Chapter-1 Clean Code

There Will Be Code

- Code represent the detail of requirements some level detail cannot ignored when it is executing that is programming
- Abstraction(quality) of our language will increase ,domain-specific languages will grow,but it will not eliminate the code
- Code should be in a formal
- Code really language which ultimately express the requirement
- never eliminate necessary code - so there will be code

Bad Code

- In the late 80s (killer app) is shut down because of bad code
- huge mess in code when added more feature code got worse , this could not manage so long - bad code will bought your product down
- obstraction many times that is **wadding** that time you go throught the bad code -strugle to find hoping some hints and clues. we will see many sensless code
- Dont leave the messy code to be cleaned later , Later equals never

Total Cost of Owning a Mess

- Messy code be very fast at beginning of a project but moving in a snail pace
- Every change in the code add the knot in code
- mess of code make productivity will be zero , management will add the more staff to the project that make more difficult for the all so that will not be solution

- **The Grand Redesign in the sky**
  - Redesign takes more time rather we can clean that code that could be better
  - Spending time keeping your code clean is not just cost efficient it matters of professional survival
- **Attitude**
  - Never blame others for your messy code , comminucate your doubts without shyness
  - In product everyone should clean code because every one wants the clean code
  - We should always explain about the messy of code to the manager it is matter of proffesionalism
- **The Primal Conundrum**
  - The way to make the deadline to finish only one way to go fast is to keep the code as clean at all time
- **The Art of Clean code**
  - Only way go fast keep your code clean
  - Writing clean code is lot like painting a picture
  - Code sense is the key ,some born with it,some fight to acquire it
  - Who writies the clean code is an artist
- **What is clean code**
  - **Bjarne Stroustrup**
    - Bad code temps to mess code grows

    - Clean code should simple and efficient
    - error handling should be completed
    - discpline of paying attention to the details
    - should as minimal dependency
- **Grady Booch**
    - Clean code is simple and direct
    - Clean code reads like well written prose(Spoken language)
    - It should be crisp(without unecessary detail) abstraction(quality) and straight forard lines of control
- **"Big" Dave Thomas**
    - Clean code can be read and enhanced by developer rather than original author
    - Smaller code is better for clean code
    - Meaningful names , code without testcase is not clean
- **Michael Feathers**
    - Clean code always looks like it was written by someone who cares
    - No one can enhance better
- **Ron Jeffries**
    - Runs all the testcase
    - contains no duplication
    - Express all the design that are in the system
    - minimize the number of entities such as classes methods fuction a

like

- **Ward Cunningham**
    - Clean code is called beautiful code
- **Schools of Thoughts**
    - To make clean code - clean variable name , clean function ,clean class etc...
    - we are only right everyone has their unique way of teaching and they are also right.
- **We are authors**
    - next time you write a line of code remeber your an author writing for readers who will judge the efforts
    - making it easy to read actually makes it easier to write
    - you cannot write the code if you cannot read the surrounding code
- **The Boy Scoute Rule**
    - Leave the campground cleaner than you found it - dont always
    - change one variable name for better
    - break up one function that is to large
    - clean up small duplications
    - clean up composite if statement

Chapter 2 - Meaningful Names

- **Use Intention-Revealing Names**
  - Name of variable,function or class should answer question -why it exist,what it does,how it is used
  - if name requireds a comment the names reveals its intent
  - it should reveal the intent
- **Avoid Disinformation**
  - Programmers must avoid leaving false clues that obsecure the meaning of code
  - Like example if you word accountList then it should be list not a word or something else
  - wrong spelling leads to disInformation
- **Make Meaningful Distinctions**
  - You cant use same name to refer to two different things in the same scope
  - the change of one word should differ in meaning and words without adding the ,a , an like adding s in the end to the words
- **Use Pronounceable Names**
  - Name should be pronuncable and communicatable
  - Example generation_time_snap is better than using gentmsmp
- **Use Searchable Names**
  - It should be easily for findable and when it is useful name but atleast it searchable
- **Avoid Encodings**

    - Encodings adds an extra burden
    - **Hungrain Notaion**
        - Should not mention the data type before the variable name, because the compiler will remember it
    - **Member prefixes**
        - You also don’t need to prefix member variables with m_ anymore your classes and function should be small enough that you dont need them
    - **Interface and Implementation**
        - I dont want my user to know that i am using the interface or abstract class something
- **Avoid Mental Mapping**
    - Reader shouldn't have to mentally translate your names into other names
    - single letter names for loop counter is traditional but it is a poor choice
    - Clarity is king write code that other should understand
- **Class Names**
    - Class name should have noun or noun phrase like Customer, WikiPage,
    - Account, and AddressParser.
    - Class name should not be verb
- **Method Names**
    - Method should have verb or verb pharse name like,,,postPayment,deletePage,

  - named for their values and prefixid with get, set and is according to java beans standard
- **Don't be cute**
  - Dont use naming for entertainment purpose
  - Choose clarity over entertainment
- **Pick one word per concept**
  - pick one word for one concept over the module
  - Example fetch,retrieve,get these three have the same meaning choose one from this and use over the module
- **Dont pun**
  - Avoid using the same word for two different concept and purpose
- **Use solution domain names**
  - Use computer science term eg math term,pattern name and algorithm names
  - people who reads your code will be programmers
- **Use Problem domain name**
  - When there is no programming name then use problem domain name
- **Use Meaningful Context**
  - Some name may be meaningful but some may not if it is not add meaningful context
- **Dont add gratuitous context**
  - dont use gratutious context
  - dont use gsd in every class name like GSDaccountAddress -17 characters

  are redudant or irrelavant

  - Add like accountAddress,customerAddress is fine because resulting name are more precius which is the pointing of all naming

----

Chapter-3 Functions

- **Small**
  - 1st rule function should be small
  - Function should hardly ever be 20 lines long
- **Blocks and indenting**
  - block within if statament , else and while statement should be one line long
  - function should not large enough to hold nested stucture
  - indent level function should not greater than one or two
- **Do one thing**
  - function should do one thing.They should do it well they should do it only
  - simply it means one function do only one thing more that the function contains then that should be spillited
  - **Sections within function**
    - generatePrimes function is divided into section such as decalaration and intialization etc...
    - it is symptoms of doing more than one things this cannot reasonable
- **One Level of Abstraction Function**

    - Statement within our functions all at the same level of abstraction
        - **Reading code from top to bottom : The Stepdown Rule**
            - function that should be in single level of abstraction it is key to keep function keep it short and to keep it to do one thing
- **Switch Statement**
    - In nature switch statement always do n things but we cannot avoid switch statement but we avoid repeation and we can do with polymorphism
    - Use them only when necessary
    - use abstract factory design pattern and create polymorphism objects
- **Use Descriptive names**
    - The smaller and more focused function is the easier it is to choose a descriptive name
    - **Ward's principle**: "You know you are working on clean code when each routine turns out to be pretty much what you expected."
    - A long descriptive name is better than the short name
    - A long descriptive name is better than long descriptive command
    - use multiple words for the function name to say what it does
- **Function Argument**
    - one or two argument ok three arguments should avoid were possible where more than three use in very special justification
    - using more argument make testing challenge
    - **Common monadic forms**

- there are two very common reason to pass single argument in a function
- 1st is when u are asking question in boolean type
- 2nd is transforming into something else and returning it

- **Flag Argument**
    - Flag arguments are ugly
    - It does one thing if the flag is true and another if the flag is false!
    - should not use flag argument
- **Dyadic Functions**
    - it will be better to convert dyadic to monadic
    - example writeField(outputStream, name) can convert to monadic by any one of these methods
    - Make the writeField a member function of outputStream class so that you can say outputStream.writeField(name)
    - Or make the outputStream a member variable of the current class so that you don’t have to pass it, like this writeField(name).
- **Triadic**
    - Should not use three arguments ,it is difficult to understand,you should think very carefull when creating a triadic argument
- **Argument objects**
    - function seems need more than two or three arguments make some argument wrapped in class of their own
    - Circle makeCircle(double x, double y, double radius);

        - Circle makeCircle(Point center, double radius);
        - both doing same thing 2nd one wraps the number of argument
    - **Arguments List**
        - it is passing list in the argument
- **Have no side effects**
    - function promise to do one thing but also do other hidden thing it will make unexpected changes to variable of its own class
    - function only do mentioned in the function name
    - **Output arguments**
        - For example-   appendFooter(s)   (it is not obvious)
        - public void appendFooter(StringBuffer report)   by seeing the function declaration only it tells what it does.
        - Better use it like   report.appendFooter();
- **Command Query operation**
    - function should either do something or answer something but not both (change state of object or return some information)
    - Doing both leads to confustion
- **Prefered Exception to returning the error code**
    - Returning error code from command function is a violation
    - when your return error code you create the problem that the caller must deal with immediately
    - **Extract try/catch Blocks**
        - Try/catch blocks that are ugly and confuse the structure of the code

and mix error processing with normal processing. So it is better to extract the bodies of the try and catch blocks out into functions of their own

- **Error Handiling is one thing**
    - function that handles error should do nothing else
    - Try keyword should be in the first line of the function, and there should be nothing after the finally/catch block
- **Dependency Magnet**
    - Some class or enum in which all the error codes are defined classes like this are dependency magnet
    - Many other class must import and use them
    - When error enum changes all those classes need to be recompiled and redeployed
    - programmers dont want to add new errors because then they have to rebuild or redeploy So they reuse old error codes instead of adding new ones
    - When you use exceptions rather than error codes, then new exceptions are derivatives of the exception class. They can be added without forcing any recompilation or redeployment
- **Dont repeat yourself**
    - There should not have duplication function by the body also not only the fuction name
    - It main aim to reduce repetition
- **Structured Programming**
    - every function should have one entry and one exit

        - By rules there should be only one return statement,no break or continue statement in loop and never ever any goto statement
    - **How do you write function like this**
        - Clean up the code once it done

Chapter-4 Comments

- Code explantion should be done using comments
- Comments are always failure because we can express without them
- needed time only use the comments
- **Comments Do Not make up for bad code**
    - When you trying to use in bad code like messy code then u can directly clean the code rather than commenting
    - Rather than spending the time in writing comments to the messy code better you can cleaning the messy
- **Explain yourself in code**
    - It takes only few seconds of thought to explain most of your intent of code by your commenting
- **Good Comment**
    - Some comments are necessary
    - only true good comment is an comment
    - **Legal Comment**
        - Sometime our coorprate coding standards fource us to write certain comments for legal reason

    - example copyrights and authorship statements are necessary
  - **Informative Comments**
    - Sometime useful to provide basic information with comments
    - use the name of function to convey the information were possible
  - **Explaination of Intent**
    - Sometime a comments goes beyond just useful information about the implementation and provide the intent behind the decision
  - **Clarification**
    - Sometimes it helps to translate the meaning of some uncertain argument or return value into something that's readable
  - **Warning of Consquence**
    - Sometime it is useful to warn other programmers about certain consquence
  - **Todo Comments**
    - Todo comments explain why the function has implementation and what the function future should be
    - TODOs are jobs that the programmer thinks should be done, but for some reason can’t do at the moment
  - **Amplification**
    - A comment is used increase the importance that may otherwise seem inconsequenial(Not important)
- **Bad Comment**
  - They are crutches or excuses for poor code or justifications for insufficient decisions

- **Mumbling**
  - If you decide to write a comment then spend the time necessary to make sure it is best comment you can write
- **Redundant comment**
  - simple function with a header comment that is completely redundant
  - unnecessary comment
- **Misleading comment**
  - Sometime with all the best intentions a programmer makes a statement in his comments isnt enough to accurate
- **Mandated Comments**
  - It is just plain silly to have a rule that says every function must have javadoc or every variable must have comment
  - It will clutter(mess of code) up the code
- **Journal comment**
  - Sometime the people add a comment to the start of module every time they edit it . so it make not sense
- **Noisy comment**
  - that comment is nothing but is noisy
- **Dont use a comment when you can use function or variable**
  - Sometimes we can able to express the comment we are giving in code with a meaningful name or function.
- **Poistion maker**
  - Sometime programmers like a mark a particular poistion in a source

file

    - Example- // Actions /////////////////////////////////// - it makes more noisy

- **Closing Brace Comment**
    - Don't use comments for marking the end to condition statement and loop statement at the end of the braces
    - while () {
    - } //while
- **Attributions and Bylines**
    - Source code control system are very good at remembering who added what and when
    - Example- /* Added by Rick */ - it used to pollute the code
- **Comment out code**
    - who see that commented-out code won't have the courage to delete it.
    - They'll think it is there for a reason and is too important to delete -bad bottle win
- **Html comment**
    - HTML in source code comments is an abomination(disguest)
- **Nonlocal Information**
    - if you must write a comment then make sure it describe the code it appear near
- **To much Information**

    - Don’t put interesting historical discussions or irrelevant descriptions of details into your comments
  - **Function Header**
    - Short function dont need much description
  - **Javadocs in Nonpublic code**
    - As useful as javadocs are for public APIs, they are anathema to code that is not intended for public consumption
  - **Inobvious connection**
    - The connection between a comment and the code it describes should be obvious(clear).

Chapter-5 Formatting

- **Purpose of Formatting**
  - Code formating important and is about communication and communction is profesional developer first order business
- **Vertical Formatting**
  - Small files are usually easier to understand than the large file
  - **The Newspaper Metaphor**
    - the topmost part of the source file should provide the highlevel concept and algorithm (Topdown apporach)
  - **Vertical Opennes between concepts**
    - Nearly all code is read left to right and top to bottom
    - There are blank lines that are separate the package declaration, the

imports and each of functions

- **Vertical Density**
  - lines of code that are tightly related should appear vertically dense(by using multiple line comment)
- **Vertical Distance**
  - When one function to next function scrolling make long so concept that are closely related should be keep vertically close to each other
    - **Variable Declaration**
      - Variable should be declared as close to their usage as possible
      - local variable should appear a top of each function
      - Control variable for loop should decalred within the loop statement
    - **Instance variable**
      - Instance variable should be decalared in top of the class
    - **Dependent function**
      - If one function call other function they should vertically close
      - the caller should be above the calle
    - **Conceptual Affinity**
      - Certain bits of code want to be near other bits.they have conceptual affinity(understandable)
      - Some concept function(different meaning in differnt field) can be kept in less vertical distance
  - **Vertical Ordering**

        - function call dependencies to point in the downward direction
        - function called should be below the function does the calling
        - we expect the most important concepts to come first and low level detail to come last
- **Horizontal formatting**
    - Limit of 120 character long could be the line
        - **Horizontal Openness and Density**
            - We use horizontal white space to associate strongly related things and disassociate things that are more weakly related
            - surrounded the assignment operartor with white space to accentuate(noticable) them
            - on the other hand should not put space between the function name and opening parthenthesis .because they are closely related
        - **Horizontal Alignment**
            - The horizontal alignment is useless
        - **Indentation**
            - Indent to show hierarchy and structure making the code easier to understand
        - **Dummy scope**
            - fooled by a semicolon silently sitting at the end of a while loop on the same line. Unless you make that semicolon visible by indenting it on it's own line, it's just too hard to see
        - **Team Rules**

        - Every programmer has his own favorite formatting rules but if he works in a team then should be in team rules

----

Chapter-6 Objects & Data Structures

- **Data abstraction**
    - Hiding implementation is about abstraction
    - Hiding implementation is not just a matter of putting a layer of functions between the variables.
- **Data/object Anti-Symmetry**
    - Object is used hide their data behind the abstraction and expose the funtion but data structure expose their data but no meaningful function
    - **Procedural code (**code is using datastructure**)**
        - Makes it easy to add new functions without changing the existing data structure
        - Makes it hard to add new data structures because all the functions must change
    - **OO code(**code using object-oriented**)**
        - Makes it hard to add new functions because all the existing classes must change
        - Makes it easy to add new classes without changing existing function
    - everythings an object is a myth , some times you really do want simple data structure with procedural operating system
    - **The law of demeter**

- the law of demeter says that a method F of a class C should only call the methods
  - class C
  - An object created by F
  - An object passed as an argument of F
  - An object instance variable of C
- other words talk to friends not strangers
- **Train Wrecks**
  - **a series of method calls or a chain where each call return object that can be used to call new method**
  - final String outputDir = ctxt.options.scratchDir.absolutePath;
  - **Why it violate the law?** The method received the parameter car, so all method calls on this object are allowed. But, calling any methods (in this case getAddress() and getStreet()) on the object returned by getOwner() is not allowed
    - Options opts = ctxt.getOptions();
    - File scratchDir = opts.getScratchDir();
    - final String outputDir = scratchDir.getAbsolutePath();
    - LoD violations occur when a module is required to navigate through multiple levels of object relationships, indicating that it is aware of too much about other modules' internal structures.
- **Hybrids**
  - hybrid structure that are half object and half data structure
  - Such hybrids make it hard to add new functions but also make it hard to add

new data structures

- **Hiding Structure**
    - objects with behavior, you shouldn't navigate through them to access their internals (Avoid Direct access)
- **Data Transfer Object**
    - the quintessential(most perfect) form of data Structure is class with public variable and no function sometimes called as DTO
    - DTO are very useful Structures especially when communicating with database or parsing message froms sockets
    - becomes the first in series of translation stage that convert raw data in a database into object in application code
    - **Beans** have private variable manipulated by getter and setter
- **Active Record**
    - Active record are special form DTO's
    - it creates hybrids between data structure and a object
    - Active record as a data strucuture and to create seperate objects that contain the business rule and that hide their internal data

----

Chapter-7 Error Handling

- Error handling is crucial but should not obscure(not well known) logic
- **Use Exception rather than return codes**
    - Return error code leads to the deeply nested structure
    - In olden day they were using error flag -problem is-the

  - caller must checks for errors immediately after the call
- **Write Your try/catch statement first**
  - try portion of a try-catch-finally statement you are starting that execution abort at any time and then resume at catch
  - Your catch has to leave your program in a consistent state, no matter what happens in the try(catch block must handle it in such a way that the program remains stable and predictable.)
  - Try to write tests that force exception and add behavior to your handler to satisfy your tests
- **Use Unchecked Exception**
  - checked exception is occurs in compile time
  - your code literally wouldnt compile if the signature didn't match what your code could do
  - python does'nt have checked exception
  - checked exception is an open/closed principle violation
  - An unchecked exception represents the error in programming logic.
  - checked exception is use low level programming if new exception checked then add throw clause if new exception in different class then also adds in the throw class encapsulation is broken because of these
- **Provide context with exception**
  - Each exception that you throw should provide enough context to determine the source and location of error
  - so you can get stack trace from any exception
  - Create informative error message and pass them along with your

exception

- If you are logging in your application, pass along enough information to be able to log the error in your catch.

- **Define Exception classes in terms of callers needs**
    - Many way to classify the error 1. classify by their source 2.Or their type of failure
    - Exception should written in a way that handles all that can go wrong when users of code calls it instead of writting from the point of failurs or type of failure
- **Define the normal flow**
    - Above steps will cleanly seperate bussiness logic from error handiling
    - instead of using exception everywhere it is better to define a flow in which the caller doesn't have to be suprised with what gets returned from the calle
- **Don't Return Null**
    - if your tempted to return null from a method consider throwing an exception or returning a special case object instead
    - All it takes is one missing null check to send an application spinning out of control.
- **Don't Pass null**
    - returning null from methods is bad but passing null into method is worse
    - When passing null as an argument we'll get a null pointer exception
    - How can fix it? , We could create a new exception type and throw it. but better way is define handler for InvalidArgumentException ... but any

way still we get runtime error

- most programming languages there is no good way to deal with a null that it passed by a caller accidentally

----

Chapter-8 Boundaries

- **Using third party code**
  - There is natural tension between the provider of an interface and the user of an interface
  - if you use a boundary interface like map,keep it inside the class or close family of classes where it is used
- **Exploring and learining boundaries**
  - Third party code helps us get more functionality derived in less time
  - it is good idea to write some tests to learn and understand how to use a third party code
  - Instead of experimenting and trying out the new stuff in our production code, we could write some tests to explore our understanding of the third-party code
- **Learning log4j**
  - log4j package rather than our own custom-build logger
  - log4j is used by developer to keep track of what happen in their software application or online services,It basically huge journal of the activity of system or application
- **Learning tests are better than free**
  - The learning tests were precise experiment that helped increases our

understanding

- When there are new releases of the third party package we run the learning tests to see whether there are behavioral difference
- learning tests verify that the third-party packages we are using work the way we expect them to

- **Using Code That Does Not Yet Exists**
  - if your code is depends upon some other code , even if that code does not exist yet,we can build an interface and connect interface with our code so that later on, the code which we are dependent upon comes into existence only the interface needs to be changed
  - one good things about writting the interface we wish we had is that it's under our control.This helps keep client code clean more readable and focused what it is trying to accomplish
- **Clean boundaries**
  - When we use code that is out of our control , special care must be taken to protect our investment and make sure further change is not too costly
  - We should avoid letting too much of our code know about the third party particularly

----

Chapter-9 Unit Tests

- **The Three Laws of TDD**
  - **First Law:-** You may not write production code until you have written failing unit test
  - **Second Law:-** You may not write more a unit test than is sufficient to fail

    and not compiling is failing (Write just enought test to fail)

  - **Third Law:-** You may not more production code than is sufficient to pass the currently failing test (Dont add extra features or code that isn't required)

- **Keeping tests clean**
  - Test code should maintained   to same standard of quality as their production code
  - As you modify the production code , old test starts to fail and mess in test code makes it hard to get those tests to pass again
  - It requires thought,design and care.It must be kept as clean as production code
  - **Test Enable the -ilities**
    - it is unit test that keeps our code flexible,maintainble and reusable
    - No matter how flexible a production code is without a test case there will be a fear that the changes will introduce bugs.
    - Test enables all the -ilities(set of qualities) because test enable change
- **Clean Test**
  - what makes clean test? Readability is more important in unit test than its production code
  - what makes test readable? the same thing that makes all code readable:clarity,simplicity and density of expression
  - The BUILD-OPERATE-CHECK2 pattern is made obvious by the structure of these tests
  - Each of the test is clearly splits into three parts

        - Build the test data
        - Operate the test data
        - Checks the operation yielded the expected results
    - Notice that the vast majority of annoying detail has been eliminated
    - The test get right to point and use only the data types and function that truly need
    - **Domain specfic testing language**
        - we build up a set of function and utilities that makes use of those APIs and that make the tests more convienient to write and easier to read
    - **A dual standard**
        - The code within the testing API does have a different set of engineering standards than the production code
        - It must be still simple and expressive but it need not be as efficeint as production code
        - there are things that you might never do in production enviroment that are perfectly fine in test enviroment
- **One Assert per Test**
    - Every test function in a JUnit(test automation framework used in java for unit testing) test should have one and only assertion statement (Statement that part of argument)
    - Because it is easy to understand
    - Number of assert it should be minimized
    - **Single concept per test**

    - We want to test a single concept in each test function
    - This test should be split up into three independent tests because it tests three independent things
- **FIRST**
  - Clean tests five rules
  - **Fast**
    - Test should be fast and run quickly,if it slow you will frequently skip those tests
    - If you don't run the test frequency you can't find the bug easily
  - **Independent**
    - Test should not depend on each other
    - when tests depend on each other then the first one to fail causes cascade of downstream failurs
  - **Repeatable**
    - Test should be repeatable in any enviroment(like without internet traveling in train)
  - **Self-Validating**
    - The test should have a boolean output.Either they pass or fail
    - You should not have to read through a log file to tell whether test pass
    - you should not have to manually compare two difference text files to see whether the test pass
  - **Timely**

    - Unit tests should be written just before the production code that makes them pass
    - if you write tests after the production code then you may find the production code hard to test

- **Unit testing-**
  - testing indvidual component of software application

Chapter-10 Classes

- **Class Organization**
  - Class should beign with list of variables , public static constant,if any should come first (you should place any constant that are public and static at the top of class)
  - private static variable followed by private instance variable
  - Start placing all public function right after constant and variables
  - After listing the public function place any private utility function that are called by the public function
- **Encapsulation**
  - Sometimes we need to make a variable or utility function protected so that it can be accessed by the test
  - if the test in the same package need to call a function or access a variable we'll make it protected or package scope
  - it looks for a way maintain privacy
- **Classes Should be small**

- The first rule of class is that they should be small
- With function we measured size by counting **physical lines**.With classes we use different measure.We **count responsibilities**
- **The Single Responsiblity Principle(SRP)**
    - The SRP states that a class or module should have **one and only reason to change**
    - We want our system to be composed of many small classes,not few largers classes
    - Each class encapsulation a single responsbility and has a single reason to change
- **cohesion(fit together well)**
    - Class should have small number of instance variable
    - Each method of class should manipulate one or more of those varibles.
    - When cohesion is high it means that the method and variable of the class are co-dependent and hang together as logical whole
    - **You should try to separate the variables and methods into two or more classes such that the new classes are more cohesive**
- **Maintaing cohesion results in many small classes**
    - Breaking large function into smaller function causes a proliferation(Rapid multiplication) of classes
    - When class losses cohesion splits them into different classes
    - breaking a large function into smaller function often give us the opportunity to split several smaller classes out as well

        - This gives our program a much better organization and a more transparent structure

- **Organizing for change**
    - In a clean system we organize our classes so as to reduce the risk of changes
    - opening a class to modification introduces the risk of breaking existing code
    - modifying a class can impact other parts of the code that depends on it
    - changes may introduce new bugs or issues
- **Isolating from change**
    - there are concrete classes which contain implementation detail(code) and abstract class
    - By minimizing coupling in this way our classes to another class design principle known as the dependency inversion principle(DIP)   (reduces the dependency of one class to another class)
    - In the essence DIP says that our classes should depend upon abstraction not on concrete detailss
    - The changes should not affect the other classes

Chapter-17 smell & Heuristics

- **Comments**
    - **Inappropriate Information**
        - Inappropriate information like source code control system,your issue tracking system or any other record-keeping system

- **Obsolete comment**
  - A comment that has gotten old,irrelevant and incorrect is obsolete
  - Comment gets old quickly , it is best not to write comment that will become obsolete.
  - if you find an obsolte comment it is best to update it or get rid(eliminate) of it as quickly as possible
- **Redundant Comment**
  - A comment with unnecessary information.
  - The comment should say that the code cannot say
- **Poorly Written Comment**
  - Going to write a comment take the time to make sure it is the best comment you can write
  - Choose your word carefully ,use correct grammar and punctuation
  - Don't ramble(writting in a confused way) ,Don't state the obvious be brief
- **Comment-out code**
  - Don't Comment out code,because coworker can be afraid of touching those code and simply rots(go bad) the code
  - When you see commented-out code delete it! dont worry the source code control system still remebers

- **Enviroment**
  - **Build requires more than one step**
    - Building a project should be single trivial operation

    - You should not have to check many little piece out from source code control
    - Build should be in one command to run and one command to check
  - **Tests require more than one step**
    - You should be able to run all the unit test with just one command
    - In the best case you can run all the tests by clicking on one button on IDE
- **Function**
  - **Too many arguments**
    - Function should have small number of argument, No argument is best followed by one ,two and three more than three is questionable should be avoided,
  - **Output arguments**
    - Reader expect arguments to be inputs not outputs
  - **Flag arguments**
    - Should avoid flag arguments passing into function
  - **Dead function**
    - Method that is never called is called dead function,Which should be discarded
- **General**
  - **Multiple language in one source file**
    - Today's modern programming enviroments make it possible to put many different language into a single source file

  - For example, a Java source file might contain snippets of XML, HTML, YAML, JavaDoc, English, JavaScript, and so on
  - This confusing at best and carelessly sloppy(lack of care) at worst
- **Obvious behavior is Unimplemented**
  - When an obvious behavior is not implemented, readers and users of the code can no longer depended on their intution about function name
  - they lose their trust in the original author and must fall back on reading the detail of code
  - the result of a function or class should not be a surprise
- **Incorrect behaviour at the boundaries**
  - It seems obvious to say that code should behave correctly
  - Test all the case look for every bondary condition(every end)
- **Overridden Safeties**
  - It is risky to override safeties
  - Turning off certain compiler warning may helps you get the build to succeed but at the risk of endless debugging session
- **Duplication**
  - DRY Principle (Dont repeat yourself)
  - kent beck made it one of the core principle of extremming programming and called it **"Once and only once"**
  - Every time you see **dplication** in the code it represent a **missed opprtunity for abstraction**
  - modules that have similar algorithms but that don't share similar

lines of code

- **Code at wrong level of abstraction**
    - It is important to create abstraction that separate higher-level general concept from lower-level detailed concept
    - We want all the lower level concept to be in the derivatives and all the higher level concept to be in the base class
    - Good software design required that we seperate concept at different level and place them in different container
- **Base Classes depending on their derivatives**
    - **(base class means whose members are inherited by another class and derived class can only have one direct base class)**
    - Base class should know nothing about their derivatives
    - In the general case we want to be able to deploy derivatives and bases in different jar files
    - And make sure the basic jar files know nothing about the contents of the derivative jar file allow us to deploy our system in discrete in independent component
- **Too much information**
    - Well defined modules have very small interface that allow you to do a lot with little.
    - A poorly defined interface provides lots of function that you must call so coupling is high
    - Good software developer learn to limit what they expose at the interface of their classes and modules
    - Hide your data.Hide your utilities function hide your constant and

your temporaries,Dont create lots of protected variable and function for your subclasses

  - Concentrate on keeping interfaces very tight and very small. Help keep coupling low by limiting information

- **Dead Code**
  - Dead code is code that isn't executed
  - when you find dead code,delete it
- **Vertical Seperation**
  - Variable and function should be defined close to where they are used
  - Local variable should be declared just above their usage and should have a vertical scope
  - private function should be defined just below their first usage
- **Inconsistency**
  - If you do something a certain way do all the similar things in the same way
  - Be carefull with convention you choose and once chosen be careful to continue to follow them
- **Clutter**
  - Variable that aren't used, function that are never called comments that add no information and so forth
  - All these things clutter and should be removed. keep your source files clean,Well organized and free of clutter
- **Artificial coupling**
  - Things that don't depend upon each other should not be artificially

coupled

    - an artificial coupling is coupling between two modules that serves no direct purpose
    - It is a result of putting a variable,constant or function

- **Feature Envy (reveals a method that would work better on different class)**
    - The method of class should be interested in the variable and function of the class **they belong to** and **not** the variable and functions of **other class**
    - Method one class should not be interesting to the method of another class
    - We want to eliminate feature envy because it exposes the internal of one class to another
- **Selector Arguments**
    - Selector argument are combine many function into one.
    - Dont use selector arguments because it will keep one large function that is need to be split
- **Obscured Intent**
    - Code should not be magic or obsucre(not well known) (it shoud not block the intent of code)
    - Code should be placed where a reader would naturally expect it to be
- **Misplaced Responsiblity**
    - One of the most important decision a software developer can **make it where to put code**

  - Code should be placed where a reader would naturally expected it to be.

- **Inappropriate Static**
  - **static method** ideal for utility funtion (that performs specfic task) **do not depend on instance data** and where **polymorphism is not needed** ex:math.max(int a,int b);
  - **nonstatic method:**It belongs to instance of class,use when the method need to access or modify instance data or should **support polymorphism** ex:method for calculating pay the varies by emplyee type
  - **Prefer nonstatic method for flexibility and polymorphism.use static methods only when you're sure they don't need to be polymorphic**
- **Use Explanatory Variables**
  - Breaks the Complex calculation into variables with meaningful names
  - example: String key="" String Value=""   Variables 'key' and 'value' make the code purpose clear
  - Use more explanory variables for better code clarity
- **Function Names Should Say What they do**
  - Function names should clarity indicate their effect and whether they modify the instance or return new one
  - if modify new instance : use names like 'addDaysTo'
  - if returning new instance: use names like 'daysLater'
  - Ensure the function name convey what it does without needing to check the implementation or documentation
- **Understand the agorithm**

    - **Before you consider yourself to be done** with function, make sure you **understand how it works**
    - It is common iteratively test and refine code make it work
    - **passing test case is not enough** ensure you **fully understand how the function work** and that the solution is correct
    - Refractor the function to make it clean and expressive which helps in understanding and confirming its correctness
- **Make logical dependencies physical**
    - If one modules depends upon another that dependency should be physical not just logical
    - The dependent module should not make assumption about the module it depends upon
    - Rather it should ask module about the information depend on
- **Prefer polymorphism to If/else or Switch/Case**
    - Consider using polymorphism instead of switch statement which are often a brute force solution
    - **One Switch Rule:** Limit to one switch statement per type of selection, The switch cases should create polymorphic objects to replace other switch statment elsewhere in the system
- **Follow Standard Convension**
    - Every team should follow a coding standard based on common industry norms
    - It is like team rule follow the team rule for formatting how to name variables,were to declare instance variable etc..
- **Replaced Magic number with named constants**

  - In general it is bad idea to have raw number in your code . You should hide them behind well named constant
  - Example : number 83000 should be hidden behind the constant SECONDS_PER_DAY
  - If you are printing 55 line per page then constant 55 should be hide behind the constant LINES_PER_PAGE
- **Be Precise**
  - When you make a decision in your code make sure you make it precisely - **Know why you have made it and how you deal with any exception**
  - **Handiling Potential Issues:**
    - Checks for null if function might return
    - Use integer for currency and handle routing properly
  - Eliminate ambiguities(unclear) and imprecision caused by disagreement
- **Struture over convention**
  - Use structural design over naming converntions to enforce design decision
  - Example:- Base Classes with abstract methods are more efficient than switch/case statement with named enumeration
- **Encapsulate conditionals**
  - Use funtion to explain the intent of boolean logic with conditional
  - Extracting functions makes boolean logic easier to understand and improves code readability

- **Avoid negative conditionals**
    - Negatives are just a bit harder to understand than positives
- **Function should do one thing**
    - Functions of this kind do more than one things and should be converted into smaller functions,
- **Dont be Arbitary(unfair)**
    - Structure code with clear reason and communicate those reason through the structure
    - Consistent stucture helps others follow conventions and prevent arbitary changes
    - Public class should be top level in their package not nested inside unrelated classes
- **Encapsulate Boundary conditions**
    - Boundary conditions are hard to keep track of
    - put the processing for them in one place
    - Dont let them leak all over the code
- **Function should descend only one level of abstraction**
    - All statement within a function should be at the same level of abstraction which should be one level below the function's name
    - **Example issue:** Mixing level of abstraction original code combined html tag syntax with bussiness logic
    - Seperate levels of abstraction can reveal additional abstraction and improves code organization
- **Keep configurable data at high levels**

            - If a constant like a default value is know as high level dont hide it low level function
            - pass the constant as an argument from high-level function to low level funtion
        - **Avoid transitive Navigation**
            - **Law of Demeter:** Modules should only know about their direct collaborators, not the entire object graph.
            - **Avoid deep navigation**
- **Test**
    - **Insufficient Test**
        - The test are insufficient   so long as there are conditions that have not been explored by the tests or calculation that have not been validated
    - **Use a coverage tool**
        - Coverage tool reports gaps in your testing strategy
        - They make it easy to find modules classes and function that are insufficiently tested
        - most IDE gives you a visual indication marking lines that are coverd green and not covered as red
    - **Don't Skip trivial(important) test**
        - they are easy to write and their documentary value in higher than cost to produce them
    - **An Ignored test is a question about ambiguity(two or more possible ways)**
        - Sometimes we are uncertain about a behavioral detail because the requirement are unclear

- **Test Boundary conditions**
    - Take special care to test boundaries conditions. We often get the middle of an algorithm right but misjudge the boundaries
- **Exhaustively Test near bugs**
    - When you find a bug in a function,it is wise to do an exhaustive(including everything possible) test of that function
- **Patterns of failure are revealing**
    - Sometime you can diagnose   a problem by finding pattern in the way the test case fail
    - complete the test cases,ordered in reasonable way,expose pattern
- **Test Coverage patterns are revealing**
    - Looking at the code that is or is not executed by the passing test gives clues to why the failing tests fail
- **Test Should be fast**
    - A slow test is a test that wont get run.
    - When things get tight,It the slow tests that will be dropped from the suite
    - So do what you must to keep test fast