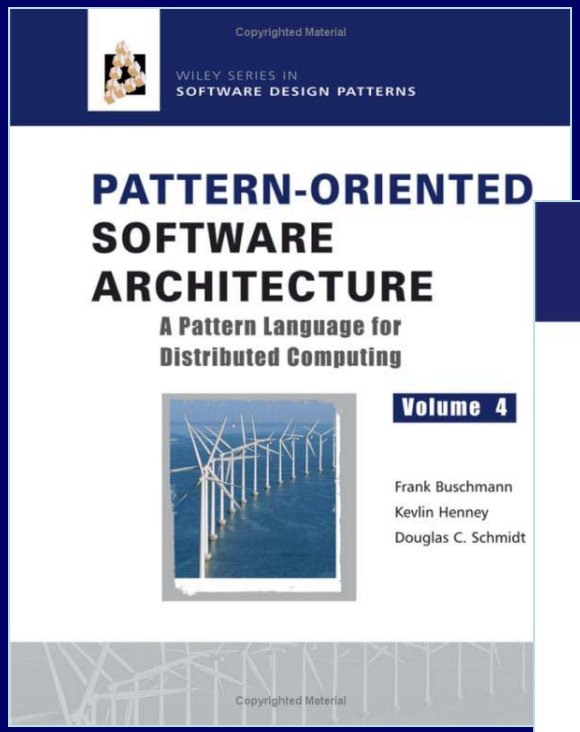
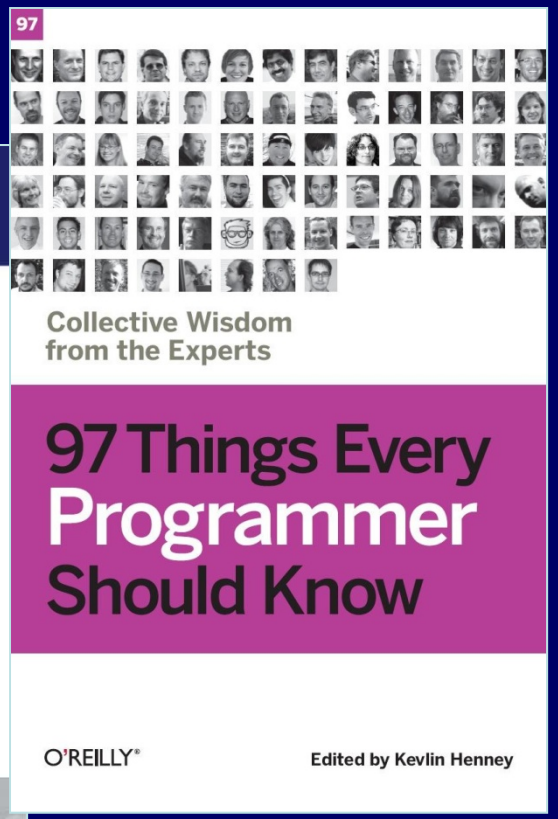
Good Object-Oriented Development

Kevlin Henney

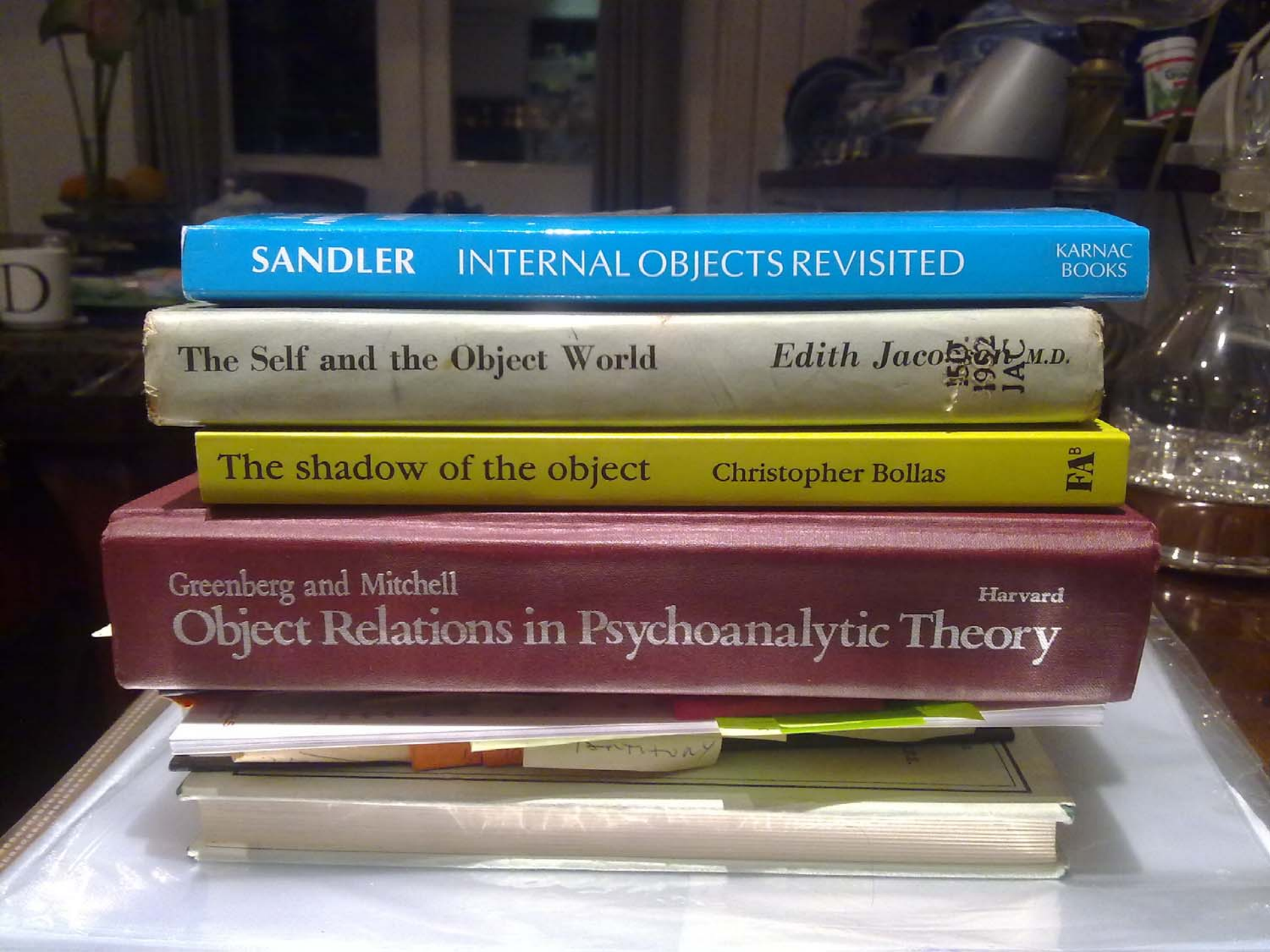
kevlin@curbralan.com

@KevlinHenney





See *http://programmer.97things.oreilly.com*
(also *http://tr.im/97tepsk* and *http://tinyurl.com/97tepsk*)
and follow *@97TEPSK*


SANDLER INTERNAL OBJECTS REVISITED
KARNAC BOOKS
The Self and the Object World
Edith Jacobson M.D.
The shadow of the object
Christopher Bollas
FAB
Greenberg and Mitchell
Harvard
Object Relations in Psychoanalytic Theory

There is one consequence of considering code as software design that completely overwhelms all others. It is so important and so obvious that it is a total blind spot for most software organizations. This is the fact that software is cheap to build.

Jack Reeves

All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.

Grady Booch

If you think good architecture is expensive, try bad architecture.

Brian Foote and Joseph Yoder


Powder
PUSH

encapsulate *enclose (something) in or as if in a capsule.*

- *express the essential feature of (someone or something) succinctly.*
- *enclose (a message or signal) in a set of codes which allow use by or transfer through different computer systems or networks.*
- *provide an interface for (a piece of software or hardware) to allow or simplify access for the user.*

The New Oxford Dictionary of English

Affordances

- The degrees of freedom a design supports in (mis)use
  - Not all affordances are desirable or intended
- Encapsulation involves the reduction of unwanted affordances
  - So, try to avoid offering interfaces that are clumsy to use or that make it easy to do the wrong thing

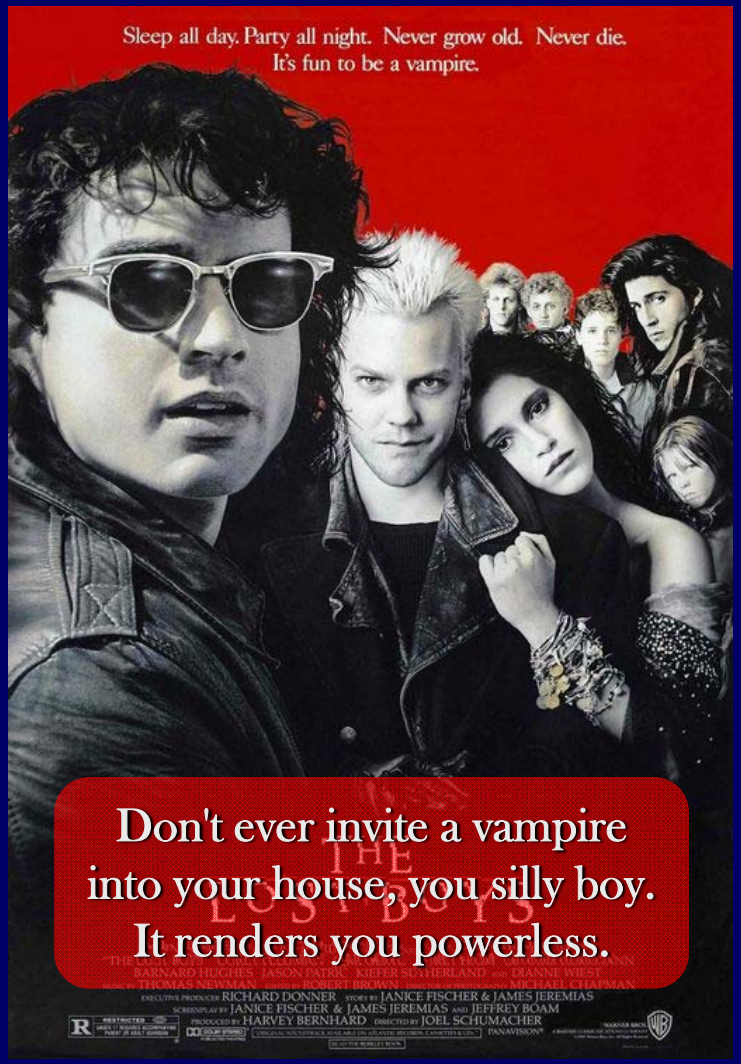

THE AMAZING
BADGER SETT
We made it 10 times as big as
real! So you can ex
underground and f


GET THE NEW LIFE!
GET! GET!!
for CHEERFUL LIFE


CONSIDERATE CONSTRUCTORS
Considerate Constructors Scheme
Improving the image
of Construction
0800 783 1423
www.ccscheme.org.uk

```
public class RecentlyUsedList
{
    public void add(String newItem)
    {
        list.remove(newItem);
        list.add(0, newItem);
    }
    public ArrayList<String> getList()
    {
        return list;
    }
    public void setList(ArrayList<String> newList)
    {
        list = newList;
    }
    private ArrayList<String> list;
}
```

```
RecentlyUsedList list = new RecentlyUsedList();
list.setList(new ArrayList<String>());
list.add("Hello, World!");
assert list.getList().size() == 1;
list.getList().clear();
assert list.getList().isEmpty();
```


Sleep all day. Party all night. Never grow old. Never die.
It's fun to be a vampire.
Don't ever invite a vampire into your house, you silly boy. It renders you powerless.
THE LOST BOYS
EXECUTIVE PRODUCER RICHARD DONNER
STORY BY JANICE FISCHER & JAMES JEREMIAS
SCREENPLAY BY JANICE FISCHER & JAMES JEREMIAS AND JEFFREY BOAM
PRODUCED BY HARVEY BERNHARD
DIRECTED BY JOEL SCHUMACHER
R
PANAVISION


STRICTLY
PRIVATE

```
public class RecentlyUsedList
{
    public boolean isEmpty()
    {
        return list.isEmpty();
    }
    public int size()
    {
        return list.size();
    }
    public void add(String newItem)
    {
        list.remove(newItem);
        list.add(0, newItem);
    }
    public void clear()
    {
        list.clear();
    }
    private List<String> list = new ArrayList<String>();
}
```

```
RecentlyUsedList list = new RecentlyUsedList();
list.add("Hello, World!");
assert list.size() == 1;
list.clear();
assert list.isEmpty();
```

```
public class RecentlyUsedList
{
    public void add(String newItem)
    {
        list.remove(newItem);
        list.add(0, newItem);
    }
    public String get(int index)
    {
        return list.get(index);
    }
    ...
}
```

```
public class RecentlyUsedList
{
    public void add(String newItem)
    {
        list.remove(newItem);
        list.add(newItem);
    }
    public String get(int index)
    {
        return list.get(size() – index – 1);
    }
    ...
}
```

```
public class RecentlyUsedList
{
    public void add(String newItem)
    {
        if(list == null)
            list = new ArrayList<String>();
        else
            list.remove(newItem);
        list.add(newItem);
    }
    public int size()
    {
        return list == null ? 0 : list.size();
    }
    public String get(int index)
    {
        if(list == null)
            throw new IndexOutOfBoundsException();
        return list.get(size() – index - 1);
    }
    private List<String> list;
}
```

Accidental complexity from unnecessary laziness

Refactoring (noun): ***a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.***

Refactor (verb): ***to restructure software by applying a series of refactorings without changing the observable behavior of the software.***

Martin Fowler, ***Refactoring***

```
public class RecentlyUsedList
{
    public RecentlyUsedList()
    {
        list = new ArrayList<String>();
    }
    public void add(String newItem)
    {
        if(list.contains(newItem))
        {
            int position;
            position = list.indexOf(newItem);
            string existingItem;
            existingItem = list.get(position);
            list.remove(position);
            list.add(0, existingItem);
        }
        else
        {
            list.add(0, newItem);
        }
    }
    public int size()
    {
        int size;
        size = list.size();
        return size;
    }
    public String get(int index)
    {
        int position;
        position = 0;
        for(String value : list)
        {
            if(position == index)
            {
                return value;
            }
            ++position;
        }
        throw new IndexOutOfBoundsException();
    }
    private List<String> list;
}
```

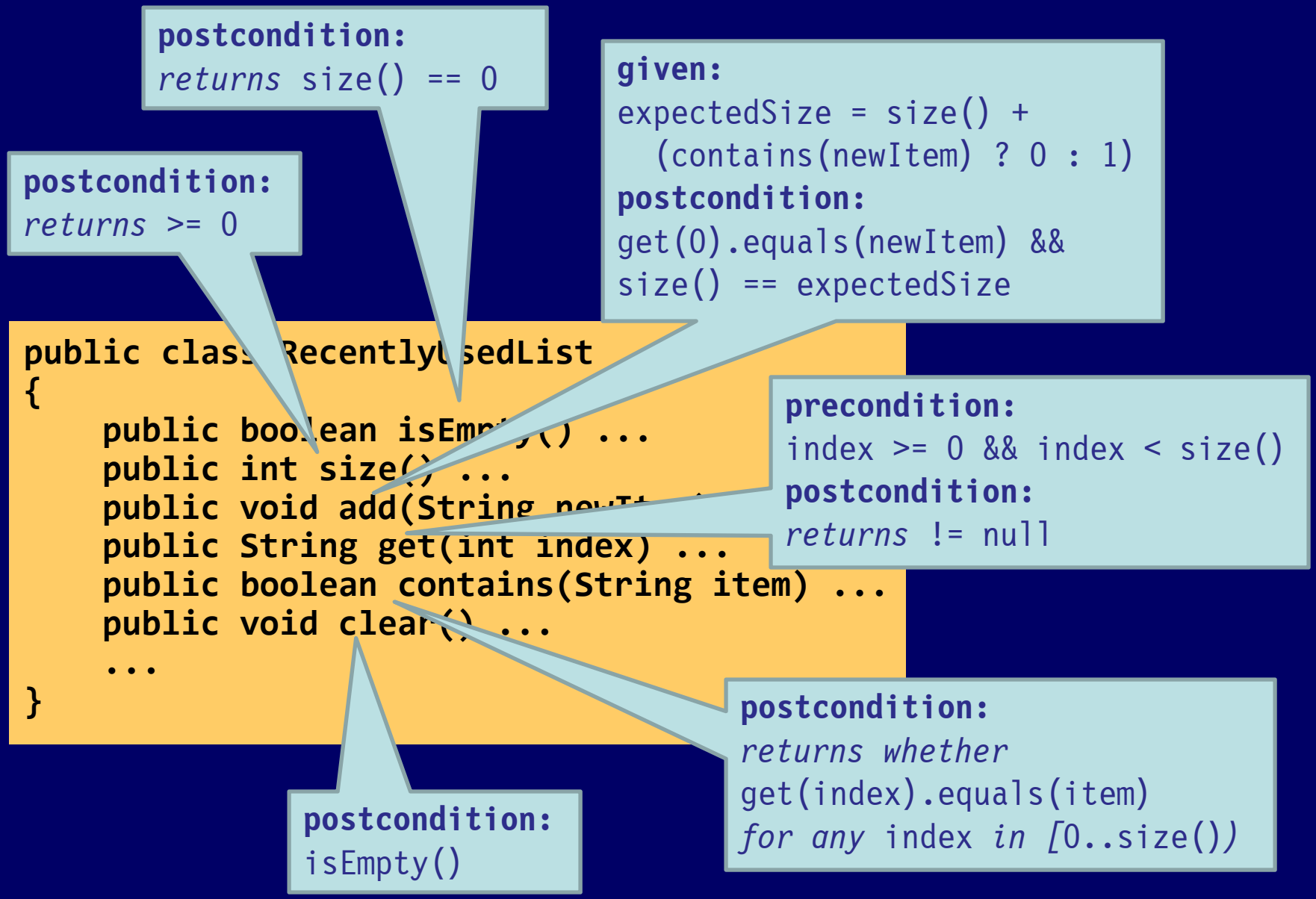
```
public class RecentlyUsedList
{
    public void add(String newItem)
    {
        list.remove(newItem);
        list.add(newItem);
    }
    public int size()
    {
        return list.size();
    }
    public String get(int index)
    {
        return list.get(size() - index – 1);
    }
    private List<String> list = new ArrayList<String>();
}
```

Construction Time Again

- A constructor is...
  - A special method for setting an object's fields to some initial values?
  - A special method for establishing a new object in a correct state?
  - A transactional method for establishing a new and valid state of the system?
- Try to avoid providing constructors that don't give useful, usable objects

The Contract Metaphor

- A contract defines a relationship by a set of expectations and constraints
  - A class can be seen in terms of a client–supplier relationship, with the client dependent on the public interface and the supplier offering the encapsulated implementation
- The contractual view reinforces the public–private separation of a class


postcondition:
returns size() == 0
postcondition:
returns >= 0
given:
expectedSize = size() +
(contains(newItem) ? 0 : 1)
postcondition:
get(0).equals(newItem) &&
size() == expectedSize
public class RecentlyUsedList
{
public boolean isEmpty() ...
public int size() ...
public void add(String newIt
public String get(int index) ...
public boolean contains(String item) ...
public void clear() ...
...
}
precondition:
index >= 0 && index < size()
postcondition:
returns != null
postcondition:
isEmpty()
postcondition:
returns whether
get(index).equals(item)
for any index in [0..size())

```
given:
expectedSize = size() + (contains(newItem) ? 0 : 1)
precondition:
newItem != null
postcondition:
get(0).equals(newItem) && size() == expectedSize
```

```
public class RecentlyUsedList
{
    public void add(String newItem) ...
    public String get(int index) ...
    ...
}
```

```
precondition:
index >= 0 && index < size()
postcondition:
returns != null
```

What would a class inheriting from *RecentlyUsedList* be permitted and disallowed from doing?

Other Contract Approaches

- Design by Contract can be useful, but has limitations
  - The specification of *Object.equals* is assertion-based but not method-centric
  - Operational complexity, re-entrancy, resource usage, etc., often need to be part of the contract
  - Tests can be used to define a contract

Everybody knows that TDD stands for Test Driven Development. However, people too often concentrate on the words "Test" and "Development" and don't consider what the word "Driven" really implies. For tests to drive development they must do more than just test that code performs its required functionality: they must clearly express that required functionality to the reader. That is, they must be clear specifications of the required functionality. Tests that are not written with their role as specifications in mind can be very confusing to read. The difficulty in understanding what they are testing can greatly reduce the velocity at which a codebase can be changed.

Nat Pryce and Steve Freeman
"Are Your Tests Really Driving Your Development?"

```
@Test
public void test()
{
    RecentlyUsedList list = new RecentlyUsedList();
    assertEquals(0, list.size());
    list.add("Aardvark");
    assertEquals(1, list.size());
    assertEquals("Aardvark", list.get(0));
    list.add("Zebra");
    list.add("Mongoose");
    assertEquals(3, list.size());
    assertEquals("Mongoose", list.get(0));
    assertEquals("Zebra", list.get(1));
    assertEquals("Aardvark", list.get(2));
    list.add("Aardvark");
    assertEquals(3, list.size());
    assertEquals("Aardvark", list.get(0));
    assertEquals("Mongoose", list.get(1));
    assertEquals("Zebra", list.get(2));
    bool thrown;
    try
    {
        list.get(3);
        thrown = false;
    }
    catch(IndexOutOfBoundsException caught)
    {
        thrown = true;
    }
    assertTrue(thrown);
}
```

```
@Test
public void test1()
{
    RecentlyUsedList list = new RecentlyUsedList();
    assertEquals(0, list.size());
    list.add("Aardvark");
    assertEquals(1, list.size());
    assertEquals("Aardvark", list.get(0));
    list.add("Zebra");
    list.add("Mongoose");
    assertEquals(3, list.size());
    assertEquals("Mongoose", list.get(0));
    assertEquals("Zebra", list.get(1));
    assertEquals("Aardvark", list.get(2));
}
@Test
public void test2()
{
    RecentlyUsedList list = new RecentlyUsedList();
    assertEquals(0, list.size());
    list.add("Aardvark");
    assertEquals(1, list.size());
    assertEquals("Aardvark", list.get(0));
    list.add("Zebra");
    list.add("Mongoose");
    assertEquals(3, list.size());
    assertEquals("Mongoose", list.get(0));
    assertEquals("Zebra", list.get(1));
    assertEquals("Aardvark", list.get(2));
    list.add("Aardvark");
    assertEquals(3, list.size());
    assertEquals("Aardvark", list.get(0));
    assertEquals("Mongoose", list.get(1));
    assertEquals("Zebra", list.get(2));
}
@Test
public void test3()
{
    RecentlyUsedList list = new RecentlyUsedList();
    assertEquals(0, list.size());
```

```
@Test
public void constructor()
{
    RecentlyUsedList list = new RecentlyUsedList();
    assertEquals(0, list.size());
}
@Test
public void add()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.add("Aardvark");
    assertEquals(1, list.size());
    list.add("Zebra");
    list.add("Mongoose");
    assertEquals(3, list.size());
    list.add("Aardvark");
    assertEquals(3, list.size());
}
@Test
public void get()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.add("Aardvark");
    list.add("Zebra");
    list.add("Mongoose");
    assertEquals("Mongoose", list.get(0));
    assertEquals("Zebra", list.get(1));
    assertEquals("Aardvark", list.get(2));
    list.add("Aardvark");
    assertEquals("Aardvark", list.get(0));
    assertEquals("Mongoose", list.get(1));
    assertEquals("Zebra", list.get(2));
    bool thrown;
    try
    {
        list.get(3);
        thrown = false;
    }
    catch(IndexOutOfBoundsException caught)
    {
        thrown = true;
    }
    asserTrue(thrown);
}
```

Constructor

Add

Get

```
@Test
public void initialListIsEmpty()
{
    RecentlyUsedList list = new RecentlyUsedList();

    assertEquals(0, list.size());
}
@Test
public void additionOfSingleItemToEmptyListIsRetained()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.add("Aardvark");

    assertEquals(1, list.size());
    assertEquals("Aardvark", list.get(0));
}
@Test
public void additionOfDistinctItemsIsRetainedInStackOrder()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.add("Aardvark");
    list.add("Zebra");
    list.add("Mongoose");

    assertEquals(3, list.size());
    assertEquals("Mongoose", list.get(0));
    assertEquals("Zebra", list.get(1));
    assertEquals("Aardvark", list.get(2));
}
@Test
public void duplicateItemsAreMovedToFrontButNotAdded()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.add("Aardvark");
    list.add("Mongoose");
    list.add("Aardvark");

    assertEquals(2, list.size());
    assertEquals("Aardvark", list.get(0));
    assertEquals("Mongoose", list.get(1));
}
@Test(expected=IndexOutOfBoundsException.class)
public void outOfRangeIndexThrowsException()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.add("Aardvark");
    list.add("Mongoose");
    list.add("Aardvark");
    list.get(3);
}
```

Initial list is empty

Addition of single item to empty list is retained

Addition of distinct items is retained in stack order

Duplicate items are moved to front but not added

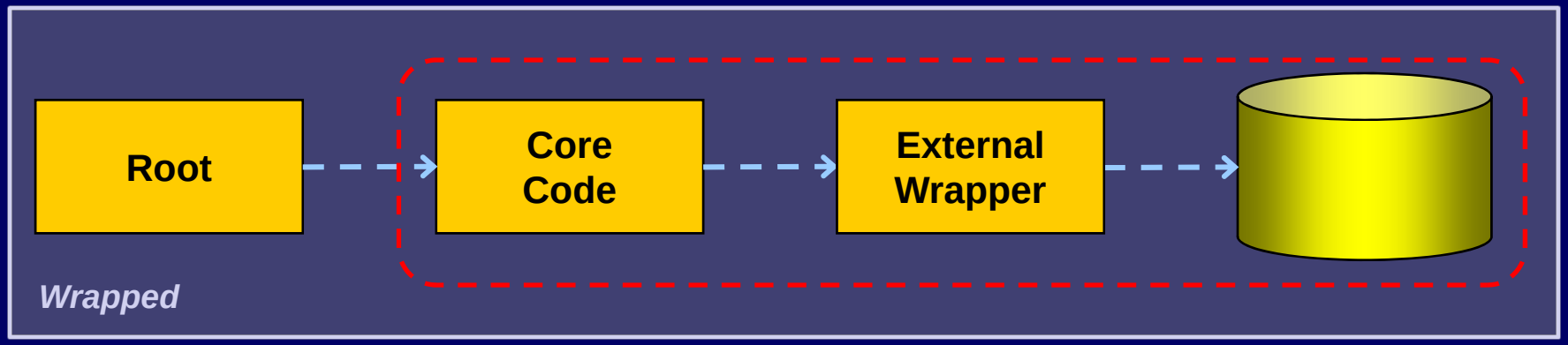
Out of range index throws exception

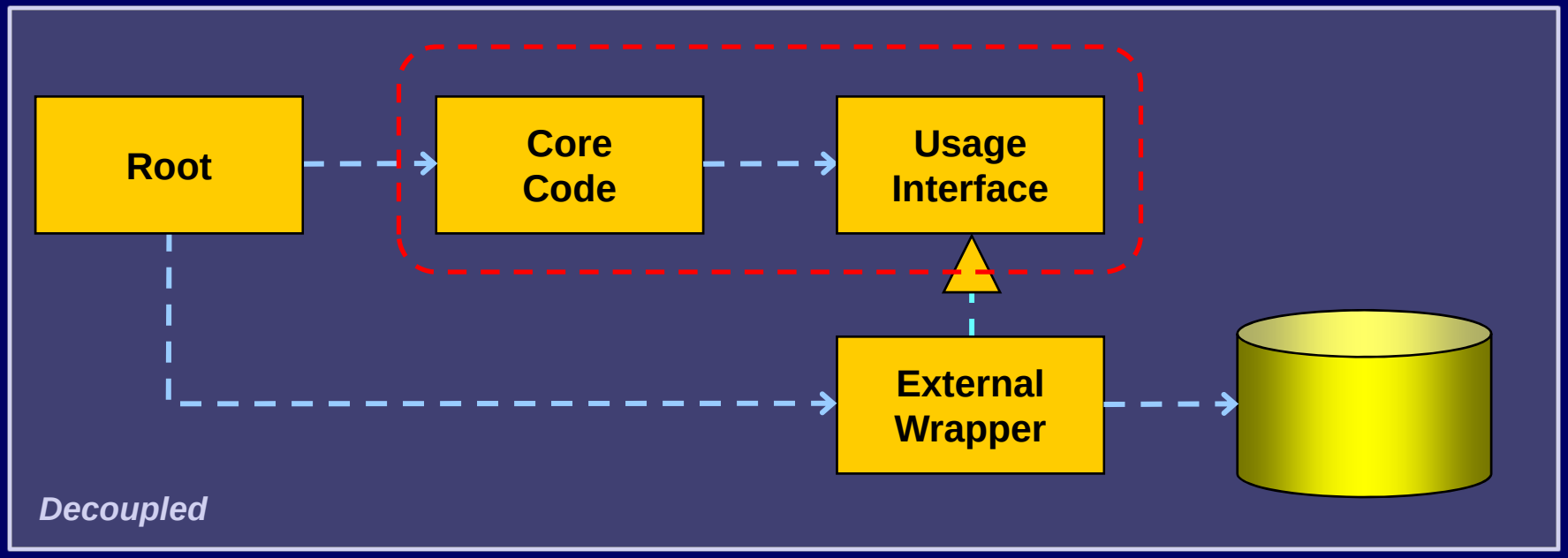
A test is not a unit test if:

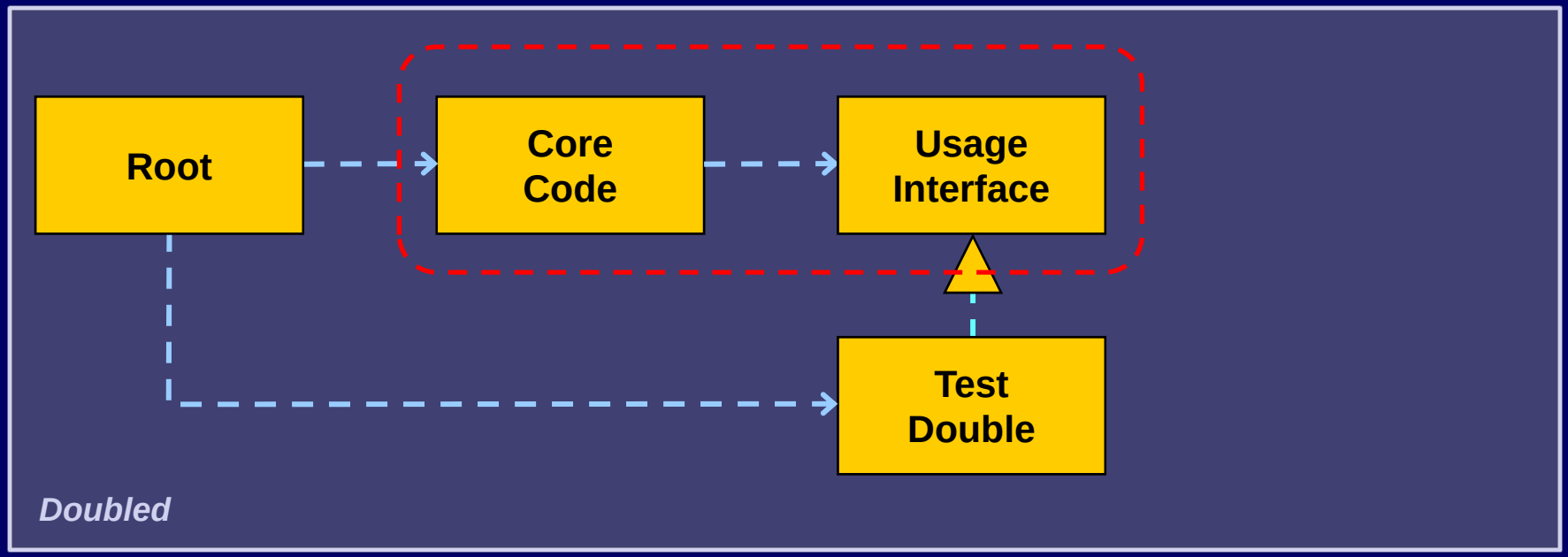
- **It talks to the database**
- **It communicates across the network**
- **It touches the file system**
- **It can't run at the same time as any of your other unit tests**
- **You have to do special things to your environment (such as editing config files) to run it.**

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"

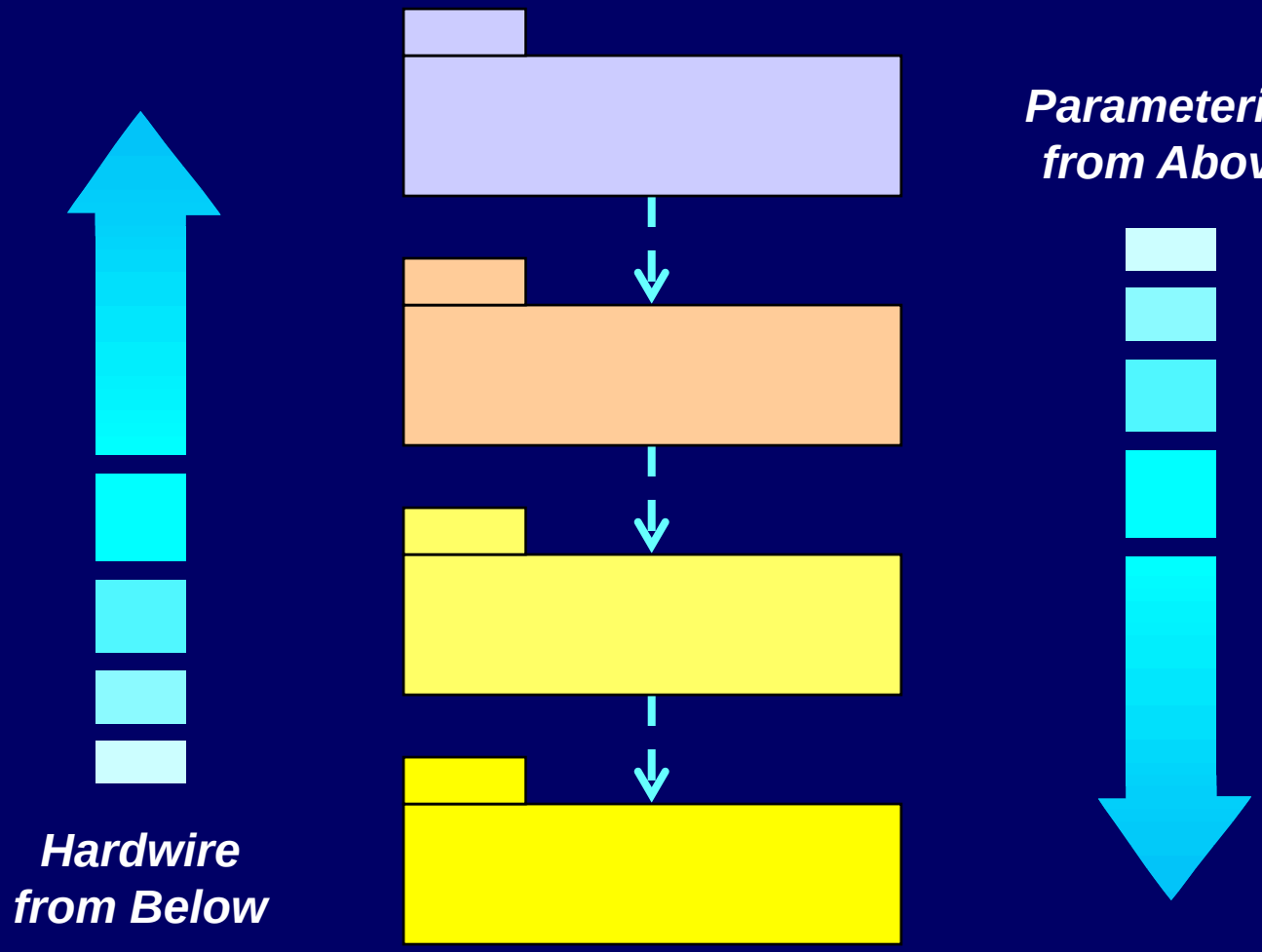

Root
Core
Code
External
Wrapper
Wrapped


Root
Core Code
Usage Interface
External Wrapper
Decoupled


Root
Core Code
Usage Interface
Test Double
Doubled


THE SINGLETON
Single Malt Scotch Whisky
of Dufftown
12
PERFECTLY BALANCED, NATURALLY RICH & SMOOTH


There can be only one.

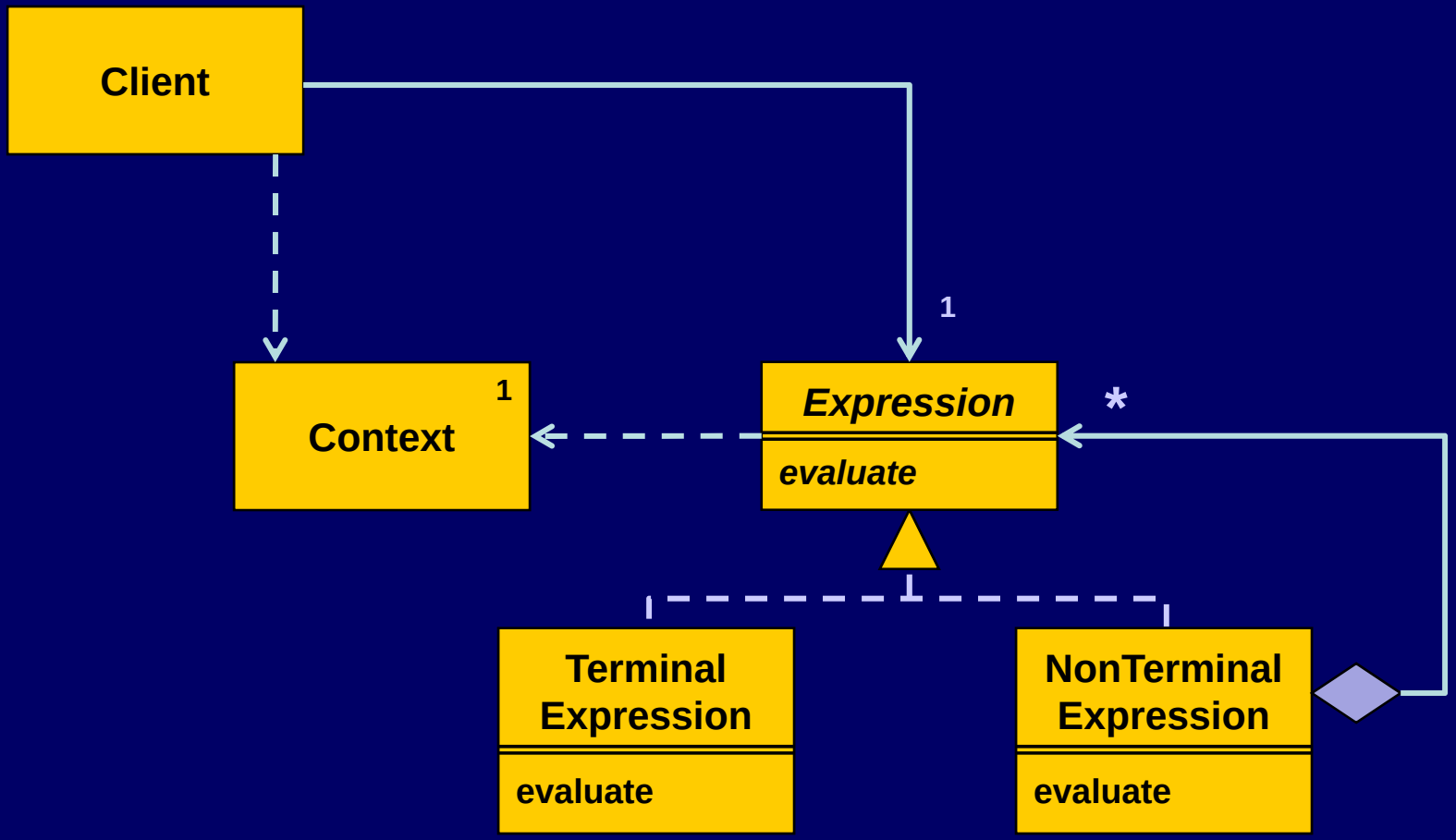

Parameterize
from Above
Hardwire
from Below

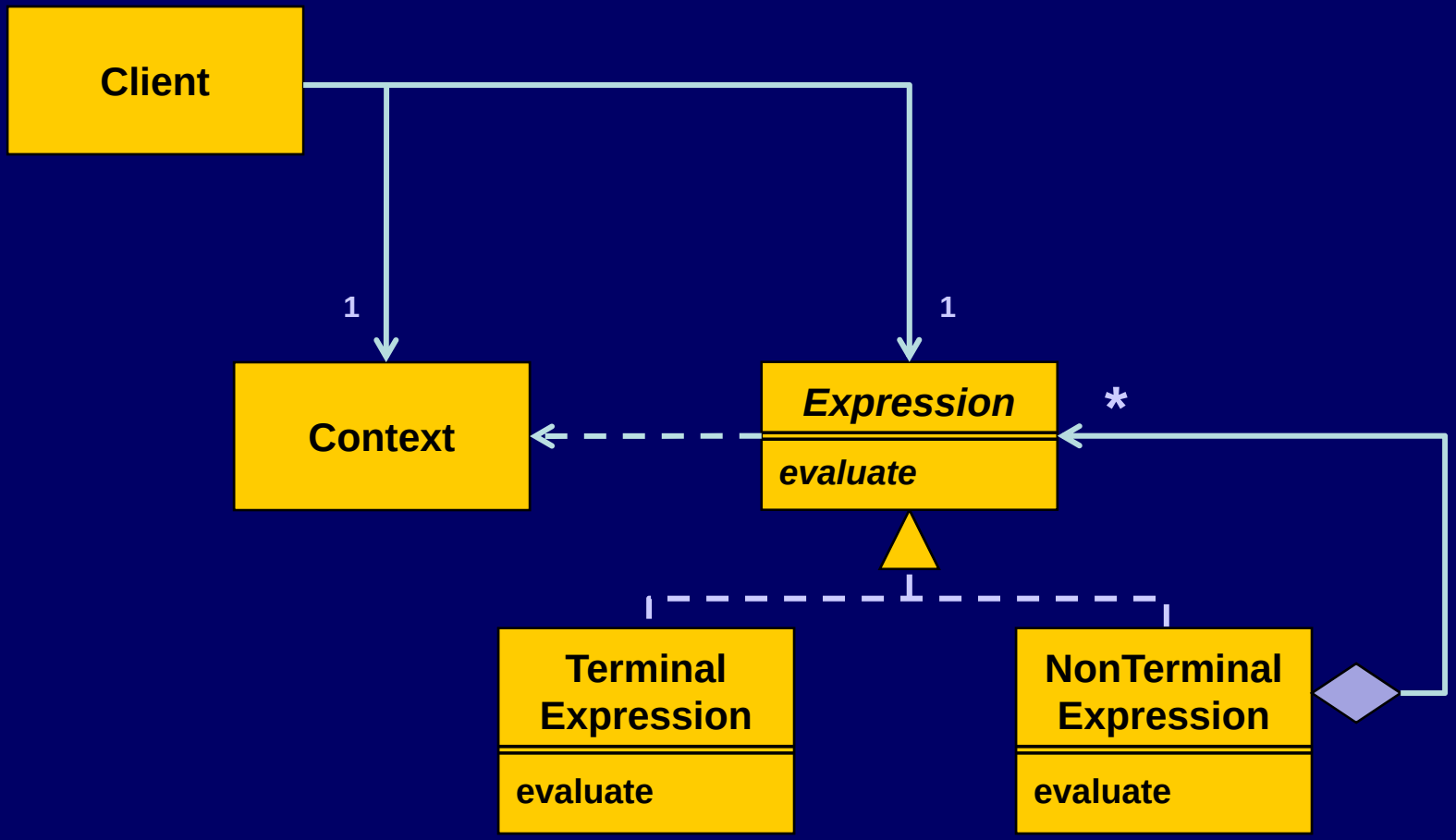
If you have a procedure with ten parameters, you probably missed some.

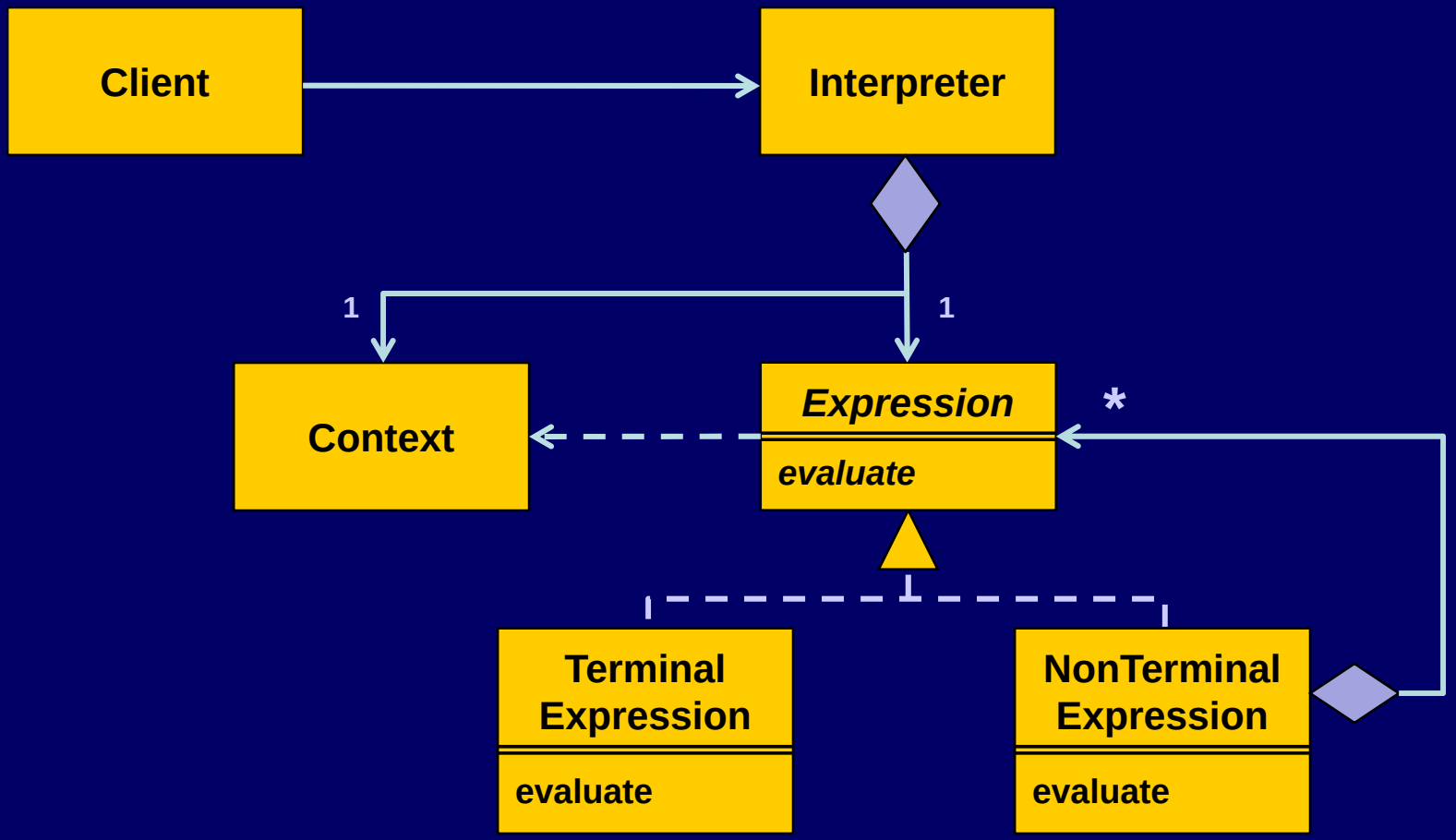
Alan Perlis

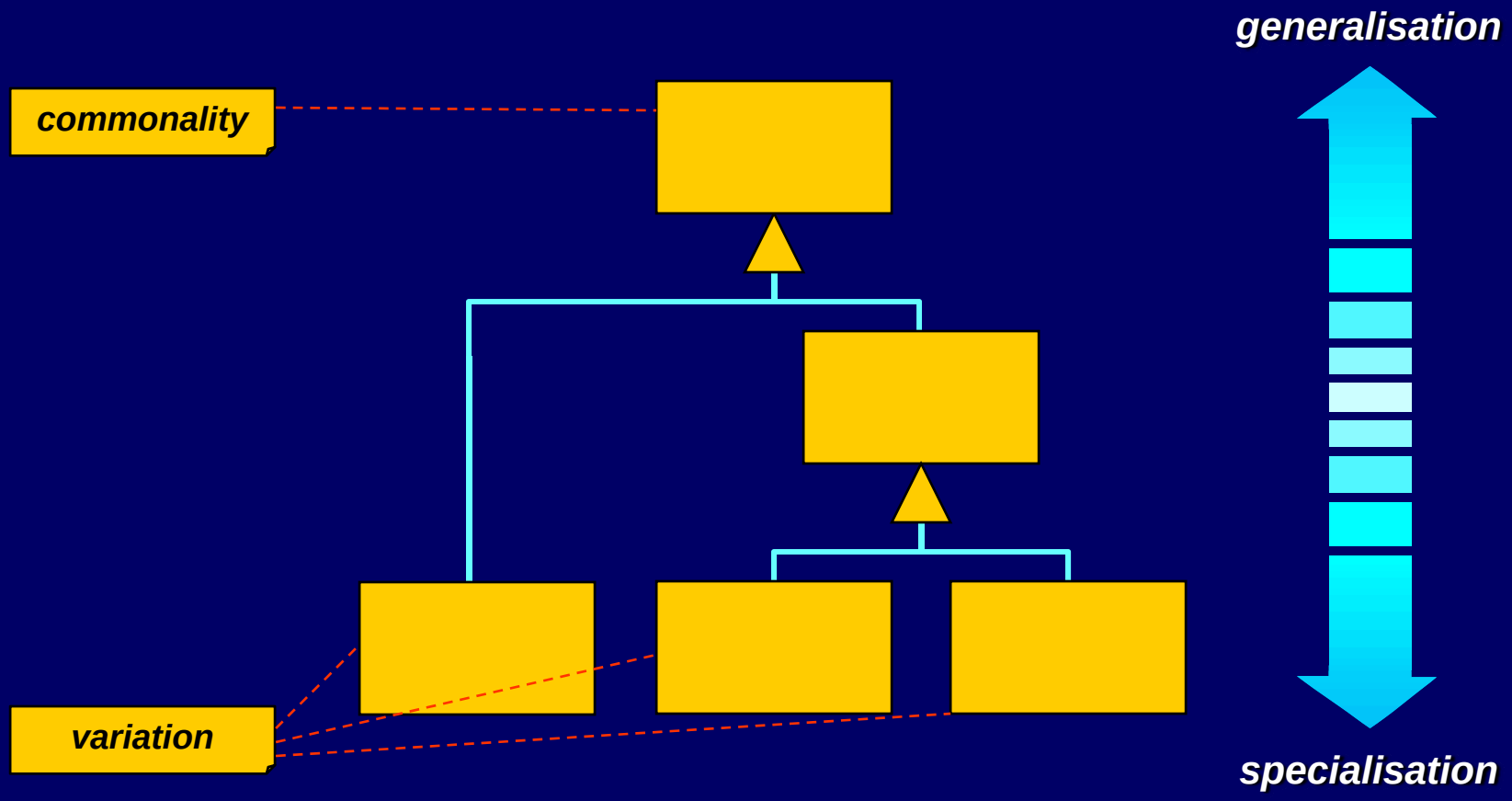
Parameter Objects

- Many kinds of parameter objects...
  - Context objects are for passing contextual information around
  - Data transfer objects (DTOs) are for batching property queries or changes
  - Value objects often start life as simple parameter objects
- Stability of the whole is greater than the stability of the individual parts


Client
1
Context
1
Expression
*
evaluate
Terminal
Expression
evaluate
NonTerminal
Expression
evaluate


Client
1
1
Context
Expression
evaluate
*
Terminal Expression
evaluate
NonTerminal Expression
evaluate


Client
Interpreter
1
1
Context
Expression
evaluate
*
Terminal Expression
evaluate
NonTerminal Expression
evaluate


commonality
variation
generalisation
specialisation

The process of moving from the specific to the general is both necessary and perilous. A doctor could, with some statistical support, generalize about men of a certain age and weight. But what if generalizing from other traits — such as high blood pressure, family history, and smoking — saved more lives? Behind each generalization is a choice of what factors to leave in and what factors to leave out, and those choices can prove surprisingly complicated.

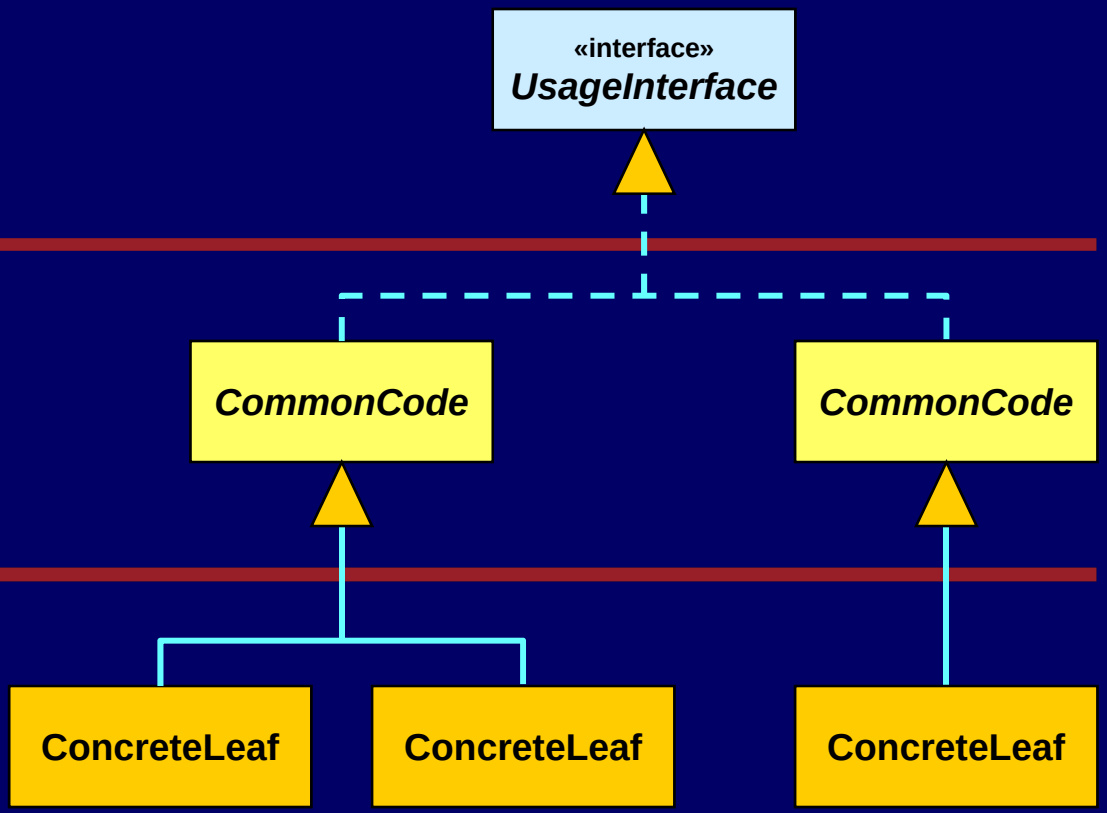
Malcolm Gladwell
"Troublemakers", *What the Dog Saw*

Sandwich Layering

Pure Interface Layer
Interfaces may extend interfaces, but there is no implementation defined in this layer.

Common Code Layer
Only abstract classes are defined in this layer, possibly with inheritance, factoring out any common implementation.

Concrete Class Layer
Only concrete classes are defined, and they do not inherit from one another.



Autognosis

An object can only access other objects through public interfaces

William Cook, "On Understanding Data Abstraction, Revisited"

It is possible to do Object-Oriented programming in Java

William Cook, "On Understanding Data Abstraction, Revisited"

Object-Oriented subset of Java: class name is only after “new”

William Cook, "On Understanding Data Abstraction, Revisited"

Liskov Substitution Principle

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.

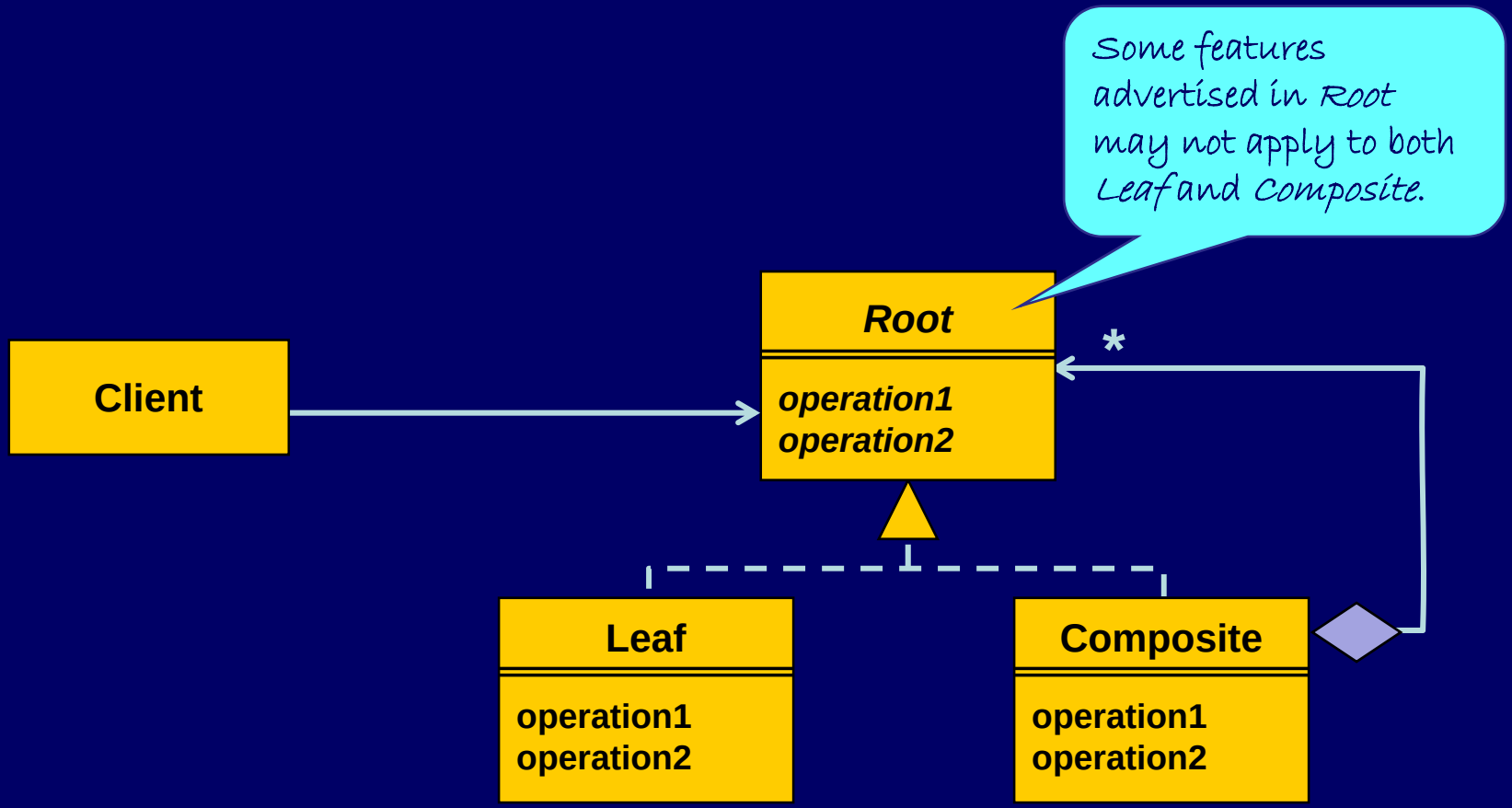
Barbara Liskov
"Data Abstraction and Hierarchy"

```
public class RecentlyUsedList extends ArrayList<String>
{
    @Override
    public void add(String newItem)
    {
        if(newItem == null)
            throw new IllegalArgumentException();
        remove(newItem);
        add(0, newItem);
    }
    ...
}
```

```
ArrayList<String> list = new RecentlyUsedList();
list.add("Hello, World!");
list.clear();
list.add("Hello, World!");
list.add("Goodbye, World!");
list.add("Hello, World!");
assert list.size() == 2;
list.add(1, "Hello, World!");
list.add(null); // throws
```

```
public class RecentlyUsedList
{
    public int size()
    {
        return list.size();
    }
    public void add(String newItem)
    {
        if(newItem == null)
            throw new IllegalArgumentException();
        list.remove(newItem);
        list.add(0, newItem);
    }
    public void clear()
    {
        list.clear();
    }
    ...
    private List<String> list = new ArrayList<string>();
}
```

Composite Compromises



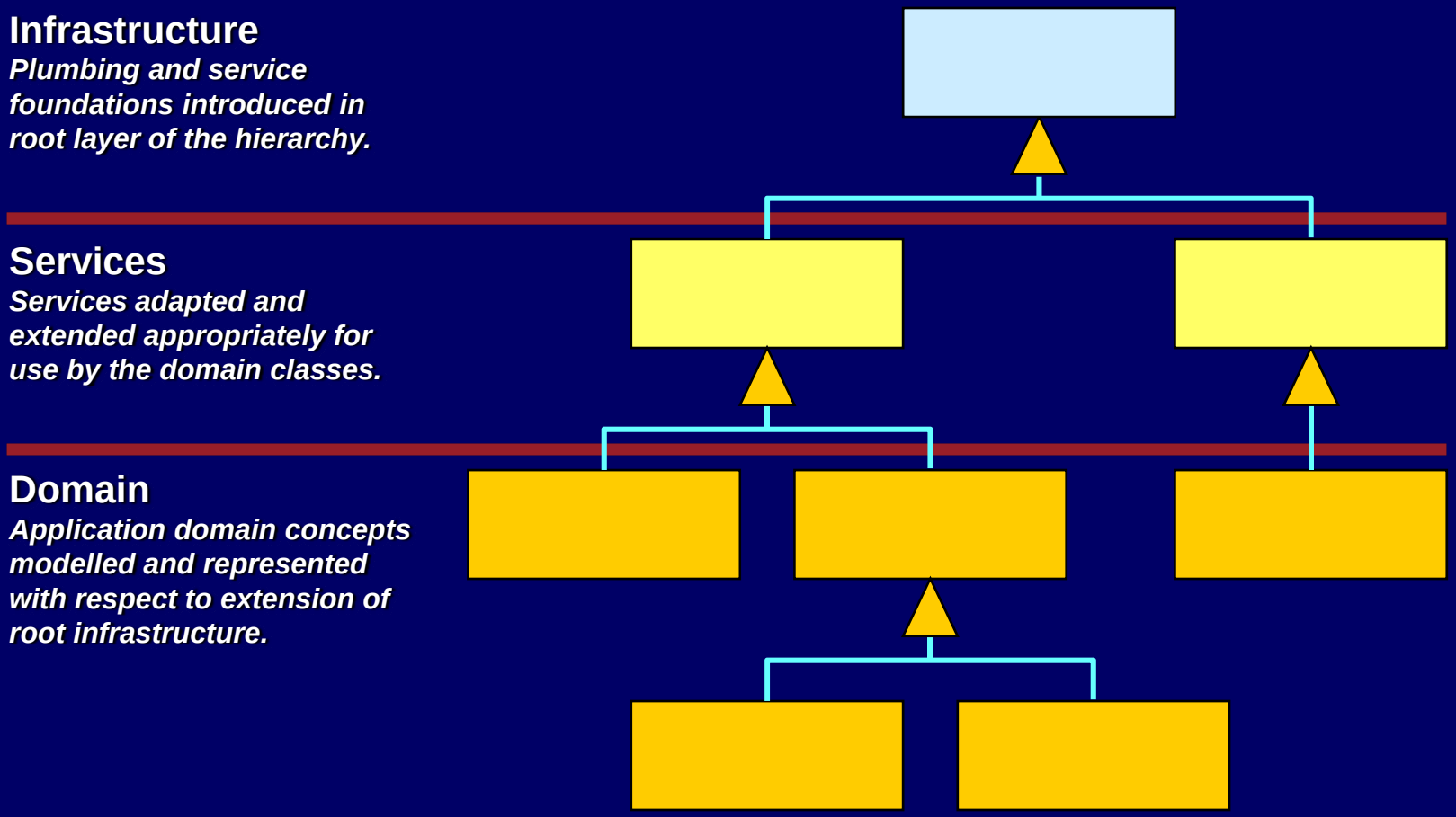

SEPARATION
JOHN BOWLBY
0 14 08 0307 6

Infrastructure + Services + Domain

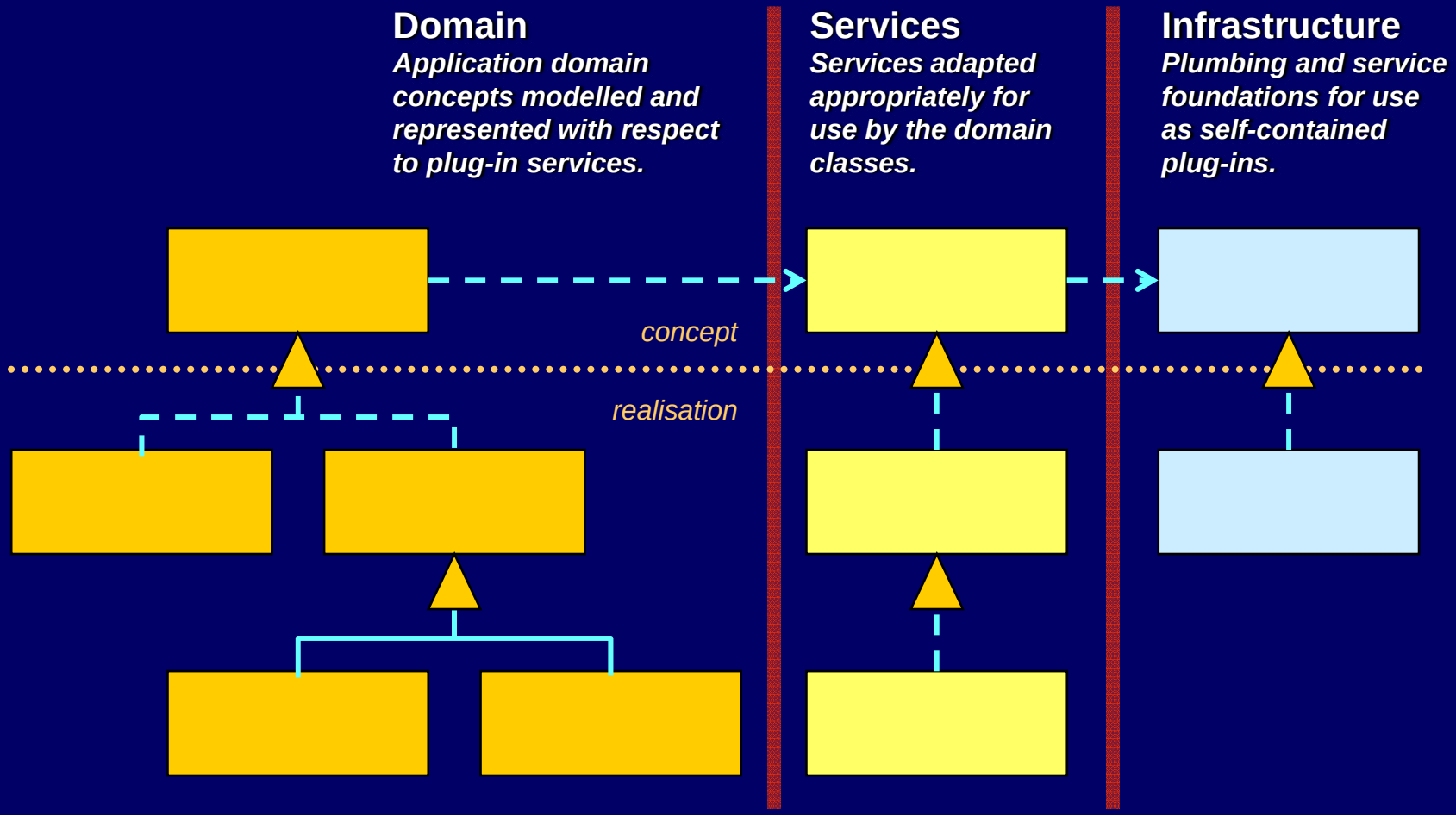
Infrastructure
Plumbing and service foundations introduced in root layer of the hierarchy.

Services
Services adapted and extended appropriately for use by the domain classes.

Domain
Application domain concepts modelled and represented with respect to extension of root infrastructure.

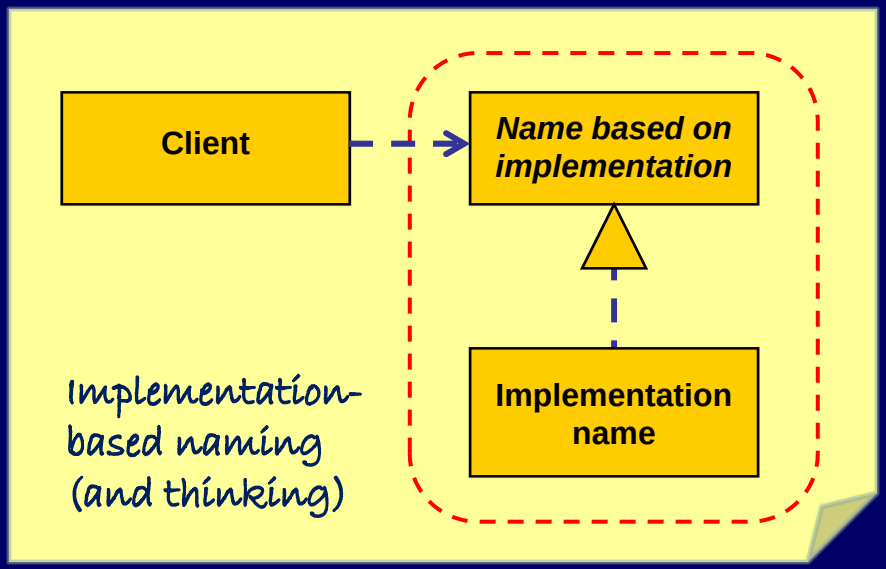


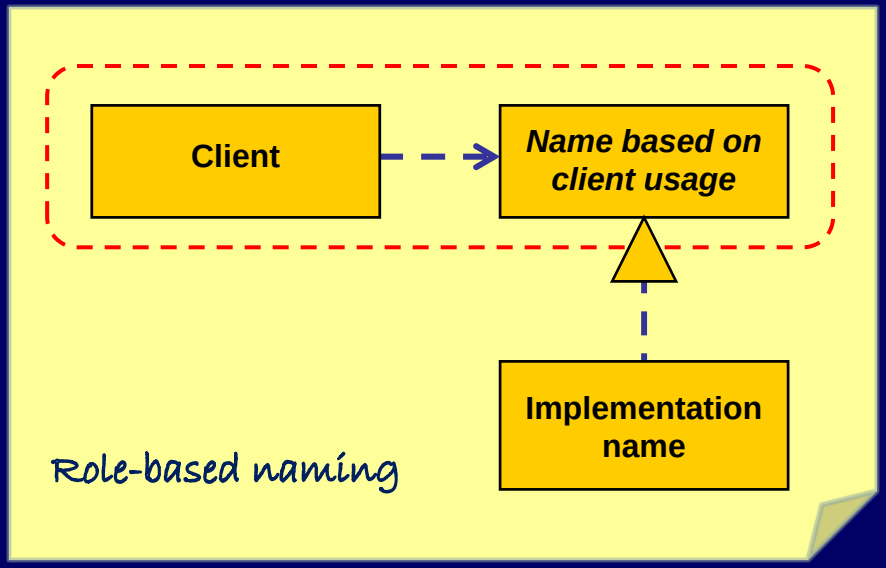
Domain × Services × Infrastructure



Selfish Objects

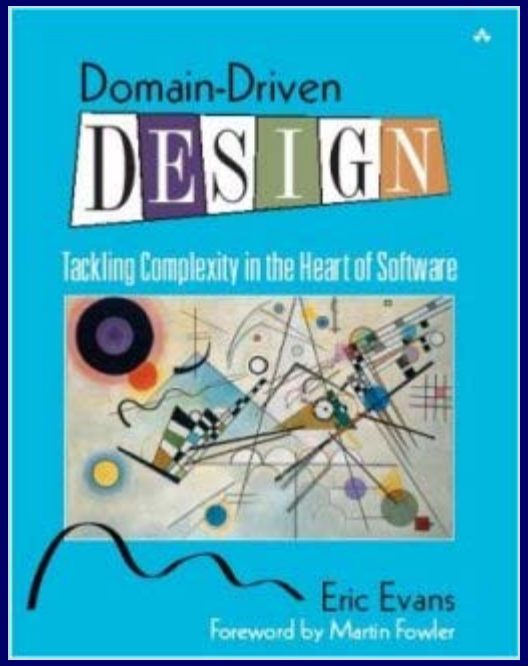
- Focusing on what objects want, not they can use or be given
  - Express external dependencies as specific and narrow plug-in interfaces
- This is in contrast to abstracting interfaces from implementations
  - Better than not abstracting interfaces at all, but often end up with a broad and unfocused façade


Client
Name based on implementation
Implementation name
Implementation-based naming (and thinking)


Client
Name based on client usage
Implementation name
Role-based naming

The Many Values of *Value*

- The term *value* is used in many overlapping and contradictory ways
  - The mechanism of *pass by value*
  - A declarative construct, e.g., *struct* in C# defines programmatic *value types*
  - A kind of object representing fine-grained information in a domain model
  - The general notion of quantity or measure of something in the real world

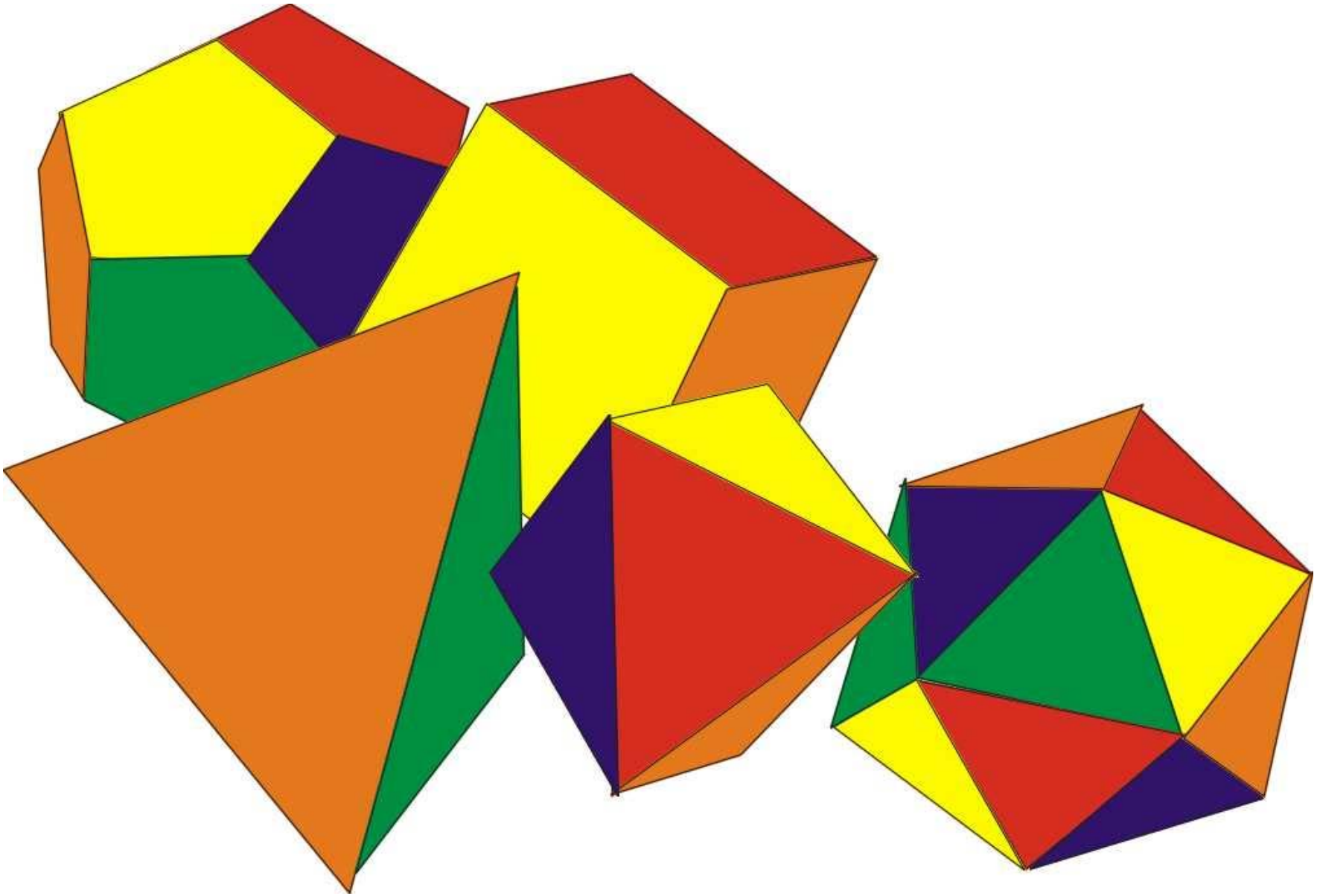


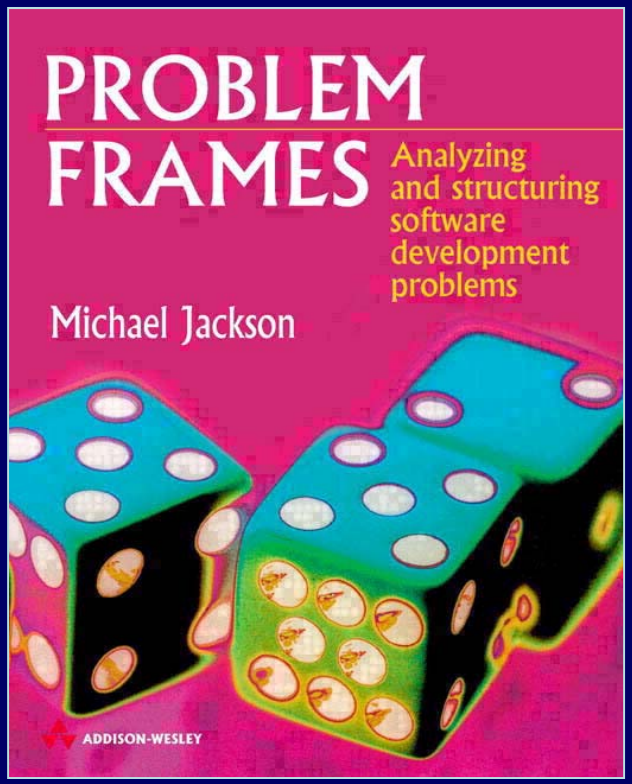
Many objects have no conceptual identity. These objects describe some characteristic of a thing. [...]

When you care only about the attributes of an element of the model, classify it as a VALUE OBJECT. Make it express the meaning of the attributes it conveys and give it related functionality. Treat the VALUE OBJECT as immutable. Don't give it any identity and avoid the design complexities necessary to maintain ENTITIES.

Complementary Perspectives

- The Platonic Perspective
  - An idealised view of what values are in terms of maths and the physical world
- The Computational Perspective
  - A model-based view of what values are in terms of programming concepts
- The Language Perspective
  - The computational view bound to the specifics of a programming language

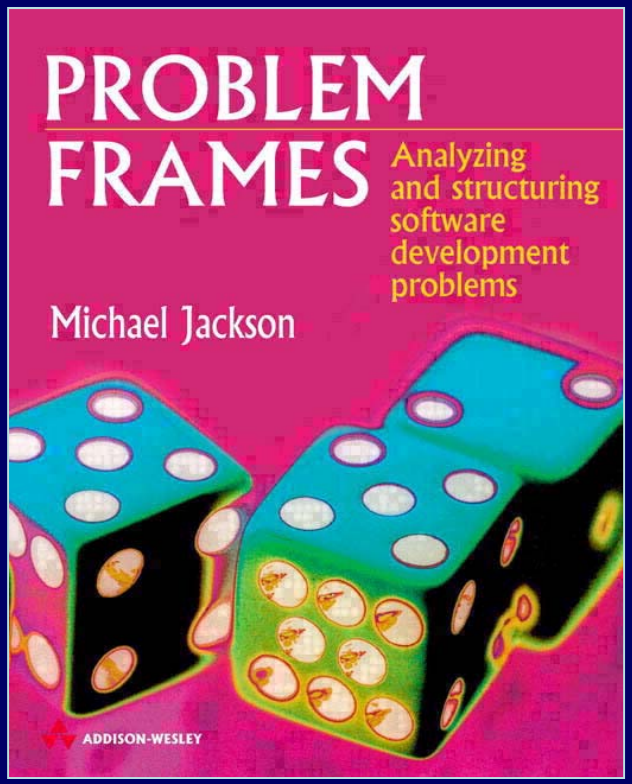




phenomenon (plural: phenomena): An element of what we can observe in the world. Phenomena may be *individuals* or *relations*. Individuals are *entities*, *events*, or *values*. Relations are *roles*, *states*, or *truths*.

individual: An individual is a *phenomenon* that can be named and is distinct from every other individual: for example, the number 17, George III, or Deep Blue's first move against Kasparov.

relationship: A kind of *phenomenon*. An association among two or more *individuals*, for example, *Mother(Lucy, Joe)*. Also, generally, any pattern or structure among phenomena of a *domain*.



Events. An event is an individual happening, taking place at some particular point in time. Each event is *indivisible* and *instantaneous*.

Entities. An entity is an individual that persists over time and can change its properties and states from one point in time to another.

Values. A value is an intangible individual that exists outside time and space, and is not subject to change.

States. A state is a relation among individual entities and values; it can change over time.

Truths. A truth is a relation among individuals that cannot possibly change over time.

Roles. A role is a relation between an event and individuals that participate in it in a particular way.

On the Origin of Species

- Value types differ in the generality and focus of their domain
  - Some are mathematical, e.g., integers
  - Some are programmatic, e.g., strings
  - Some are real world, e.g., ISBNs
- Value types reflect constraints
  - E.g., ISBNs have a well-formedness rule
  - E.g., *int* is a bounded subset of integers

Systems of Values

- **Operations, relationships and constraints form systems of values**
  - **E.g., a point in time is a value, as is the difference between two points in time, but *time point*, *time period* and *time interval* are not equivalent types**
  - **E.g., distance divided by time yields speed (and displacement divided by time yields velocity)**

povo, *sm*.

1. Conjunto de indivíduos que falam (em regra) a mesma lingua, têm costumes e hábitos idênticos, uma história e tradições communs.
2. Os habitantes duma localidade ou região; povoação.
3. V. *povoado*.
4. Multidão.
5. V. *plebe*.

Minidicionário da Língua Portuguesa

Whole Value

Besides using the handful of literal values offered by the language (numbers, strings, true and false) and an even smaller complement of objects normally used as values (date, time, point), you will make and use new objects with this pattern that represent the meaningful quantities of your business. These values will carry whole, useful chunks of information from the user interface to the domain model.

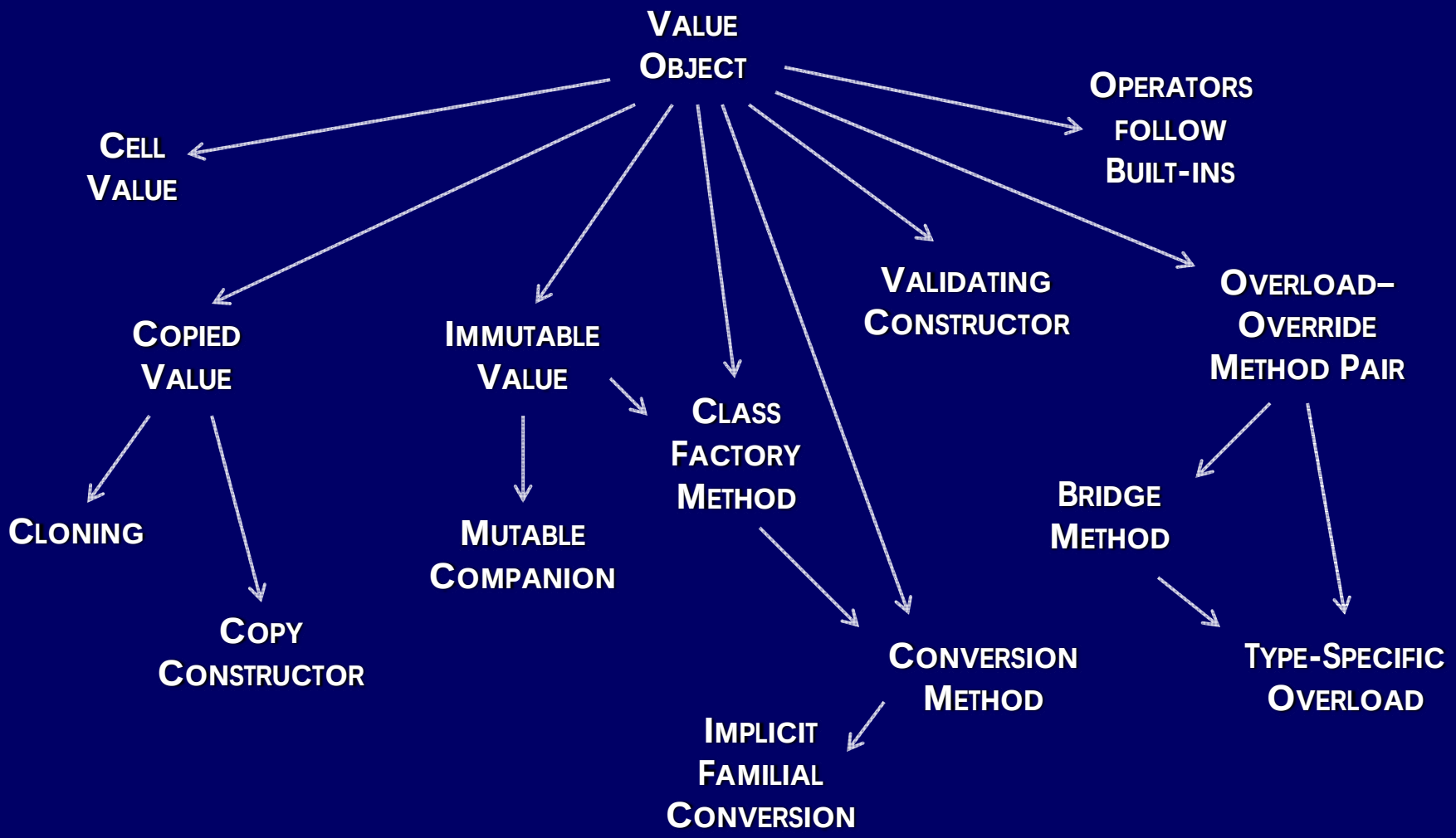
Construct specialized values to quantify your domain model and use these values as the arguments of their messages and as the units of input and output. Make sure these objects capture the whole quantity, with all its implications beyond merely magnitude; but keep them independent of any particular domain. (The word *value* here implies that these objects do not have identity of importance.)

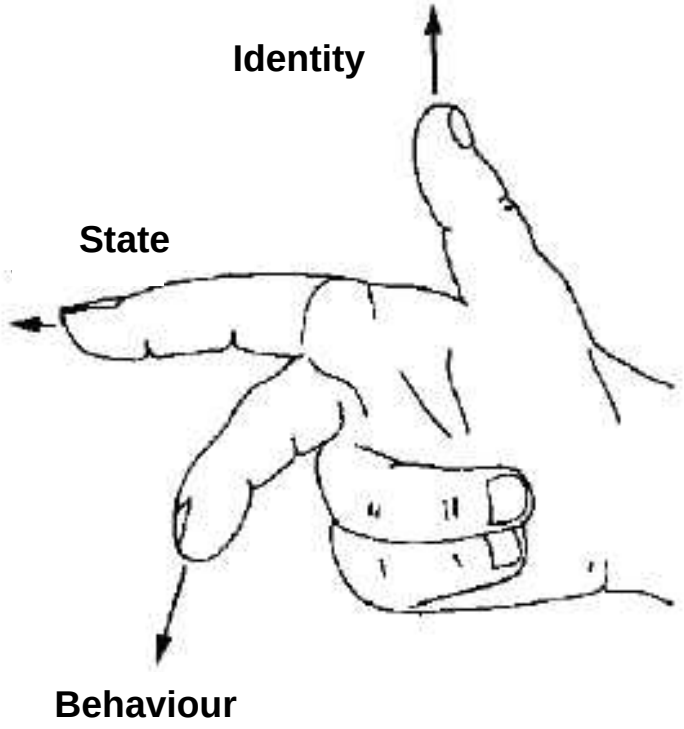
Ward Cunningham
"The CHECKS Pattern Language of Information Integrity"

Values as Objects

- From a programming perspective, we can model values as objects
  - Hence *value objects* and *value types*
  - Value objects have significant state but insignificant identity
- But there is no dichotomy or conflict between *values* and *objects*
  - A *value object* is a kind or style of *object* that realises a *value*

Patterns of Value



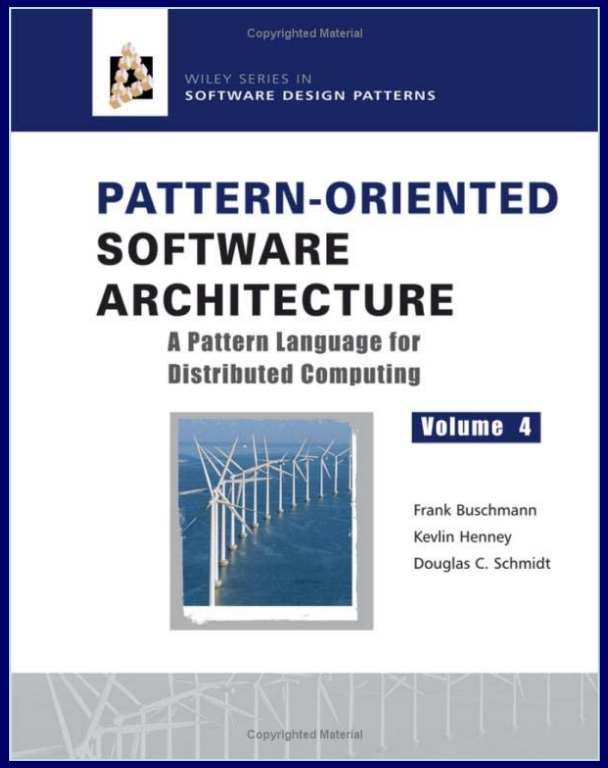

Identity
State
Behaviour


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

Referential transparency and **referential opaqueness** are properties of parts of computer programs. An expression is said to be referentially transparent if it can be replaced with its value without changing the program (in other words, yielding a program that has the same effects and output on the same input). The opposite term is referentially opaque.

While in mathematics all function applications are referentially transparent, in programming this is not always the case. The importance of referential transparency is that it allows a programmer (or compiler) to reason about program behavior. This can help in proving correctness, simplifying an algorithm, assisting in modifying code without breaking it, or optimizing code by means of memoization, common subexpression elimination or parallelization.

http://en.wikipedia.org/wiki/Referential_transparency_(computer_science)



Immutable Value

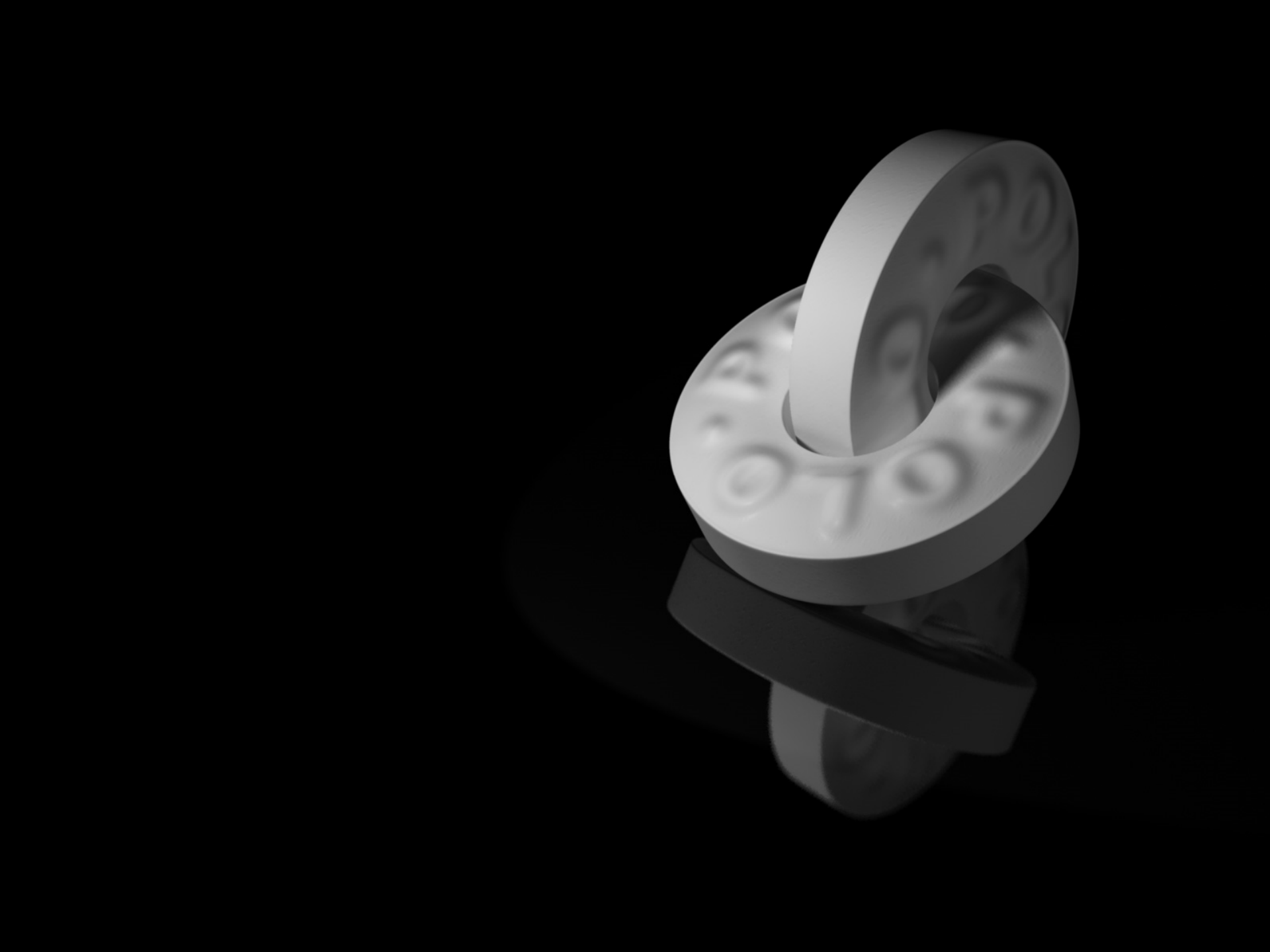
Define a value object type whose instances are immutable.

Copied Value

Define a value object type whose instances are copyable.

Value Object Smells

- Anaemia
  - Little behaviour beyond field access
  - Failure to enforce value invariants, i.e., constructors that allow construction of invalid values
- Multiple stereotypes
  - Strong service behaviour and external dependencies, e.g., ISBN class that checks for existence of a book



General POLO Anatomy

- Construction...
  - Constructor enforces validity
- Comparison...
  - Equality is a fundamental concept
  - Total ordering may or may not apply
- Classification and conversion...
  - Neither a subclass nor a superclass be
  - May support conversions

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getDayInMonth() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setDayInMonth(int newDayInMonth) ...
    ...
}
```

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setWeekInYear(int newWeek) ...
    public void setDayInYear(int newDayInYear) ...
    public void setDayInMonth(int newDayInMonth) ...
    public void setDayInWeek(int newDayInWeek) ...
    ...
}
```

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setWeekInYear(int newWeek) ...
    public void setDayInYear(int newDayInYear) ...
    public void setDayInMonth(int newDayInMonth) ...
    public void setDayInWeek(int newDayInWeek) ...
    ...
    private int year, month, dayInMonth;
}
```

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setWeekInYear(int newWeek) ...
    public void setDayInYear(int newDayInYear) ...
    public void setDayInMonth(int newDayInMonth) ...
    public void setDayInWeek(int newDayInWeek) ...
    ...
    private int daysSinceEpoch;
}
```

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

Lucius Cary

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    ...
}
```

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public Month getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public DayOfWeek getDayInWeek() ...
    ...
}
```

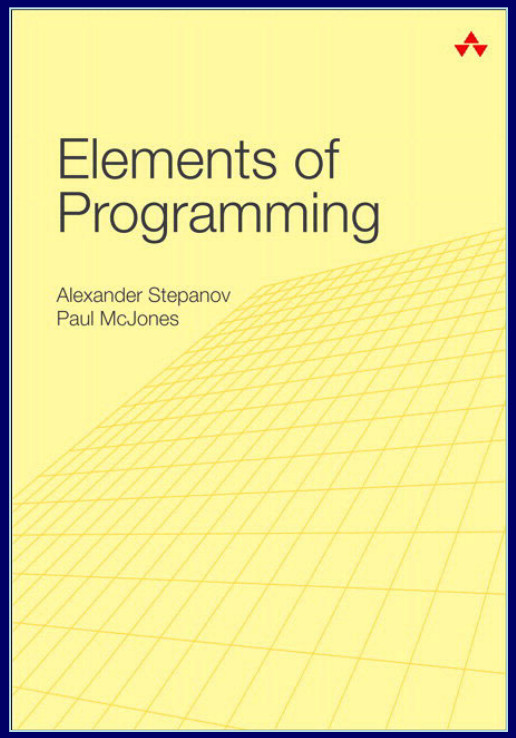
```
public final class Date implements ...
{
    ...
    public int year() ...
    public Month month() ...
    public int weekInYear() ...
    public int dayInYear() ...
    public int dayInMonth() ...
    public DayOfWeek dayInWeek() ...
    ...
}
```

Language Defines a Context

- Design is context sensitive
  - And design detail is, therefore, affected by choice of programming language
- Different languages encourage and enable different choices and styles
  - Culture and idioms, features for immutability, transparency of copying, presence of nulls, support for operator overloading, ease of equality, etc.

Two values of a value type are *equal* if and only if they represent the same abstract entity. They are *representationally equal* if and only if their datums are identical sequences of 0s and 1s.

If a value type is uniquely represented, equality implies representational equality.



Reflexivity: I am me.

Symmetry: If you're the same as me, I'm the same as you.

Transitivity: If I'm the same as you, and you're the same as them, then I'm the same as them too.

Consistency: If there's no change, everything's the same as it ever was.

Null inequality: I am not nothing.

No throw: If you call, I won't hang up.

Hash equality: If we're the same, we both share the same magic numbers.

equals

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.

```
@Test
public void identicallyConstructedValuesCompareEqual()
...
@Test
public void differentlyConstructedValuesCompareUnequal()
...
@Test
public void valuesCompareUnequalToNull()
...
@Test
public void identicallyConstructedValuesHaveEqualHashCodes()
...
```

```
@Test
public void identically_constructed_values_compare_equal()
...
@Test
public void differently_constructed_values_compare_unequal()
...
@Test
public void values_compare_unequal_to_null()
...
@Test
public void identically_constructed_values_have_equal_hash_codes()
...
```

We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.

David L Parnas

"On the Criteria to Be Used in Decomposing Systems into Modules"