SUMMARY

CLEAN CODE SUMMARY

AGILE SOFTWARE CRAFTMANSHIP GUIDELINES - **DEVELOPER DECONSTRUCTED**

SUPERGLOO SUMMARY SERIES

Summary Guide Disclaimer

Martin, Robert C. (2008-08-01). Clean Code: A Handbook of Agile Software Craftsmanship . Pearson Education.

partner@supergloo.com

http://www.supergloo.com

Special Offers

For deeper exploration and learning of software engineering, you are also encouraged to visit http://www.supergloo.com/ for free resources and significant discount coupons of online training.

Clean Code Chapter 1

Even if you believe tools will allow all code to be generated automatically someday, it still represents the details of the software requirements. So, code will remain.

The level of abstraction in languages will continue to increase as well as a rise in the number of domain-specific languages. The increase in abstraction is good and will still not eliminate code.

Bad Code

Bad code can bring companies down. While this might seem extreme, any programmer with a few years experience has been impeded by bad code.

Writing bad code can happen for a variety of reasons, but let’s not fool ourselves into believing it will be cleaned up later. We’ve all said we’d go back and clean it up later, but we didn’t know LeBlanc’s law: Later equals never.

The Total Cost of Owning a Mess

Messy code can slow things down. The degree of the slowdown can be significant.

The Grand Redesign in the Sky

What happens if this slowdown eventually leads to management buying into the idea of a complete rewrite?

Two teams emerge: the Tiger team and Maintenance team. The Tiger team chooses only the best and brightest to write the new system while everyone else must continue to maintain the current system. Now the two teams are in a race.

By the time the new system is done, the original members of the Tiger team are long gone, and the current members are demanding that the new system is redesigned because it’s such a mess.

Attitude

How does good code evolve into bad? There are many explanations and excuses for it such as requirements changed, schedules too tight, stupid managers, irate customers, useless marketing types.

We need to stop blaming everyone and everything but ourselves. What if you were a doctor and had a patient with demands. The patient demanded you stop hand-washing in preparation for surgery because it was taking too much time. Although the patient is the boss, the doctor should refuse to comply.

Programmers should avoid bending to the will of managers who don’t understand the risks of making code messes.

The Primal Conundrum

All developers feel the pressure to meet deadlines. But, real professionals know this is wrong. The only way to make the deadline and continue to go fast is to keep the code as clean as possible at all times.

The Art of Clean Code?

How do I write clean code?

Writing clean code is a lot like painting a picture which means the answer to how to write clean code is vague.

Most of us appear to know when a picture is painted well or badly. But being able to recognize good art from bad does not mean that we know how to paint. Similar to being able to recognize clean code vs. messy code does not mean knowing how to write clean code!

Writing clean code requires the use of techniques applied through an acquired code-sense of “cleanliness.”

A programmer who writes clean code is like an artist who creates clean code from blank screen canvas.

What is Clean Code?

The author as well-known and deeply experienced programmers what they thought. Their responses included descriptions such as readability, elegant, detailed, focused, undistracted, *etc*.

Schools of Thought

Within martial artists schools, students and teachers do not all agree about which one is best or even which technique is best within a particular martial art. No school is entirely right. Bias within schools does not invalidate the teachings of a different school. In a similar light, the techniques and teachings of clean code are the way the art of programming is practiced.

The Boy Scout Rule

Code needs to be kept clean over time. Over time, apply the boy scout rule of leaving things better than you found it.

Conclusion

Just like art books can not promise to make you an artist, this book cannot promise to make you a good programmer. This book can give you tools, techniques, and thought processes of other programmers.

Meaningful Names Chapter 2

Simple rules for creating good names:

Use Intention-Revealing Names

The name of a variable, function, or class, should tell you why it exists, what it does, and how it is used. If a name requires a comment, it does not reveal intention.

Avoid Disinformation

Programmers must avoid using names that obscure the meaning of the code. For example, avoid using *hp*, *aix*, and *sco* because they are variants of Unix platforms. Another example is referring to a group of accounts as an *accountList* unless it's actually a *List*.

Make Meaningful Distinctions

Programmers can create problems naming code solely to satisfy a compiler or interpreter. For example, the practice of creating a variable named *klass* just because the name *class* is reserved by the compiler

Number-series naming (a1, a2, .. aN) is not meaningful.

If you have a *Product* class and other classes called *ProductInfo* or *ProductData*, you have made the names noisy because differences in meaning are not clear.

Use Pronounceable Name

Make names pronounceable. A variable name of *genymdhms* (generation date, year, month, day, hour, minute, and second) is not pronounceable.

Use Searchable Names

The common practice of using single-letter names create a particular problem because they are not easy to locate by searching in code.

Avoid Encodings

Encoding names with type indicators such as *String* in a variable *phoneString* is unneeded and creates noise. Type indicator encoding is especially unneeded in languages such as Java because Java is a strongly typed language and the compiler forces correct typing anyhow.

Member Prefixes

Prefixing member variables with conventions such as *m_* is not needed anymore.

Interfaces and Implementations

Preceding and interface with letter I as seen in *IShapeFactory* for example is a distraction.

Avoid Mental Mapping

A problem arises from a choosing to use neither problem domain terms nor solution domain terms. It forces readers to mentally translate names. Example using the letter r to indicate a URL.

Class Names

Classes and objects should noun names and should avoid using verbs such as Manager, Processor, *etc*.

Method Names

Use verbs in method names like *postPayment*, *deletePage*, or *save*.

Don’t Be Cute

If names are too clever, they will be memorable only to people who share the author’s sense of humor, so choose clarity over entertainment value.

Pick One Word per Concept

It’s confusing to have *fetch*, *retrieve*, and *get* as equivalent methods of different classes. Stay consistent with abstract concepts.

Don’t Pun

Using the same name for two different purposes is essentially a pun and should be avoided. Example: using *add* that adds to a collection in one case, while in another, *add* concatenates a string to another string.

Use Solution Domain Names

People who read your code will be programmers, so use computer science (CS) terms, algorithm names, pattern names, math terms, and so forth vs. problem domain names. The name *AccountVisitor* means more to a programmer familiar with the VISITOR pattern.

Use Problem Domain Names

Building on previous rule, use problem domain names when solution domain names are not available.

Add Meaningful Context

If you have variables named *firstName*, *lastName*, *street*, *city*, *state*, and *zipcode*, but saw the *state* variable being used alone in a method? Would it be clear the context is an address?

Don’t Add Gratuitous Context

Add no more context to a name than is necessary and shorter names are usually better than longer ones, if they are clear.

Final Words

Choosing good names a teaching issue rather than a technical, business, or management issue and many programmers don’t learn to do it very well.

Functions Chapter 3

A function is a type of procedure or routine in computer programs. What makes good functions?

Small

- Lines should not be 150 characters long.
- Functions should not be 100 lines long.
- Functions should hardly ever be 20 lines long.

Blocks and Indenting

- Blocks within statements should be one line long.

- The indent level of a function should not be greater than one or two.

Do One Thing

The following advice has appeared in one form or another for over 30 years:

Functions should do only one thing and do it well.

One Level of Abstraction per Function

To make sure our functions are doing “one thing,” we need to make sure that the statements within our function are all at the same level of abstraction.

To determine abstraction level, use the “Step Down” rule. This rule is reading the program as though it were a set of TO paragraphs, each of which is describing the current level of abstraction and referencing subsequent TO paragraphs at the next level down.

Making the code read like a top-down set of TO paragraphs is a useful technique for keeping the abstraction level consistent.

Switch Statements

Switch statements may be tolerated using the following guidelines:

- appear only once
- used to create polymorphic objects
- hidden behind an inheritance relationship so that the rest of the system can’t see them

Use Descriptive Names

Choosing good names rely on small functions that do one thing. And with this kind of function in place, consider the following:

- Don’t be afraid of using long names
- Don’t be afraid of spending time to choose a descriptive name
- Be consistent in naming

Function Arguments

How many arguments should functions allow? The ideal number of arguments is zero (niladic), followed by one (monadic), followed closely by two (dyadic). Avoid three arguments where possible and do not use more than three (polyadic).

Other argument considerations:

- Do not use flag arguments such as booleans because it implies the function doing more than one thing

- Two arguments make the function more complicated to understand and three arguments even more so

- Wrap multiple arguments into a class of their own

Have No Side Effects

Ensure functions have no side effects and especially side effects that include temporal coupling. Temporal coupling creates dependencies between code and timing.

Command Query Separation

Functions should either perform an action or answer a question, but not both.

Prefer Exceptions to Returning Error Codes

And when using exceptions, it’s often preferable to extract try/catch block include into functions.

Don’t Repeat Yourself

Avoid duplication. (For summary version of this principle, check out my summary guide The Pragmatic Programmer Summary on Amazon)

Conclusion

How do you write functions as previously described? Writing functions within these guidelines doesn’t happen right away. It requires iterating over versions.

Comments Chapter 4

Comments are used to address shortcomings in our code. But proceed with caution, because nothing can be more helpful (or cluttering or even damaging) than code comments.

Warning: over time, comments rarely evolve with code because they are not often maintained. Fnd Truth can only be found in the code and not the comments.

Comments Key Factors:

- Comments Do Not Make Up for Bad Code
- Explain Yourself in Code (not comments)

What attributes make good comments?

1) Legal Comments (copyright, all rights reserved, etc.)

2) Informative Comments (example: comment for intention of regular expression matching because regular expression code can be confusing to some people.)

3) Clarification

4) Warning of Consequences

5) TODO comments are sometimes reasonable, and IDEs can help find them

6) Amplification of importance

7) Javadocs

What attributes are found in bad comments?

1) Mumbling (take time to construct comments)

2) Redundant

3) Misleading

4) Mandated (re: rules declaring every function must have a comment)

5) Journaling (redundant with source code control systems)

6) Noise (re: documenting the default constructor when it’s obvious in code)

7) Scary Noise (blatantly incorrect)

8) When the comment could be avoided by better naming or succinctly written code

9) Position markers (example: declaring a section of particular kinds of functions)

10) Closing Brace Comments (re: try to shorten your functions instead)

11) Attributions and Bylines (again, redundant with source code control systems)

12) Commented out code

13) HTML comments. Do not use HTML in comments

14) Nonlocal Information

15) No Obvious Connection (re: the connection between the comment and the code should be obvious)

16) Function Headers (short functions do not need much description)

17) Javadocs in Nonpublic Code

Formatting Chapter 5

We want people to realize and appreciate the neatness, consistency and attention to detail in our code.

Purpose of Code Formatting

In essence, code formatting is about professional communication.

Formatting guidelines include:

Vertical Formatting

1) File size consistency

2) Openness Between Concepts (example: line breaks between the end of one function and beginning of new)

3) Density (tightly related code should appear together)

4) Distance (closely related concepts should vertically close to each other to avoid jumping around)

5) Ordering (generally speaking, function call dependencies should point in the downward direction)

Horizontal Formatting

1) Openness and Density (example: spaces around assignment operators)

2) Alignment (horizontal length can imply need to refactor into more succinct code such as extracting new methods)

3) Indentation (indent to signify hierarchy of scoping; re: methods should be indented to first level)

4) Dummy scores (example: a *while* code block should be broken into lines rather than consolidating into one line)

Team Rules

Teams should agree on formatting rules.

Objects and Data Structures Chapter 6

Data Abstraction

Hiding implementation is about abstractions and not just a matter of putting a layer of functions between the variables. Abstractions through layers allows the manipulation of the essence of data rather than requiring to know and expose the implementation.

Data/Object Anti-Symmetry

The distinction, or anti-symmetry between objects and data structures such as data transfer objects (DTOs): objects hide their data behind abstractions and expose functions to operate on their data while data structures expose their data for convenience and contain no meaningful functions. Avoid mixing these two constructs together into a structure called a Hybrid.

The Law of Demeter

A heuristic called the Law of Demeter says a module should not know about the innards of the objects it manipulates.
http://en.wikipedia.org/wiki/Law_of_Demeter

Conclusion

Objects are best utilized to expose behavior rather than data. Conversely, data structures such as DTOs are designed to expose data rather than behavior.

Error Handling Chapter 7

Techniques and considerations for writing code that handles errors with grace and style include: Use Exceptions Rather Than Return Codes Take advantage of exception handling in languages that support them.

Write Your Try-Catch-Finally Statement First The code in *try* blocks is similar to transactions because the *catch* block can leave a program in a consistent state.   This helps define expectations, no matter what could go wrong during execution.

Use Unchecked Exceptions The debate of unchecked vs. checked exceptions is over.

Provide Context with Exceptions Provide enough context to determine the source and location of the error.

Define Exception Classes in Terms of a Caller's Needs The most important concern should be how exceptions are caught. Also, wrapping code to throw your own exceptions rather than coding to specific third-party API exceptions is good practice.

Define the Normal Flow

Don’t use exceptions which clutters the business logic.

Don’t Return Null

If you are tempted to return null from a method, consider throwing an exception or returning a SPECIAL CASE object instead.

Don’t Pass Null

Returning null from methods is bad, but passing null into methods is worse.

Boundaries Chapter 8

It’s rare to be in control of all software in our systems. Therefore, we need practices for keeping software clean when crossing software boundaries.

Using Third-Party Code

Avoid passing 3rd-party interfaces at boundaries in your system. Use wrapper code instead.

Exploring and Learning Boundaries

Sometimes third-party code can help us get more functionality than writing out own. It may be in our best interest to write tests for the third-party code we use because we can write them to help learn third-party APIs. So, in this case, they cost us nothing.

Using Code That Does Not Yet Exist

To keep from being blocked, explore writing your own interface for working with boundaries of code that does not exist yet.

Clean Boundaries

When using code outside our control, special care must be taken to ensure possible future change is not too costly. Boundary code needs clear separation through wrappers/adapters and tests that define expectations.

Unit Tests Chapter 9

The Agile and Test Driven Development (TDD) movements have encouraged many programmers to write automated unit tests. But, many programmers have missed some of the more subtle, and important, points of writing good tests.

The Three Laws of TDD

1) No writing production code until you have written a failing unit test.

2) No writing more of a unit test than is sufficient to fail. Not compiling is failing.

3) No writing more production code than is sufficient to pass the currently failing test.

Keeping Tests Clean

Do not fall into the trap of having different quality standards for test code vs. production code.

Clean Tests

Three factors to make tests clean: 1) Readability 2) Readability
3) Readability

Characteristics of readable code: clarity, simplicity, and density of expression.

One Assert Per Test

Although it may seem draconian, many believe unit tests should only have one assert per test.

Five Rules of Clean Tests 1) Fast 2) Independent 3) Repeatable 4) Self-Validating (means boolean output) 5) Timely **Final Advice**

Keep test code clean.

Classes Chapter 10

Class Organization Ordering (if present)

1) Public static constants

2) Private static variables

3) Private instance variables

4) Public functions (followed immediately by any private function used by a particular public function; re: follows previously described step down rule)

Encapsulation

Keep variables and utility functions private, but do not be fanatic about it. Make protected if needed by the test in the same package.

Classes Should Be Small!

As already noted with functions, smaller is the primary rule when it comes to designing classes.

The Single Responsibility Principle

The Single Responsibility Principle (SRP) states that a class or module should have only one reason to change. By trying to identify reasons to change (aka: responsibilities) we recognize and create better abstractions in code.

Cohesion

Classes should have a small number of instance variables, and each method of a class should manipulate one or more of those variables. Maintaining this cohesion results in many small classes.

Organizing for Change

One way to organize for change is to support an object oriented design principle called “Open-Closed Principle” or OCP. Design classes to be open for extension but closed for modification.

Also, we isolate from change by utilizing concrete classes of implementation details with abstract concept classes and interfaces.

Systems Chapter 11

Rather than focus on any frameworks such as Enterprise Java Beans (EJB), EJB2, JNDI, Spring and AspectJ, this chapter will concentrate on the essence of what these frameworks are attempting to deliver to keep things more current. Frameworks attempt to systemize software development through separating concerns such as startup, scaling up and out, testing and decision making.

Startup Process

Segment the startup process of starting the application. Or, in other words, separate the concern of starting an application.

Ways to separate concern of startup:

The main Method

main builds the objects necessary for the system and passes them to the application.

Factories

Consider the Abstract Factory pattern that provides interfaces for creating related or dependent objects without specifying their concrete classes.

Dependency Injection (DI)

An object should not take responsibility for instantiating dependencies itself. Instead, the object should delegate this responsibility to another mechanism, thereby inverting the control. DI supports the Single Responsibility Principle (SRP) mentioned earlier in this book.

DI is also known as Inversion of Control (IoC).

Scaling Up

It is a myth that we can get systems “right the first time.” Instead, we should implement only today’s stories, then refactor and expand as needed. System architectures can grow incrementally if the proper separation of concerns is maintained.

Test Drive the System Architecture

The power of separating concerns can not be overstated. It facilitates being able to test drive your architecture and puts you in a position to evolve it from simple to sophisticated. It also allows you adopt new technologies on demand.

Optimize Decision Making

No one person can make all the decisions, so modularity and separation of concerns make decentralized management and decision making possible.

Use Standards Wisely, When They Add Demonstrable Value

Be wary of strongly hyped standards if it distracts your team from implementing value for customers.

Systems Need Domain Specific Languages (DSLs)

DSLs are separate, small scripting languages or APIs in standard languages. They are intended to permit code to be written so that it reads like a domain expert might write.

Conclusion

Systems design should use the simplest thing that can work rather than an invasive architecture.

Emergence Chapter 12

According to Kent Beck, there are four simple rules to follow to facilitate the emergence of good design. A design is simple if it follows these rules:

1) Runs all tests
2) Contains no duplication
3) Expresses the intent of the programmer
4) Minimizes the number classes and methods

Runs All the Tests

A simple way to verify the system works as intended is running comprehensive tests passing all of the time. Making systems testable is a way to lead toward a design where classes are small and follow SRP. Testable systems also provide confidence during refactoring efforts.

No Duplication

Duplication represents additional work, risk, and unnecessary complexity.

Expressive

The majority of the cost of a software project is in long-term maintenance. So, a convoluted code base that is difficult to maintain increases costs. Instead, code should clearly express the intent of its author.

This book has already covered ways to be expressive such as:

1) Good naming
2) Keep functions and classes small
3) Use standard nomenclature
4) well written unit tests
5) maintain an attitude, desire, and effort to be expressive

Concurrency Chapter 13

It is much easier to write code that executes in a single thread than concurrent programs.

Why Concurrency?

Concurrency is a decoupling strategy because it helps us segment what gets done from when it gets done. In single-threaded applications, the what and when are tightly coupled.

Decoupling what from when can improve both throughput and structure, but concurrency is difficult.

Myths and misconceptions

- Concurrency always improves performance
- Design does not change when writing concurrent programs
- Understanding concurrency issues is not important when working with a container such as Web or EJB container

Warnings

- Concurrency incurs some overhead (writing additional code)
- Correct concurrency is complex
- Concurrency bugs are difficult to repeat
- Concurrency often requires a fundamental change in design strategy

What makes concurrency so difficult?

Unlike single-threaded applications, concurrency allows multiple processing paths that can lead to different results.

Concurrency Defense Principles

Techniques to defend your systems from concurrency issues:

1) Single Responsibility Principle

For concurrency, this principle may be applied in the following ways:

- Concurrency code could have independent life cycle, change and tuning development paths
- Concurrency has unique challenges
- Keep concurrent code separated from other code

2) Limit Data Scope

3) Use copies of data

Copy objects and treating them as read-only when possible.

4) Threads should be as independent as possible

Example: perhaps a thread doesn’t share data with any other thread

5) Know Your Library

Is your choice of collection class, 3rd-party library, *etc.* thread-safe?

6) Know Your Execution Model

Models include bound resources, mutual exclusion, starvation, deadlock, and livelock.

Known Concurrent Execution Model Issues

- Producer-Consumer - http://en.wikipedia.org/wiki/Producer-consumer
- Readers-Writers - https://en.wikipedia.org/wiki/Readers-writers_problem
- Dining Philosophers - https://en.wikipedia.org/wiki/Dining_philosophers_problem

7) Beware Dependencies Between Synchronized Methods

When you must use more than one method on a shared object, three ways to make the code correct:

- Client-Based Locking
- Server-Based Locking
- Adapted Server (an intermediary to perform locking)

8) Keep Synchronized Sections Small

9) Writing Correct Shut-down Code is Hard

10) Write Tests With Potential to Expose Threaded Code

11) Treat Spurious Failures as Potential Threading Issues

12) Get Non-threaded Code Working First

13) Make Your Threaded Code Pluggable

Possible examples include allowing configuration of threaded code such as being able to set the number of threads to use and configurable setting of number test iterations.

14) Make Your Threaded Code Tunable

Getting the correct balance of threads usually involves some trial-and-error.

15) Make Sure to Experiment

Examples include running with more threads than processor cores and running on different platforms.

16) Instrument Code and Try to Force Failures

Successive Refinement Chapter 14

The author walks through a Java code case study with the primary points being clean code takes iterations. It's not enough to get code working. Programmers (and their managers) must budget time to iterate over working code to make improvements.

JUnit Internals Chapter 15

Another Java case study is analyzed. This chapter concentrates on the source code of the prominent JUnit testing library. Similar to the previous chapter, the emphasis seems to be refactoring and applying principles and recommendations described in earlier chapters.

Refactoring SerialDate Chapter 16

A final Java case study in which the author performs a code review of SerialDate found in the JCommon library. Again, the takeaway is applying principles and recommendations described in earlier chapters of this book.

Smells and Heuristics Chapter 17

The following list of software smells and heuristics is from both the author and Martin Fowler’s book “Refactoring”. The list is divided into categories such as code comments, environment, *etc.*

Code Comments

1) Inappropriate Information

2) Obsolete

3) Redundant

4) Poorly Written

5) Commented-Out Code

Environment

1) Build Requires More Than One Step

2) Tests Require More Than One Step

Functions

1) Too Many Arguments

2) Output Arguments

3) Flag Arguments (booleans)

4) Dead Functions (unused code)

General

1) Multiple Languages in One Source File
2) Obvious Behavior Is Unimplemented
3) Incorrect Behavior at the Boundaries
4) Overridden Safeties (i.e. overriding serialVersionUID in Java)
5) Duplication
6) Code at Wrong Level of Abstraction
7) Base Classes Depending on Their Derivatives
8) Too Much Information
9) Dead Code
10) Vertical Separation
11) Inconsistency
12) Clutter
13) Artificial Coupling
14) Feature Envy (classes should be interested in what they have rather than

other classes)

15) Selector Arguments

16) Obscured Intent

17) Misplaced Responsibility

18) Inappropriate Static

19) Use Explanatory Variables

20) Function Names Should Say What They Do

21) Understand the Algorithm

22) Make Logical Dependencies Physical

23) Prefer Polymorphism to If/ Else or Switch/ Case

24) Follow Standard Conventions

25) Replace Magic Numbers with Named Constants

26) Be Precise

27) Structure over Convention

28) Encapsulate Conditionals

29) Avoid Negative Conditionals

30) Functions Should Do One Thing

31) Hidden Temporal Couplings

32) Don’t Be Arbitrary

33) Encapsulate Boundary Conditions

34) Functions Should Descend Only One Level of Abstraction

35) Keep Configurable Data at High Levels

36) Avoid Transitive Navigation

Java

1) Avoid Long Import Lists by Using Wildcards

2) Don’t Inherit Constant

3) Constants versus Enums (don’t use enums)

Names

1) Choose Descriptive Names

2) Choose Names at the Appropriate Level of Abstraction

3) Use Standard Nomenclature Where Possible

4) Unambiguous Names

5) Use Long Names for Long Scopes

6) Avoid Encodings (prefixes such as m_)

7) Names Should Describe Side-Effects

Tests

1) Insufficient Tests
2) Use a Coverage Tool!
3) Don’t Skip Trivial Tests
4) An Ignored Test Is a Question about an Ambiguity
5) Test Boundary Conditions
6) Exhaustively Test Near Bugs
7) Patterns of Failure Are Revealing
8) Test Coverage Patterns Can Be Revealing
9) Tests Should Be Fast

Conclusion and Special Offers

We hoped you enjoyed this summary. If you leave an honest review on Amazon or wherever you obtained this book, it is appreciated and will be used for future iterations of this book.

Thank you in advance.

For additional exploration in software development training and eduction, you are also encouraged to visit http://www.supergloo.com for free resources and significant discount coupons of online training.

You are encouraged to check out the full version of the book if you haven’t already done so.