Programmer's Dozen

Thirteen Recommendations for Refactoring, Repairing and Regaining Control of Your Code

Kevlin Henney

kevlin@curbralan.com

Presented at *JavaZone 2003*, Oslo, 18th September 2003.

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
  - Motivation and principles
- A Programmer's Dozen
  - Thirteen code-level recommendations
- Outroduction
  - Wrap up

> *We think in generalities, but we live in detail.*
>
> **Alfred North Whitehead**

There is no shortage of technical wisdom on how to develop clear and robust code, so why is the expression on many a programmer's face so paned (sic) when staring at all those source windows? There are companies whose development culture does not encourage pursuit of knowledge of practice: a penny-wise pound-foolish approach. However, there are many companies and developers that want to push themselves to the state of the art, but seem swamped and bemused by how much state there really is to that art.

This tutorial offers a concrete thirteen-point list of practices (zero through twelve) that can be applied out-of-the-box to reduce code size and complexity, acting as both guidelines for new code and indicators for refactoring. The short list has no ambition to be all that you needed to know about design but were afraid to ask, but it does offer an easily memorable and easily practiced set of guidelines that offer the greatest immediate return on investment — the most bang for your buck, oomph for your euro or kerrang for your kroner.

Introduction

programmer	*a person who writes computer programs.*
dozen	*a group or set of twelve.*

The New Oxford Dictionary of English

Asymmetric bounds are most convenient to program in a language like C in which arrays start from zero: the exclusive upper bound of such an array is equal to the number of elements! Thus when we define a C array with 12 elements, 0 is inclusive lower bound and 12 the exclusive upper bound for the subscripts of the array.

Andrew Koenig

*A **baker's dozen** contains thirteen items as opposed to the familiar twelve. This dates from the time when bakers were subject to heavy fines if they served under-weight bread. To avoid this danger bakers provided a surplus number of loaves, the thirteenth loaf in the dozen being called the vantage loaf.*

Lock, Stock & Barrel

Minimalism

- A movement in art, architecture, music and interior design
- More generally, a tendency towards less rather than more
  - Lowest point on a curve or shortest distance

> *It isn't necessary for a work to have a lot of things to look at, to compare, to analyse one by one, to contemplate. The thing as a whole, its quality as a whole, is what is interesting. The main things are alone and are more intense, clear and powerful.*
>
> **Donald Judd**

4

The appeal to "omit needless words" is made by Strunk and White concerning written composition:

> Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all sentences short, or avoid all detail and treat subjects only in outline, but that every word tell.

Whilst we may muse about engineering, architectural or craft metaphors for software development, there is no denying that, in essence, programming is writing. It is a form of communication that has two distinct audiences: us and the machine. Although there are times when it might not feel this way, the machine is easily pleased, demanding little more than well-formed code. We, however, are a little more complex and discerning: we demand that our communication communicate. We can make an appeal similar to Strunk and White's: omit needless code.

Less Code, More Software

- Creeping featurism and code verbosity do not add to the used behaviour of software
  - Functionality is weakly correlated with the number of lines of code written
- On the other hand, minimalism is not the reduction of source code to line noise

> *The difference between a good and a poor architect is that the poor architect succumbs to every temptation and the good one resists it.*
>
> **Ludwig Wittgenstein**

This sentence no verb. The previous sentence is an example of compression. It could not have been made more direct by adding anything as its structure is intimately bound up in its meaning. In fact, its structure is its meaning. Taking time out to add things would only weaken it and miss the point entirely. This is compression; it is about intent, it is about sufficiency, and it is about time it was considered a valuable property of code. It is not a question of style versus substance: the style *is* the substance.

Richard Gabriel offers the following on compression:

> Compression is the characteristic of a piece of text that the meaning of any part of it is "larger" than the piece has by itself. This is accomplished by the context being rich and each part of the text drawing on that context — each word draws part of its meaning from its surroundings. A familiar example outside programming is poetry whose heavily layered meanings can seem dense because of the multiple images it generates and the way each new image or phrase draws from several of the others.

Remove to Improve

- Yes, it can be OK to delete code
  - Controlled and considered code implosion shows that a project is maturing
- But remember, minimalism is not nihilism
  - Refactoring is a disciplined practice

> *The minimum could be defined as the perfection that an artefact achieves when it is no longer possible to improve it by subtraction. This is the quality that an object has when every component, every detail, and every junction has been reduced or condensed to the essentials. It is the result of the omission of the inessentials.*
>
> **John Pawson**

6

As a discipline of composition much of what can be said for writing natural language is directly applicable to code. There is no virtue in long-windedness and, by way of balance, there is also no virtue in code that is unreadably terse. As ever the appeal is for well-written code. Code that is simple and clear, brief and direct.

As a measure of productivity or functionality, *lines of code* is meaningless. Its use as such a measure is irresponsible. Yes, it is a measure, but not of these things, just as weight is not a useful measure of height, eye colour or intelligence. It is a measure of bulk, of mass. It gives you an idea of how much you are dealing with, but not actually how much it does. The correlation is weak, and only works in the limit with well-factored code. There are many examples of code that have the same functionality as an order of magnitude less in size.

Refactoring is the discipline of changing code structure to improve it, while retaining the same behaviour. It can be considered a matter of revision, and often one of removal.

Some Recommendations

- For programmers that care, good practice recommendations can be a problem
  - There are so many!
  - Being bombarded from all sides, it is easy to become overwhelmed
- A reduced list is a better place to start

> *To achieve simplicity paradoxically requires an enormous amount of effort.*
>
> **John Pawson**

Conscientious developers are often looking for that extra edge that allows them to keep their code live and clean. For guidelines that help them to reduce the amount of source code in their system, to avoid duplication and unnecessary centralization in their code structure, to decouple tangled hierarchies and overly intimate packages, and to write encapsulated interfaces rather than the assembleresque *getter–setter* style that is unfortunately so popular in many systems.

However, the number of recommendations available can be overwhelming. The Programmer's Dozen aims to offer enough for guidance, and enough to embrace other more specific practices, but not so much that the recommendations form a complex shopping list.

Rules, Considerations or ...?

- Recommendations are weaker than rules, but are stronger than considerations
  - A *rule* is a 'shall' or a 'must' that should rarely if ever be broken, but must be concrete
  - A *recommendation* is more of a 'should', and is best phrased as a constructive guideline
  - A *consideration* is something to keep in mind, and can be very concrete or quite abstract

The distinction between *rules* and *recommendations* is drawn from the original Ellemtel *C++ Rules and Recommendations*:

> Programming standards must be valid both for newcomers and experts at the same time. This is sometimes very difficult to fulfill. We have solved this problem by differentiating our guidelines into rules and recommendations. Rules should almost never be broken by anyone, while recommendations are supposed to be followed most of the time, unless a good reason was at hand. This division gives experts the possibility to break a recommendation, or even sometimes a rule, if they badly need to.

To this we can add the weaker notion of *considerations*, as well as the stronger notion of *law*. In this case a law should be considered with respect to the physical notion of law rather than the political one. A correctly formulated physical law cannot be broken, whereas laws governing a society are closer to what we have defined as rules — clearly quite breakable. An example of a binding set of laws in programming would be the syntax of a given language. Any attempt to break such laws is met not so much with punishment as unfulfilment.

Recommendation 0

Follow Consistent Form

> *This principle, that of parallel construction, requires that expressions similar in context and function be outwardly similar. The likeness of form enables the reader to recognize more readily the likeness of content and function.*
>
> **Strunk and White**

Idiomatic Code

- Suitable idioms in code should be followed to improve communication
  - Idiolects reduce general comprehensibility
- However, this does not mean slavish adherence to unreasonable convention

idiom
- Linguistic usage that is grammatical and natural to native speakers
- The characteristic vocabulary or usage of a specific human group or subject
- The characteristic artistic style of an individual, school, etc.

In all fields of communication we must recognise the importance of idiom. Compression appeals to common idiom (e.g. the use of the + operator to represent addition or concatenation) without resorting either to private language (e.g. `#define` your own 'keywords' and certain bizarre applications of operator overloading) or to lowest common denominator baby talk — don't write code for novices unless you are writing code for novices.

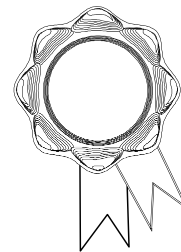
Idioms define a usage, and offer a meaning, that is shared by a community of programmers above and beyond the level of language syntax. Semantics and additional intent are communicated through idioms. This is why the mastery of any programming language, no matter how complex or simple it first appears, often takes a lot longer than many are prepared to admit.

Recommendation 1
Don't Break Contracts

> *Politicians are the same everywhere. They promise to build a bridge even when there's no river.*
>
> **Nikita Khrushchev**

Clarity and Substitutability

- A contract imposes a set of constraints on a client–supplier relationship
- Some — sometimes all — of the interface agreement can be phrased assertively
  - E.g. preconditions and postconditions
- Contracts also mean that a subclass should be substitutable for its superclass



The basis of the contract model is to formalise the relationship between the object user and supplier at the contract level, rather than just at the implementation. It is then the role of the developer to supply a class implementation that satisfies this contract, and for the user to use it in the agreed fashion.

Type systems provide part of the story when it comes to establishing interface usage, but at best they can provide no more than compile time confidence in the structure; in many systems there is plenty left to hit the fan at runtime if interface usage is well formed but otherwise incorrect. A contractual view establishes further limits on an interface by defining the legal requirements on the behaviour of operations:

- Contracts can be expressed formally using preconditions and postconditions to define boundaries of correct behaviour.
- Contracts can be expressed by example, a more empirical approach, often based on specification by example and unit tests.
- Contracts may simply be legislated in documentation.

Recommendation 2

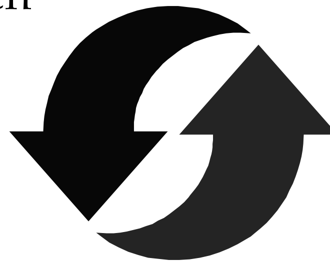
Express Independent Ideas Independently

What do you get when you cross a mosquito with a rock climber?

Nothing, you can't cross a vector and a scalar.

Cohesion and Orthogonality

- Do not place unrelated concepts together in the same class or package
  - Overachieving interfaces are weaker and more complex, not stronger and simpler
- Cyclic dependencies between layers and packages lead to noosely coupled code
  - Dependencies are propagated unnecessarily



A concerted focus on dependency management will deliver you some tangible benefits in the short term — development times, build times, lunch times — and reduce the cost of change in the long term. The loose coupling keeps the code supple and more stable as, over time, the genuine sources of variation, and therefore parameterization, become apparent and needed.

A cyclic package dependency is where the elements of one package depend, directly or indirectly, on the contents of another which in turn depends, directly or indirectly, on the first package. Cyclic dependencies imply a strong coupling between two or more development components. This can affect the evolution and architectural integrity of a system. In terms of packages, it means that a package cannot be treated as an isolated whole as it is intimately related to any changes in another package and vice-versa, i.e. together they form a single, less cohesive unit of release.

An example of this is the way in which `java.io` and `java.lang` are interdependent (`java.io.Serializable` is used extensively by `java.lang` types, for instance), rather than being layered as you would expect in a decoupled design, i.e. `java.io` depending on `java.lang` but not vice-versa. In other words, Java systems have no uniquely primitive package.

Recommendation 3

Encapsulate

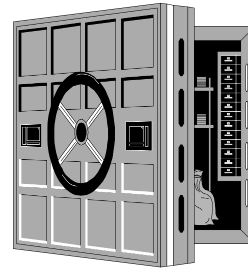
encapsulate *enclose (something) in or as if in a capsule.*

- *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

In a Capsule

- Encapsulation is more than just combining data and function with protection...
  - The self-containedness of an element
  - An element's ease of use, so the surface area and simplicity of its interface
  - Extent that design decisions *leak* from an element
  - The reinforcement of constraints



Objects are said to be encapsulated, which literally means "within a capsule". There are two slightly different interpretations of this in conventional OO, both of which apply: objects have crisp, well-defined boundaries that hide their representation, i.e. the capsule has a surface separating the outside world from its inside; data and function are packaged together in an object, i.e. it is both of these that are put in the capsule.

Classes are the representative elements of design for objects, and it is classes that are written and written against by the developer. At face value, the two statements above make encapsulation appear to be no more than wrapping data in operations with a convenient syntax. However, it goes much further, and is not restricted to OO designs. Encapsulation is a property of a design rather than a mechanism. It refers generally to an approach that makes components self-contained and complete. An encapsulated class ensures correct constraints for an object from the moment it is created to the time it is discarded. Self-containedness includes exception safety, thread safety, ease of use, transparency, etc. An encapsulated design attempts to match constraints with affordances.

Recommendation 4

Restrict Mutability of State

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

Lucius Cary

Less Chance for Change

- The less opportunity there is for state changes, the less that can go wrong
  - Hence, reduce scope of modifiable variables and the methods available to modify objects
- Restricted state mutability improves encapsulation
  - E.g. immutable values are more encapsulated than mutables when objects are shared

Substitutability, normally considered only in the context of conventional inheritance, is also possible with respect to the mutability of an object. Indeed, substitutability of objects sometimes only makes sense when mutability is considered, for example the substitutability of an ellipse for a circle is not reasonable when they are freely modifiable, but makes perfect sense if they are immutable.

What reduced mutability offers is a confidence in code that state changes on an object can either not happen or can happen only in certain places. Given the number of bugs that arise from unwanted state change, such a confidence is worth having.

Recommendation 5
Parameterize from Above

> *While moon sets*
> *atop the trees,*
> *leaves cling to rain.*
>
> **Bashō**

Parameterize with Parameters

- Pass in parameters explicitly rather than having them global or pulled in
  - Beware Singletons, Monostates, *static*s, etc.
  - Communicate through constructor arguments or generic parameters, as appropriate
  - Decentralise configuration constants
- Callout interfaces define the dependencies of each part that could be configured
  - E.g. the Mock Object unit testing technique

20

Why is programming through interfaces considered to be good practice? And why, increasingly, is it considered to be poor practice to use Singleton objects? These two questions have the same answer: *flexibility*. This is a loose word, so here is a more precise answer: accommodation of alternative implementations.

A system should be testable and reconfigurable from a build perspective. Singletons and `static`s prevent this flexibility and should generally be eliminated and avoided. They look like a better idea than they actually are.

Recommendation 6

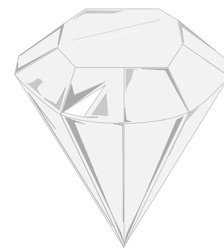
Favour Symmetry over Asymmetry

As the sunrise to the night,
As the north wind to the clouds.

Percy Bysshe Shelley

Symmetry and Simplicity

- Structures that are regular can be used and recalled more easily than irregular ones
  - There is less to know, less to remember
- Symmetry is one such form of design compression
  - A symmetric design has features that are balanced opposites, e.g. pairs of methods



22

Quoted in Jon Bentley's *Bumper Sticker Computer Science*, Andy Huber cautioned: "Avoid asymmetry". On the other hand, Victor Hugo proclaimed: "Symmetry is boredom, and boredom is the very source of death. Despair yawns." ("La symétrie, c'est l'ennui, et l'ennui est le fond même du deuil. Le désespoir baîlle."). So what is the best guidance for software design? The bias should be towards symmetry, with enough asymmetry to make a difference, but not so much as to cause indifference.

A rule is easier to recall than its exceptions, and a rule that appears only to be exceptions is harder still. In other words, there is less information in a symmetric design than in an asymmetric one. For this reason, symmetry tends to have a simplifying effect, whether in the detail of code or in a system's gross architecture. Gratuitous asymmetry creates the semblance of disorder in a design: everything is different, nothing is balanced; to describe and understand it requires protracted effort; to recall and apply it is tedious and error prone.

Symmetry is about balance. The expectation that when a particular feature is present, its logical counterpart will also be there. Where there is the capability for output there is also the capability for input. Where a resource is acquired it can also be released. Where there is a `commit` there is also a `rollback`.

Recommendation 7

Flow Don't Jump

Still glides the Stream, and shall for ever glide;
The Form remains, the Function never dies.

William Wordsworth

Avoiding Jumpy Code

- Discontinuous control flow structures should be used sparingly
  - Easy to get addicted to *break* and early *return*s — and *continue* and even *goto* (where it exists)
  - *throw* is exceptional
- Control flow that doesn't flow can be hard to comprehend and to refactor
  - Structured programming is not yet dead

It's tough being a salmon. A life spent at sea is a prelude to a Special Forces survival course for mating, jumping uphill against the natural flow of a stream. To look at the control flow of some programs is to stand by a turbulent stream like a hopeful bear waiting to knock a salmon out with its paws.

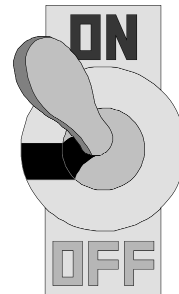
By definition flow should, well, flow. The basic ingredients of continuous control flow are sequence (where one statement follows on to the next), selection (`if else` and `switch`) and iteration (`while`, `for` and `do while`). These are the basic ingredients of structured programming, and the lessons learnt from structured programming should not be forgotten. Sometimes discontinuous control flow mechanisms seem attractive (`return`, `break`, `continue` and `goto`), offering apparent short cuts. Early `return` and `break` can sometimes be justified, but `continue` and `goto` should be considered surplus to requirements. `throw` is exceptional when it comes to discontinuous flow — it may at first appear that exceptions are in bed with `goto` et al, but it turns out that they're sleeping mostly with the opposition.

Recommendation 8
Sharpen Fuzzy Logic

> *This above all: to thine own self be true,*
> *And it must follow, as the night the day,*
> *Thou canst not then be false to any man.*
>
> **William Shakespeare**

Poor Logic is False Economy

- Many verbose coding practices are justified because they simplify debugging
  - This is solving the wrong problem
- Apply standard logical transformations
  - Elimination and introduction
- Reduce the incidence of flag variables and Boolean arguments
  - E.g. *enable(false)*



Like overly jumpy control flow, a piece of logic can sometimes accumulate mass like a snowball. Before you know it, you haven't a clue what is going on and the logic has gone fuzzy. This recommendation could have been named "review fuzzy logic", but there would have been no point: the only thing to do with complex logic is to simplify it.

There is something about Boolean logic that sometimes challenges us. Given that there are only two values to play with — as opposed to arithmetic with its choice of infinities — you would think that there couldn't be too much scope for confusion. Perhaps it is precisely because of this pared down set of choices that we can so easily miss a trick, confidently writing a piece of code to do something when we had fully intended for the code not to do it. The fact that looking at how we express logic is sometimes seen as a minor detail not worthy of attention is misleading: the opposite is true. In software detail matters, and Boolean logic is the bread and butter of programming so it has a special place in the skill set of any programmer and the text of any program.

Recommendation 9

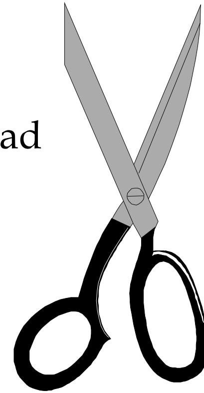
Hand Redundant Code its Notice

Omit needless words.

Strunk and White

Subtle and Elaborate No-ops

- Code may be less than the sum of its parts
  - Redundant code has no genuine effect
- Most redundant code acts as a speed bump to understanding
  - Unreachable code, locking non-thread affected code, unused code, etc.



Some redundancy can be useful where it acts as a checksum rather than as duplication. For example, including the day along with a date — e.g. Saturday 6th April 2002 — could be considered redundant: using either Zeller's congruence or a calendar you can find out the day. However, the calculation is non-trivial and the redundancy serves as a checksum when arranging appointments and the like. It ensures that any thinkos — e.g. the day is right, but you had another month on the brain when you wrote out the date — are obvious.

Appropriate redundancy also rights any representation misinterpretations: Saturday 06/04/02 is correct when read using little-endian date ordering but is catchably incorrect if the reader assumes middle-endian (it would be Tuesday) or big-endian (it would be Sunday). The extra point of triangulation allows you to fix your position precisely and with confidence. It is where the redundancy is genuinely doing nothing that there is a problem.

Recommendation 10

Let the Code Take Some Decisions

> *His had been an intellectual decision founded on his conviction that if a little knowledge was a dangerous thing, a lot was lethal.*
>
> **Tom Sharpe**

Empowering Code

- There is no need to spell out every decision taken by a program in minute detail
  - Control flow and logic can degenerate to a patchwork of special cases
- Code can also appear to do more than the sum of its parts
  - Indirect selection mechanisms, such as polymorphism, lookup tables and grouping objects into common collections

30

How do you get your program to do something? You write out specifically the code that performs the task. How do you make your program take alternate actions depending on some context or condition? The obvious answer is to write all the decisions and options out explicitly. Decisions and multiple options encoded as `if else` and `switch` statements certainly spell everything out in detail, but sometimes that can be just a little — if not a lot — more detail than is strictly necessary.

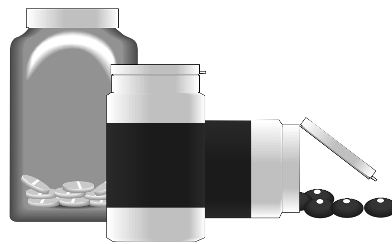
OK, to clarify, of course it is the code that will ultimately make all the decisions — it is unlikely, and extremely inefficient, that you will be involved in the intimate execution of your program when it is deployed — but what is meant by "letting the code make the decisions" is related to your point of view when writing one piece of code versus another. It is a matter of emergent behaviour, arising from the collaboration of sufficient parts, versus screenfuls of hand-knitted logic. It is a matter of pushing mechanism down to the appropriate level.

Recommendation 11
Consider Duplicate Code to be a Mistake

> *You cannot step twice into the same river.*
>
> **Heraclitus**

DRY Code

- There is a need to cure the common code
- Visible duplication is a problem best solved by refactoring the commonality
  - Into a class, a function or a block surrounded by clearer logic
  - And don't forget to visit your local library



Duplicate code has a bad smell and violates the catchily named DRY principle (Don't Repeat Yourself). The weirdness that is quantum entanglement applies at the subatomic level but not in source code: a code fix in one place does not automagically update any of its duplicates.

The accumulation of duplicate code also has an obvious side effect: it increases the amount of source code. Contrary to benighted management belief, more is not better in this case. Every problem has an optimal code size: too short and the code is cryptic line noise; too long and you cannot see the wood for the trees. Duplication can obscure code's intent, causing you to wade rather than run through it, slowed by the detail and a haunting sense of déjà vu — "Is this code actually the same as that code... or is it subtly different?"

Duplicate code should be refactored away at the earliest possible opportunity, removing the doubt in the programmer's mind and obstacles from the path of continued development.

Recommendation 12

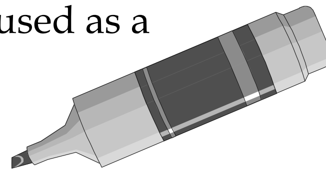
Prefer Code to Comments

1. *If a program is incorrect, it matters little what the documentation says.*
2. *If documentation does not agree with the code, it is not worth much.*
3. *Consequently, code must largely document itself. If it cannot, rewrite the code rather than increase the supplementary documentation. Good code needs fewer comments than bad code does.*
4. *Comments should provide additional information that is not readily obtainable from the code itself. They should never parrot the code.*
5. *Mnemonic variable names and labels, and a layout that emphasizes logical structure, help make a program self-documenting.*

Kernighan and Plauger

Code that is not Code

- Comments are typically ignored both by programmers and by compilers
  - Say it with code where possible
  - Comments that are wrong have negative value
- Comments should detail the non-obvious, not the obvious or fixable
  - The more that comments are used as a matter of habit, the less value they have overall



The speed that you can read code is affected by a number of factors, some of which depend on you and some of which depend on the code. For instance, your ability to parse and grasp code will be lower in the post-lunch siesta part of the afternoon than in the up-'n'-at-'em, caffeine-injected start to the day. You can attain a higher speed when code is based on familiar patterns rather than someone else's programming idiolect. Similarly, the broader your vocabulary of such idioms, the more you can take advantage of them. These observations form the basis of many of the recommendations, most obviously *Follow Consistent Form* and *Don't Break Contracts*. But there are other speed bumps that can slow you down: comments — code that is not code.

Comments are a problem in many systems. Not so much their absence as their presence. Sure, a good comment can be useful, but given the general hit-versus-miss rate across much commercial code, such comments seem to be few and far between. The lingering belief that comments are somehow *always* a 'good thing' seems to be adding drag rather than thrust to a lot of source code. This does not mean that all comments are necessarily bad, just that if you took a production system at random and removed all of the comments from its source, the result would probably be an improvement.

Outroduction

Remember that there is no code faster than no code.

Taligent's Guide to Designing Programs

In Closing

- Baroquecratic practices and just-in-case code have a br(e)aking effect
  - The measure of code is in software behaviour not in grams, furlongs or KLOCs
- Developmental requirements are as important as functional and operational
  - Attention to developmental requirements makes other requirements easier to address

The recommendations are not a substitute for thinking. They are channels for it.

Articles by Kevlin Henney, available from in HTML or PDF from `http://www.curbralan.com`:

"Creating Stable Assignments", *C++ Report*, June 1998.

"For the Sake of Simplicity", *EXE*, July 1999.

"Collections for States", *EuroPLoP*, July 1999.

"Something for Nothing", *Java Report*, December 1999.

"Substitutability", *C++ Report*, May 2000.

"Matters of State", *Java Report*, June 2000.

"C++ Patterns: Executing Around Sequences", *EuroPLoP*, July 2000.

"Collections for States", *Java Report*, August 2000.

"A Tale of Two Patterns", *Java Report*, December 2000.

"Another Tale of Two Patterns", *Java Report*, March 2001.

"Making an Exception", *Application Development Advisor*, May 2001.

"A Tale of Three Patterns", *Java Report*, October 2001.

"Omit Needless Code", *Overload*, October 2001.

"Generic Decoupling", *C/C++ Users Journal* online, November 2001.

"The Perfect Couple", *Application Development Advisor*, November–December 2001.

"The Safe Stacking of Cats", *C/C++ Users Journal* online, February 2002.

"The Imperial Clothing Crisis", *Overload*, February 2002.

"Six of the Best", *Application Development Advisor*, May 2002.

"The Rest of the Best", *Application Development Advisor*, June 2002.

"State Government", *C/C++ Users Journal* online, June 2002.

"Null Object", *EuroPLoP*, July 2002.

"The Importance of Symmetry", published as "Symmetrie in Java", JavaSPEKTRUM, July–August 2002.

"The Temptations of Symmetry", published as "Die Gratwanderung zwischen Symmetrie und Asymmetrie", JavaSPEKTRUM, September–October 2002.

"Value-Based Programming in Java", *JAOO*, September 2002.

"Methods for States", *VikingPLoP*, September 2002.

"Inside Requirements", *Application Development Advisor*, May–June 2002.

Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997.

Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 2000.

Kent Beck, *Test-Driven Development*, Addison-Wesley, 2003.

Joshua Bloch, *Effective Java*, Addison-Wesley, 2001.

Don Box, Keith Brown, Tim Ewald and Chris Sells, *Effective COM*, Addison-Wesley, 1999.

Stewart Brand, *How Buildings Learn*, Phoenix, 1994.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996.

James O Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

Tom DeMarco, *Slack*, Broadway Books, 2001.

Martin Fowler, *Analysis Patterns*, Addison-Wesley, 1997.

Martin Fowler, *Refactoring*, Addison-Wesley, 1999.

Richard P Gabriel, *Patterns of Software*, Oxford, 1996.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison-Wesley, 1995.

Mats Henricson and Erik Nyquist, *Industrial Strength C++*, Prentice Hall, 1997.

Andrew Hunt and David Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000.

Butler W Lampson, "Hints for Computer System Design", *Operating Systems Review*, October 1983.

Tim Mackinnon, Steve Freeman and Philip Craig, "Endo-Testing: Unit Testing with Mock Objects", *XP 2000*.

Bertrand Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.

Donald A Norman, *The Design of Everyday Things*, MIT Press, 1988.

Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000.

Herb Sutter, *Exceptional C++*, Addison-Wesley, 2000.

Taligent's Guide to Designing Programs, Addison-Wesley, 1994.