Clean Code in aNutshell

50 Best Practices to
Start Crafting Quality Software
Immediately

Topics

- Comments
- Single Responsibility
- Naming
- Function
- Overuse Static
- Magic Number
- Long if
- Too Much Inheritance
- Don’t Pass Null
- Exception
- Code Indentation
- Code Grouping
- Deep Nested
- Length Limit
- Code Format
- Prefer Inline
- Return Early
- Null Coalescing
- 1-50-500 Rule

Comments

- Comments are often lies waiting to happen. The code should speak for itself.

- Trying to explain the code functionality in comment actually showcases our inability to write good code.

- Instead of explaining the intent in comments, we should …
  - Select self explanatory object names which can explain the intent correctly.
  - A good variable name can itself explain for what the variable will be used or what kind of value will it store.
  - A good function or a class name can very well explain what purpose is it achieving.

Comment Example

- Bad Code

```
// Example: "Tue, 02 Apr 2003 22: 18: 49 GMT"
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65)) {

}
```

- Good Code

```
if (employee.isEligibleForFullBenefits()) {

}
```

If code is readable you don't need comments

- Bad Code

```
// Check to see if the customer is eligible
// for Platinum Credit Card
if (customer.flags && sal.salary > 20000)
```

- Good Code

```
if (customer.isEligibleForPlatinumCard())
```

Rule of Thumb

Spent more

time & energy

in creating a

self-explanatory code

with

meaningful

`object` names

rather than writing more

~~*comments.*~~

Explain your intention in comments

- It is a good practice to provide a comment for a particular logic decision to understand why it was taken.This helps the programmer to understand why the logic was implemented in this way.

- This comments also opens up the possibility to think of an alternative way to do the same which might improve up readability or performance.

- Example

```
// if we sort the array here the logic becomes simpler in
   calculatePayment() method.

// sorting customers in descending order of income to
   identify eligible customer for platinum card offer
```

Never Leave Code Commented

- Leaving the code commented will raise many doubts for the programmer. Like
    - Why was it commented?
    - Was there any functionality which was left undone or it is just to be removed?
    - Is this code prepared for the next features?
- Version Control can keep code change. Use version control wisely.

Rule of Thumb

Never

leave your

code commented

after making

changes

if you have a good

version control system.

Use Intension-Revealing Name

- Intention revealing names helps us understand what the variable, function or a class does , why it exists and how it is used.

- It takes time to figure out a good name, however it saves equal amount of time writing comments to explain what it does.

- Good intention revealing name saves more effort than it takes to create it.

Meaningful Name

- Avoid Disinformation Name.
- Use Pronounceable Name.
- Use Searchable Name.
- Don’t be cute!
- Avoid Encodings, i.e. :- *phoneString, ageInt*.
- Member Prefix. i.e. :- *m_name*.
- Class and Object name should noun name.
- Use verbs in Method Name. i.e. :- *postPayment()*
- Pick one word per concept.
- Use Problem and / or Solution Domain Name.
- Add Meaningful Context.

Intension Name Example

- Bad Code

```
int m; // Number of months the customer has
defaulted in
```

- Good Code

```
int defaultAttemptsInMonths;
int accountCreationInMonths;
```

Intension Name Example

- Bad Code

```
public List <int[]> getDetails() {
    List <int[]> list1 = new ArrayList<int>();
    for (int[] y : theList)
        if (y[0] == 2)
            list1.add(y);
    return list1;
}
```

Intension Name Example

- Good Code

```
public List <int[]> getDefaultedCustomers() {
    List <int[]> defaultedCustomers = new ArrayList<int[]>();
    for (int[] accNum : customerAccountLists)
        if (accNum[DEFAULT_VALUE] == MAX_ATTEMPTS)
            defaultedCustomers(accNum);
    return defaultedCustomers;
}
```

Use Pronounceable Name Example

- Bad Code

```
private DateTime cusPaySchd;
private DateTime balUpdTmp;
```

- Good Code

```
private DateTime customerPaymentSchedule;
private DateTime balanceUpdateTimestamp;
```

Functions

- The smaller the function, the better.
- A function should only do one thing.
- Statements within our function should be at the same level of abstraction.
- Functions must only do what the name suggests and nothing else.
- Use descriptive names.
- Function should either perform an action or answer a question, but not both.
- Don’t Repeat Yourself (DIY). Avoid duplicate code and/or different code but do the same thing.

Function Should be Small

- Larger Function

```
protected static Map<String, String> getHttpHeaders(HttpServletRequest request) {
  Map<String, String> httpHeaders = new HashMap<String, String>();

  if (request == null || request.getHeaderNames() == null) {
    return httpHeaders;
  }

  Enumeration names = request.getHeaderNames();

  while (names.hasMoreElements()) {
    String name = (String)names.nextElement();
    String value = request.getHeader(name);
    httpHeaders.put(name.toLowerCase(), value);
  }

  return httpHeaders;
}
```

Function Should be Small

- Smaller Function

```
protected static Map<String, String> getHttpHeaders(HttpServletRequest request) {
  if ( isInValidHeader(request) ) {
    return Collections.emptyMap();
  }
  return extractHeaders(request);
}

private static boolean isInValidHeader(HttpServletRequest request) {
  return (request == null || request.getHeaderNames() == null);
}

private static Map<String, String> extractHeaders(HttpServletRequest request) {
  Map<String, String> httpHeaders = new HashMap<String, String>();
  for ( String name : Collections.list(request.getHeaderNames()) ) {
    httpHeaders.put(name.toLowerCase(), request.getHeader(name));
  }
  return httpHeaders;
}
```

Less Parameters are Better

- When a function has more than 2 or 3 arguments then a class should be created for some of the arguments that seem to form a group.

- Bad Code

```
void updateCustomerData(string Name, int Age, date DOB,
string Address)
```

- Good Code

```
void updateCustomerData( CustomerInfo customer )
```

Have no Side Effect

- Bad Code

```
private int totalCustomers.
// ...
void countCustomer(Criteria searchCriteria) {
    Customer [] customers;
    // get customers by criteria
    self.totalCustomers = customers.count();
}
```

- Good Code

```
int getTotalCustomers(Criteria searchCriteria) {
    Customer [] customers;
    // get customers by criteria
    return customers.count();
}
```

Do not Overuse of Static

- Static method turn bad when its become more complex.
- If the code becomes hard wired to the static methods, there is no easy way to replace the reference to the static methods with something else.
- If you are testing your code using automated test, convert static method to something easily mocked.

Overuse of Static Example

- Bad Code

```
public class Utility{
    public static int doSomething() {
        //…
    }
}

public class Client{
    public void foo() {
        //…
        Utility.doSomething();
    }
}
```

Overuse of Static Example

- Good Code

```
public class Utility {
    public int doSomething() {
        //…
    }
}
```

```
public class Client {
    private final Utility utility;
    public Client(Utility aUtility) {
        utility = aUtility;
    }
    public void foo() {
        //…
        utility.doSomething();
        //…
    }
}
```

Magic Number - Replace with const, enum, var

- Magic numbers are the hardcoded values used in the code.

- If the value is used at 10 different places in the code and if it changes then code will have to be changed at all those 10 places.

- A good practice to assign the value to a variable or a constant and then use that variable in the code where ever required.

- This increase another programmers more understand your code and avoid conflict by duplicate values.

Magic Number Example

- Bad Code

```
if (nnumberOfATMTransactions > 5) {      // <- Magic
Number
    accNum.chargeTransactionFee();
}
```

- Good Code

```
const MAX_TRANSACTION_LIMIT = 5;      // <- Replace it

if (nnumberOfATMTransactions > MAX_TRANSACTION_LIMIT) {
    accNum.chargeTransactionFee();
}
```

Long `if` Condition - Replace with Function

- Replace long if conditions with a function. This approach makes the code more readable and easier to maintain.

- Bad Code

```
if !( ((num_defaults == 4) && (num_months == 1)) || ((num_defaults == 4) &&
    (num_months == 3)) || (( num_defaults == 4) && (num_months == 4)) ||
    ((num_defaults == 4) && (num_months == 5)) || (num_defaults == 5)) {

}
```

- Good Code

```
function bool isCustomerDefault(num_months, num_defaults) {
    return
        !( ((num_defaults == 4) && (num_months == 1)) || ((num_defaults == 4)
      && (num_months == 3)) || (( num_defaults == 4) && (num_months == 4))
       || ((num_defaults == 4) && (num_months == 5)) || (num_defaults == 5));
}

if (isCustomerDefault(month, defaults)) { . . . }
```

Following the Single Responsibility Principle. No Large Classes.

- Single Responsibility Principle(SRP) states that the class should have a single reason to change. A class should not have too many responsibilities.

- if the code is kept clean, it will be much easier to identify the function and quicker to maintain.

- Messed up code like messed up library of books. If the books of different categories are properly maintained in different shelves it will be easier to find the one required.

SRS Example

- More than single responsibility, more coupling between 2 purposes.

```
public class CustomerBalAndStatUpdater {
    public int creditBalance() { ... }
    public void updateStatement() { ... }
}
```

- Just one responsibility, easier to read, test, and replace

```
public class CustomerBalanceUpdate {
    public int creditBalance() { ... }
}
public class CustomerStatementUpdate {
    public void updateStatement() { ... }
}
```

Rule of Thumb

To have a

small class

which does just

one single job

so that it is

easier to maintain

and

understand.

Clean Class

- Encapsulation :- keep variables and utility function private. Make protected if needed by the test in the same package.

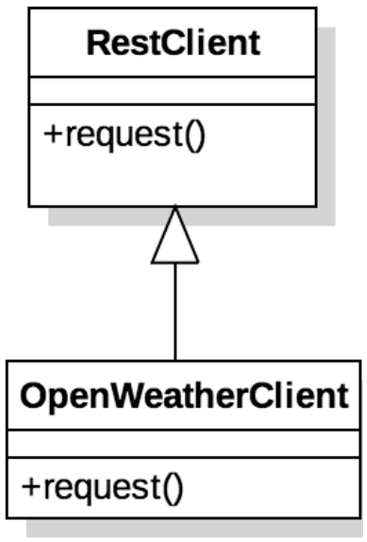
- Classes should be small.

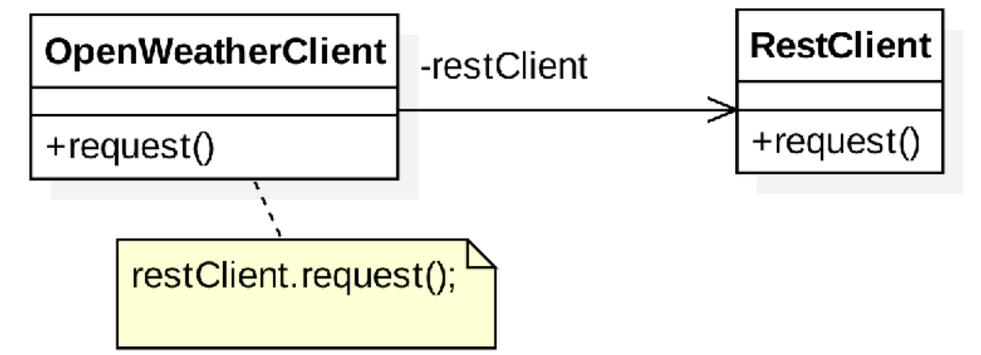
- Keep SOLID.

- Cohesion :- Classes should have a small number of instance variables. Maintaining this cohesion results in many small classes.

Too Much Inheritance

- Both composition and inheritance are ways to reuse code to get additional functionality.
- Unit testing is easy in composition because we know what all methods we are using from other class and we can mock it up.
- One of the best OO design principles is to use composition over inheritance.





Note: In inheritance, we **depend heavily on superclass** and **don't know what all methods of superclass will be used**. So we will have to test all the methods of superclass. This is an **extra work** and we need to do it **unnecessarily** because of inheritance.

Inheritance Example

- Inheritance

```
public class Vehicle {
    private Engine engine;
    public Vehicle(engine: Engine) {
        this.engine = engine;
    }
    public getEngine(): Engine {
        return this.engine;
    }
}

public class Car extends Vehicle {
    public void startEngine() {
        getEngine().start();
    }
}
```

Inheritance Example

- Composition (delegate)

```
public class Vehicle {
    private Engine engine;
    public Vehicle(engine: Engine) {
        this.engine = engine;
    }
    public getEngine(): Engine {
        return this.engine;
    }
}

public class Car {

    private Verhicle verhicle;
    public Car(vehicle: Vehicle) {
        this.verhicle = verhicle;
    }

    public void startEngine() {
        verhicle.getEngine().startEngine();
    }
}
```

Don’t Pass null

- "Null" should not be passed unless an API you are working with requires "Null" as an argument.

- It’s better to use “empty” versions of the type that’s being expected, e.g. an empty array, string or object. This way, the receiving code doesn’t have to check the type.

- Simply do not return or pass `null`.

Don’t Pass null

- In Code we trust

```
public class CustomerAssetCalculator {
    public double totalAsset(double currAccBal, double fixedDepositBal) {
      return (currAccBal + fixedDepositBal);
    }
    ...
}
```

- In Code we don’t trust

```
public class CustomerAssetCalculator {
    public double totalAsset (double currAccBal, double fixedDepositBal) {
        if (currAccBal == null || fixedDepositBal == null) {
            throw InvalidArgumentException(
                “Invalid argument for CustomerAssetCalculator");
        }
        return (currAccBal + fixedDepositBal);
    }
    ...
}
```

Don’t Return null

- Bad Code

```
public class CustomerManager {

    public List<Customer> getActiveCustomers() {
      List<Customer> activatedCustomers = null;
      for (eachCustomer in customers) {
        if (eachCustomer.isActivated) {
          if (activatedCustomers == null) {
            activatedCustomers = new List<Customer>();
          }
          activatedCustomers.Append(eachCustomer);
        }
      }
      return activatedCustomers;
    }
}
```

- Good Code

```
public class CustomerManager {

    public List<Customer> getActiveCustomers() {
      List<Customer> activatedCustomers = new List<Customer>();
      foreach (Customer eachCustomer in customers) {
        if (eachCustomer.isActivated) {
          activatedCustomers.Append(eachCustomer);
        }
      }
      return activatedCustomers;
    }
}
```

Indent Code

- Bad Code

```
function addNewCustomer() {
  if ($ meetsEligibilityCriteria) {
            add_it_now();
            send_welcome_kit();
    }
       else {
             list_missing_details();
    }
        finalize();
}
```

- Good Code

```
function addNewCustomer() {
    if ($ meetsEligibilityCriteria) {
        add_it_now();
        send_welcome_kit();
    }
    else {
        list_missing_details();
    }
    finalize();
}
```

Code Grouping

```
extension ViewController {

    func configurePinningCertificate() {
        let serverTrustPolicies: [String: ServerTrustPolicy] = [
            “service.company.com": .pinCertificates(
                certificates: ServerTrustPolicy.certificates(),
                validateCertificateChain: true,
                validateHost: true
            ),
            “www.google.com": ServerTrustPolicy.disableEvaluation
        ]

        let trustPolicyManager = ServerTrustPolicyManager(policies: serverTrustPolicies)
        self.manager = SessionManager(
            configuration: URLSessionConfiguration.default,
            serverTrustPolicyManager: trustPolicyManager
        )
    }
}
```

Avoid Deep Nesting

- Bad Code

```
function generate_statement() {
    // ...
    if (is_writable(folder)) {
        if (fp = fopen(file_path, "w")) {
            if (statement = generate_current_statement()) {
                if (fwrite(fp, statement)) {
                    // ...
                } else {
                    return false;
                }
            } else {
                return false;
            }
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

Avoid Deep Nesting

- Good Code (with whitespace code grouping)

```
function generate_statement() {
    // ...
    if (!is_writable(folder)) {
        return false;
    }

    if (!fp = fopen(file_path,"w")) {
        return false;
    }

    if (!statement = generate_current_statement()) {
        return false;
    }

    if (fwrite(fp, statement)) {
        return true;
    }

    return false;
}
```

Limit Line Length

- Line length should be limited to an extent where it can be read without scrolling the bar horizontally.

- Our eyes are comfortable reading lines of code which are not horizontally long but are rather tall and narrow.

- Bad Code

```
$ query = "SELECT id, username, first_name, last_name, status FROM users LEFT JOIN us
  USING (users.id, user_posts.user_id) WHERE post_id = '123'";
```

- Good Code

```
$ query = "SELECT id, username, first_name, last_name, status" +
          "FROM users LEFT JOIN user_posts" +
               “USING (users.id, user_posts.user_id) " +
          "WHERE post_id = '123'";
```

Formatting Arrays & Repetitive Variable Declarations

- Bad Code

```
let reviews = [
    "publish_posts": "publish_posts",
    "edit_posts": "edit_blog-reviews",
    "edit_others_posts": "edit_others_blog-reviews",
    "delete_posts": "delete_blog-reviews",
    "edit_post": "edit_blog-review",
    "delete_post": "delete_blog-review"
    "read_post": "read_blog-review"
]
```

Formatting Arrays & Repetitive Variable Declarations

- Good Code

```
let reviews = [
    "publish_posts"     : "publish_posts",
    "edit_post"         : "edit_blog-review",
    "edit_posts"        : "edit_blog-reviews",
    "edit_others_posts": "edit_others_blog-reviews",
    "delete_post"       : "delete_blog-review"
    "delete_posts"      : "delete_blog-reviews",
    "read_post"         : "read_blog-review"
]
```

Return Early & Often

- Return early and often is a good practice of writing a clean code.
- If the code is doing some check it should not wait to process a bunch of code before it fails and returns

- Bad Code

```
if (firstConditionWasMet() {
    if (secondConditionWasMet()) {
        doSecondTask();
    } else {
        doFirstTask();
    }
} else {
    return;
}
```

- Good Code

```
if (firstConditionWasMet() {
    doFirstTask();
    return;
}

if (secondConditionWasMet()) {
    doSecondTask();
    return;
}
```

Prefer Inline Logic

- Inline logics evaluate and return results within a single line. Inline logics are very useful for defining simple algebraic expressions.

- Bad Code

```
private bool getMonth(int month_num) {
    if (month_num == 10) {
        return true;
    } else {
        return false;
    }
}
```

- Good Code

```
private bool getMonth(int month_num) {
    return month_num == 10
}
```

Null Coalescing Operator

- Bad Code

```
private CustomerAddress getCustomerAddress() {
    if (this.customer == null) {
        return new CustomerAddress();
    } else {
        return this.customer.address;
    }
}
```

- Good Code

```
private CustomerAddress getCustomerAddress() {
    return this.customer.address ?? new CustomerAddress();
}
```

Null Coalescing Operator - Swift

- Bad Code

```
let user: Account?
// ...
let request = Request()
request.header[“token”] = user!.hashCredential()
request.header[“role-request”] = user!.role
// ...
```

- Good Code

```
let user: Account?
// ...
if let credential = user {
    let request = Request()
    request.header[“token”] = credential.hashCredential()
    request.header[“role-request”] = credential.role
    // ...
} else {
     fatelerror(“Require user credential!”)
}
```

Declare Variables Close to Use

- Bad Code

```
int month_num = getMonths();
// ...
getStatementInfo();

// ...
string NomineeName;
deleteDecreaseCustomer();

//...
if (month_num == 10) {

}
```

- Good Code

```
// ...
getStatementInfo();

// ...
string NomineeName;
deleteDecreaseCustomer();

//...
int month_num = getMonths();
if (month_num == 10) {

}
```

Prefer Returning Empty Collections instead of Null

- Bad Code

```
function List<Customer> getCustomers() {
      return getCustomerFromDatabase();
 }

 List<Customer> customers = getCustomers();
 if (customers != null) {
      for (Customer c: customers) {
          // ...
      }
 }
```

- Good Code

```
 function List<Customer> getCustomers() {
     List<Customer> customers = getCustomerFromDatabase();
     if (customers == null) {
         return Collections.emptyList();
     } else {
         return customers;
     }
}

List<Customer> customers = getCustomers();
for (Customer c: customers) {
     // ...
}
```

Boundaries

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

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

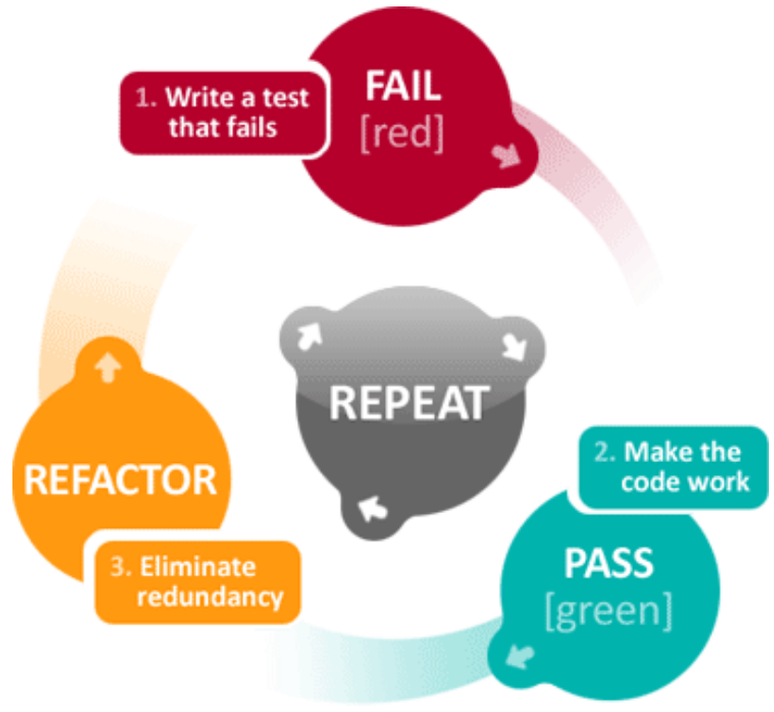
- Clean Boundaries. When using code outside our control, special care must be taken to ensure possible future change is not too costly.

The Three Laws of TDD

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

2. No writing more of a unit test than is sufficient to fail.

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



Keep Test Clean

- Three factors to make tests clean:

    1. Readability

    2. Readability

    3. Readability

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

Five Rules of Clean Test

1. Fast
2. Independent
3. Repeatable
4. Self-Validating
5. Timely

Emergence

- There are 4 simple rules to follow to facilitate the emergence of good design…

1. Runs all tests
2. Contains no duplication.
3. Expresses the intent of the programmers.
    - Good naming
    - Keep functions and classes small.
    - Use standard nomenclature.
    - Well written unit tests
    - Maintain an attitude, desire, and effort to be expressive.
4. Minimized the number of classes and methods.

10-50-500 Rule

- A simple rule to keep the code clean and maintainable is: 10-50-500
  - 10: No package can have more than 10 classes.
  - 50: No method can have more than 50 lines of code.
  - 500: No class can have more than 500 lines of code.

Smells and Heuristics

- Code Comments
  - Inappropriate Information
  - Obsolete
  - Redundant
  - Poorly Written
  - Commented-Out Code
- Environment
  - Build Requires More Than One Step
  - Tests Require More Than One Step
- Functions
  - Too Many Arguments
  - Output Arguments
  - Flag Arguments (booleans)
  - Dead Functions (unused code)
- General
  - Multiple Languages in One Source File
  - Obvious Behavior Is Unimplemented
  - Incorrect Behavior at the Boundaries
  - Overridden Safeties (i.e. overriding serialVersionUID in Java)
  - Duplication
  - Code at Wrong Level of Abstraction
  - Base Classes Depending on Their Derivatives
  - Too Much Information
  - Dead Code
  - Vertical Separation
  - Inconsistency
  - Clutter

Smells and Heuristics

- Artificial Coupling
- Feature Envy (classes should be interested in what they have rather than other classes)
- Selector Arguments
- Obscured Intent
- Misplaced Responsibility
- Inappropriate Static
- Use Explanatory Variables
- Function Names Should Say What They Do
- Understand the Algorithm
- Make Logical Dependencies Physical
- Prefer Polymorphism to If/ Else or Switch/ Case
- Follow Standard Conventions
- Replace Magic Numbers with Named Constants
- Be Precise
- Structure over Convention
- Encapsulate Conditionals
- Avoid Negative Conditionals
- Functions Should Do One Thing
- Hidden Temporal Couplings
- Don’t Be Arbitrary
- Encapsulate Boundary Conditions
- Functions Should Descend Only One Level of Abstraction
- Keep Configurable Data at High Levels
- Avoid Transitive Navigation
- Java
  - Avoid Long Import Lists by Using Wildcards
  - Don’t Inherit Constant

Smells and Heuristics

    - Constants versus Enums (don’t use enums)
- Names
    - Choose Descriptive Names
    - Choose Names at the Appropriate Level of Abstraction
    - Use Standard Nomenclature Where Possible
    - Unambiguous Names
    - Use Long Names for Long Scopes
    - Avoid Encodings (prefixes such as m_)
    - Names Should Describe Side-Effects
- Tests
    - Insufficient Tests
    - Use a Coverage Tool!
    - Don’t Skip Trivial Tests
    - An Ignored Test Is a Question about an Ambiguity
    - Test Boundary Conditions
    - Exhaustively Test Near Bugs
    - Patterns of Failure Are Revealing
    - Test Coverage Patterns Can Be Revealing
    - Tests Should Be Fast

Conclusion

- Codes are sources of compilation to create computer binary. But its used by programmers to maintain the product.
- Writing Clean Code provides longevity, scalability, and reliability of code.
- Clean coding practices should be a habit rather than a one off occurrence.
- Codes should be treated like a painting or artwork. Just like artist or craftsman who really care about quality.
- You can assure that end product will be better in term of performance and maintenance.
- With Clean Code, coding across multiple locations or integration with peers becomes very easy since everyone follows a standard approach.