clean code notes

bj

August 2019

1 5S principles

- *Seiri* - organization ("sorting") knowing where things are (e.g. suitable naming)
- *Seiton* - tidiness ("systemize") A place for everything, and everything in its place; a piece of code should be where you expect to find it
- *seiso* - cleaning ("shine") keep the code clean/litter free (extra comments/commented out code, etc.)
- *seiketsu* - standardization the group agrees about how to keep the workspace clean
- *shutsuke* - self discipline force yourself to follow the practices!

Golden idea - build "machines" that are more maintainable in the first place rather than rushing and fixing after

"An ounce of prevention is worth a pound of cure"

"We should view our code as the beautiful articulation of noble efforts of design"

2 Clean Code

- Minimal
- Focused
- Like well written prose
- elegant, efficient
- does not tempt additions of messy code; like a building with broken windows
- should be readable by humans
- looks like its written by someone who cares

- an object or method should only do one thing
- each line/method does what you initially expected it to after reading

3 Meaningful Names

- Use intention revealing names
  example:
  int d; // elapsed time in days – BAD;
  other options:
  int elapsedTimeInDays;
  int daysSinceCreation;
  int daysSinceModification;
  int fileAgeInDays;

  Method Example:
  public List<int[]> getThem() vs. public List<int[]> getFlaggedCells()
  List<int[]> list1 vs List<int[]> flaggedCells
  if (cell[0] == 4) vs if (cell[STATUS_VALUE] == FLAGGED)

  even better:
  List<Cell> flaggedCells = new ArrayList<Cell>();
  if (cell.isFlagged()) flaggedCells.add(cell);
- Avoid words whose "entrenched" meaning vary from the intended meaning
- avoid multiple names which vary in tiny/small ways (XYZControllerHandlingStrings vs XYZControllerStorageStrings)
- avoid numeric variables (a1[i] = a1[i] vs. destination[i] = source[i])
- use pronounceable names (Date genymdhms vs. generationTimestamp)
- use searchable names
  the length of a name should correspond to the size of its scope
- avoid encodings
  encoding type or scope information to names adds extra burden of deciphering (e.g. hungarian notation, member prefixes (your class should be small enough that you dont need them!))
- interfaces and implementations
  use ClassType –> ClassTypeImpl instead of IClassType –> ClassType
- avoid mental mapping
  clarity is king!
- class names should be a noun (since they are an object, after all)

- methods should have a verb or verb phrase names, and use getThing, setThing, isStatus (accessors, mutators, and predicates)
- use one word per concept (e.g. dont have three *fetch*, *retrieve*, *get* or *controller*, *manager*, *driver*)
- add meaningful context (e.g. it should be clear what a variable applies to based on its namespace/ class its a part of etc.)

Our goal is to make the code as easy as possible to understand

4 Functions

functions should be small functions should be smaller than that! Functions should hardly ever be 20 lines long

Functions should do one thing. they should do it well. they should do it only!

One way you can know if a function is doing more than one thing is if you can extract another function from it *with a name that is not merely a restatement of its implementation*

The step down rule - the code should read like a top-down narrative. Every function should be be followed by the next level of abstraction one at a time

- To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.
  - To include the setups, we include the suite setup if this is a test suite, then we include the regular setup.
    - To include the suite setup, we search the parent hierarchy for the 'SuiteSetup' page and add an include statement with the path of that page
      - To search the parent hierarchy...

4.1 Function arguments

Functions should have as few arguments as possible. At most they should have 3. Output arguments are harder to understand than input arguments – instead functions should generally just return your desired output

- Flag arguments are bad (e.g. booleans into function). Instead, make two functions (e.g. - render(bool isSuite) vs. renderForSuite() and renderForSingleTest())
- For functions with two arguments, be wary when there is no 'natural' ordering of the two inputs - e.g. x,y vs. row,col –> its not immediately obvious which form you're using
  Try instead to use argument objects:

Circle makeCircle(double x, double y, radius) –> Circle makeCircle(Point center, double radius)

- For "monads" (single argument functions), they should have a verb/noun relationship (do "verb" on argument "noun")
- for dyads (2 argument functions), "keyword encoding" is good, e.g. the arguments fill in the "keywords" of the function name –> assertExpectedEqualsActual(expected, actual)
- functions should either do something or answer something - not both. e.g. change the state of an object or return information about the object
- prefer exceptions over error codes –> they lead to many nested if statements. However, try catch blocks are ugly, so extract the body of try/catch blocks to functions of their own. Error handling is its own thing, hence it should have its own function
- we can negate dijkstra's rule ("every function should have only one entry and exit") and have break/continue/return/gotos in a function *as long as the function is short* and their effects are immediately obvious

Don't repeat yourself!

These rules probably won't be followed when you initially write the function - its like a rough draft that will need revisions

5 Comments

- Inaccurate comments are worse than no comments at all. e.g. code is updated and comments are no longer accurate
- comments do not make up for bad code
- code should express itself instead, e.g.
  // Check to see if the employee is eligible for full benefits
  if (employee.flags & HOURLY_FLAG && employee.age > 65)
  vs.
  if (employee.isElegibleForBenefits())
- informative comments - should convey information that CANNOT be conveyed with code/naming
  e.g. a regular expression:
  // format matched: kk:mm:ss EEE, MMM dd, yyy

```
Pattern timeMatcher = Pattern.compile("\\d*:$\\$d* $\\$w*")
```

- clarifying code that you cannot change or alter (e.g. part of a library or something) is good

- TODOs are also fine - jobs that you think should be done but can't at the moment for some reason
- Amplification - emphasizes the importance of something that at first glance may seem trivial
- a comment that requires you to look elsewhere to find its meaning is bad
- redundant comments are bad - they are just distracting noise
- don't leave commented out lines of code - delete them! that's what source control is for
- instead of marking ends of loops and such; try and make the function/section shorter
- position markers (marking related sections/functions) are generally noise
- don't add systemwide information to a local comment - if it contains information that could change elsewhere besides the current code, there's no guarantee that comment will be changed later and hence could become incorrect

Don't use a comment when you can use a function or variable

6 Formatting

Choose a simple set of rules and consistently apply those rules. Automated tools exist to help do that for you.

Files shouldn't be too long (around a max of 200/300 lines)

Code should be ordered from most abstract/general to most detailed from top to bottom

Different complete thoughts/expressions should be separated by a blank line

Lines of code that are tightly related should appear vertically dense; e.g., not separated too far by useless comments

Concepts that are closely related should be kept vertically close to each other. You shouldn't have to hunt to find where things used by other things are in the same file.

If one function calls another, they should be vertically close. The caller should be above the callee if possible.

Variables should be declared as close to their usage as possible. For our goal of very short functions, they should be able to be declared at the top of the function

Lines should have a max of around 80-120 characters

If you are tempted to align a group of declarations and assignments horizontally on each line, it may instead be a sign that they need to be split up. The problem is the length of the list, not the lack of alignment

Ignoring indentation for short if statements/functions/loops is generally bad

7 Objects and Data Structures

Data structures should enforce access policies.

Classes should expose abstract interfaces to the *essence* of the data, without having to know its implementation

ex:

```
public interface Vehicle {
    double getPercentFuelRemaining();
}
```

vs.

```
public interface Vehicle {
    double getFuelTankCapacityInGallons();
    double getGallonsOfGasoline();
}
```

In which case the former is better since it represents an abstract concept which we want rather than implementation details.

The worst option is to blithely add getters and setters.

Objects should hide their data behind abstractions and expose functions that operate on that data. Data structures should expose their data and have no meaningful functions.

Procedural code (using data structures) makes it easy to add new functions without changing the existing data structures. Object Oriented (using objects) on the other hand, makes it easy to add new classes without changing existing functions.

The opposite is also true

Procedural code makes it hard to add new data structures because all the functions must change. OO code makes it hard to add new functions because all the classes must change.

Law of Demeter: a module should not know about the innards of the objects it manipulates. An object should not expose its internal structure through accessors.

Avoid creating "Hybrid" structures - half object and half data structure (have some public variables but also functions).

Ex: The following are bad:

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
final String outputDir = ctxt.getAbsolutePathOfScratchDirectoryOption();
final String outputDir = ctxt.getScratchDirectoryOption().getAbsolutePath();
```

Instead, we should ask why/what is the purpuse of getting this path (for example to use as a file name for writing out to) and do that instead:

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

8 Error Handling

- Error handling is important, **but if it obscures logic, its wrong**.
- Use exceptions rather than return codes
  - Return codes add clutter (many, many nested if/else status checks and error logs)
- write the *try-catch-finally* first.
  - They define a scope to work with, and the catch block adds a "consistent state" to always leave the program/method with
- Provide context with exceptions - enough information to know where and WHY the exception occurred - not just a stack trace
- Don't let exceptions break up the "normal flow" of the program - e.g.

```
try {
    MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
    m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
    m_total += getMealPerDiem();
}
```

  should instead be

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

  That is, the no meal expenses found thing should be handled internally. One solution is to have *getMeals()* to return an object, which, depending on if meal expenses exist or not, will be one of two types both of which inherit from a parent class defining their interface. This is called the *Special Case Pattern*.

- Don't return null
  - This results in many annoying *if null* checks. It's also possible to accidentally miss adding a null check while coding which could break your program at runtime.
- Don't pass null - again, same idea with having *if null* checks.

9 Using Third Party Code/Libraries

- We want to avoid having to re-write our code if a library changes. Hence, its good to wrap libraries in our own class/API so that we only need to change our internal wrapper and not the whole system.
- Wrapping the library is also useful if you're writing code using an API that doesn't exist yet (e.g. waiting for someone to finish or something). You can use your own wrapper and fill the wrapper with dummy code for now.

10 Tests

- Tests should be kept clean/are just as important as the production code. If tests aren't clean, they become hard to add to ==> you'll eventually stop adding tests because its too hard ==> you can't test new production code ==> you won't want to change production code because every addition is a possible bug ==> your entire codebase rots.
- Tests should run relatively fast, otherwise you wont want to run them as often, hence you don't catch bugs as quickly. On the other side, they don't have to be as fast as production code so don't be afraid to sacrifice some performance for readability.

11 Classes

- A class should have as few *responsibilities* as possible as opposed to number of lines of code, e.g. it should do one thing ONLY (single responsibility principle)
- You should be able to describe the class in around 25 words without using "if", "and", "or", "but" - e.g.
  - The SuperDashboard provides access to the component that last held the focus, and it also allows us to track the version and build numbers
    - the *and* indicates the existence of multiple responsibilities
- Goal should be to have many small classes, all with a single responsibility, instead of one large class
- Private method behavior that applies only to a small subset of the class can be an indicator that something needs to change/ violates the single responsibility principle. e.g. change

```
public class Sql {
    public Sql(String table, Column[] columns)
```

```
    public String create()
    public String insert(Object[] fields)
    public String selectAll()
    public String findByKey(String keyColumn, String keyValue)
    public String select(Column column, String pattern)
    public String select(Criteria criteria)
    public String preparedInsert()
    private String columnList(Column[] columns)
    private String valuesList(Object[] fields, final Column[] columns)
    private String selectWithCriteria(String criteria)
    private String placeholderList(Column[] columns)
}
```

to

```
abstract public class Sql {
    public Sql(String table, Column[] columns)
    abstract public String generate();
}
public class CreateSql extends Sql {
    public CreateSql(String table, Column[] columns)
    @Override public String generate()
}
public class SelectSql extends Sql {
    public SelectSql(String table, Column[] columns)
    @Override public String generate()
}
public class InsertSql extends Sql {
    public InsertSql(String table, Column[] columns, Object[] fields)
    @Override public String generate()
    private String valuesList(Object[] fields, final Column[] columns)
}
public class SelectWithCriteriaSql extends Sql {
    public SelectWithCriteriaSql(
        String table, Column[] columns, Criteria criteria)
    @Override public String generate()
}
public class SelectWithMatchSql extends Sql {
    public SelectWithMatchSql(
    String table, Column[] columns, Column column, String pattern)
    @Override public String generate()
}
...
```

etc.

The code in each class becomes "excruciatingly simple." This means

  - It takes practically 0 effort to understand it
  - adding to the class won't break anything else (just add a new inherited class)

- Dependency Inversion Principle - classes should depend on abstractions, NOT implementation details - it means we should be able to change our concrete implementation details without breaking the other classes that use it.

12 Systems

The startup process/logic should be separate from the runtime logic.

- One solution is to do all construction in *main* or modules called by main, then run the rest of the system assuming everything in main has initialized correctly.

For objects that will be created while the program is already initialized and running, we can use an *abstract factory*

- E.g. when the program starts, the *factory* is initialized, and then while the program is running the *factory* is responsible for creating new objects of that type.

Dependency Injection - an object should not be responsible for creating its required dependencies - instead they should be *injected* into the object by an outside authority.

- I.E. the class has setters or constructor arguments that pass in its dependencies; the class itself does nothing with their instantiation.

Tightly coupled dependencies/logic make it hard to extend/add to the code since you have to work around the coupled dependencies.

"An optimal system architecture consists of modularized domains of concern, each of which is implemented with Plain Old Java (or other) Objects. The different domains are integrated together with minimally invasive Aspects or Aspect-like tools. This architecture can be test-driven, just like the code."

Obviously the book is Java oriented, but the idea is there.

13 Emergence

Four rules of simple design - A design is "simple" if it follows these rules:

1. Runs all the tests
2. Contains no duplication

3. Expresses the intent of the programmer
4. Minimizes the number of classes and methods

- Refactoring the code - each time we add some code we should go back and check if we've dirtied the existing code - if so we need to clean up/refactor.
- No duplication - it adds additional work and complexity if you need to change something. Also, lines that look very similar but differ slightly can count as duplication. Even small amounts of repeated code can be cleaned:

```
void scaleToOneDimension(...){
    if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
        return;
    float scalingFactor = ...;
    ...
    RenderedOp newImage = ImageUtilities.getScaledImage(image, scalingFactor,
        scalingFactor);
    image.dispose();
    System.gc();
    image = newImage;
}
void rotate(int degrees) {
    RenderedOp newImage = ImageUtilities.getRotatedImage(image, degrees);
    image.dispose();
    System.gc();
    image = newImage();
}
```

Could be changed to

```
void scaleToOneDimension(...) {
    if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
        return;
    float scalingFactor = ...;
    ...
    replaceImage(ImageUtilities.getScaledImage(image, scalingFactor,
        scalingFactor));
}
void rotate(int degrees) {
    replaceImage(ImageUtilities.getRotatedImage(image, degrees));
}
void replaceImage(RenderedOp newImage) {
    image.dispose();
    System.gc();
```

```
    image = newImage();
}
```

Additionally, the fact that we need *replaceImage* shows we violate the single responsibility principle; so the replaceImage method might be extracted into its own class.

- The *template method* pattern also helps eliminate duplication. For different sections of code that are largely the same with minor differences, break the code up into multiple higher level functions, and have different function implementations in the places that differ slightly.

  ex:

```
class VacationPolicy
{
    public void accrueVacation() {
        calculateBaseVacationHours();
        alterForLegalMinimums();
        applyToPayroll();
    }
    void calculateBaseVacationHours() {...}
    abstract void alterForLegalMinimums(); // to be implemented
    void applyToPayroll() {...}
}
class USVacationPolicy extends VacationPolicy
{
    @Override void alterForLegalMinimums() {
        ...
    }
}
class EUVacationPolicy extends VacationPolicy
{
    @Override void alterForLegalMinimums() {
        ...
    }
}
```

- Expressing the intent of the programmer - Try to be careful and clear with your code (should be obvious what a class/function does, good names, short methods/classes) - basically take PRIDE in your code and have good craftsmanship - it should be clear that you tried and care about the code.

- Minimal classes and methods - although the above suggests adding many, many classes/methods/etc. in the name of expressiveness, don't go overboard - our goal is to have as few methods and classes while at the same time keeping the overall system small.

14 Concurrency

- concurrency related code should be kept separate from other code.
- limit the scope of data as much as possible to avoid race conditions - e.g. only access shared data in one point of the code (no duplication!) so you can't accidentally forget to protect/lock the data elsewhere.
  - Change object design of shared data so that you don't force objects using the data to manage sharing as well
- Use copies instead of sharing when possible (e.g. local copy to a thread to be merged later in the main thread)
- threads should be as independent as possible - share as little with other threads as possible
- most concurrent problems are variations of the reader-writer, producer-consumer, and dining philosophers problems
- keep critical sections as small as possible

15 Successive Refinement

- Your first working code should be considered a "rough draft"
  - You then clean/refine this rough draft, possibly multiple times, to get a clean version.
- When you have different types with similar methods inside of a class, its a sign to break out into a new class.
- Instead of making one large change to the code which then breaks it, make smaller changes and test as you go as they're easier to fix
- "Much of good software design is simply about partitioning - creating appropriate places to put different kinds of code. This separation of concerns makes the code much simpler to understand and maintain."

16 JUnit Internals

This chapter goes through an example of cleaning up a Java class, JUnit ComparisonCompacter.

Some examples of improvements:

- Extracting boolean statements to function to clarify their MEANING/INTENT
  - E.g.

```
if (expected == null || actual == null || stringsAreEqual()) {
    ...
```

    to

```
if (shouldNotCompact()) {
    ...
bool shouldNotCompact() {
    return expected == null || actual == null || stringsAreEqual();
}
```

- clarifying function names to expose side effects/actions besides just what the name implies
  - E.g.

```
String compact() {
...
```

    to

```
String formatCompactedComparison() {
...
```

- clarifying/renaming variable names
  - E.g. changing *suffixIndex* to *suffixLength*
- Extracting error handling/checks from the logic/flow of a function

Follow the Boy Scout Rule - always leave code cleaner than you found it.

17 Refactoring SerialDate

Again, a chapter on cleaning up a Java class. This time JCommon org.jfree.date

Some examples of improvements:

- Replacing multiple related constants with an enum

- changing names for clarification
  - one example was to change *date.addDays()* to *date.plusDays()* to imply that the function does not modify the *date* object but instead returns a new date with the addition of days sent to *plusDays* (think like immutable strings in Java)
- Moving code apart from unrelated areas to related areas (e.g. a new or existing class)
  - Example - a search for use of a group of variables shows they are only used in one derivative class - then move them to that class!
- Adding a factory class to generate instances with specific constraints/ details
- deleting redundant comments
- adding "explaining temporary variables" to complicated algorithms for clarity
- replacing magic numbers with explanatory variables

18 Smells and Heuristics

This chapter is a list of indicators ("Bad Smells") that indicate you should refactor/rewrite/remove code, and general rules/heuristics to follow. I skipped some entries that I though felt redundant/restatements of other points already covered.

18.1 Comments

18.1.1 Inappropriate Information

Don't add comments about things unrelated to the current code - e.g. file/changelist histories, or meta data like authors, last modified, etc.

```
/*
 * This comment is clutter!
 *   Changelog:
 *   01/01/01 - first change
 *   02/01/01 - second change
 *   03/01/01 - third change
 *   04/01/01 - fourth change
 */
public void doStuff() {
 ...
}
```

18.1.2 Obsolete Comment

Outdated comments are bad; ones that contain misleading/incorrect information make your code *worse*

```
/*
 * This comment states the function does thing 1
 * but it actually now does thing 2:
 *
 *    This function does thing 1
 */
public void doThing2() {
...
}
```

18.1.3 Redundant Comment

Comments that says the same thing as the code are useless

```
/*
 * Does thing 1
 * can throw myException
 */
public doThing1() throws myException {
...
}
```

18.1.4 Poorly written comment

Bad grammar or poor phrasing make the code more confusing and harder to read. Comments should be *brief* and *consise*

```
/*
 * Run on sentence and is unnecesarily verbose:
 *
 * When you call doThing1() it runs the doThing algorithm
 * which consisihs of the foloowung forst you apply
 * the first step second you apply the seoncdiosj step
 * third you doe this thing but if the other thing happens
 * and finally you eat a chicken sandwhich
 */
public void doThing1() {
...
}
```

18.1.5 Commented Out Code

Commented out code should be deleted - version control should be your backup if you decide something needs to be restored.

```
// This code was rewritten ages ago but you're afraid
// to delete it because what if your new stuff doesn't
// work how will I ever survive?
//
// public void useless() {
//   ...
// }
```

18.2 Environment

18.2.1 Build Requires more than One Step

Building a project should require a single command. Should be one command to download the project and dependencies and another single command to build it.

```
// good! easy
git clone my_git_repo.git
cd my_git_repo && make
```

18.2.2 Tests Require More Than One Step

All tests should run with a single command. This makes it easy to run tests hence encouraging you to run them often. Running them often means you will find bugs faster.

```
// easy!
make test
```

18.3 Functions

18.3.1 Too Many Arguments

Functions should have as few args as possible; at max 3. "More than 3 is very questionable and should be avoided with prejudice"

```
// chances are if you need this many arguments the function is too
// complex and should be broken down
void myLongComplexFunction(int arg1, int arg2, char *pointer1, bool ifPointer1,
        char *pointer2, MY_ENUM functionCondition1) {
...
}
```

18.3.2 Output Arguments

The auther here says "Output arguments are counterintuitive. Readers expect arguments to be inputs, not outputs." I disagree with this rule - what if the point of your function is to fill an argument with data? It's faster to fill an argument pointer than to return an entire struct.

```
// this seems fine to me - no issue here.
void calculateMyValue(int arg1, int arg2, struct *result);
```

18.3.3 Flag Arguments

Boolean arguments indicate that a function does more than one thing E.g. "if this reason, then do thing 1, else do thing 2" - shows that thing 1 and thing 2 should be their own functions.

```
void doThing1(int arg1, bool doConditionalThing) {
    statement a;
    statement b;

    if (doConditionalThing) {
        do extra stuff here
    }
}
```

Could be broken up as

```
void doThing1(int arg1) {
    statement a;
    statement b;
}
void doExtraStuff() {
    do extra stuff here
}
```

18.3.4 Dead Function

Functions that are never called should be deleted; they're the equivalent of commented out code. Again, source control can always be used to restore if necessary.

```
// this code used to be called earlier but
// changes have been made elsehwere and now
// this code isn't used and is extra bloat
void usedEarlierButNotNow() {
...
}
```

18.4 General

18.4.1 Obvious Behavior Is Unimplemented

It should be intuitive the functions available based on a class or system description.

```
Day day = DayDate.StringToDay(String dayName);
```

It's intuitive that dayName should be able to take any variation of a weekday, like "Monday", "monday", "mon", or "M"

18.4.2 Overridden Safeties

Things like ignoring failing tests, disabling or ignoring compiler safeting warnings/errors can lead to bugs or errors.

For example, compiling the following with warnings disabled vs. all warnings enabled with gcc 8.3.0 (example from here)

```
// myfile.c
#include <stdio.h>

int main()
{
    printf("Program run!\n");
    int i=10;
}
...
gcc myfile.c       // no warnings given
gcc -Wall myfile.c // warning: unused variable 'i' [-Wunused-variable]
```

Though I am dissapointed it didn't say anything about not returning a value from *int main*. If you're feeling really strict, compile with *gcc -Wall -Werror* to have compilation fail completely if any warnings are present.

18.4.3 Duplication

"Every time you see duplication in the code, it represents a missed opportunity for abstraction." Duplication should be turned into another function or another class. Additionally, switch cases and if-else chains can represent similar ideas which should be replaced with polymorphism (slightly different classes inheriting from the same parent type)"

```
void a() {
    doThing1;
    doThing2;
    anotherThing1;
    anotherThing2;
```

```
}
void b() {
    doThing3;
    doThing4;
    anotherThing1;
    anotherThing2;
}
```

Could instead be

```
void a() {
    doThing1;
    doThing2;
    doAnotherThing();
}
void b() {
    doThing3;
    doThing4;
    doAnotherThing();
}
void doAnotherThing() {
    anotherThing1;
    anotherThing2;
}
```

In terms of switch cases/if-else:

```
void internalClassFunction() {
    switch (internalClassCondition)
    {
        case A:
            doThingForA()
        case B:
            doThingForB()
        case C:
            doThingForC()
    }
}
```

Could instead be:

```
class Thing {
    virtual void internalClassFunction() = 0;
}
...
class ThingTypeA {
    void internalClassFunction() {
```

```
        doThingForA();
    }
}
class ThingTypeB {
    void internalClassFunction() {
        doThingForB();
    }
}
class ThingTypeC {
    void internalClassFunction() {
        doThingForC();
    }
}
```

18.4.4 Code at Wrong Level of Abstraction

Higher level concepts should be separated from lower level concepts. This applies to classes, source files, modules, etc.

```
public interface Stack {
    Object pop() throws EmptyException;
    void push(Object o) throws FullException;
    double percentFull();
    class EmptyException extends Exception{}
    class FullException extends Exception{}
}
```

percentFull() is a higher level concept than the push-pop implementation of a regular stack, which may not have a concept of 'full' (a regular stack has a dynamic size).

18.4.5 Base Classes Depending on Their Derivatives

A base class should know and care nothing about its children. An exception is if there are a set of known possible child types that the parent class chooses from.

I can't really think of a code example for this... (in terms of actual dependance, anyways)

18.4.6 Too Much Information

"A well defined interface does not offer many functions to depend on, so coupling is low. A poorly defined interface provides lots of functions that you must call to get something done, so coupling is high."

Having multiple options to do things implies you can do more than one thing, which breaks the rule that a class/function should do one thing only.

See section 11

18.4.7 Vertical Separation

Functions/variables spread far away from where they are used makes the code confusing and cumbersome to read.

```
void myFunc() {
    func1();
    func2();

    statementA;
    statementB;
}
void otherFunc() {
    statementC;
    statementD;
    ...
}
void yetAnotherFunc() {
    statementE;
    statementF;
    ...
}
void func1() {
    ...
    ...
}
void func2() {
    ...
    ...
}
```

Should reorder *func1* and *func2* closer to *myfunc*

```
void myFunc() {
    func1();
    func2();

    statementA;
    statementB;
}
void func1() {
    ...
    ...
}
void func2() {
    ...
    ...
```

```
}
void otherFunc() {
    statementC;
    statementD;
    ...
}
void yetAnotherFunc() {
    statementE;
    statementF;
    ...
}
```

18.4.8 Inconsistency

Multiple types of functions/variables and casings makes the code unintuitive to parse and look for similar things.

Given the following function

```
void processVerificationRequest(request);
```

It should be intuitive that the function handling deletion requests would be called

```
void processDeletionRequest(request);
```

As opposed to *processVerificationRequest* and *handleDeletionRequest* for example, which makes no sense in the fact that their names don't necessarily imply they are related.

18.4.9 Artificial Coupling

Placing constants/enums/functions/variables in code where it is convenient instead of logically located. This is lazy - discipline yourself!

```
static datatype myGeneralHelperFunction(item) {
    ...
}

void MyUnrelatedClass::internalFunction() {
    equation_result = myGeneralHelperFunction(item);
    ...
}
```

The general helper function should be moved as an internal class function if its actually related to the class; otherwise it should be extracted into related utility functions or its own file - this makes it easy to use if it ends up being useful elsewhere!

18.4.10 Feature Envy

When a class wants to access way too much of another class's internals - it *wants* to be a part of that other class and get all of its data. Sometimes, this is unavoidable howerver in order to avoid tight internal coupling and provide proper abstraction.

```
public class HourlyEmployeeReport {
    private HourlyEmployee employee ;
    public HourlyEmployeeReport(HourlyEmployee e) {
        this.employee = e;
    }
    String reportHours() {
        return String.format(
            "Name: %s\tHours:%d.%1d\n",
            employee.getName(),
            employee.getTenthsWorked()/10,
            employee.getTenthsWorked()%10);
    }
}
```

Here the reportHours function "envies" the data inside of employee; however we don't want to expose how the employee report is formatted (string.format) to the employee, who doesn't care about this information. Thus we will leave it as is.

18.4.11 Obscured Intent

Non-expressive and shortened names are illegible to everyone except the author:

```
public int m_otCalc()
```

vs.

```
public int overTimePay()
```

The latter is much more expressive and legible. The author states hungarian notation (m_ prefix) is unnecssary and confusing.

18.4.12 Inappropriate Static

Static should not be used on functions that could be modified/run differently via polymorphism. For example,

```
HourlyPayCalculator.calculatePay(employee, overtimeRate)
```

could change and be inherited from for different kinds of employees or other reasons, and thus should not be static.

18.4.13 Lack of Explanatory Variables

Long or complicated calculations/equations should be broken down into smaller steps using explanatory variables.

```
if(headerPattern.matcher(line).find())
...
```

vs

```
Matcher match = headerPattern.matcher(line);
if(match.find())
...
```

the latter is more legible than the former.

18.4.14 Function Names Should Say What They Do

Ambiguous function names make code hard to read.

```
Date newDate = date.add(5);
```

vs.

```
Date newDate = date.daysLater(5);
```

The second is explicit and makes sense while the first is ambiguous.

18.4.15 Understand the Algorithm

If you don't know how the algorithm works your code will reflect that and will be confusing just as you are confused about the algorithm. A solution to fix this is refactor the code into easy to digest steps so it is clear what each step is and does.

18.4.16 Make Logical Dependencies Physical

Modules dependent on other modules shouldn't make 'Logical' assumptions about the other module; they should be explicitly asked for in code.

```
public class HourlyReporter {
  private HourlyReportFormatter formatter;
  private final int PAGE_SIZE=55;
  public void generateReport(employees) {
    for (HourlyEmployee e : employees) {
      addLineItemToPage(e);
      if(page.size() == PAGE_SIZE) {
        printAndClearItemList();
      }
    }
}
```

The *HourlyReporter* assumes that the PAGE_SIZE is 55. This should be the formatterś concern, not the managerś. The *HourlyReporter* is making a logical assumption that should be the job of the formatter's instead, which should contain a method *getMaxPageSize()* for the *HourlyReporter* to use.

18.4.17 Follow Standard Specifications

If your team has an agreed upon style guide, follow it.

18.4.18 Replace Magic Numbers with Named Constants

The reader shouldn't have to guess or assume what a seemingly innocent valueś purpose is. The exception would be for known or obvious formulas/code, such as

```
double milesWalked = feetWalked/5280.0;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

18.4.19 Encapsulate Conditionals

Logic contained in if/for/while cases should be logically clear. For example

```
if (shouldBeDeleted(item))
```

is much clearer than

```
if (timer.hasExpired() && !timer.isRecurrent())
```

Additionally, prefer positive to negative conditionals (e.g. *isFalse()* instead of *!isTrue()* as the exclamation point is small and has a possibility of being missed by the reader.

18.4.20 Hidden Temporal Couplings

When the order/structure of calling functions matters, it should be explicitly required that they are called in that order.

```
public void dive(String reason) {
  saturateGradient();
  reticulateSplines();
  diveForMoog(reason);
}
```

vs

```
public void dive(String reason) {
  Gradient gradient = saturateGradient();
  List<Spline> splines = reticulateSplines(gradient);
  diveForMoog(splines, reason);
}
```

The latter enforces the order in which the functions are called due to their arguments, which is good.

18.4.21 Encapsulate Boundary Conditions

Putting boundary conditions/checks in a function keeps you from having to rewrite said cases in multiple places. The more places they are hard coded the more places you have to accidently miss them and add off-by-one bugs.

```
if (level + 1 < tags.length) {
  parts = new Parse(body, tags, level + 1, offset + endTag)
```

vs

```
int nextLevel = level + 1;
if (nextLevel < tags.length) {
  parts = new Parse(body, tags, nextLevel, offset + endTag);
```

Adding *nextLevel* prevents us from having to write *level + 1* twice which reduces the chance of adding bugs/errors.

18.4.22 Functions Descending More Than One Level of Abstraction

A function should only go down one level of logic. This mixes with a function should do one thing only. Example:

```
public String render() throws Exception {
  StringBuffer html = new StringBuffer(''hr'');
  if (size > 0) html.append('' size = \''''').append(size + 1).append(`\'''');
  html.append(`>`);

  return html;
}
```

This function contains both html structure and html syntax (two different abstractions) in the same function and instead be broken up, e.g.

```
...
HtmlTag hr = new HtmlTag(`hr`);
...
```

18.4.23 Keep Configurable Data at High Levels

The more likely a piece of data is to change the easier it should be to do so, e.g. kept at the highest level possible such as program/command line arguments.

18.4.24 Avoid Transitive Navigation

If module A interacts with module B which interacts with module C, A shouldnt́ need to know about C (transitive property).

Ex, we dont́ want to have to call

```
a.getB().getC().doSomething()
```

It should instead be written so that we just call

```
a.doSomething()
```

18.5 Names

18.5.1 Choose Names at the Appropriate Level of Abstraction

Names shouldnt́ give away implementation. ex

```
public interface Modem {
  boolean dial(String phoneNumber);
  ...
```

What if the modem isnt́ over a phone line (like hard wired, usb, coaxial, etc.)? Having 'dial' and 'phoneNumber' exposes the implementation. It should instead be

```
public interface Modem {
  boolean connect(String connectionLocator);
  ...
```

18.5.2 Use Standard Nomenclature Where Possible

Its easier to understand names if they already exist in a known pattern/syntax/definition that everyone is already familiar with.

Ex. if using the *Decorator Pattern*, an appropriate name would be *AutoHangupModemDecorator*.

18.5.3 Use Long Names for Long Scopes

How long of a scope a variable has should determine how long its name is. E.g., the name i is fine in a short for loop but would be inappropriate for longer use in a function. If used longer it implies there is more importance to the variable and hence it deserves a more descriptive name.

18.5.4 Avoid Encodings

Things like hungarian notation (m_<variable>) are outdated and are just extra clutter.

18.5.5 Names Should Describe Side Effects

"Don't use a simple verb to describe something that does more than a simple action"

```
public ObjectOutputStream getOos() throws IOException {
  if (m_oos == null) {
    m_oos = new ObjectOutputStream(m_socket.getOutputStream());
  }
  return m_oos;
}
```

Here the function name *getOos* doesn't include the fact that it creates a new oos if it is null. A better name is *createOrReturnOos*.

18.6 Tests

18.6.1 An Ignored Test Is a Question About an Ambiguity

If you aren't running it that implies either it doesn't work or you're not sure how it should function and don't know what "working" means.

18.6.2 Exhaustively Test Near Bugs

If you found a bug in one area, there's a good chance there's other bugs in that same area.

18.6.3 Notice Patterns in Multiple Failed Tests

For example, all tests with inputs larger than five characters failed indicates you could probably fix all of them with one solution.