# Five Considerations for Software Developers


Kevlin Henney

*kevlin@curbralan.com*


Presented at *Jfokus*, Stockholm, 30th January 2008.


Kevlin Henney

`kevlin@curbralan.com`
`kevlin@acm.org`

Curbralan Ltd

`http://www.curbralan.com`
*Voice*: +44 117 942 2990
*Fax*: +44 870 052 2289

## Agenda

- Introduction
- *Consideration 1*: Economy
- *Consideration 2*: Visibility
- *Consideration 3*: Spacing
- *Consideration 4*: Symmetry
- *Consideration 5*: Emergence
- Outroduction



What general qualities in a software architecture help to promote its success? We can of course focus on fitness for purpose, cost of change, organisational acceptance, and so on, but are there broad considerations that can be kept in mind when looking at the structural and developmental side of an architecture?

Those involved in software have a lot to keep in mind as they negotiate the worlds inside and outside of their code and the relationship between them. For those interested in improving the state of their art there are many (many) sources of specific recommendations they can use to sharpen their practice. This talk takes a step back from the busy, overpopulated and often overwhelming world of such recommendations to focus on five general considerations that can inform more detailed recommendations and specific decisions.

# Introduction

*Software engineering is the science and art of designing and making, with economy and elegance, applications, bridges, frameworks, and other similar structures so that they can safely resist the forces to which they may be subjected.*

**A Definition for Software Engineers**

## Of Beer and Code

- Questions and answers...
  - *Who?* Frank Buschmann, Charles Weir and me
  - *When?* OOPSLA, October, 2001
  - *Where?* Four Green Fields, Tampa, Florida
  - *How?* Because we wanted to avoid downtown, and Alan O'Callaghan suggested this place
  - *What?* Guinness! And, uh, a discussion on elegance and style in code, which led to five considerations
  - *Why?* Because Charles asked an NP-hard question and it seemed like a fun idea to find some answers



The five considerations under discussion stem from a discussion over beer about qualities of code and coding that could easily be taught to programmers.

## Considering Considerations

- A consideration is not a rule
  - And it is also weaker than the conventional notion of a recommendation
  - It is... a consideration
- A consideration takes a point of view
  - It may be general; it may be specific
- A system of considerations can offer a coherent and unified set of views
  - Together they can guide specific recommendations



Of course, it is important to distinguish between these qualities as hard and fast rules — which they are not — and their role as considerations in a balanced system of interacting considerations — which they are. There is no claim that these are universal properties, just that together they form a useful framework for discussing design, a vehicle for relaying concepts.

# *Consideration 1*: Economy

*Continuing existence or cessation of existence: those are the scenarios. Is it more empowering mentally to work towards an accommodation of the downsizings and negative outcomes of adversarial circumstance, or would it be a greater enhancement of the bottom line to move forwards to a challenge to our current difficulties, and, by making a commitment to opposition, to effect their demise?*

**Tom Burton, *Long Words Bother Me***

*To be, or not to be: that is the question:*
*Whether 'tis nobler in the mind to suffer*
*The slings and arrows of outrageous fortune,*
*Or to take arms against a sea of troubles,*
*And by opposing end them?*

**William Shakespeare, *Hamlet***

# Maximalism

```
interface Iterator
{
    boolean set_to_first_element();
    boolean set_to_next_element();
    boolean set_to_next_nth_element(in unsigned long n) raises(…);
    boolean retrieve_element(out any element) raises(…);
    boolean retrieve_element_set_to_next(out any element, out boolean more) raises(…);
    boolean retrieve_next_n_elements(in unsigned long n, out AnySequence result, out boolean more) raises(…);
    boolean not_equal_retrieve_element_set_to_next(in Iterator test, out any element) raises(…);
    void remove_element() raises(…);
    boolean remove_element_set_to_next() raises(…);
    boolean remove_next_n_elements(in unsigned long n, out unsigned long actual_number) raises(…);
    boolean not_equal_remove_element_set_to_next(in Iterator test) raises(…);
    void replace_element(in any element) raises(…);
    boolean replace_element_set_to_next(in any element) raises(…);
    boolean replace_next_n_elements(in AnySequence elements, out unsigned long actual_number) raises(…);
    boolean not_equal_replace_element_set_to_next(in Iterator test, in any element) raises(…);
    boolean add_element_set_iterator(in any element) raises(…);
    boolean add_n_elements_set_iterator(in AnySequence elements, out unsigned long actual_number) raises(…);
    void invalidate();
    boolean is_valid();
    boolean is_in_between();
    boolean is_for(in Collection collector);
    boolean is_const();
    boolean is_equal(in Iterator test) raises(…);
    Iterator clone();
    void assign(in Iterator from_where) raises(…);
    void destroy();
};
```



This supposedly general-purpose and "reusable" interface is drawn from the specifically unused CORBA Collections Service. Leaving aside the issue of whether it makes sense to define a general-purpose traversal interface for arbitrary collection objects on a network — it doesn't — this interface makes an impressive meal of the apparently simple task of iterating over a collection.

In Java and C# the methods on iterator interfaces can be counted on the fingers of one hand. In C++ the diversity of iterator capabilities gives rise to a whole style of programming that is based on efficiency and directness of expression. Neither of these observations can be considered true of the interface shown above, which uses vagueness and indecision to justify an uncommitted, complex and singularly inappropriate general-purpose interface for remote iteration. It is perhaps worth noting that other CORBA services define iterators that are significantly tighter and more sensible than — and significantly unlike — the example above.

## Minimalism

- Clarity is often achieved by reducing clutter
  - But don't encode the code
- Compression can come from careful abstraction
  - Compression relates to directness of expression
  - Abstraction concerns the removal of certain detail
- Abstraction is a matter of choice and quality of abstraction relates to compression and clarity
  - Of itself, abstraction is neither "good" nor "bad"
  - Encapsulation is a vehicle for abstraction



Abstraction is an essential skill in working with complex concepts: information overload and distraction by unrelated items can be reduced by filtering out inappropriate details and focusing on some essential core aspects. The choice of details determines the quality of an abstraction: a poor abstraction leaves too much in or removes the wrong details.

Compression also arises from well considered composition, which refocuses on the elements of composition, which in turn are the products of a particular choice of abstraction. Compression and abstraction are sometimes seen as opposites, but in truth they are complements that balance a design through necessary tension: sometimes in conflict; sometimes in reinforcement.

Abstraction is also balanced by its natural counterpart, concretion. In searching for appropriate models, some of the detail removed by the process of abstraction needs to be reintroduced at an appropriate level or juncture.

## JUTLAND: A Minimal Experiment

- Java Unit Testing: Light, Adaptable 'N' Discreet
  - A simple experiment and demonstration of design principles, as well as a teaching aid
- Smaller than classic JUnit, but less intrusive, more extensible and more idiomatic
  - Smaller in terms of concepts, lines of code, classes
  - Uses a plug-in micro-architecture based on one of three kinds of interface: declared and static (listener and policy interfaces), exception-based (failure detection) or introspected (test execution)



The JUTLAND framework is a simple unit-testing framework designed to explore what a minimal and idiomatically designed unit-testing framework for Java could look like.

Although relatively small and loosely coupled, JUnit has evolved to become larger and more tightly coupled than is strictly necessary for the task it undertakes. Classic JUnit also does not make the best use of Java language features in its design. For example, common fixture initialisation is via an overridden method, `setUp`, rather than taking advantage of constructors, the natural vehicle for object initialisation. Similarly, the intrusion of inheritance is also used for tearing down a test fixture via `tearDown`, rather than using a reflected method.

Although it is based on a quite different approach, more like TestNG or NUnit, JUnit 4 still carries the baggage of its predecessor versions, and certain decisions are not obviously improvements.

# JUTLAND's Nanokernel

```java
package jutland.kernel;
import java.lang.reflect.Method;
public class Tester implements Runnable
{
    public Tester(TestListener listener, TestPolicy policy, Class... testClasses)
    {
        this.listener    = listener;
        this.testClasses = testClasses;
        this.policy      = policy;
    }
    public void run()
    {
        for(Class<?> testClass : testClasses)
        {
            if(testClass != null)
            {
                try
                {
                    listener.startTestClass(testClass);
                    runTestClass(testClass);
                }
                catch(RuntimeException caught)
                {
                    listener.testFault(caught);
                }
                finally
                {
                    listener.endTestClass(testClass);
                }
            }
        }
    }
    private void runTestClass(Class<?> testClass)
    {
        for(Method testMethod : testClass.getMethods())
        {
            if(policy.isTestMethod(testMethod))
            {
                try
                {
                    listener.startTestMethod(testMethod);
                    runTestMethod(testClass, testMethod);
                }
                finally
                {
                    listener.endTestMethod(testMethod);
                }
            }
        }
    }
    ...
```

```java
    ...
    private void runTestMethod(Class<?> testClass, Method testMethod)
    {
        Class expectedException = policy.expectedException(testMethod);
        Object testObject = null;
        try
        {
            testObject = testClass.newInstance();
            testMethod.invoke(testObject, (Object[]) null);
            if(expectedException == null)
                listener.testSuccess();
            else
                listener.testFailure(null);
        }
        catch(Throwable caught)
        {
            Throwable cause = caught.getCause();
            if(cause == null)
                listener.testFault(caught);
            else if(expectedException != null && expectedException.isInstance(cause))
                listener.testSuccess();
            else
                listener.testFailure(cause);
        }
        dispose(testClass, testObject);
    }
    private void dispose(Class<?> testClass, Object testObject)
    {
        if(testObject != null)
        {
            try
            {
                Method disposer = testClass.getMethod("dispose", (Class[]) null);
                disposer.invoke(testObject, (Object[]) null);
            }
            catch(NoSuchMethodException caught)
            {
            }
            catch(Throwable caught)
            {
                listener.testFault(caught);
            }
        }
    }
    private TestListener listener;
    private TestPolicy policy;
    private Class[] testClasses;
}
```



JUTLAND is based on a simple kernel of code responsible for executing a number of test classes. Explicit policies, reflection and exceptions are used to define the plug-in interfaces. Only the executable code in the kernel is shown in the fragment above; the explicit interfaces of the `TestListener` and `TestPolicy` can be deduced from the usage.

Note that the symmetric arrangement of elements within `try finally` blocks is intentional not accidental: in many cases this symmetry would be mistaken. In this case it ensures that an attempt is always to report an episode, regardless of the quality of implementation of the listener involved.

# *grep*

```
int grep(char *regexp, FILE *f, char *name)
{
    int n, nmatch;
    char buf[BUFSIZ];

    nmatch = 0;
    while (fgets(buf, sizeof buf, f) != NULL) {
        n = strlen(buf);
        if (n > 0 && buf[n-1] == '\n')
            buf[n-1] = '\0';
        if (match(regexp, buf)) {
            nmatch++;
            if (name != NULL)
                printf("%s:", name);
            printf("%s\n", buf);
        }
    }
    return nmatch;
}
```

```
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}

int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}

int matchstar(int c, char *regexp, char *text)
{
    do {
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}
```



This simple and simplified version of `grep` is taken from Kernighan and Pike's *The Practice of Programming*.

## *eval*

```
(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))))
```



This simple and simplified version of Scheme's `eval` is often used to highlight the elegance of meta-circular approaches to language interpretation.

## Decremental Development

- Don't include the unused or repeat yourself like a broken record (or CD... whatever)
  - *Eliminate Waste* (Lean Development), *Don't Repeat Yourself* (Pragmatic Programming), *Once And Once Only* (Extreme Programming), *Omit Needless Code* and *Unify Duplicate Code* (Programmer's Dozen)
- Refactoring, encapsulation and libraries are tools that help in the search for the minimum
  - Hill-descending techniques
  - Refactoring is gradual, stable and goal oriented

13

To quote Dennis Ritchie and Ken Thompson:

> There have always been fairly severe size constraints on the Unix operating system and its software. Given the partially antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy but a certain elegance of design.

This perspective applies during the envisioning of a design, its realisation and its reconsideration. The more code there is, the harder it is to reason about. This suggests that the code of a system that goes into production should be essential and focused on clear communication — a view that is undermined by the view that code is a disposable instruction to the machine rather than the machine a disposable means to execute code.

# *Consideration* 2: Visibility

*Design involves assumptions about the future of the object designed, and the more the future resembles the past the more accurate the assumptions are likely to be. But designed objects themselves change the future into which they will age.*

**Henry Petroski, *To Engineer is Human***

## The Nature of Software

- Software exists in an abstract space
  - Absence of physicality makes its development hard to estimate accurately from our physical perspective
  - Any use of physical space for exposition or representation is by necessity metaphorical
- Practices need to emphasise making the invisible more visible
  - E.g. the role of a shared and recognised architecture
  - E.g. the use of patterns in design
  - E.g. communicating intent in code



Although certainly structured, the space in which software is conceived is not a physical one. This means that it is as neutral and open to many forms of visualisation as it is susceptible to false metaphors and invisibility. For example, when we draw parallels between software concepts and the worlds of physics or building architecture, the mapping between ideas is metaphorical rather than direct, which means that in some cases ideas may correspond simply but falsely.

The absence of physicality and consequent loss of visibility also presents problems when it comes to estimation of scope and scale of tasks. Humans are geared up to perceive the world physically, so the loss of physical dimension has us reaching for physically related metaphors to navigate and estimate the world of software.

In this light, we can see that increasing visibility of the invisible or implicit is a significant consideration in development. Significant decisions and differences need to be emphasised and made more rather than less obvious. The very notion of following idioms to highlight familiar patterns of usage and structure, whether actual design patterns or matters of convention such as naming, is in this vein. It is also worth noting that many uses of the term *idiom* are not actually idiomatic, in the sense that they are not the practice common to a significant grouping of people or that they are actually foreign rather than idiomatic usage, i.e. borrowed in from other languages. Many "idioms" are just techniques whose authors aspire for them to be idiomatic.

## Visibility in Development Processes

- Open source and agile development place value on visible steps and the use of feedback
  - Architectural progress is visible and empirical
- Incremental development is based on stepped progression and accumulation
  - Don't just iterate, come up with something!
- Scenario-driven development focuses on usable functionality as the measure of an increment
  - Don't just produce stuff, produce usable stuff!

16

The empirical nature of software development is in contrast to the defined nature of many processes that have been (and still are) often applied to software development. The contrast is one that inevitably creates mismatch and tension between "predicted" outcomes and schedules and actual development. The open and uncertain nature of the many forces on software development, whether technical or functional, suggests that a development process needs to expose rather than smother risks and uncertainties, and to make its progress visible in terms that can be considered meaningful and faithful.

Even if news on progress is bad news, it is better to have this than no news, and more honest than wrong news. Reasonable decisions cannot be taken in the absence of accurate information. Decisions on progress and direction need to be taken on many different time scales, which suggests that feedback loops at different levels of development and orders of time are essential to ensure accuracy, whether minute to minute or month to month.

Iterative and incremental lifecycles navigate development based on such feedback looks, with agile processes taking the next step by ensuring a rich network of feedback at different scales. Scenario-driven development, whether through stories or use cases, ensures that macro-development steps are taken primarily in terms of visible functionality and not software artefacts, which have less visibility to stakeholders.

## Concretion of Implied Concepts

- Discovery of types for values, management and control, collectives, domains, etc
  - Implied concepts or collocated capabilities can be made more visible by recognising these as objects of distinct and explicit types, i.e. usage becomes type
- For example...
  - Strings for keys and codes become types in their own right, e.g. ISBNs, SQL statements, URLs
  - Recurring value groupings become whole objects
  - Behaviour based on *statics* implies a manager object



There are often implied concepts in code. For instance, few examples of `int` in code are actually just plain 32-bit signed integers free of interpretation. There is often an implied and associated concept being represented. This is often made clear(er) by their naming, context and usage. However, sometimes the enforcement of this implied concept is tedious and the visibility of the concept low.

Although ISBNs may reasonably be communicated via strings, this is not necessarily the best in-program representation for their manipulation. They are not arbitrary strings free of constraints and interpretation, so wrapping a string in a type that enforces their well formedness increases the visibility of the concept in the code, making the implicit explicit. It also reduces duplicate code and unspoken assumptions, and increases opportunities for change of representation. Likewise, groupings of individual items that are passed around together and treated as a whole — e.g. three integers representing a date — or where the grouping is more stable than the stability of the individual items — "If you have a procedure with ten parameters, you probably missed one" (Alan Perlis) — suggest encapsulation of the concept as a named type rather than as a ragtag assortment of individual values.

# *Consideration 3*: Spacing

*"That's a great deal to make one word mean," Alice said in a thoughtful tone. "When I make a word do a lot of work like that," said Humpty Dumpty, "I always pay it extra."*

**Lewis Carrol, *Through the Looking-Glass, and What Alice Found There***

## Locality and Separation

- Spacing introduces separation between parts, making parts more distinct and focused
  - The conceptual spacing between components
- Agility is not just about process, organisation, tools, skills and attitude: architecture matters
  - Most systems are rife with coupling and excuses
- Class hierarchies can become so jammed with purpose as to cause cognitive gridlock
  - E.g. inheritance layers from infrastructure to domain



With perhaps the exception of user interfaces, any discussion of space in software is inevitably based on a metaphorical view rather than one that is perceived directly through human senses. Diving too deeply into this metaphor without coming up for a reality check can lead to some false conclusions. However, there is still plenty of depth to be had.

Spacing is present in many aspects of software and its development, whether in the literal interpretation of spacing in source code or the more conceptual spacing to be found in separating concerns in module design. Of course, there is a balance to be found: don't get lost in whitespace or fragments.

Spacing can enhance visibility, letting each concept stand alone and more clearly distinct from others. For example, spacing is what divides uncohesive wholes into more cohesive individual parts, each more clearly understood. The loosening of coupling is also an act of introducing space between parts. And, of course, with spacing comes boundaries and, therefore, interfaces. Such separations support parallel work, stable protocols, simplified testing and the ability to change the parts behind the interfaces freely and independently.

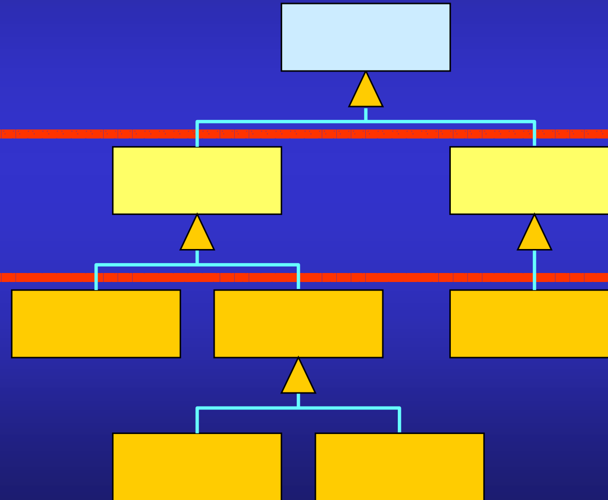
In terms of cohesion, placing definitions all together in a single place reduces visibility and overwhelms any sense of locality. Spacing definitions apart, e.g. in separate files, improves both visibility and locality, assuming that the spacing is not arbitrary. On the other hand, too much spacing leads to a loss of visibility, i.e. fragmentation.

# 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.*





The class hierarchy shown above is not an uncommon approach to organising hierarchies, with infrastructural plumbing at the root, additional adaptation beneath that, before reaching the actual domain classes of interest. This does exhibit layering of concepts, and therefore some spacing between them, in a way not found in hierarchies that mash incidental detail, plumbing and domain concepts together. However, this use of inheritance to accumulate through layers steadily increases the dependencies that any leaf class — and its authors, clients and tests — must contend with. So, in spite of its other qualities of separation, there is a degree of hierarchy lock-in and overhead that arises from insufficient separation between the conceptual layers.

## Domain × Services × Infrastructure

**Domain**
*Application domain concepts modelled and represented with respect to plug-in services.*

**Services**
*Services adapted appropriately for use by the domain classes.*

**Infrastructure**
*Plumbing and service foundations for use as self-contained plug-ins.*

concept

realisation

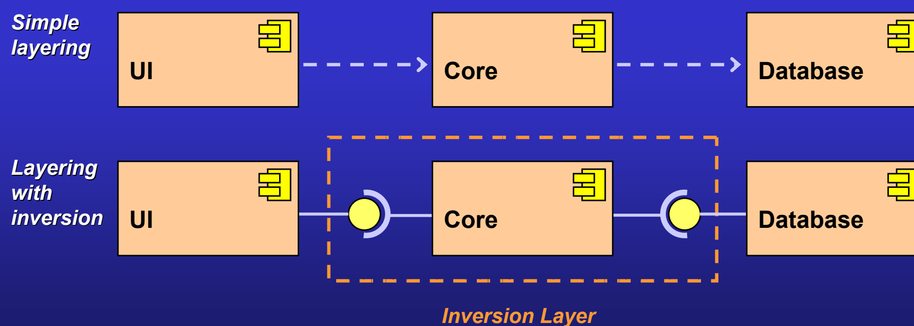


Space through separation and composition offers orthogonality and greater composability. The focus is more naturally on the domain first and infrastructure as a tertiary detail, and not necessarily one that has to intrude much past the services layer. Delegation through conceptual roots expressed as pure interfaces leads to a stricter and more separated layering that is more supple.

Orthogonality is often the key to unblocking the obstacles arising from information overload. So, favour Strategy and other pluggable approaches that push behaviour out of a class hierarchy over those that push behaviour down, such as Template Method, and split classes and hierarchies along role lines.

# Inversion Layers

- Dependency inversion eliminates the transitive dependencies of many layered architectures
  - Leads to a more testable, plug-in design style






Moving to a design where essential dependencies are visible and nonessential ones hidden can be considered a common soundbite behind much OO thinking. However, the approach advocated here is stronger than the usual approach: place an object or group of objects at the centre of their own apparent universe, with clear spacing between them and any external dependencies, bridging through self-centred interfaces rather than concrete types. A self-centred interface is one that is defined by the object demanding the services rather than one defined by a separate party and adopted by an object. Such selfishness leads to a highly localised, open and testable architectural style, sidestepping the problems of Singleton objects and the transitive coupling through layers common in many architectures.

Some developers have already come across this style in the context of introducing Mock Objects for testing database dependencies, or the use of Dependency Injection in the context of specific middleware platforms, or the choice to base a system's extension on plug-ins. However, this approach applies more broadly than each of these specific instances. A more object self-centred approach to class design should generally — not just specifically — be considered the way to do business with objects. It leaves more space to think, to breathe and to change one's mind. It also helps to contain assumptions and reduce the span of changes. There is considerable evidence that span of changes is broadly indicative of system health, i.e. shotgun maintenance.

# *Consideration 4*: Symmetry

*Due or just proportion; harmony of parts with each other and the whole; fitting, regular, or balanced arrangement and relation of parts or elements; the condition or quality of being well-proportioned or well-balanced. In stricter use [...]: Exact correspondence in size and position of opposite parts; equable distribution of parts about a dividing line or centre. (As an attribute either of the whole, or of the parts composing it.)*

**Oxford English Dictionary**

## In Search of Balance

- Symmetry is with respect to an aspect, a point of view, a part, a domain, a formalism, etc
  - Symmetry has various definitions, ranging from a formal view of invariance to a more everyday one based on completeness, consistency and balance
- Making symmetry is significantly more important to most designs than breaking it
  - It is sometimes easy to see one as the other — many examples of symmetry breaking from one point of view are examples of preservation from another

24

Just as symmetry should not be overplayed, neither should asymmetry or the reduction of symmetry be presented as an end goal or virtuous destination of design. Perfect symmetry is not only difficult to achieve: in software development it is also somewhat difficult to define! However, blindly breaking symmetry is unlikely to have a useful effect: experience suggests that most software systems are in need of more rather than less symmetry.

In software, as in physics, symmetry breaking is an effect rather than a cause, so we must look deeper. We can see many examples where the recognition and fixing of a broken symmetry has led to an improvement. For example, regularity of naming convention and argument lists emphasises likeness through likeness of form. Its informational nature makes software more easily reversible than the real world, so time's arrow need not command the same reverence it does in the real world: actions can be undone or redone with impunity.

Symmetry breaking has an important role to play, but one that is subordinate to the making of symmetry in a practical domain so often lacking in it. A disproportionate number of API design problems can be characterised in terms of broken symmetry. Some problems have competing symmetric and asymmetric solutions. For example, in the problem of the dining philosophers one solution is to break symmetry by requiring that one philosopher uses a different hand to the others, and another is to make the abstract concept of management concrete, i.e. introduce a waiter.

## In Search of Alignment

- Alignment between two domains or views is a common form of symmetry
  - "The elements of symmetry of the causes are to be found in the effects produced" (Pierre Curie)
- For example...
  - Aligning problem and solution domains (e.g. DSLs)
  - Aligning architecture and organisational structure (Conway's "Law")
  - Aligning architectural partitioning and stability



A correspondence in structure between the domain of the problem and the domain of the solution, i.e. *modelarity*, makes comprehension of the designed modular structure and the problem domain easier. Do not underestimate the pressure that a desire for alignment can exert on design and other concepts or artefacts in use. This is as true in everyday life as it is in software.

For example, *island* and *isle* are conceptually related and, one would assume, etymologically related. However, they are false cognates, with *isle* influencing the spelling of *island*: *isle* came to Middle English via Old French (*ile*, now *île* in Modern French) from the Latin (*isle*, from *insula*); *island* came from Old English *igland* (or *egland* or *eigland*), where *ig* itself meant *island*, via Middle English *ilond*. A similar effect can be seen today in the increasing tendency to spell (and recognise as correct) the spelling of *minuscule* as *miniscule*, after the increasingly common pattern of the *mini-* prefix.

We can also see alignment in the evolution of old measurement units. Unrelated units tend to align with one another over time, especially when subject to other forces, e.g. monetary. For example, in the reign of Elizabeth I the English mile was lengthened by nearly a tenth from its traditional Roman length to one that aligned more conveniently with the perch, the unit used to measure land and define ownership.

Regularity and irregularity explains why children tend take longer to learn how to use strong verbs — where the past tense is formed via vowel shift, e.g. "I run" becomes "I ran" — than weak verbs — where the past tense is formed via suffixing, e.g. "I walk" becomes "I walked".

## JUnit Symmetries and Asymmetries

- Recursively consistent execution capability in the JUnit test runner UI for Eclipse
  - By simple selection, can choose to execute all the tests in a project, a package, a class or a method
  - Not present on the default runner UIs for JUnit 3.8
- The family of assertions pair up nicely...
  - *assertTrue* and *assertFalse*, *assertNull* and *assertNotNull*, *assertSame* and *assertNotSame*, ...
  - But sadly, and noticeably, neither *assertEquals* nor *assertArrayEquals* have a partner



Over time JUnit has become more symmetric in terms of its use and feature set. For example, `assertTrue` was once not accompanied by `assertFalse` (and before that it was named simply `assert`, which was set to clash with the J2SE 1.4 keyword). The symmetries highlighted above as missing are standard in NUnit, which can be considered a later generation of the same design family. Note that although there is a strong case for `assertEquals` and `assertArrayEquals` to have negating counterparts, the same is not true for `assertThat`, whose constraint-based approach supersedes the need for such balance.

A tendency to align and improve perceived symmetry can be seen as a desire to reduce friction from minor differences by emphasising similarity and fit. This is particularly true of both API and UI design.

As an aside, the act of refactoring with respect to functional tests is one of symmetry preservation: functional behaviour is invariant while the structure of the code changes.

# *Consideration 5*: Emergence

*Recognize that you are not assembling a building from components like an erector set, but that you are instead weaving a structure which starts out globally complete, but flimsy; then gradually making it stiffer but still rather flimsy; and only finally making it completely stiff and strong.*

**Christopher Alexander, *A Pattern Language***

## The Product of the Parts

- Complex and sophisticated behaviour can arise from simple rules
  - Emergent properties can be counterintuitive and at odds with appearances
  - Control without control
- Emergent behaviour can be undesirable as well as desirable
  - Without awareness of emergence, a development can become mired in patchwork special-case code and a victim rather than a user of queuing theory

28

Not all aspects and levels of a system's behaviour have to be defined painstakingly and explicitly, enumerated in patchworks of special cases. Behaviour can arise out of combination and collaboration: it does not itself need to be modular in definition. One of the most common examples of emergent behaviour is that of flocking, complex behaviour that relies on three simple rules:

> *Separation*: don't crowd or bump into your neighbours.
>
> *Alignment*: go in the same direction as your neighbours.
>
> *Cohesion*: move as close as possible to your neighbours.

However, emergence is not magic and should not appear as an excuse for muddled or vague notions of behaviour. Most obviously, it can find itself in tension with the consideration of visibility.

Federation is a common approach to establishing emergent behaviour that is localised, scalable, coherent and resilient. For example, the DNS system for domain-name lookup eschews centralised control in favour of an approach that is locally defined but globally scalable. This approach can be found in many Broker-based architectures.

Emergent behaviour is also responsible for surprising behaviour, typically non-linear in nature — which, to humans, is often surprising. For example, gridlocked traffic, virtual memory page thrashing, effect on schedule of adding more (untrained) staff to a late project, and so on.

## Command(-less) and Control(-free)

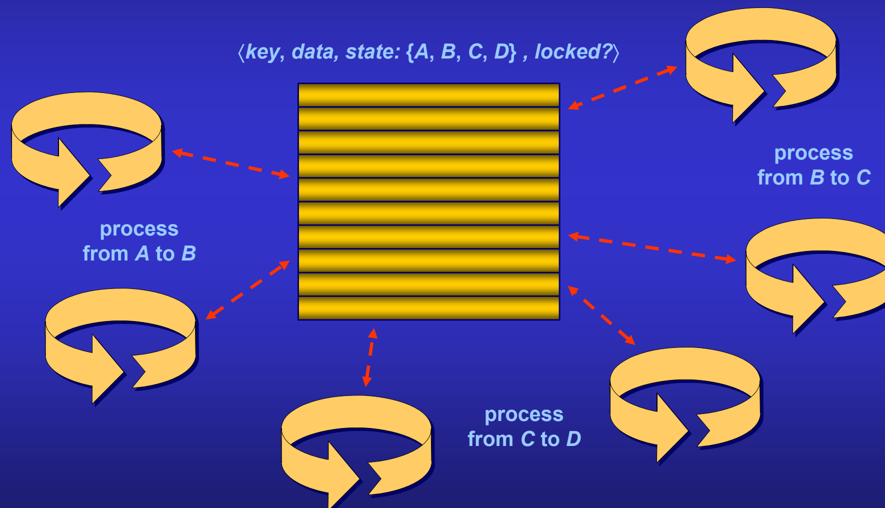
- Sometimes the most effective way to achieve a desired effect is to give up tight control
  - E.g. self-organising versus micro-managed teams
  - E.g. make a problem visible to encourage its solution — build problems, bug count or age — as opposed to making its solution a commanded responsibility
  - E.g. take decisions through polymorphism not *if*
  - E.g. a sequence of elements does not need to have *sort* applied to make it sorted: start from nothing and add elements so that a sorted order is preserved

29

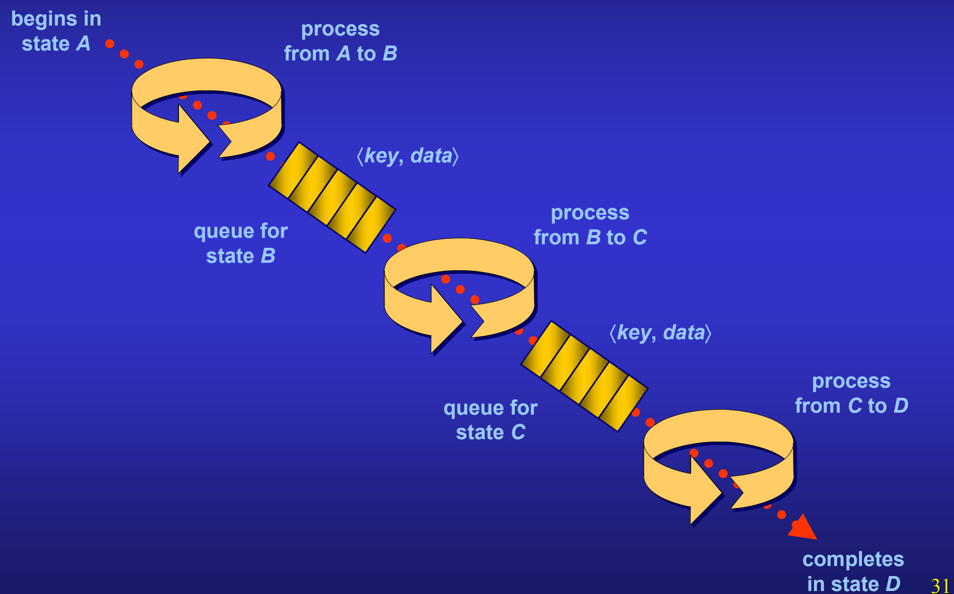
There are many cases that demonstrate that sometimes the best way to achieve an outcome is not to pursue it actively from the point of view of controlling. This counterintuitive approach can be found in action in the form of self-organising teams favoured by agile processes. Likewise, there are counterintuitive ways of tackling certain problems that involve stepping back and controlling other variables, as opposed to controlling effects. For example, the number of outstanding defects in a bug database can easily plateau, reaching steady state. One way to reduce this is not to tackle the count directly, or reward reduction of count, but to tackle the age of bugs and both focus on and reward the reduction of the mean age of defects logged in the database.

It is all too easy to fall into the trap of writing out every apparent decision that a program takes in terms of explicitly structured control flow. It is not simply the problem of being overwhelmed by battalions of special cases, but also the problem of obfuscating even the simplest decision — leading in turn to a loss of economy.

Polymorphism, whether declared or handcrafted, is one of the most common approaches to distributing decision behaviour away from the point of use. This manifests itself most obviously in class hierarchies, but also through template parameters and function pointers. Introducing appropriate spacing also polymorphism to play out at compile and link time, through the judicious use of platform-specific build paths and dynamically linked libraries.




Consider the problem of handling a large number of records that can be identified by their key and further characterised in terms of the data they hold and the state that they are in. They follow a simple state model where one state is processed into the next. The arrangement of worker threads shown, each independently drawing from a common repository, has a decentralised feel to it, following a Blackboard-style architecture. Because of the threading an additional locked flag is required to supplement the state flag.




Reviewing the problem suggests that a better way to characterise the problem is as one of transformation rather than one of state. A Pipes and Filters structure presents a more directly aligned architecture with this characterisation of the problem domain, one that leads to a reduction in the state needed to manage the lifecycle of each record. Instead of treating records as static and operated on by opportunistic threads, records pass from thread to thread via queues. The queues now represent the states and the threads the transitions. Decision structures have disappeared and the state management and lifecycle is an emergent rather than explicit property. This approach is more scalable and composable than the previous one.

## Design Discovery

- A RUF-then-refine rather than a BUF approach helps to converge on a good design
  - A vision of what is needed, and a set of possible outcomes in mind, can make a big difference...
  - But this is not the same as a single fixed and overarching scheme
- For example...
  - Evolving a unit-testing framework from *assert*
  - The JUnit *Money* problem in C++ or Ruby, for which there are lighter solutions than the typical Java ones



If software development could be considered a defined domain, fixed-plan development would not only be possible but would also be more generally effective. However, its multi-variable nature tends to defeat any up-front planning that goes beyond establishing a vision of design and some of the possible paths and techniques that can be used in the detail. This is not to say that there should be no up-front activity — exploration and a critical level of understanding are needed before diving headfirst into the solution space — but that the goal and scope of up-front activity should be clarified and bound.

A simple example of discovery and response is the ability to rename an identifier. It appears trivial but is significant in revealing the intended concepts that underpin a chosen solution. Given the typically metaphorical nature of names, names matter because of the metaphoric entailment. This is one simple undertaking that must be possible as a design is refined through experience.

The Money problem, simply put, is to construct a class that can be used to represent single or mixed current amounts. The typical Java version of the Money problem relies on the Composite and the Double Dispatch patterns plus a host of additional helper methods, and can be extended to take advantage of the Null Object pattern. An idiomatic C++ version based on the STL requires only a single value class with no class hierarchy, and either a `map` or `vector` of `pair` for representation. An idiomatic Ruby version can rely on the convenience of `Hash` objects.

# Outroduction

*The only thing to do with good advice is to pass it on. It is never any use to oneself.*

**Oscar Wilde**

# Any Other Business