Kevin San Gabriel
301342241
CMPT 376W Milan Tofiloski
Journal Entry #21

Reduce code The size of a codebase/file should be as small as possible so that compilation of the code is quick and so that there is no redundant code.

Avoid clever code Code should be kept simple so that it can be read easily by others as well as the author of the code.

Make it small and cohesive Small and cohesive code will reduce the number of responsibilities on the author to maintain the code.

Eliminate duplication Duplication of code is unneeded redundancy which contributes nothing.

Eliminate dependencies Strive to remove dependences so that there is a lower chance of the code breaking if a part is modified

Write self documenting code so that anyone who reads that code, including the author will be able to understand it easier

Code should be understandable in seconds clearly express the meaning of the code by making it both readable and meaningful

Avoid primitive obsession code will become more flexible if there is focus on higher level abstractions

Checkin frequently, take small steps compile often, save often, commit often so that there is a shorter feedback cycle, less painful merges and so that other developers are kept in the loop

Kevin San Gabriel
301342241
CMPT 376W Milan Tofiloski
Journal Entry #21

Keep code at one level of abstraction methods should do one thing, delegate to other methods that each do one thing

Part 2:

Simplicity High complexity and less simplicity in the code will make the author stand out as a skilled programmer.
Clarity Code that is unclear will create an illusion of abstraction. The code will be hard to interpret. This will be useful especially if readers of the code are not supposed to be reading the code.
Brevity Long code will create the need for more lines and less white space. When viewed, the code will look more thorough and complex.
Humanity Code that is not written for people will not be easy to read for developers. Other developers will probably not be interested in giving feedback on the code.

Part 3:

Small Changes When refactoring, make one change at a time. Making too many changes at once introduces bugs.

For example, when creating methods for a class, focus on implementing one method at a time

Implementing multiple methods will create confusion on the dependencies of each method. The result will be a spaghetti like series of calls.

Kevin San Gabriel
301342241
CMPT 376W Milan Tofiloski
Journal Entry #21

Slow down Read code slowly

When reading code, look at code one line at a time

Looking at code one line at time will reduce the chances of missing something

Duplicate code If you have duplicate code, focus on moving it all to one place before making changes

If you have duplicate code, comment it out and keep track of it before making changes

This will ensure that the entire program does not break if you mess up with that code

Stay on Task by Taking Notes Work on most important smells, however, don't address a smell as soon as possible.

If you're implementing a method and come across another method which would look nicer if it were implemented like the method you're currently implementing, take notes on that method and complete your current task.

This will ensure you do not introduce bugs by getting sidetracked

Hide the mess If you have code which is a mess, hide it from the rest of the application and put it in an interface

Kevin San Gabriel
301342241
CMPT 376W Milan Tofiloski
Journal Entry #21

If there is untested code, use Junit on the untested code.

Keep track of messy code will keep things organized so that you can test the code.

Don't be afraid to take a step back Important advantage of coding in small steps is that if you make a mistake, you can easily reverse it

If you implement one line of a method, don't be afraid to erase it. You can always re-write that one line easily

Being able to easily reverse mistakes in a code will save you time in debugging potential bugs that may arise as a result not coding in small steps

Understand the code before you change it Before you change code, the first step you should take is to understand it.

If you want to implement a method differently, first understand how you implemented it the first time

Understand code before you change it will make it easier to implement it again afterwards in a different way