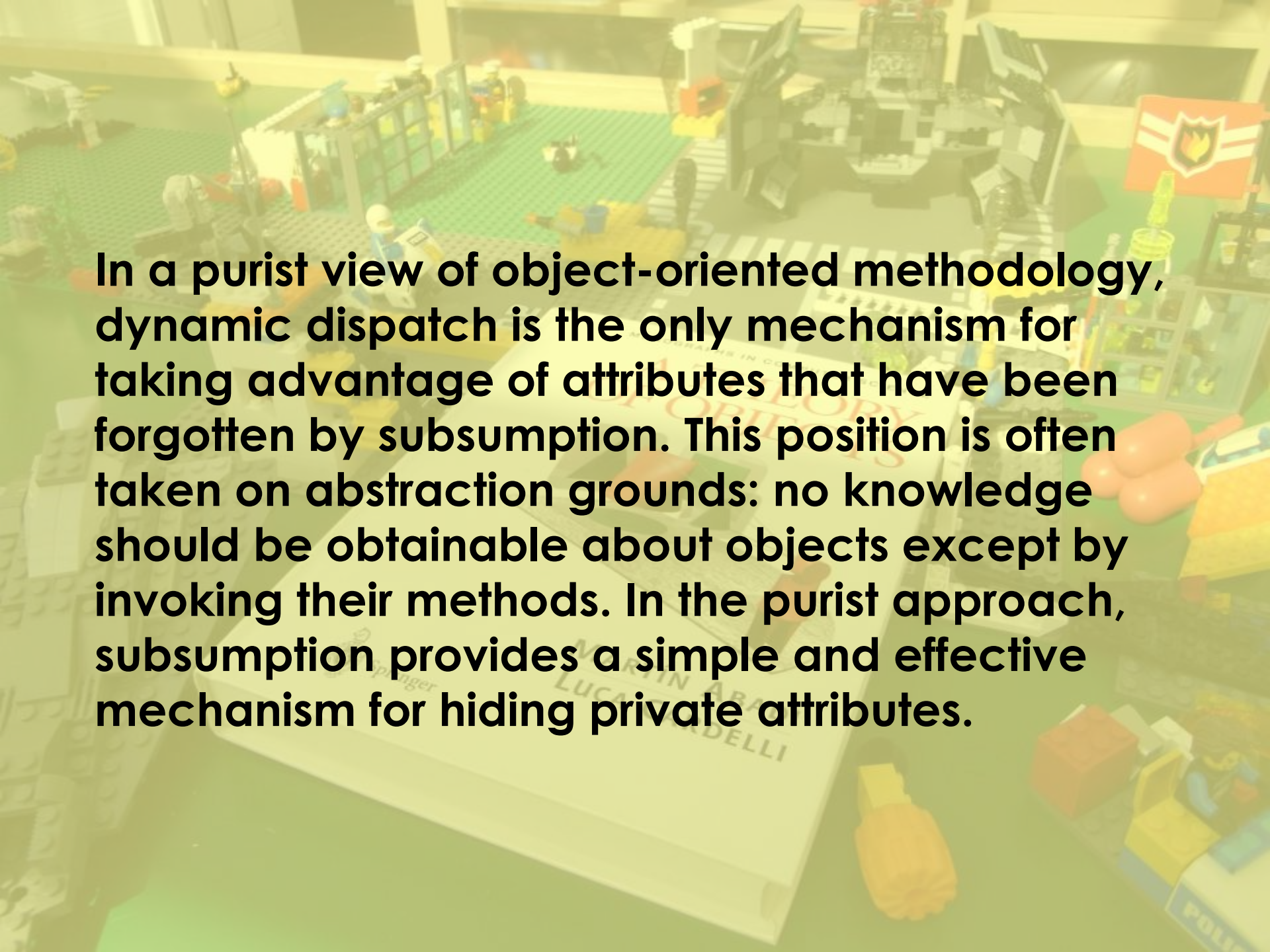
Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

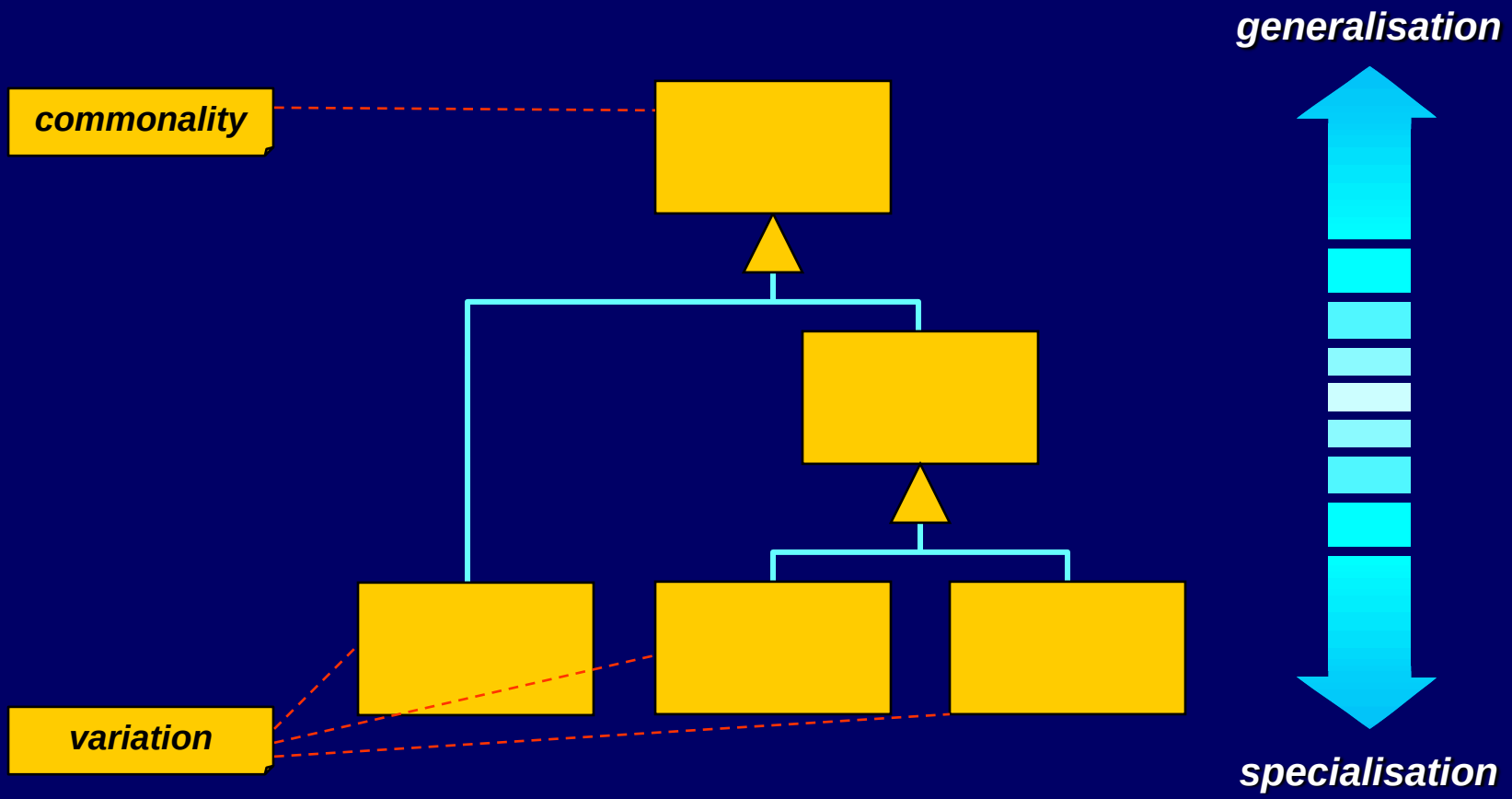

MONOGRAPHS IN COMPUTER SCIENCE
A THEORY OF OBJECTS
Springer
MARTÍN ABADI
LUCA CARDELLI



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

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;
    }
}
```

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

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

```
List<string> list = new RecentlyUsedList();
list.Add("Hello, World!");
list.Clear();
list.Add("Hello, World!");
list.Add("Goodbye, World!");
list.Add("Hello, World!");
Debug.Assert(list.Count == 2);
list.Insert(1, "Hello, World!");
list.Add(null); // throws
```

```
given:
expectedSize = Count + (Contains(newItem) ? 0 : 1)
precondition:
newItem != null
postcondition:
this[0]== newItem && Count == expectedSize
```

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

```
precondition:
index >= 0 && index < Count
postcondition:
returns != null
```

What would a class derived from *RecentlyUsedList* be permitted to do and be disallowed from doing?

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"

OO ≡ ADT?

OO ≠ ADT

```
typedef struct RecentlyUsedList RecentlyUsedList;
```

```
RecentlyUsedList * create();
void destroy(RecentlyUsedList *);
bool isEmpty(const RecentlyUsedList *);
int size(const RecentlyUsedList *);
void add(RecentlyUsedList *, int toAdd);
int get(const RecentlyUsedList *, int index);
bool equals(const RecentlyUsedList *, const RecentlyUsedList *);
```

```
struct RecentlyUsedList
{
    int * items;
    int length;
};
```

```
RecentlyUsedList * create()
{
    RecentlyUsedList * result = (RecentlyUsedList *) malloc(sizeof(RecentlyUsedList));
    result->items = 0;
    result->length = 0;
    return result;
}

void destroy(RecentlyUsedList * self)
{
    free(self->items);
    free(self);
}

bool isEmpty(const RecentlyUsedList * self)
{
    return self->length == 0;
}

int size(const RecentlyUsedList * self)
{
    return self->length;
}

static int indexOf(const RecentlyUsedList * self, int toFind)
{
    int result = -1;
    for(int index = 0; result == -1 && index != self->length; ++index)
        if(self->items[index] == toFind)
            result = index;
    return result;
}

static void removeAt(RecentlyUsedList * self, int index)
{
    memmove(&self->items[index], &self->items[index + 1], (self->length - index - 1) * sizeof(int));
    --self->length;
}

void add(RecentlyUsedList * self, int toAdd)
{
    int found = indexOf(self, toAdd);
    if(found != -1)
        removeAt(self, found);
    self->items = (int *) realloc(self->items, (self->length + 1) * sizeof(int));
    self->items[self->length] = toAdd;
    ++self->length;
}

int get(const RecentlyUsedList * self, int index)
{
    return self->items[self->length - index - 1];
}

bool equals(const RecentlyUsedList * lhs, const RecentlyUsedList * rhs)
{
    return lhs->length == rhs->length && memcmp(lhs->items, rhs->items, lhs->length * sizeof(int)) == 0;
}
```

```
struct RecentlyUsedList
{
    std::vector<int> items;
};
```

```
extern "C"
{
    RecentlyUsedList * create()
    {
        return new RecentlyUsedList;
    }
    void destroy(RecentlyUsedList * self)
    {
        delete self;
    }
    bool isEmpty(const RecentlyUsedList * self)
    {
        return self->items.empty();
    }
    int size(const RecentlyUsedList * self)
    {
        return self->items.size();
    }
    void add(RecentlyUsedList * self, int toAdd)
    {
        std::vector<int>::iterator found =
            std::find(self->items.begin(), self->items.end(), toAdd);
        if(found != self->items.end())
            self->items.erase(found);
        self->items.push_back(toAdd);
    }
    int get(const RecentlyUsedList * self, int index)
    {
        return self->items[self->items.size() - index - 1];
    }
    bool equals(const RecentlyUsedList * lhs, const RecentlyUsedList * rhs)
    {
        return lhs->items == rhs->items;
    }
}
```

If we want to emphasize the programmatic aspect of a type that has an associated `operator==`, we say "objects compare equal", but never "objects are equal". [...]

We deliberately avoid equivocal phrases such as "objects are equal", "objects are the same", or "objects are identical".

John Lakos

Normative Language to Describe Value Copy Semantics

http://www.open-std.org/jtc1/sc22/WG21/docs/papers/2007/n2479.pdf

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 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

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.***

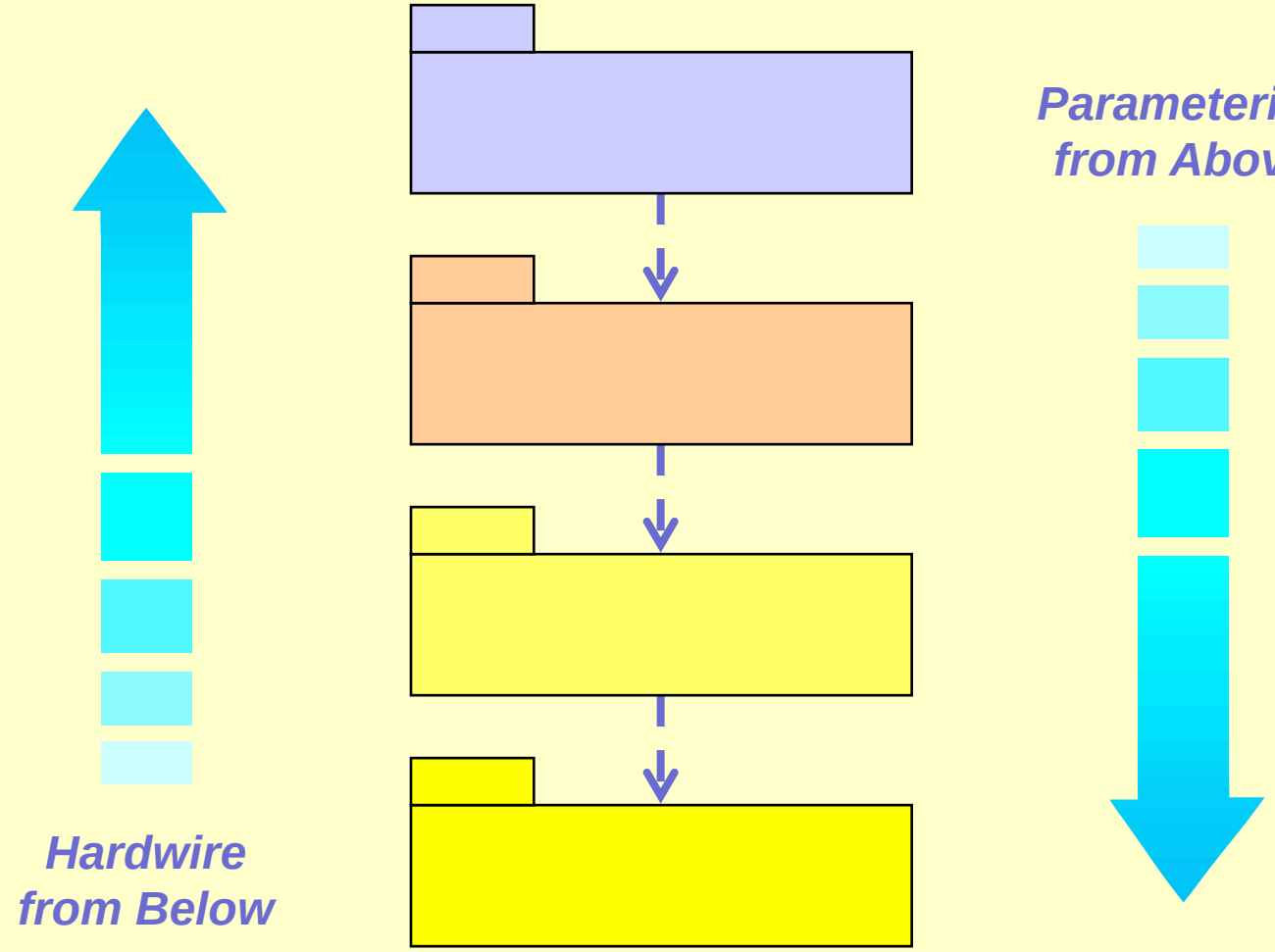
http://en.wikipedia.org/wiki/Dependency_inversion_principle

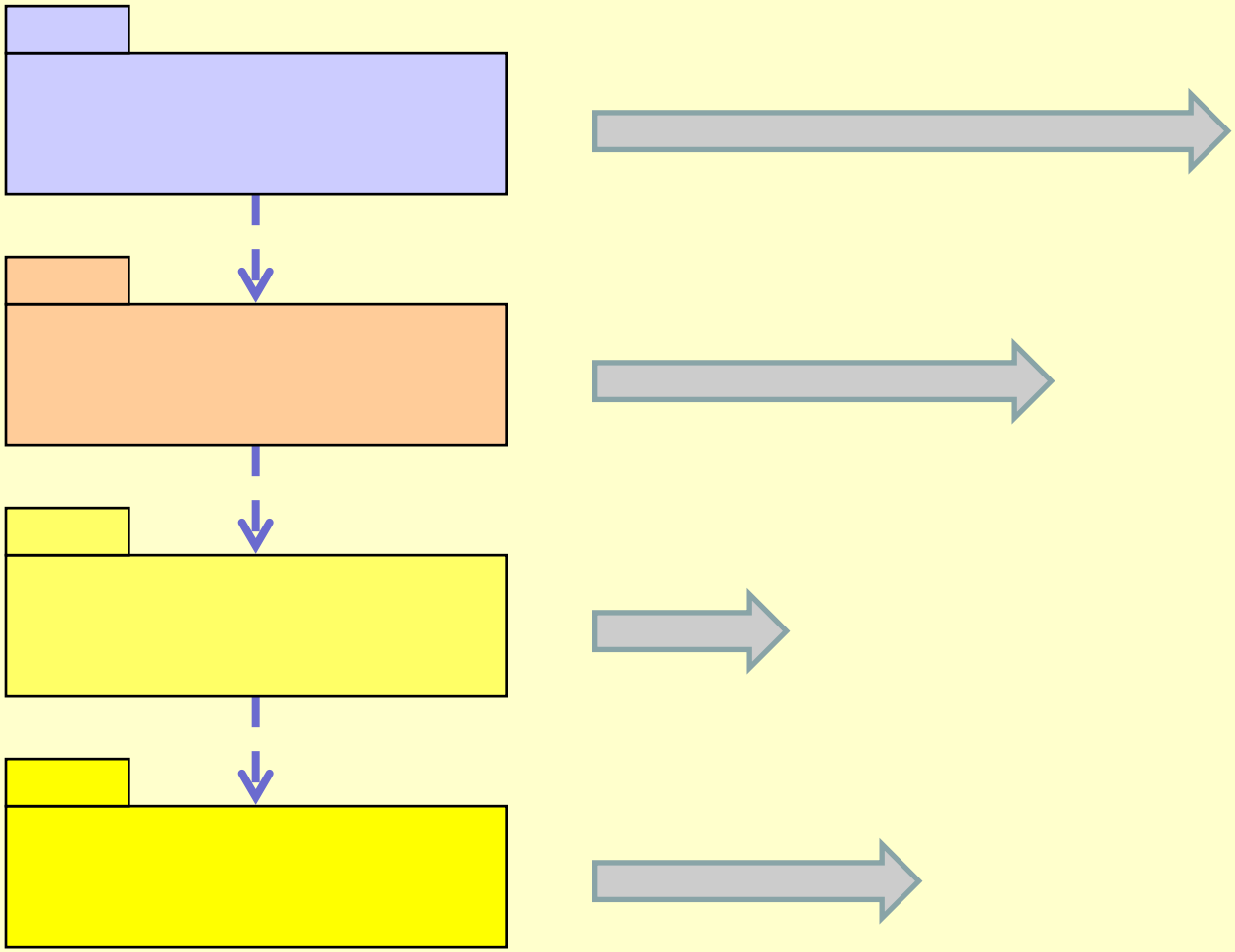
inversion, *noun*

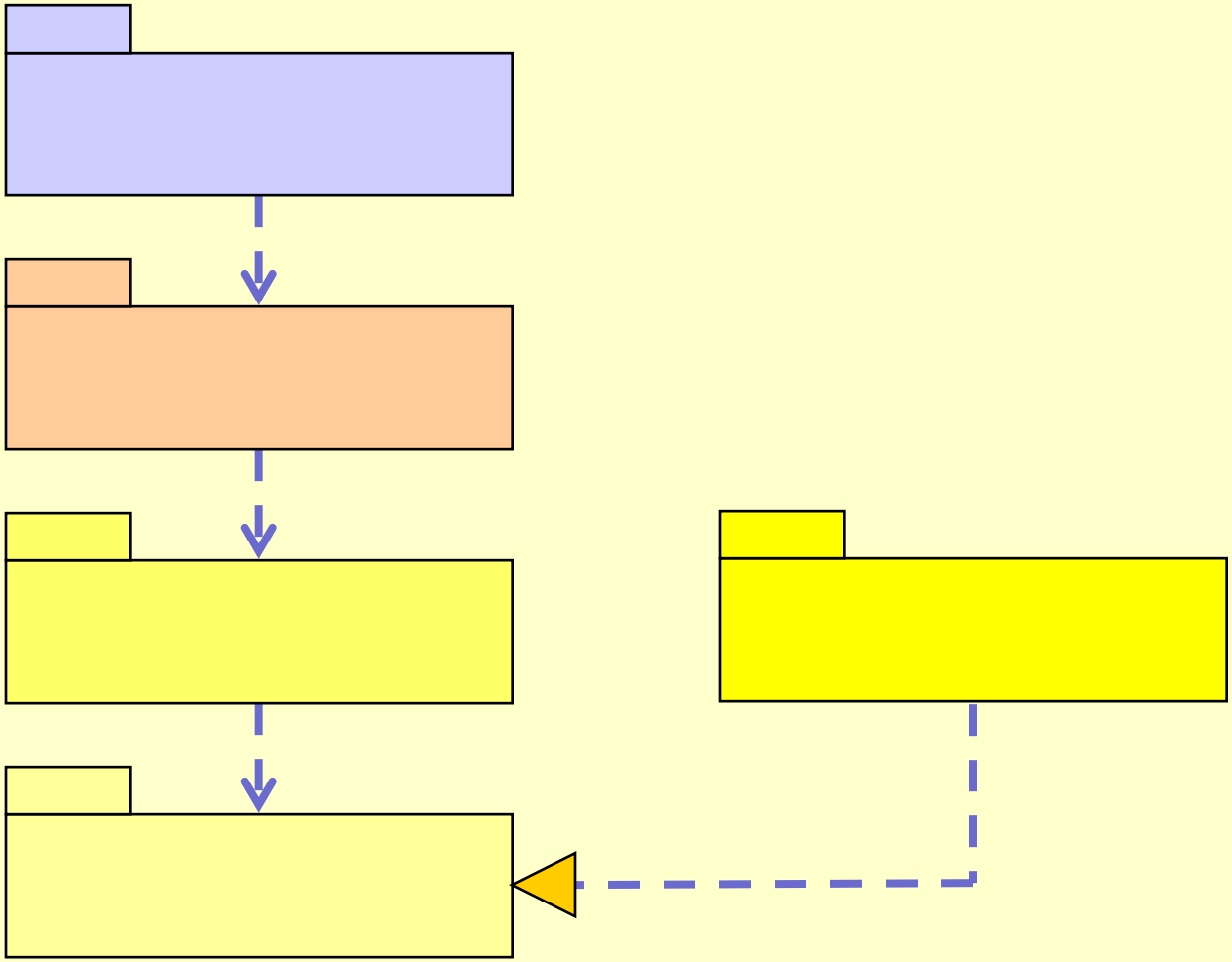
- the action of inverting or the state of being inverted
- reversal of the normal order of words, normally for rhetorical effect
- an inverted interval, chord, or phrase
- a reversal of the normal decrease of air temperature with altitude, or of water temperature with depth

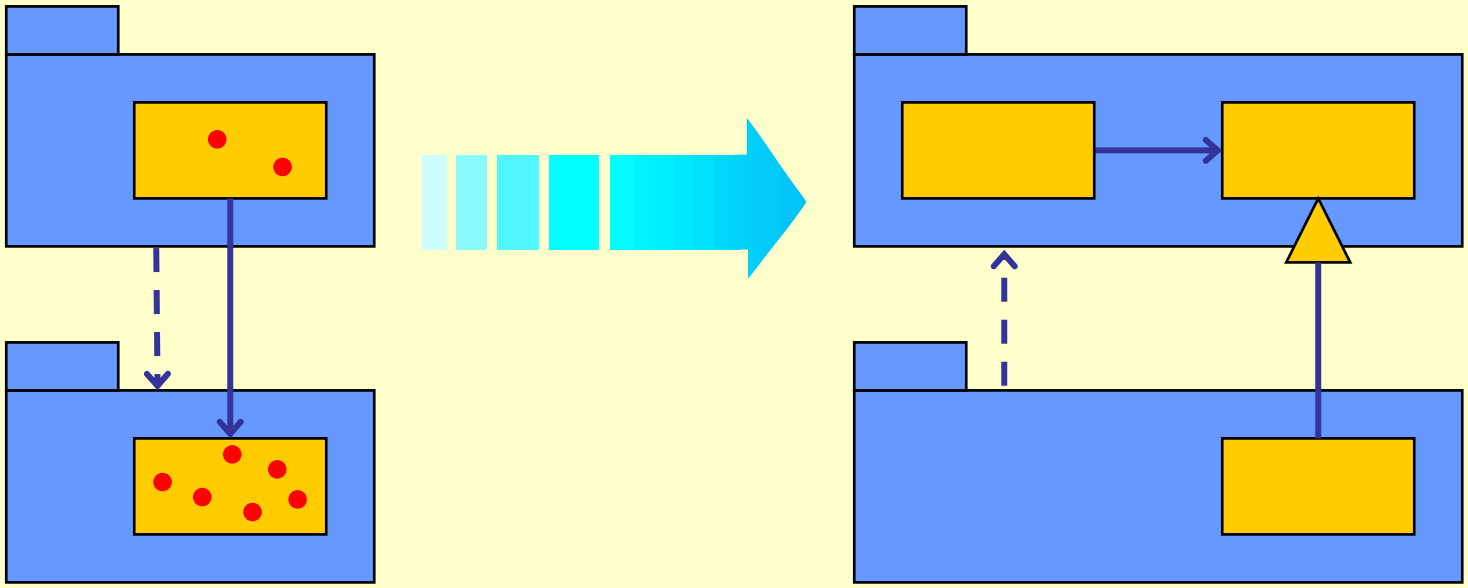
Concise Oxford English Dictionary


THE SINGLETON
Single Malt Scotch Whisky
of Dufftown
12


Parameterize
from Above
Hardwire
from Below


Rate of change





Scenario buffering by dot-voting possible changes and then readjusting dependencies

S

O

L

I

D

F

L

U

I

D

Functional

L

U

I

D

I still have a deep fondness for the Lisp model. It is simple, elegant, and something with which all developers should have an infatuation at least once in their programming life.

Kevlin Henney

"A Fair Share (Part I)", *CUJ C++ Experts Forum*, October 2002

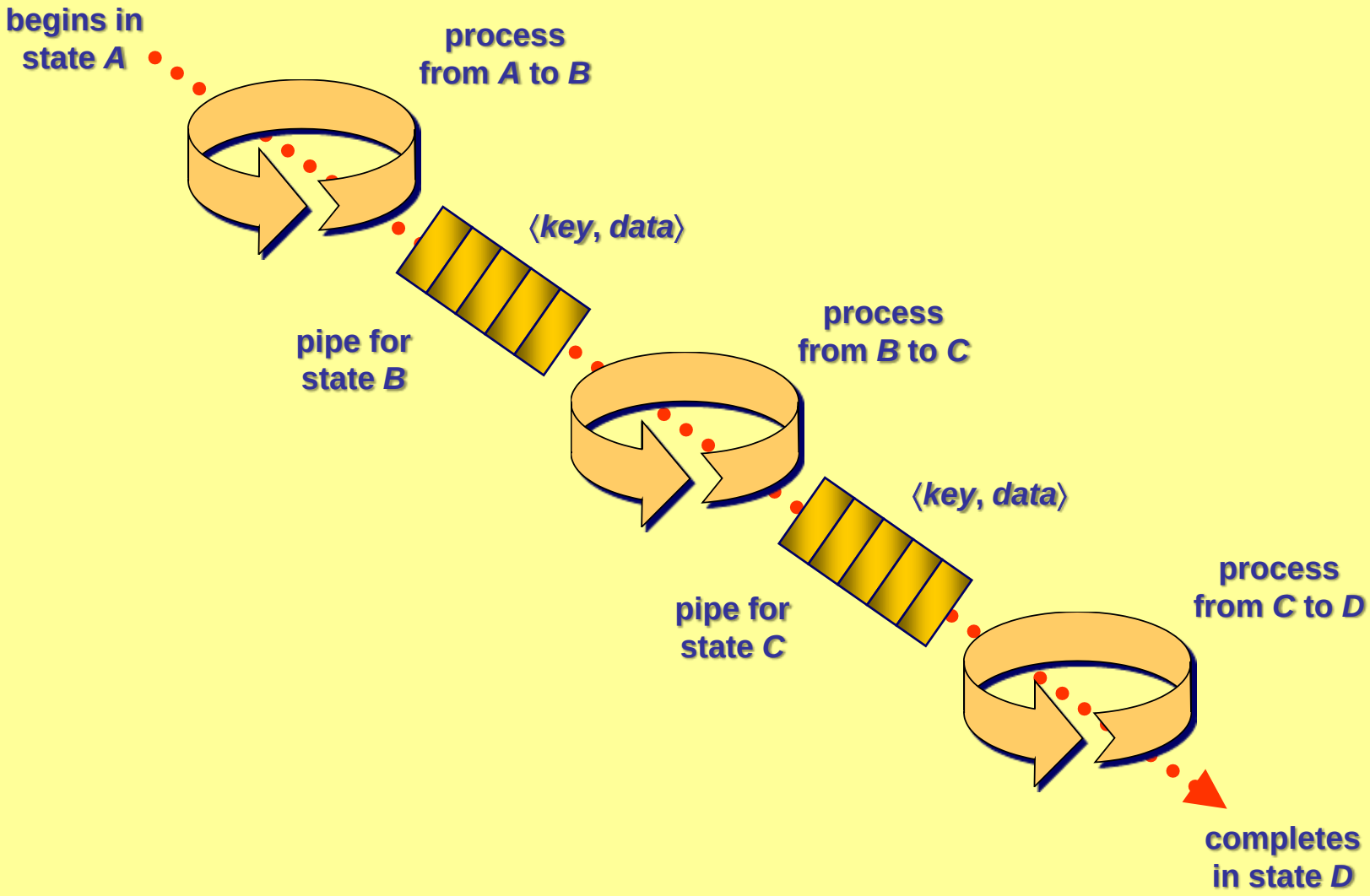

WILEY SERIES IN
SOFTWARE DESIGN PATTERNS
PATTERN-ORIENTED
SOFTWARE
ARCHITECTURE
A Pattern Language for
Distributed Computing
Volume 4
Frank Buschmann
Kevlin Henney
Douglas C. Schmidt

Pipes and Filters

Some applications process streams of data: input data streams are transformed stepwise into output data streams. However, using common and familiar request/response semantics for structuring such types of application is typically impractical. Instead we must specify an appropriate data flow model for them.

Therefore:

Divide the application's task into several self-contained data processing steps and connect these steps to a data processing pipeline via intermediate data buffers.


begins in
state A
process
from A to B
⟨key, data⟩
pipe for
state B
process
from B to C
⟨key, data⟩
pipe for
state C
process
from C to D
completes
in state D

Functional

Loose

U

I

D

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.

Alan Kay

One of the most pure object-oriented programming models yet defined is the Component Object Model (COM). It enforces all of these principles rigorously. Programming in COM is very flexible and powerful as a result. There is no built-in notion of equality. There is no way to determine if an object is an instance of a given class.

William Cook
"On Understanding Data Abstraction, Revisited"

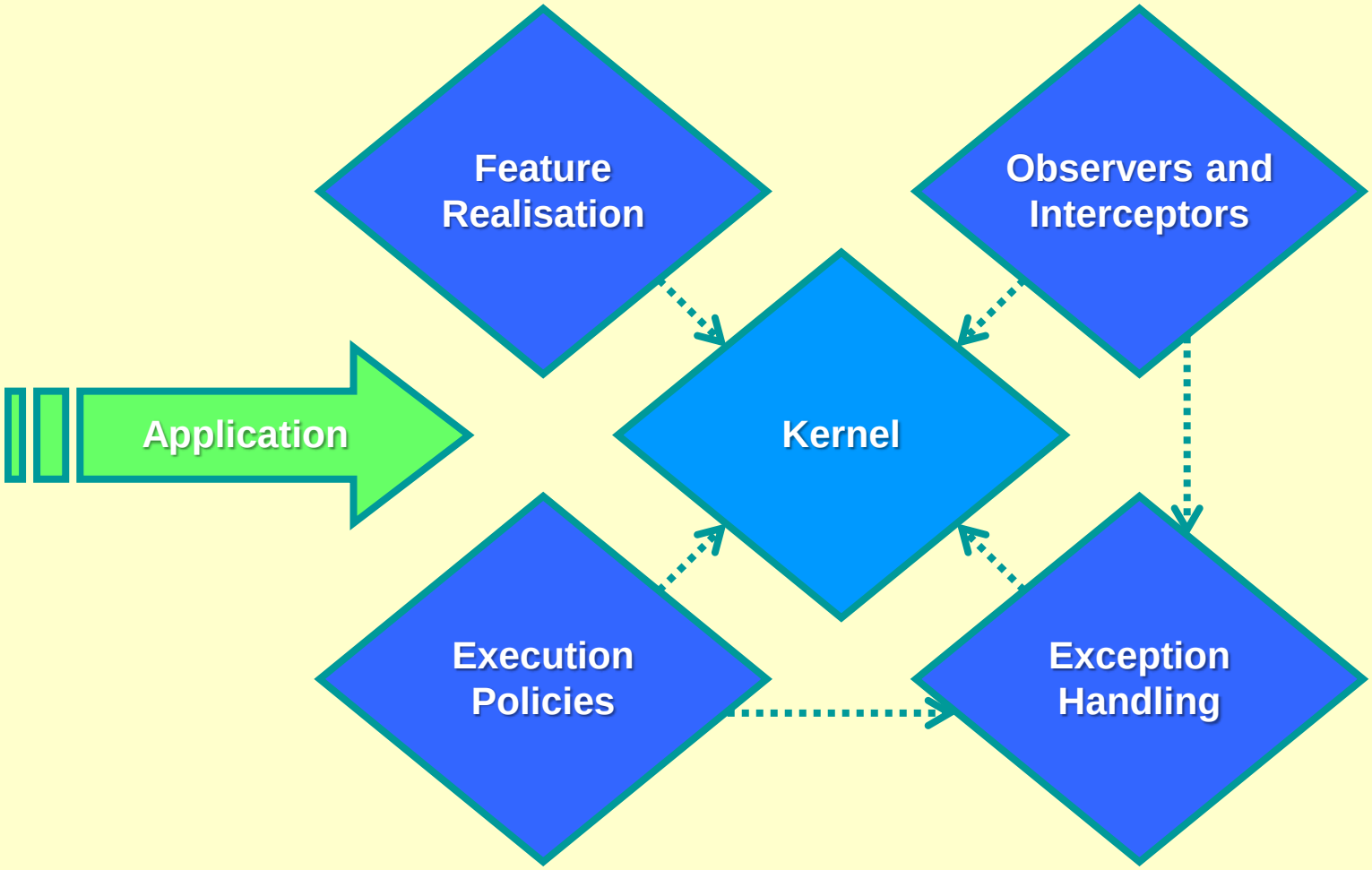
Event-Based, Implicit Invocation

The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register [or declare] an interest in an event by associating a procedure with it. When the event is announced, the system itself invokes all of the procedures that have been registered for the event. Thus an announcement "implicitly" causes the invocation of procedures in other modules.

Architecturally speaking, the components in an implicit invocation style are modules whose interfaces provide both a collection of procedures (as with abstract data types) and a set of events.

Mary Shaw & David Garlan

Software Architecture: Perspectives on an Emerging Discipline


Feature Realisation
Observers and Interceptors
Application
Kernel
Execution Policies
Exception Handling

Functional

Loose

Unit Testable

I

D

Test early.

Test often.

Test automatically.

Andrew Hunt and David Thomas
The Pragmatic Programmer

```
public static class Year
{
    public static bool IsLeap(int year) ...
}
```

```
namespace Leap_year_spec
{
    [TestFixture]
    public class A_year_is_a_leap_year
    {
        [Test] public void If_it_is_divisible_by_4_but_not_by_100() ...
        [Test] public void If_it_is_divisible_by_400() ...
    }

    [TestFixture]
    public class A_year_is_not_a_leap_year
    {
        [Test] public void If_it_is_not_divisible_by_4() ...
        [Test] public void If_it_is_divisble_by_100_but_not_by_400() ...
    }

    [TestFixture]
    public class A_year_is_not_considered_valid
    {
        [Test] public void If_it_is_0() ...
        [Test] public void If_it_is_negative() ...
    }
}
```

```
namespace Leap_year_spec
{
    [TestFixture]
    public class A_year_is_a_leap_year
    {
        [Test] public void If_it_is_divisible_by_4_but_not_by_100() ...
        [Test] public void If_it_is_divisible_by_400() ...
    }

    [TestFixture]
    public class A_year_is_not_a_leap_year
    {
        [Test] public void If_it_is_not_divisible_by_4() ...
        [Test] public void If_it_is_divisble_by_100_but_not_by_400() ...
    }

    [TestFixture]
    public class A_year_is_not_considered_valid
    {
        [Test] public void If_it_is_0() ...
        [Test] public void If_it_is_negative() ...
    }
}
```

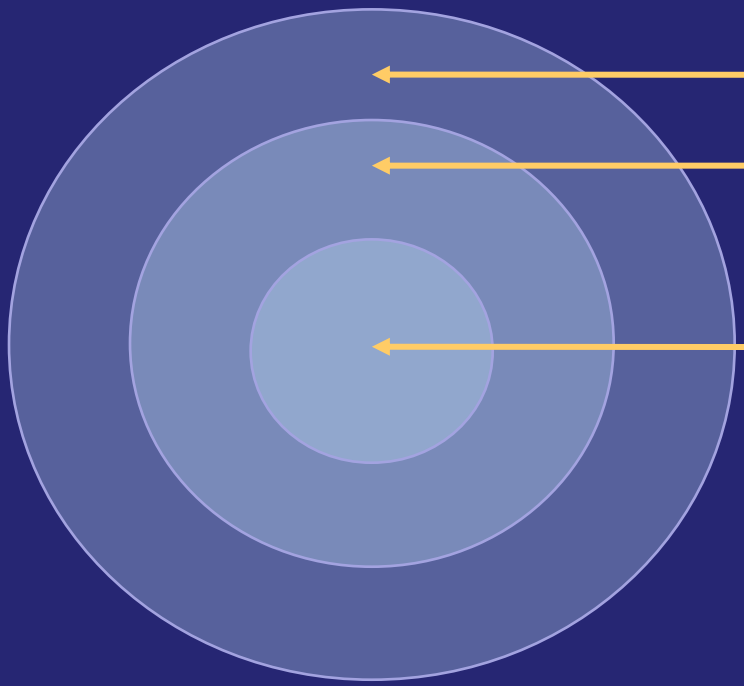
A test is not a unit test if:

- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it.

Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes.

Michael Feathers, "A Set of Unit Testing Rules"

http://www.artima.com/weblogs/viewpost.jsp?thread=126923



Necessarily not unit testable, such as interactions with external dependencies

Unit testable in theory, but not unit testable in practice

Unit testable in practice

Functional

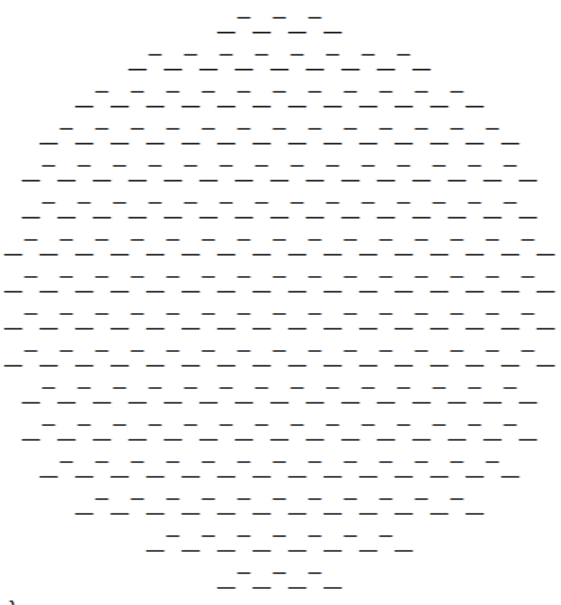
Loose

Unit Testable

Introspective

D

```
#define _ -F<00||--F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
```



```
}
```

```
(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))))
```

```
def eval(x, env=global_env):
    "Evaluate an expression in an environment."
    if isa(x, Symbol):              # variable reference
        return env.find(x)[x]
    elif not isa(x, list):          # constant literal
        return x
    elif x[0] == 'quote':           # (quote exp)
        (_, exp) = x
        return exp
    elif x[0] == 'if':              # (if test conseq alt)
        (_, test, conseq, alt) = x
        return eval((conseq if eval(test, env) else alt), env)
    elif x[0] == 'set!':            # (set! var exp)
        (_, var, exp) = x
        env.find(var)[var] = eval(exp, env)
    elif x[0] == 'define':          # (define var exp)
        (_, var, exp) = x
        env[var] = eval(exp, env)
    elif x[0] == 'lambda':          # (lambda (var*) exp)
        (_, vars, exp) = x
        return lambda *args: eval(exp, Env(vars, args, env))
    elif x[0] == 'begin':           # (begin exp*)
        for exp in x[1:]:
            val = eval(exp, env)
        return val
    else:                           # (proc exp*)
        exps = [eval(exp, env) for exp in x]
        proc = exps.pop(0)
        return proc(*exps)

isa = isinstance

Symbol = str
```

```
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)
```

```
namespace Leap_year_spec
{
    [TestFixture]
    public class A_year_is_a_leap_year
    {
        [Test] public void If_it_is_divisible_by_4_but_not_by_100() ...
        [Test] public void If_it_is_divisible_by_400() ...
    }

    [TestFixture]
    public class A_year_is_not_a_leap_year
    {
        [Test] public void If_it_is_not_divisible_by_4() ...
        [Test] public void If_it_is_divisble_by_100_but_not_by_400() ...
    }

    [TestFixture]
    public class A_year_is_not_considered_valid
    {
        [Test] public void If_it_is_0() ...
        [Test] public void If_it_is_negative() ...
    }
}
```

```
namespace Leap_year_spec
{
    [TestFixture]
    public class A_year_is_a_leap_year
    {
        [Test] public void If_it_is_divisible_by_4_but_not_by_100() ...
        [Test] public void If_it_is_divisible_by_400() ...
    }

    [TestFixture]
    public class A_year_is_not_a_leap_year
    {
        [Test] public void If_it_is_not_divisible_by_4() ...
        [Test] public void If_it_is_divisble_by_100_but_not_by_400() ...
    }

    [TestFixture]
    public class A_year_is_not_considered_valid
    {
        [Test] public void If_it_is_0() ...
        [Test] public void If_it_is_negative() ...
    }
}
```

Functional

Loose

Unit Testable

Introspective

'Dempotent

Idempotence is the property of certain operations in mathematics and computer science, that they can be applied multiple times without changing the result beyond the initial application. The concept of idempotence arises in a number of places in abstract algebra (in particular, in the theory of projectors and closure operators) and functional programming (in which it is connected to the property of referential transparency).

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

Asking a question should not change the answer.

Bertrand Meyer

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


beyond
SHARED PATH
Please consider
other path users
WA
THIS IS A WORK
ALL USERS SHO
CYCLISTS ARE A
No liability will be accep

When it is not necessary to change, it is necessary not to change.

Lucius Cary

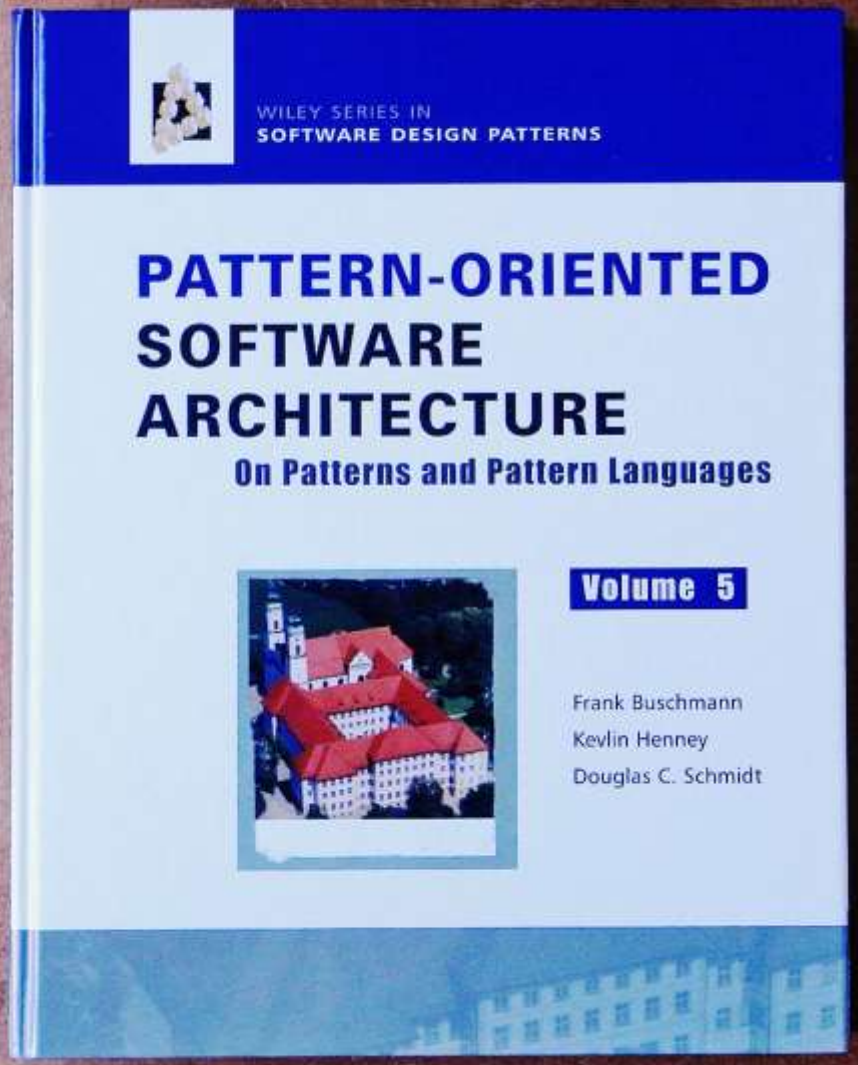
Functional

Loose

Unit Testable

Introspective

'Dempotent



At some level the style becomes the substance.