

# Index

## detection, [237–238](#)  
++ (pre- or post-increment) operator, [325](#), [326](#)

## A

aborted computation, [109](#)  
abstract classes, [149](#), [271](#), [290](#)  
ABSTRACT FACTORY pattern, [38](#), [156](#), [273](#), [274](#)  
abstract interfaces, [94](#)  
abstract methods  
    adding to ArgumentMarshaler, [234–235](#)  
    modifying, [282](#)  
abstract terms, [95](#)  
abstraction  
    classes depending on, [150](#)  
    code at wrong level of, [290–291](#)  
    descending one level at a time, [37](#)  
    functions descending only one level of, [304–306](#)  
    mixing levels of, [36–37](#)  
    names at the appropriate level of, [311](#)  
    separating levels of, [305](#)  
    wrapping an implementation, [11](#)  
abstraction levels  
    raising, [290](#)  
    separating, [305](#)  
accessor functions, Law of Demeter and, [98](#)

accessors, naming, [25](#)  
Active Records, [101](#)  
adapted server, [185](#)  
affinity, [84](#)  
*Agile Software Development: Principles, Patterns, Practices (PPP)*, [15](#)  
algorithms  
    correcting, [269–270](#)  
    repeating, [48](#)  
    understanding, [297–298](#)  
ambiguities  
    in code, [301](#)  
    ignored tests as, [313](#)  
amplification comments, [59](#)  
analysis functions, [265](#)  
“annotation form”, of AspectJ, [166](#)  
Ant project, [76, 77](#)  
AOP (aspect-oriented programming), [160, 163](#)  
APIs. *See also* [public APIs](#)  
    calling a `null`-returning method from, [110](#)  
    specialized for tests, [127](#)  
    wrapping third-party, [108](#)  
applications  
    decoupled from Spring, [164](#)  
    decoupling from construction details, [156](#)  
    infrastructure of, [163](#)  
    keeping concurrency-related code separate, [181](#)  
arbitrary structure, [303–304](#)  
`args` array, converting into a `list`, [231–232](#)  
`Args` class

constructing, [194](#)  
implementation of, [194–200](#)  
rough drafts of, [201–212](#), [226–231](#)

`ArgsException` class  
listing, [198–200](#)  
merging exceptions into, [239–242](#)

argument(s)  
flag, [41](#)  
for a function, [40](#)  
in functions, [288](#)  
monadic forms of, [41](#)  
reducing, [43](#)

argument lists, [43](#)  
argument objects, [43](#)

argument types  
adding, [200](#), [237](#)  
negative impact of, [208](#)

`ArgumentMarshaler` class  
adding the skeleton of, [213–214](#)  
birth of, [212](#)

`ArgumentMarshaler` interface, [197–198](#)

arrays, moving, [279](#)  
art, of clean code, [6–7](#)  
artificial coupling, [293](#)

AspectJ language, [166](#)

aspect-oriented programming (AOP), [160](#), [163](#)

aspects  
in AOP, [160–161](#)  
“first-class” support for, [166](#)

assert statements, [130–131](#)  
`assertEquals`, [42](#)  
assertions, using a set of, [111](#)  
assignments, unaligned, [87–88](#)  
atomic operation, [323–324](#)  
attributes, [68](#)  
    authors of JUnit, [252](#)  
    programmers as, [13–14](#)  
authorship statements, [55](#)  
automated code instrumentation, [189–190](#)  
automated suite, of unit tests, [124](#)

## B

bad code, [3–4](#). *See also* [dirty code](#); [messy code](#)  
    degrading effect of, [250](#)  
    example, [71–72](#)  
    experience of cleaning, [250](#)  
    not making up for, [55](#)  
bad comments, [59–74](#)  
banner, gathering functions beneath, [67](#)  
base classes, [290](#), [291](#)  
BDUF (Big Design Up Front), [167](#)  
beans, private variables manipulated, [100–101](#)  
Beck, Kent, [3](#), [34](#), [71](#), [171](#), [252](#), [289](#), [296](#)  
behaviors, [288–289](#)  
Big Design Up Front (BDUF), [167](#)  
blank lines, in code, [78–79](#)  
blocks, calling functions within, [35](#)  
Booch, Grady, [8–9](#)  
boolean, passing into a function, [41](#)

boolean arguments, [194](#), [288](#)  
boolean map, deleting, [224](#)  
boolean output, of tests, [132](#)  
bound resources, [183](#), [184](#)  
boundaries  
    clean, [120](#)  
    exploring and learning, [116](#)  
    incorrect behavior at, [289](#)  
    separating known from unknown, [118–119](#)  
boundary condition errors, [269](#)  
boundary conditions  
    encapsulating, [304](#)  
    testing, [314](#)  
boundary tests, easing a migration, [118](#) “Bowling Game”, [312](#)  
Boy Scout Rule, [14–15](#), [257](#)  
    following, [284](#)  
    satisfying, [265](#)  
broken windows metaphor, [8](#)  
bucket brigade, [303](#)  
BUILD-OPERATE-CHECK pattern, [127](#)  
builds, [287](#)  
business logic, separating from error handling, [109](#)  
bylines, [68](#)  
byte-manipulation libraries, [161](#), [162–163](#)

## C

*The C++ Programming Language*, [7](#)  
calculations, breaking into intermediate values, [296](#)  
call stack, [324](#)

Callable interface, [326](#)  
caller, cluttering, [104](#)  
calling hierarchy, [106](#)  
calls, avoiding chains of, [98](#)  
caring, for code, [10](#)  
Cartesian points, [42](#)  
CAS operation, as atomic, [328](#)  
change(s)  
    isolating from, [149–150](#)  
    large number of very tiny, [213](#)  
    organizing for, [147–150](#)  
    tests enabling, [124](#)  
change history, deleting, [270](#)  
check exceptions, in Java, [106](#)  
circular wait, [337](#), [338–339](#)  
clarification, comments as, [57](#)  
clarity, [25](#), [26](#)  
class names, [25](#)  
classes  
    cohesion of, [140–141](#)  
    creating for bigger concepts, [28–29](#)  
    declaring instance variables, [81](#)  
    enforcing design and business rules, [115](#)  
    exposing internals of, [294](#)  
    instrumenting into ConTest, [342](#)  
    keeping small, [136](#), [175](#)  
    minimizing the number of, [176](#)  
    naming, [25](#), [138](#)  
    nonthread-safe, [328–329](#)

as nouns of a language, [49](#)  
organization of, [136](#)  
organizing to reduce risk of change, [147](#)  
supporting advanced concurrency design, [183](#)  
classification, of errors, [107](#)  
clean boundaries, [120](#)  
clean code  
    art of, [6–7](#)  
    described, [7–12](#)  
    writing, [6–7](#)  
clean tests, [124–127](#)  
cleanliness  
    acquired sense of, [6–7](#)  
    tied to tests, [9](#)  
cleanup, of code, [14–15](#)  
clever names, [26](#)  
client, using two methods, [330](#)  
client code, connecting to a server, [318](#)  
client-based locking, [185](#), [329](#), [330–332](#)  
`clientScheduler`, [320](#)  
client/server application, concurrency in, [317–321](#)  
Client/Server nonthreaded, code for, [343–346](#)  
client-server using threads, code changes, [346–347](#)  
`ClientTest.java`, [318](#), [344–346](#)  
closing braces, comments on, [67–68](#)  
Clover, [268](#), [269](#)  
clutter  
    Javadocs as, [276](#)  
    keeping free of, [293](#)  
code, [2](#)

bad, [3–4](#)  
Beck’s rules of, [10](#)  
commented-out, [68–69](#), [287](#)  
dead, [292](#)  
explaining yourself in, [55](#)  
expressing yourself in, [54](#)  
formatting of, [76](#)  
implicity of, [18–19](#)  
instrumenting, [188](#), [342](#)  
jiggling, [190](#)  
making readable, [311](#)  
necessity of, [2](#)  
reading from top to bottom, [37](#)  
simplicity of, [18](#), [19](#)  
technique for shrouding, [20](#)  
third-party, [114–115](#)  
width of lines in, [85–90](#)  
at wrong level of abstraction, [290–291](#)  
code bases, dominated by error handling, [103](#)  
code changes, comments not always following, [54](#)  
code completion, automatic, [20](#)  
code coverage analysis, [254–256](#)  
code instrumentation, [188–190](#) “code sense”, [6](#), [7](#)  
code smells, listing of, [285–314](#)  
coding standard, [299](#)  
cohesion  
    of classes, [140–141](#)  
    maintaining, [141–146](#)  
command line arguments, [193–194](#)

commands, separating from queries, [45–46](#)  
comment header standard, [55–56](#)  
comment headers, replacing, [70](#)  
commented-out code, [68–69](#), [287](#)  
commenting style, example of bad, [71–72](#)  
comments  
    amplifying importance of something, [59](#)  
    bad, [59–74](#)  
    deleting, [282](#)  
    as failures, [54](#)  
    good, [55–59](#)  
    heuristics on, [286–287](#)  
    HTML, [69](#)  
    inaccurate, [54](#)  
    informative, [56](#)  
    journal, [63–64](#)  
    legal, [55–56](#)  
    mandated, [63](#)  
    misleading, [63](#)  
    mumbling, [59–60](#)  
    as a necessary evil, [53–59](#)  
    noise, [64–66](#)  
    not making up for bad code, [55](#)  
    obsolete, [286](#)  
    poorly written, [287](#)  
    proper use of, [54](#)  
    redundant, [60–62](#), [272](#), [275](#), [286–287](#)  
    restating the obvious, [64](#)  
    separated from code, [54](#)

TODO, [58–59](#)  
too much information in, [70](#)  
venting in, [65](#)  
writing, [287](#)

“communication gap”, minimizing, [168](#)

Compare and Swap (CAS) operation, [327–328](#)

ComparisonCompactor module, [252–265](#)  
defactored, [256–261](#)  
final, [263–265](#)  
interim, [261–263](#)  
original code, [254–256](#)

compiler warnings, turning off, [289](#)

complex code, demonstrating failures in, [341](#)

complexity, managing, [139–140](#)

computer science (CS) terms, using for names, [27](#)

concepts  
keeping close to each other, [80](#)  
naming, [19](#)  
one word per, [26](#)  
separating at different levels, [290](#)  
spelling similar similarly, [20](#)  
vertical openness between, [78–79](#)

conceptual affinity, of code, [84](#)

concerns  
cross-cutting, [160–161](#)  
separating, [154, 166, 178, 250](#)

concrete classes, [149](#)

concrete details, [149](#)

concrete terms, [94](#)

concurrency

- defense principles, [180–182](#)
- issues, [190](#)
- motives for adopting, [178–179](#)
- myths and misconceptions about, [179–180](#)

concurrency code

- compared to nonconcurrency-related code, [181](#)
- focusing, [321](#)

concurrent algorithms, [179](#)

concurrent applications, partition behavior, [183](#)

concurrent code

- breaking, [329–333](#)
- defending from problems of, [180](#)
- flaws hiding in, [188](#)

concurrent programming, [180](#)

*Concurrent Programming in Java: Design Principles and Patterns*, [182](#),  
[342](#)

concurrent programs, [178](#)

concurrent update problems, [341](#)

`ConcurrentHashMap` implementation, [183](#)

conditionals

- avoiding negative, [302](#)
- encapsulating, [257–258](#), [301](#)

configurable data, [306](#)

configuration constants, [306](#)

consequences, warning of, [58](#)

consistency

- in code, [292](#)
- of enums, [278](#)
- in names, [40](#)

consistent conventions, [259](#)

constants

versus enums, [308–309](#)

hiding, [308](#)

inheriting, [271](#), [307–308](#)

keeping at the appropriate level, [83](#)

leaving as raw numbers, [300](#)

not inheriting, [307–308](#)

passing as symbols, [276](#)

turning into enums, [275–276](#)

construction

moving all to `main`, [155](#), [156](#)

separating with factory, [156](#)

of a system, [154](#)

constructor arguments, [157](#)

constructors, overloading, [25](#)

consumer threads, [184](#)

ConTest tool, [190](#), [342](#)

context

adding meaningful, [27–29](#)

not adding gratuitous, [29–30](#)

providing with exceptions, [107](#)

continuous readers, [184](#)

control variables, within loop statements, [80–81](#)

convenient idioms, [155](#)

convention(s)

following standard, [299–300](#)

over configuration, [164](#)

structure over, [301](#)

using consistent, [259](#)  
convoluted code, [175](#)  
copyright statements, [55](#)  
cosmic-rays. *See* [one-offs](#)  
`CountDownLatch` class, [183](#)  
coupling. *See also* [decoupling](#); [temporal coupling](#); [tight coupling](#)  
    artificial, [293](#)  
    hidden temporal, [302–303](#)  
    lack of, [150](#)  
coverage patterns, testing, [314](#)  
coverage tools, [313](#) “crisp abstraction”, [8–9](#)  
cross-cutting concerns, [160](#)  
Cunningham, Ward, [11–12](#)  
cuteness, in code, [26](#)

## D

dangling `false` argument, [294](#)  
data  
    abstraction, [93–95](#)  
    copies of, [181–182](#)  
    encapsulation, [181](#)  
    limiting the scope of, [181](#)  
    sets processed in parallel, [179](#)  
    types, [97](#), [101](#)  
data structures. *See also* [structure\(s\)](#).  
    compared to objects, [95](#), [97](#)  
    defined, [95](#)  
    interfaces representing, [94](#)  
    treating Active Records as, [101](#)

data transfer-objects (DTOs), [100–101](#), [160](#)  
database normal forms, [48](#)  
`DateInterval` enum, [282–283](#)  
`DAY` enumeration, [277](#)  
`DayDate` class, running `SerialDate` as, [271](#)  
`DayDateFactory`, [273–274](#)  
dead code, [288](#), [292](#)  
dead functions, [288](#)  
deadlock, [183](#), [335–339](#)  
deadly embrace. *See* [circular wait](#)  
debugging, finding deadlocks, [336](#)  
decision making, optimizing, [167–168](#)  
decisions, postponing, [168](#)  
declarations, unaligned, [87–88](#)  
DECORATOR objects, [164](#)  
DECORATOR pattern, [274](#)  
decoupled architecture, [167](#)  
decoupling, from construction details, [156](#)  
decoupling strategy, concurrency as, [178](#)  
default constructor, deleting, [276](#)  
degradation, preventing, [14](#)  
deletions, as the majority of changes, [250](#)  
density, vertical in code, [79–80](#)  
dependencies  
    finding and breaking, [250](#)  
    injecting, [157](#)  
    logical, [282](#)  
    making logical physical, [298–299](#)  
    between methods, [329–333](#)  
    between synchronized methods, [185](#)

Dependency Injection (DI), [157](#)  
Dependency Inversion Principle (DIP), [15](#), [150](#)  
dependency magnet, [47](#)  
dependent functions, formatting, [82–83](#)  
derivatives  
    base classes depending on, [291](#)  
    base classes knowing about, [273](#)  
    of the exception class, [48](#)  
    moving `set` functions into, [232](#), [233–235](#)  
    pushing functionality into, [217](#)  
description  
    of a class, [138](#)  
    overloading the structure of code into, [310](#)  
descriptive names  
    choosing, [309–310](#)  
    using, [39–40](#)  
design(s)  
    of concurrent algorithms, [179](#)  
    minimally coupled, [167](#)  
    principles of, [15](#)  
design patterns, [290](#)  
details, paying attention to, [8](#)  
DI (Dependency Injection), [157](#)  
Dijkstra, Edsger, [48](#)  
dining philosophers execution model, [184–185](#)  
DIP (Dependency Inversion Principle), [15](#), [150](#)  
dirty code. *See also* [bad code](#); [messy code](#)  
dirty code, cleaning, [200](#)  
dirty tests, [123](#)

disinformation, avoiding, [19–20](#)  
distance, vertical in code, [80–84](#)  
distinctions, making meaningful, [20–21](#)  
domain-specific languages (DSLs), [168–169](#)  
domain-specific testing language, [127](#)  
`DoubleArgumentMarshaler` class, [238](#)  
DRY principle (Don't Repeat Yourself), [181](#), [289](#)  
DTOs (data transfer objects), [100–101](#), [160](#)  
dummy scopes, [90](#)  
duplicate `if` statements, [276](#)  
duplication  
    of code, [48](#)  
    in code, [289–290](#)  
    eliminating, [173–175](#)  
    focusing on, [10](#)  
    forms of, [173](#), [290](#)  
    reduction of, [48](#)  
    strategies for eliminating, [48](#)  
dyadic argument, [40](#)  
dyadic functions, [42](#)  
dynamic proxies, [161](#)

## E

`e`, as a variable name, [22](#)  
Eclipse, [26](#)  
edit sessions, playing back, [13–14](#)  
efficiency, of code, [7](#)  
EJB architecture, early as over-engineered, [167](#)  
EJB standard, complete overhaul of, [164](#)  
EJB2 beans, [160](#)

EJB3, Bank object rewritten in, [165–166](#)  
“elegant” code, [7](#)  
emergent design, [171–176](#)  
encapsulation, [136](#)  
    of boundary conditions, [304](#)  
    breaking, [106–107](#)  
    of conditionals, [301](#)  
encodings, avoiding, [23–24, 312–313](#)  
entity bean, [158–160](#)  
enum(s)  
    changing MonthConstants to, [272](#)  
    using, [308–309](#)  
enumeration, moving, [277](#)  
environment, heuristics on, [287](#)  
environment control system, [128–129](#)  
envying, the scope of a class, [293](#)  
error check, hiding a side effect, [258](#)  
Error class, [47–48](#)  
error code constants, [198–200](#)  
error codes  
    implying a class or enum, [47–48](#)  
    preferring exceptions to, [46](#)  
    returning, [103–104](#)  
    reusing old, [48](#)  
    separating from the `Args` module, [242–250](#)  
error detection, pushing to the edges, [109](#)  
error flags, [103–104](#)  
error handling, [8, 47–48](#)  
error messages, [107, 250](#)  
error processing, testing, [238–239](#)

errorMessage method, [250](#)  
errors. *See also* [boundary condition errors](#); [spelling errors](#); [string comparison errors classifying](#), [107](#)

Evans, Eric, [311](#)

events, [41](#)

exception classification, [107](#)

exception clauses, [107–108](#)

exception management code, [223](#)

exceptions

- instead of return codes, [103–105](#)
- narrowing the type of, [105–106](#)
- preferring to error codes, [46](#)
- providing context with, [107](#)
- separating from Args, [242–250](#)
- throwing, [104–105](#), [194](#)
- unchecked, [106–107](#)

execution, possible paths of, [321–326](#)

execution models, [183–185](#)

Executor framework, [326–327](#)

ExecutorClientScheduler.java, [321](#)

explanation, of intent, [56–57](#)

explanatory variables, [296–297](#)

explicitness, of code, [19](#)

expressive code, [295](#)

expressiveness

- in code, [10–11](#)
- ensuring, [175–176](#)

Extract Method refactoring, [11](#)

*Extreme Programming Adventures in C#*, [10](#)

*Extreme Programming Installed*, [10](#)

“eye-full”, code fitting into, [79–80](#)

## F

factories, [155–156](#)

factory classes, [273–275](#)

failure

    to express ourselves in code, [54](#)

    patterns of, [314](#)

    tolerating with no harm, [330](#)

false argument, [294](#)

fast tests, [132](#)

fast-running threads, starving longer running, [183](#)

fear, of renaming, [30](#)

Feathers, Michael, [10](#)

feature envy

    eliminating, [293–294](#)

    smelling of, [278](#)

file size, in Java, [76](#)

final keywords, [276](#)

F.I.R.S.T. acronym, [132–133](#)

First Law, of TDD, [122](#)

FitNesse project

    coding style for, [90](#)

    file sizes, [76, 77](#)

    function in, [32–33](#)

    invoking all tests, [224](#)

flag arguments, [41, 288](#)

focussed code, [8](#)

foreign code. *See* [third-party code](#)

formatting

horizontal, [85–90](#)

purpose of, [76](#)

Uncle Bob's rules, [90–92](#)

vertical, [76–85](#)

formatting style, for a team of developers, [90](#)

Fortran, forcing encodings, [23](#)

Fowler, Martin, [285, 293](#)

frame, [324](#)

function arguments, [40–45](#)

function call dependencies, [84–85](#)

function headers, [70](#)

function signature, [45](#)

functionality, placement of, [295–296](#)

functions

breaking into smaller, [141–146](#)

calling within a block, [35](#)

dead, [288](#)

defining private, [292](#)

descending one level of abstraction, [304–306](#)

doing one thing, [35–36, 302](#)

dyadic, [42](#)

eliminating extraneous `if` statements, [262](#)

establishing the temporal nature of, [260](#)

formatting dependent, [82–83](#)

gathering beneath a banner, [67](#)

heuristics on, [288](#)

intention-revealing, [19](#)

keeping small, [175](#)

length of, [34–35](#)  
moving, [279](#)  
naming, [39](#), [297](#)  
number of arguments in, [288](#)  
one level of abstraction per, [36–37](#)  
in place of comments, [67](#)  
renaming for clarity, [258](#)  
rewriting for clarity, [258–259](#)  
sections within, [36](#)  
small as better, [34](#)  
structured programming with, [49](#)  
understanding, [297–298](#)  
as verbs of a language, [49](#)  
writing, [49](#)  
futures, [326](#)

## G

Gamma, Eric, [252](#)  
general heuristics, [288–307](#)  
generated byte-code, [180](#)  
generics, improving code readability, [115](#)  
get functions, [218](#)  
getBoolean function, [224](#)  
GETFIELD instruction, [325](#), [326](#)  
getNextId method, [326](#)  
getState function, [129](#)  
Gilbert, David, [267](#), [268](#)  
given-when-then convention, [130](#)  
glitches. *See* [one-offs](#)

global setup strategy, [155](#)  
“God class”, [136–137](#)  
good comments, [55–59](#)  
goto statements, avoiding, [48](#), [49](#)  
grand redesign, [5](#)  
gratuitous context, [29–30](#)

## H

hand-coded instrumentation, [189](#)  
`HashTable`, [328–329](#)  
headers. *See* [comment headers](#); [function headers](#)  
heuristics  
    cross references of, [286](#), [409](#)  
    general, [288–307](#)  
    listing of, [285–314](#)  
hidden temporal coupling, [259](#), [302–303](#)  
hidden things, in a function, [44](#)  
hiding  
    implementation, [94](#)  
    structures, [99](#)  
hierarchy of scopes, [88](#)  
HN. *See* [Hungarian Notation](#)  
horizontal alignment, of code, [87–88](#)  
horizontal formatting, [85–90](#)  
horizontal white space, [86](#)  
HTML, in source code, [69](#)  
Hungarian Notation (HN), [23–24](#), [295](#)  
Hunt, Andy, [8](#), [289](#)  
hybrid structures, [99](#)

# I

`if` statements

    duplicate, [276](#)

    eliminating, [262](#)

`if-else` chain

    appearing again and again, [290](#)

    eliminating, [233](#)

ignored tests, [313](#)

implementation

    duplication of, [173](#)

    encoding, [24](#)

    exposing, [94](#)

    hiding, [94](#)

    wrapping an abstraction, [11](#)

*Implementation Patterns*, [3](#), [296](#)

implicity, of code, [18](#)

import lists

    avoiding long, [307](#)

    shortening in `SerialDate`, [270](#)

imports, as hard dependencies, [307](#)

imprecision, in code, [301](#)

inaccurate comments, [54](#)

inappropriate information, in comments, [286](#)

inappropriate static methods, [296](#)

`include` method, [48](#)

inconsistency, in code, [292](#)

inconsistent spellings, [20](#)

incrementalism, [212–214](#)

indent level, of a function, [35](#)

indentation, of code, [88–89](#)  
indentation rules, [89](#)  
independent tests, [132](#)  
information  
    inappropriate, [286](#)  
    too much, [70](#), [291–292](#)  
informative comments, [56](#)  
inheritance hierarchy, [308](#)  
inobvious connection, between a comment and code, [70](#)  
input arguments, [41](#)  
instance variables  
    in classes, [140](#)  
    declaring, [81](#)  
    hiding the declaration of, [81–82](#)  
    passing as function arguments, [231](#)  
    proliferation of, [140](#)  
instrumented classes, [342](#)  
insufficient tests, [313](#)  
integer argument(s)  
    defining, [194](#)  
    integrating, [224–225](#)  
    integer argument functionality, moving into `ArgumentMarshaler`,  
[215–216](#)  
    integer argument type, adding to `Args`, [212](#)  
    integers, pattern of changes for, [220](#)  
IntelliJ, [26](#)  
intent  
    explaining in code, [55](#)  
    explanation of, [56–57](#)  
    obscured, [295](#)

intention-revealing function, [19](#)  
intention-revealing names, [18–19](#)  
interface(s)  
    defining local or remote, [158–160](#)  
    encoding, [24](#)  
    implementing, [149–150](#)  
    representing abstract concerns, [150](#)  
    turning `ArgumentMarshaler` into, [237](#)  
    well-defined, [291–292](#)  
    writing, [119](#)  
internal structures, objects hiding, [97](#)  
intersection, of domains, [160](#)  
intuition, not relying on, [289](#)  
inventor of C++, [7](#)  
Inversion of Control (IoC), [157](#)  
`InvocationHandler` object, [162](#)  
I/O bound, [318](#)  
isolating, from change, [149–150](#)  
`isxxxxArg` methods, [221–222](#)  
iterative process, refactoring as, [265](#)

## J

jar files, deploying derivatives and bases in, [291](#)  
Java

    aspects or aspect-like mechanisms, [161–166](#)  
    heuristics on, [307–309](#)  
        as a wordy language, [200](#)

Java 5, improvements for concurrent development, [182–183](#)  
Java 5 Executor framework, [320–321](#)

Java 5 VM, nonblocking solutions in, [327–328](#)  
Java AOP frameworks, [163–166](#)  
Java programmers, encoding not needed, [24](#)  
Java proxies, [161–163](#)  
Java source files, [76–77](#)  
javadocs  
    as clutter, [276](#)  
    in nonpublic code, [71](#)  
    preserving formatting in, [270](#)  
    in public APIs, [59](#)  
    requiring for every function, [63](#)  
`java.util.concurrent` package, collections in, [182–183](#)  
JBoss AOP, proxies in, [163](#)  
JCommon library, [267](#)  
JCommon unit tests, [270](#)  
JDepend project, [76, 77](#)  
JDK proxy, providing persistence support, [161–163](#)  
Jeffries, Ron, [10–11, 289](#)  
jiggling strategies, [190](#)  
JNDI lookups, [157](#)  
journal comments, [63–64](#)  
JUnit, [34](#)  
JUnit framework, [252–265](#)  
Junit project, [76, 77](#)  
Just-In-Time Compiler, [180](#)

## K

keyword form, of a function name, [43](#)

## L

- l, lower-case in variable names, [20](#)
- language design, art of programming as, [49](#)
- languages
  - appearing to be simple, [12](#)
  - level of abstraction, [2](#)
  - multiple in one source file, [288](#)
  - multiples in a comment, [270](#)
- last-in, first-out (LIFO) data structure, operand stack as, [324](#)
- Law of Demeter, [97–98](#), [306](#)
- LAZY INITIALIZATION/EVALUATION idiom, [154](#)
- LAZY-INITIALIZATION, [157](#)
- Lea, Doug, [182](#), [342](#)
- learning tests, [116](#), [118](#)
- LeBlanc's law, [4](#)
- legacy code, [307](#)
- legal comments, [55–56](#)
- level of abstraction, [36–37](#)
- levels of detail, [99](#)
- lexicon, having a consistent, [26](#)
- lines of code
  - duplicating, [173](#)
  - width of, [85](#)
- list(s)
  - of arguments, [43](#)
  - meaning specific to programmers, [19](#)
  - returning a predefined immutable, [110](#)
- literate code, [9](#)
- literate programming, [9](#)
- Literate Programming*, [141](#)

livelock, [183](#), [338](#)  
local comments, [69–70](#)  
local variables, [324](#)  
    declaring, [292](#)  
    at the top of each function, [80](#)  
lock & wait, [337](#), [338](#)  
locks, introducing, [185](#)  
`log4j` package, [116–118](#)  
logical dependencies, [282](#), [298–299](#)  
LOGO language, [36](#)  
long descriptive names, [39](#)  
long names, for long scopes, [312](#)  
loop counters, single-letter names for, [25](#)

## M

magic numbers  
    obscuring intent, [295](#)  
    replacing with named constants, [300–301](#)  
main function, moving construction to, [155](#), [156](#)  
managers, role of, [6](#)  
mandated comments, [63](#)  
manual control, over a serial ID, [272](#)  
Map  
    adding for `ArgumentMarshaler`, [221](#)  
    methods of, [114](#)  
maps, breaking the use of, [222–223](#)  
marshalling implementation, [214–215](#)  
meaningful context, [27–29](#)  
member variables  
    `f` prefix for, [257](#)

prefixing, [24](#)  
renaming for clarity, [259](#)  
mental mapping, avoiding, [25](#)  
messy code. *See also* [bad code](#); [dirty code](#) total cost of owning, [4–12](#)  
method invocations, [324](#)  
method names, [25](#)  
methods  
    affecting the order of execution, [188](#)  
    calling a twin with a flag, [278](#)  
    changing from static to instance, [280](#)  
    of classes, [140](#)  
    dependencies between, [329–333](#)  
    eliminating duplication between, [173–174](#)  
    minimizing assert statements in, [176](#)  
    naming, [25](#)  
    tests exposing bugs in, [269](#)  
minimal code, [9](#)  
misleading comments, [63](#)  
misplaced responsibility, [295–296](#), [299](#)  
MOCK OBJECT, assigning, [155](#)  
monadic argument, [40](#)  
monadic forms, of arguments, [41](#)  
monads, converting dyads into, [42](#)  
Monte Carlo testing, [341](#)  
Month enum, [278](#)  
MonthConstants class, [271](#)  
multithread aware, [332](#)  
multithread-calculation, of throughput, [335](#)  
multithreaded code, [188](#), [339–342](#)  
mumbling, [59–60](#)

mutators, naming, [25](#)  
mutual exclusion, [183](#), [336](#), [337](#)

## N

named constants, replacing magic numbers, [300–301](#)

name-length-challenged languages, [23](#)

names

- abstractions, appropriate level of, [311](#)
- changing, [40](#)
- choosing, [175](#), [309–310](#)
- of classes, [270–271](#)
- clever, [26](#)
- descriptive, [39–40](#)
- of functions, [297](#)
- heuristics on, [309–313](#)
- importance of, [309–310](#)
- intention-revealing, [18–19](#)
- length of corresponding to scope, [22–23](#)
- long names for long scopes, [312](#)
- making unambiguous, [258](#)
- problem domain, [27](#)
- pronounceable, [21–22](#)
- rules for creating, [18–30](#)
- searchable, [22–23](#)
- shorter generally better than longer, [30](#)
- solution domain, [27](#)
- with subtle differences, [20](#)
- unambiguous, [312](#)
- at the wrong level of abstraction, [271](#)

naming, classes, [138](#)  
naming conventions, as inferior to structures, [301](#)  
navigational methods, in Active Records, [101](#)  
near bugs, testing, [314](#)  
negative conditionals, avoiding, [302](#)  
negatives, [258](#)  
nested structures, [46](#)  
Newkirk, Jim, [116](#)  
newspaper metaphor, [77–78](#)  
niladic argument, [40](#)  
no preemption, [337](#)  
noise  
    comments, [64–66](#)  
    scary, [66](#)  
    words, [21](#)  
nomenclature, using standard, [311–312](#)  
nonblocking solutions, [327–328](#)  
nonconcurrency-related code, [181](#)  
noninformative names, [21](#)  
nonlocal information, [69–70](#)  
nonpublic code, javadocs in, [71](#)  
nonstatic methods, preferred to static, [296](#)  
nonthreaded code, getting working first, [187](#)  
nonthread-safe classes, [328–329](#)  
normal flow, [109](#)  
null  
    not passing into methods, [111–112](#)  
    not returning, [109–110](#)  
    passed by a caller accidentally, [111](#)  
null detection logic, for `ArgumentMarshaler`, [214](#)

NullPointerException, [110](#), [111](#)

number-series naming, [21](#)

## O

*Object Oriented Analysis and Design with Applications*, [8](#)

object-oriented design, [15](#)

objects

compared to data structures, [95](#), [97](#)

compared to data types and procedures, [101](#)

copying read-only, [181](#)

defined, [95](#)

obscured intent, [295](#)

obsolete comments, [286](#)

obvious behavior, [288–289](#)

obvious code, [12](#)

“Once and only once” principle, [289](#)

“ONE SWITCH” rule, [299](#)

one thing, functions doing, [35–36](#), [302](#)

one-offs, [180](#), [187](#), [191](#)

OO code, [97](#)

OO design, [139](#)

Open Closed Principle (OCP), [15](#), [38](#)

by checked exceptions, [106](#)

supporting, [149](#)

operand stack, [324](#)

operating systems, threading policies, [188](#)

operators, precedence of, [86](#)

optimistic locking, [327](#)

optimizations, LAZY-EVALUATION as, [157](#)

optimizing, decision making, [167–168](#)

orderings, calculating the possible, [322–323](#)  
organization

- for change, [147–150](#)
- of classes, [136](#)
- managing complexity, [139–140](#)

outbound tests, exercising an interface, [118](#)

output arguments, [41](#), [288](#)

- avoiding, [45](#)
- need for disappearing, [45](#)

outputs, arguments as, [45](#)

overhead, incurred by concurrency, [179](#)

overloading, of code with description, [310](#)

## P

paperback model, as an academic model, [27](#)

parameters, taken by instructions, [324](#)

`parse` operation, throwing an exception, [220](#)

partitioning, [250](#)

paths of execution, [321–326](#)

pathways, through critical sections, [188](#)

pattern names, using standard, [175](#)

patterns

- of failure, [314](#)
- as one kind of standard, [311](#)

performance

- of a client/server pair, [318](#)
- concurrency improving, [179](#)
- of server-based locking, [333](#)

permutations, calculating, [323](#)

persistence, [160](#), [161](#)  
pessimistic locking, [327](#)  
phraseology, in similar names, [40](#)  
physicalizing, a dependency, [299](#)  
Plain-Old Java Objects. *See* [POJOs](#) platforms, running threaded code, [188](#)  
pleasing code, [7](#)  
pluggable thread-based code, [187](#)  
POJO system, agility provided by, [168](#)  
POJOs (Plain-Old Java Objects)  
    creating, [187](#)  
    implementing business logic, [162](#)  
    separating threaded-aware code, [190](#)  
    in Spring, [163](#)  
    writing application domain logic, [166](#)  
polyadic argument, [40](#)  
polymorphic behavior, of functions, [296](#)  
polymorphic changes, [96–97](#)  
polymorphism, [37](#), [299](#)  
position markers, [67](#)  
positives  
    as easier to understand, [258](#)  
    expressing conditionals as, [302](#)  
    of decisions, [301](#)precision  
        as the point of all naming, [30](#)  
predicates, naming, [25](#)  
preemption, breaking, [338](#)  
prefixes  
    for member variables, [24](#)  
    as useless in today's environments, [312–313](#)

pre-increment operator, `++`, [324](#), [325](#), [326](#)  
“prequel”, this book as, [15](#)  
principle of least surprise, [288–289](#), [295](#)  
principles, of design, [15](#)  
`PrintPrimes` program, translation into Java, [141](#)  
private behavior, isolating, [148–149](#)  
private functions, [292](#)  
private method behavior, [147](#)  
problem domain names, [27](#)  
procedural code, [97](#)  
procedural shape example, [95–96](#)  
procedures, compared to objects, [101](#)  
process function, repartitioning, [319–320](#)  
`process` method, I/O bound, [319](#)  
processes, competing for resources, [184](#)  
processor bound, code as, [318](#)  
producer consumer execution model, [184](#)  
producer threads, [184](#)  
production environment, [127–130](#)  
productivity, decreased by messy code, [4](#)  
professional programmer, [25](#)  
professional review, of code, [268](#)  
programmers  
    as authors, [13–14](#)  
    conundrum faced by, [6](#)  
    responsibility for messes, [5–6](#)  
    unprofessional, [5–6](#)  
programming  
    defined, [2](#)  
    structured, [48–49](#)

programs, getting them to work, [201](#)  
pronounceable names, [21–22](#)  
protected variables, avoiding, [80](#)  
proxies, drawbacks of, [163](#)  
public APIs, javadocs in, [59](#)  
puns, avoiding, [26–27](#)  
`PUTFIELD` instruction, as atomic, [325](#)

## Q

queries, separating from commands, [45–46](#)

## R

random jiggling, tests running, [190](#)  
range, including end-point dates in, [276](#)  
readability  
    of clean tests, [124](#)  
    of code, [76](#)  
    Dave Thomas on, [9](#)  
    improving using generics, [115](#)

readability perspective, [8](#)

readers  
    of code, [13–14](#)  
    continuous, [184](#)

readers-writers execution model, [184](#)

reading  
    clean code, [8](#)  
    code from top to bottom, [37](#)  
    versus writing, [14](#)  
reboots, as a lock up solution, [331](#)

recommendations, in this book, [13](#)  
redesign, demanded by the team, [5](#)  
redundancy, of noise words, [21](#)  
redundant comments, [60–62](#), [272](#), [275](#), [286–287](#)  
`ReentrantLock` class, [183](#)  
refactored programs, as longer, [146](#)  
refactoring  
    Args, [212](#)  
    code incrementally, [172](#)  
    as an iterative process, [265](#)  
    putting things in to take out, [233](#)  
    test code, [127](#)  
*Refactoring* (Fowler), [285](#)  
renaming, fear of, [30](#)  
repeatability, of concurrency bugs, [180](#)  
repeatable tests, [132](#)  
requirements, specifying, [2](#)  
`resetId`, byte-code generated for, [324–325](#)  
resources  
    bound, [183](#)  
    processes competing for, [184](#)  
    threads agreeing on a global ordering of, [338](#)  
responsibilities  
    counting in classes, [136](#)  
    definition of, [138](#)  
    identifying, [139](#)  
    misplaced, [295–296](#), [299](#)  
    splitting a program into main, [146](#)  
return codes, using exceptions instead, [103–105](#)

reuse, [174](#)  
risk of change, reducing, [147](#)  
robust clear code, writing, [112](#)  
rough drafts, writing, [200](#)  
`runnable` interface, [326](#)  
run-on expressions, [295](#)  
run-on journal entries, [63–64](#)  
runtime logic, separating startup from, [154](#)

## S

safety mechanisms, overridden, [289](#)  
scaling up, [157–161](#)  
scary noise, [66](#)  
schema, of a class, [194](#)  
schools of thought, about clean code, [12–13](#)  
scissors rule, in C++, [81](#)  
scope(s)

- defined by exceptions, [105](#)
- dummy, [90](#)
- envying, [293](#)
- expanding and indenting, [89](#)
- hierarchy in a source file, [88](#)
- limiting for data, [181](#)
- names related to the length of, [22–23](#), [312](#)
- of shared variables, [333](#)

  
searchable names, [22–23](#)  
Second Law, of TDD, [122](#)  
sections, within functions, [36](#)  
selector arguments, avoiding, [294–295](#)

self validating tests, [132](#)  
Semaphore class, [183](#)  
semicolon, making visible, [90](#)  
“serial number”, `SerialDate` using, [271](#)  
`SerialDate` class  
    making it right, [270–284](#)  
    naming of, [270–271](#)  
    refactoring, [267–284](#)  
`SerialDateTests` class, [268](#)  
serialization, [272](#)  
server, threads created by, [319–321](#)  
server application, [317–318](#), [343–344](#)  
server code, responsibilities of, [319](#)  
server-based locking, [329](#)  
    as preferred, [332–333](#)  
    with synchronized methods, [185](#)  
“Servlet” model, of Web applications, [178](#)  
`Servlets`, synchronization problems, [182](#)  
`set` functions, moving into appropriate derivatives, [232](#), [233–235](#)  
`setArgument`, changing, [232–233](#)  
`setBoolean` function, [217](#)  
setter methods, injecting dependencies, [157](#)  
setup strategy, [155](#)  
`SetupTeardownIncluder.java` listing, [50–52](#)  
shape classes, [95–96](#)  
shared data, limiting access, [181](#)  
shared variables  
    method updating, [328](#)  
    reducing the scope of, [333](#)  
shotgun approach, hand-coded instrumentation as, [189](#)

shut-down code, [186](#)  
shutdowns, graceful, [186](#)  
side effects  
    having none, [44](#)  
    names describing, [313](#)  
Simmons, Robert, [276](#)  
simple code, [10](#), [12](#)  
Simple Design, rules of, [171–176](#)  
simplicity, of code, [18](#), [19](#)  
single assert rule, [130–131](#)  
single concepts, in each test function, [131–132](#)  
Single Responsibility Principle (SRP), [15](#), [138–140](#)  
    applying, [321](#)  
    breaking, [155](#)  
    as a concurrency defense principle, [181](#)  
    recognizing violations of, [174](#)  
    server violating, [320](#)  
    SQL class violating, [147](#)  
    supporting, [157](#)  
    in test classes conforming to, [172](#)  
    violating, [38](#)  
single value, ordered components of, [42](#)  
single-letter names, [22](#), [25](#)  
single-thread calculation, of throughput, [334](#)  
SINGLETON pattern, [274](#)  
small classes, [136](#)  
*Smalltalk Best Practice Patterns*, [296](#)  
smart programmer, [25](#)  
software project, maintenance of, [175](#)  
software systems. *See also* [system\(s\)](#).

compared to physical systems, [158](#)  
SOLID class design principle, [150](#)  
solution domain names, [27](#)  
source code control systems, [64](#), [68](#), [69](#)  
source files  
    compared to newspaper articles, [77–78](#)  
    multiple languages in, [288](#)  
*Sparkle* program, [34](#)  
spawned threads, deadlocked, [186](#)  
special case objects, [110](#)  
SPECIAL CASE PATTERN, [109](#)  
specifications, purpose of, [2](#)  
spelling errors, correcting, [20](#)  
`SpreadsheetDateFactory`, [274–275](#)  
Spring AOP, proxies in, [163](#)  
Spring Framework, [157](#)  
Spring model, following EJB3, [165](#)  
Spring V2.5 configuration file, [163–164](#)  
spurious failures, [187](#)  
`sql` class, changing, [147–149](#)  
square root, as the iteration limit, [74](#)  
SRP. *See* [Single Responsibility Principle](#)  
standard conventions, [299–300](#)  
standard nomenclature, [175](#), [311–312](#)  
standards, using wisely, [168](#)  
startup process, separating from runtime logic, [154](#)  
starvation, [183](#), [184](#), [338](#)  
static function, [279](#)  
static import, [308](#)  
static methods, inappropriate, [296](#)

*The Step-down Rule*, [37](#)

stories, implementing only today's, [158](#)

STRATEGY pattern, [290](#)

string arguments, [194](#), [208–212](#), [214–225](#)

string comparison errors, [252](#)

StringBuffers, [129](#)

Stroustrup, Bjarne, [7–8](#)

structure(s). *See also* [data structures](#)

    hiding, [99](#)

    hybrid, [99](#)

    making massive changes to, [212](#)

    over convention, [301](#)

structured programming, [48–49](#)

SuperDashboard class, [136–137](#)

swapping, as permutations, [323](#)

switch statements

    burying, [37](#), [38](#)

    considering polymorphism before, [299](#)

    reasons to tolerate, [38–39](#)

switch/case chain, [290](#)

synchronization problems, avoiding with servlets, [182](#)

synchronized block, [334](#)

synchronized keyword, [185](#)

    adding, [323](#)

    always acquiring a lock, [328](#)

    introducing a lock via, [331](#)

    protecting a critical section in code, [181](#)

synchronized methods, [185](#)

synchronizing, avoiding, [182](#)

synthesis functions, [265](#)

system(s). *See also* [software systems](#)

- file sizes of significant, [77](#)
- keeping running during development, [213](#)
- needing domain-specific, [168](#)

system architecture, test driving, [166–167](#)

system failures, not ignoring one-offs, [187](#)

system level, staying clean at, [154](#)

system-wide information, in a local comment, [69–70](#)

## T

tables, moving, [275](#)

target deployment platforms, running tests on, [341](#)

task swapping, encouraging, [188](#)

TDD (Test Driven Development), [213](#)

- building logic, [106](#)
- as fundamental discipline, [9](#)
- laws of, [122–123](#)

team rules, [90](#)

teams

- coding standard for every, [299–300](#)
- slowed by messy code, [4](#)

technical names, choosing, [27](#)

technical notes, reserving comments for, [286](#)

TEMPLATE METHOD pattern

- addressing duplication, [290](#)
- removing higher-level duplication, [174–175](#)
- using, [130](#)

temporal coupling. *See also* [coupling](#)

- exposing, [259–260](#)

- hidden, [302–303](#)
- side effect creating, [44](#)
- temporary variables, explaining, [279–281](#)
- test cases
  - adding to check arguments, [237](#)
  - in ComparisonCompactor, [252–254](#)
  - patterns of failure, [269](#), [314](#)
  - turning off, [58](#)
- test code, [124](#), [127](#)
- TEST DOUBLE, assigning, [155](#)
- Test Driven Development. *See* [TDD](#)
- test driving, architecture, [166–167](#)
- test environment, [127–130](#)
- test functions, single concepts in, [131–132](#)
- test implementation, of an interface, [150](#)
- test suite
  - automated, [213](#)
  - of unit tests, [124](#), [268](#)
  - verifying precise behavior, [146](#)
- testable systems, [172](#)
- test-driven development. *See* [TDD](#)
- testing
  - arguments making harder, [40](#)
  - construction logic mixed with runtime, [155](#)
- testing language, domain-specific, [127](#)
- testNG project, [76](#), [77](#)
- tests
  - clean, [124–127](#)
  - cleanliness tied to, [9](#)

commented out for `SerialDate`, [268–270](#)  
dirty, [123](#)  
enabling the `-ilities`, [124](#)  
fast, [132](#)  
fast versus slow, [314](#)  
heuristics on, [313–314](#)  
ignored, [313](#)  
independent, [132](#)  
insufficient, [313](#)  
keeping clean, [123–124](#)  
minimizing assert statements in, [130–131](#)  
not stopping trivial, [313](#)  
refactoring, [126–127](#)  
repeatable, [132](#)  
requiring more than one step, [287](#)  
running, [341](#)  
self validating, [132](#)  
simple design running all, [172](#)  
suite of automated, [213](#)  
timely, [133](#)  
writing for multithreaded code, [339–342](#)  
writing for threaded code, [186–190](#)  
writing good, [122–123](#)

Third Law, of TDD, [122](#)  
third-party code integrating, [116](#)  
    learning, [116](#)  
    using, [114–115](#)  
    writing tests for, [116](#)

this variable, [324](#)

Thomas, Dave, [8](#), [9](#), [289](#)

thread(s)

- adding to a method, [322](#)
- interfering with each other, [330](#)
- making as independent as possible, [182](#)
- stepping on each other, [180](#), [326](#)
- taking resources from other threads, [338](#)

thread management strategy, [320](#)

thread pools, [326](#)

thread-based code, testing, [342](#)

threaded code making pluggable, [187](#)

- making tunable, [187–188](#)
- symptoms of bugs in, [187](#)
- testing, [186–190](#)
- writing in Java [5](#), [182–183](#)

threading

- adding to a client/server application, [319](#), [346–347](#)
- problems in complex systems, [342](#)

thread-safe collections, [182–183](#), [329](#)

throughput

- causing starvation, [184](#)
- improving, [319](#)
- increasing, [333–335](#)
- validating, [318](#)

`throws` clause, [106](#)

tiger team, [5](#)

tight coupling, [172](#)

time, taking to go fast, [6](#)

Time and Money project, [76](#)

file sizes, [77](#)  
timely tests, [133](#)  
timer program, testing, [121–122](#)  
“TO” keyword, [36](#)  
TO paragraphs, [37](#)  
TODO comments, [58–59](#)  
tokens, used as magic numbers, [300](#)  
Tomcat project, [76](#), [77](#)  
tools  
    ConTest tool, [190](#), [342](#)  
    coverage, [313](#)  
    handling proxy boilerplate, [163](#)  
    testing thread-based code, [342](#)  
train wrecks, [98–99](#)  
transformations, as return values, [41](#)  
transitive navigation, avoiding, [306–307](#)  
triadic argument, [40](#)  
triads, [42](#)  
`try` blocks, [105](#)  
`try/catch` blocks, [46–47](#), [65–66](#)  
`try-catch-finally` statement, [105–106](#)  
tunable threaded-based code, [187–188](#)  
type encoding, [24](#)

## U

ubiquitous language, [311–312](#)  
unambiguous names, [312](#)  
unchecked exceptions, [106–107](#)  
unencapsulated conditional, encapsulating, [257](#)  
unit testing, isolated as difficult, [160](#)

unit tests, [124](#), [175](#), [268](#)  
unprofessional programming, [5–6](#)  
uppercase c, in variable names, [20](#)  
usability, of newspapers, [78](#)  
use, of a system, [154](#)  
users, handling concurrently, [179](#)

## V

validation, of throughput, [318](#)  
variable names, single-letter, [25](#)  
variables  
    1 based versus zero based, [261](#)  
    declaring, [80](#), [81](#), [292](#)  
    explaining temporary, [279–281](#)  
    explanatory, [296–297](#)  
    keeping private, [93](#)  
    local, [292](#), [324](#)  
    moving to a different class, [273](#)  
    in place of comments, [67](#)  
    promoting to instance variables of classes, [141](#)  
    with unclear context, [28](#)  
venting, in comments, [65](#)  
verbs, keywords and, [43](#)  
version class, [139](#)  
versions, not deserializing across, [272](#)  
vertical density, in code, [79–80](#)  
vertical distance, in code, [80–84](#)  
vertical formatting, [76–85](#)  
vertical openness, between concepts, [78–79](#)

vertical ordering, in code, [84–85](#)

vertical separation, [292](#)

## W

wading, through bad code, [3](#)

Web containers, decoupling provided by, [178](#)

what, decoupling from when, [178](#)

white space, use of horizontal, [86](#)

wildcards, [307](#)

*Working Effectively with Legacy Code*, [10](#)

“working” programs, [201](#)

workmanship, [176](#)

wrappers, [108](#)

wrapping, [108](#)

writers, starvation of, [184](#)

“Writing Shy Code”, [306](#)

## X

XML

deployment descriptors, [160](#)

“policy” specified configuration files, [164](#)