**“Meaningful Names” for Classes, Functions and Attributes**

Giving of good meaningful names to Classes, Functions and Attributes is one of the most crucial aspects of good programming. We generally tend to pay less attention on it and spend less time thinking of a proper name, and the reason for this is that we don’t understand how important it is to give meaningful names. In this post we will see some of the things along with examples that we need to keep in mind before coming up with a name.

**Name Should be Intention Revealing**

|  |
| --- |
| **transeferMoney(User user1, User user2, long amount)**  From which user to which user is the money transfered ?  **List<User> userList;**  What is the purpose of creating this list ? Where exactly will it be used?  **for(int i=0 ; i <=19 ; i ++){**  **chooseARandomPlayerAndAddToTeam();**  **}**  Do we know why exactly are we iterating till 19? why not more or less? |

**This can be corrected as**

**transeferMoney(User giver, User receiver, long amount)**

**List<User> houseOwners;**

**for(int i=0 ; i <=MAX\_NUMBER\_OF\_TEAM\_MEMBERS ; i ++){**

**chooseARandomPlayerAndAddToTeam();**

**}**

Things to keep in mind

1. What is the type of object
2. What is the significance of Constants or Strings that are used in the code

**Make meaningful Distinctions**

**Class UserService{**

**.....**

**}**

We know that it does something related to User objects. But what exactly ?

**function searchGyms(double lat, double long, long distance)**

**function searchGyms(double lat, double long, long distance, ProgramType programType)**

**function searchGyms(Sting zip, long distance)**

When do we need to call which method is not explicit from the name of the method. One has to go through the parameters to find the exact match.

**public void preparePizza(){**

**Food food1 = getDough();**

**Food food2 = getToppings();**

**....**

**}**

How are food1 and food2 different

**Class UserAuthenticationService{**

**......**

**}**

**function searchGymsForLatLongWithDistance(double lat, double long, long distance)**

**function searchGymsForLatLongWithDistanceAndProgramType(double lat, double long, long distance, ProgramType programType)**

**function searchGymsWithinGivenDistanceFromZip(Sting zip, long distance)**

**public void preparePizza(){**

**Food pizzaDough = getDough();**

**Food pizzaToppings = getToppings();**

**....**

**}**

Rules

* The word variable should never appear in a variable name. The word table should never appear in a table name.
* How is NameString better than Name? Would a Name ever be a floating point number?
* In the absence of specific conventions, the variable moneyAmount is indistinguishable from money, customerInfo is indistinguishable from customer, accountData is indistinguishable from account, and theMessage is indistinguishable from message.

**Class Names**

Class names must be nouns like User, Vehicle, Employee, Account etc. You should not be using generic names that give no extra information information like AccountProcessor, AddressManager, AddressInfo etc.

**Function/Method names**

Methods should always have/be a verb like validate , transferMoney, authenticate etc.

Use static factory methods as compared to overloaded constructors eg

Employee.createNewWithDefaultVaues();

- or -

Employee.createNewWithFistNameAndLastNameString firstName,String lastName)

**Pick one word per context**

Pick one word for a concept and stick to it and whole team should follow it. For example - a finder method can be named as “findByEmail” , “findByAge” or “findByFirstNameAndLastName” - or it can be renamed as “fetchByEmail” , “fetchByAge” , “fetchByFirstNameAndLastName” - it can also be named as “getByEmail” , “getByAge”.

Imagine if mostly in the code you have written methods with the name find\*, but have occasionally used get\* or fetch\*. Now every time one has to use a methods he/she will have to scroll through autosuggest of all the methods names and will then be able to decide if such a method exists or not.

Same holds true for Class names as well. We mostly search for Classes using the shortcuts in IDEs and it becomes easy if Class names are according the expected notations.

Some other Important Rules

* Use computer science (CS) terms, algorithm names, pattern names, math terms, and so forth eg. name AccountVisitor means a great deal to a programmer who is familiar with the VISITOR pattern.
* Don’t prefix the name of the classes in your module with the short form of module name. eg TBBUserAuthenticationService , TBBAccountService

**Functions**

**Small**

* Keep your functions small
* Try to make them even smaller than that.

Not only does this keep the enclosing function small, but it also adds documentary value because the function called within the block can have a nicely descriptive name.

**Do One Thing**

* FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY.

public static String renderPageWithSetupsAndTeardowns( PageData pageData, boolean isSuite) throws Exception {

if (isTestPage(pageData))

includeSetupAndTeardownPages(pageData, isSuite);

return pageData.getHtml();

}

Is the function doing one thing or three things?

If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing.

Error handing is one thing. Thus, a function that handles errors should do nothing else. This implies (as in the examplGhafoorie above) that if the keyword try exists in a function, it should be the very first word in the function and that there should be nothing after the catch/finally blocks.

**One Level of Abstraction per Function**

* We need to make sure that the statements within our function are all at the same level of abstraction. More details are declared in “Level of Abstraction” section.
* We want every function to be followed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions.

**Is having a “Switch Statement” okay?**

**Problems with Switch statements**

* Wherever there is a switch statement, that code has high probability of having similar switch statements at other places in the code. Whenever you change any one of these statements, you will have to make the same change at places also.
  + Switch statement clearly does more than one thing.
  + It violates the Single Responsibility(SRP) because there is more than one reason for it to change.
  + It violates the Open Closed (OCP) because it must change whenever new types are added.

**Solution**

* The most common way of avoiding switch statements is by using polymorphism.

getAverageMonthlySalary(employee){

switch(employee.getDesignation()) {

case MANAGER

salaryCalculator.getHighPayrollSalary();

break;

case HR

salaryCalculator.getMediumPayrollSalary();

break;

case DEVELOPER

salaryCalculator.getLowPayrollSalary();

break;

}

This can be replaced with

public abstract class Employee{

public abstract long getAverageMonthlySalary();

}

public Manager extends Employee{

public long getAverageMonthlySalary(){

...........

}

}

public HumanResourceManager extends Employee{

public long getAverageMonthlySalary(){

...........

}

}

public Developer extends Employee{

public long getAverageMonthlySalary(){

...........

}

}

employee.getAverageMonthlySalary();

* There are a lot of times when we are not able to avoid “switch” statements. The workaround to this is replacing it with an ABSTRACT FACTORY. This factory will be creating instances of similar types. So the switch statement will be hidden behind this factory rather than being duplicated throughout the code.

It is okay to have a switch statement in code if it is not repeated anywhere else in the code.

**Separation of Command and Query functions**

Functions should either be performing some operations on the class attributes/arguments etc, or it should answer something, but it should never do both the things at the same time.

changeDesignationIfEmployeeStillPresent(long employeeId, String newDesignation);

now this can be changed to a better version

boolean isEmployeeStillPresentWithCompany = isEmployeeStillPresent(employeeId);

if(isEmployeeStillPresentWithCompany ){

changeDesignation(employeeId, “Manager”);

}

**Have No Side Effects**

Functions should do the only one thing that the name of the functions communicates.

e.g.

checkIfAllowedToAvailLeaves(int numbeOfDays){

boolean isClientOkayWithLeaves = getClientConfirmationOnLeaves(numbeOfDays);

boolean isTeamOkayWithLeaves = getTeamsConfirmationOnLeaves(numbeOfDays);

if(isClientOkayWithLeaves && isTeamOkayWithLeaves && (numberOfDays > 5)) {

deductOneWeeksSalary();

}

}

**Function Arguments**

Our target should be reduce the number of function arguments to as less as possible. Best is the have functions with no arguments, then is to have functions with one argument and then with two. We should never exceed the number of arguments than three. More detail about it is mentioned here.

**Organizing Functions**

We must place public function first then the private functions that are called by that function. Some people try to keep all the public function at the starting, followed by the protected, package protected and private function. But I follow the practice i.e. explain by Uncle Bob i.e. to keep the public function first followed by the package protected and private functions it calls

**Function Arguments**

Our target should be reduce the number of function arguments to as less as possible. Best is the have functions with no arguments, then is to have functions with one argument and then with two. We should never exceed the number of arguments than three.

Here is the explanation for this recommendation

* Functions with no arguments is always easier to understand than the one with one argument. And a function with one argument is always easier to understand e.g.   
    
  processSalary(); is easier to understand that processSalary(Timesheet timeSheet)   
  and  
  processSalary(Timesheet timeSheet) is easier to understand than processSalary(Timesheet timeSheet, YearlyHolidays holidays)
* More number of argument even make testing difficult as you will have to verify various combinations of arguments. It becomes more difficult when you have same type of argument repeated 2-3 times.
* Passing of “flag arguments” is really ugly and should be totally avoided. Signature of the functions that contains “flag argument clearly shows that the function does more than one thing.
* You can reduce the number of arguments by creating objects combining various related arguments. This is not a bad idea or cheating as the responsibility of valid values of these arguments is of the new Object.

**One Argument Functions**

One might expect these functions to do one of the following

* Asking a question about the argument isHaivingHighPayroll(employee);
* Asking the function to do some processing on the argument and then returning some transformed result, or not returning anything. eg getCurrentMonthsSalary(monthlyHolidayDetails)

We should try to avoid function with one argument that doesn’t fall in one of the above two cases.

Also, we should not be using an output argument

**Output Arguments**

It is very difficult to understand output arguments as we generally expect the output value to be returned by a return statement

e.g. includePageResponse(outputStream)

We should try to replace such functions with a function that return the output

eg above can be replaced with   
byte[] finalHtmlPage = getPageResponse()

outputStream.set(finalHtmlPage);

**Pass & Returning “null”**

* Passing of null as arguments to functions is really a bad practice unless some API expects it.
* You should not return a null from you function. If you don’t have anything to return, return a dummy/default/special Object.

**Level Of Abstraction**

We need to follow this rule both for classes and functions

**Level of Abstraction for Functions**

This is a very important point that we need to keep in mind while writing our functions. If the functions follow the right level of abstraction then they are very clean and are easier to understand.

Functions should just appeal as a bullet points of a writing. Lets see an example for this.

Suppose that in a simple e-commerce we want to place an online order. The things we want to happen when an online order is placed are

**Place Order Story description**

1. Validate Credit Card details for purchase
   1. validate that the card exists
   2. verify that user has enough balance
2. Charge the credit card for the purchased product
   1. Get Total payable amount.
      1. Get product price
      2. Get delivery charges if applicable
      3. Calculate taxes ;
      4. Get Total amount by adding price + delivery charges + taxes;
   2. Fetch the relevant card bank based on the Credit Card type
   3. Ask the bank to charge the total amount
3. Save the order information in database
   1. Save the Order Number and Transaction Id
   2. Save the Billing information
4. Process for home delivery
   1. save the order information in delivery system so that it can be processed by it asynchronously.

Place Order Code

public void processOrder(Product product, CreditCard creditCard){

validateCreditCardIsCapableOfPurchasing(product,creditCard);

Order order = chargeForPurchaesdProduct(product,creditCard);

saveOrderInformationInSystem(product, order);

processForHomeDelivery(creditCard.getUser());

}

public void validateCreditCardIsCapableOfPurchasing(Product product, CreditCard creditCard) {

checkIfCardIsValid(creditCard);

checkIfCardHasEnoughMoney(creditCard, product.getPrice());

}

public void chargeForPurchaesdProduct(Product product, CreditCard creditCard) {

double totalPrice = getTotalChargablePriceProduct(product);

Bank bank = creditCard.getAssociatedBank();

bank.makeNewPurchaseTransaction(creditCard, totalPrice);

}

**Level Of Abstraction for Classes**

The super/abstract classes must not include functions that might not be implemented by some of the implementation or which doesn’t belong to that level. For example

class List<T>{

pubic void add(t)

public void remove(t);

public T get(int index);

public void sort(Comparator comparator);

}

As you see in the above code sort should not be part of the List. It should be part of some subclass like SortabeList etc.

**Classes**

**Organizing members of Class**

Its suggested to keep all the variables first arranges in the order of public, protected, package protected and private. Then you should have public functions that are followed by the protected, package protected and private functions that function calls.

Fundamental Rules

* Just like functions, keep your classes small.
* Try to make the classes smaller.
* For functions the size of the functions was based on the statements/things it does, but for classes we determine the size based on the responsibilities it has.

For example

public interface UserService {

public boolean checkUserNameAndPassword(String emailAddress, String password);

public void sendPasswordResetMail(String emailAddress);

public void changePassword(String emailAddress, String newPassword);

public void sendAccountVerificationMail(String emailAddress);

public User crateNewUser(String emailAddress, String password, Date dateOfBirth );

public User findUserByEmailAddress(String emailAddress);

public User getUser(long userId);

public void update (User user);

public String createSingleSignonId(User user);

public String getUserPreferences(long userId);

public String updatePreferences(UserPreferences userPreferences);

public String resetPreferencesToDefault(long userId);

public List<BankAccount> getBankAccountsOfUser(long userId);

public BankAccount createNewBankAccountForUser(User user);

}

Even a simple class like above is handling different responsibilities. The various responsibilities that the above service handles are user authentication, user management, user preferences management and user bank account management. Hence the above class is eligible for getting splitted into four different Classes.

**Single Responsibility Principle**

* Single Responsibility Principle states that should be only one reason for a class to change.
* This principle states that if a class has two reasons to changes then we will have to split the classes into two different classes. Each class will handle only one responsibility and in future if we need to make a change, we are going to make it in the class that handles it.
* So we want our system to be composed of many small classes, each handling a single responsibility.

**Cohesion**

The variables and methods defined in the class should be very much interdependent on one another. There are few rules that we need to keep in mind to achieve high cohesion

* Classes should have less number of variables
* Each method should be using at least one variable and we should target that methods should use more and more variables
* This rule goes hand in hand with the rule of passing less number of arguments mentioned **here**.
* A method which doesn’t use any of the instance variable is eligible of getting moved to any of the classes and this is what we want to avoid.
* Strategy of having high cohesion and having less arguments leads to having more number of instance variable with only a small subset of methods using those. Not these methods can be extracted out in a separate class.

**Data Structures and Objects**

Uncle Bob brings up this very important point. A lot a developer are not clear that how data structures are different from Objects. This can be correlated to the difference between “Procedural Programming” and “Object Oriented Programming”

By using the most common dependency injection framework “spring” most of us have unintentionally moved toward toward Procedural Programming. Application that use spring/EJB’s are mostly divided into layers i.e. Service, DAO/Repository etc. and we just move the data i.e. POJOs from one layer to other. Most these POJOs have just getter and setters as functions and that is because we want to conform to the Java Bean guidelines.

Lets compare the two examples below

class Employee{

String firstName;

String lastName;

int tenureWithCompanyInDays;

long salaryPerAnum;

int numberOfWorkingDaysTillDate;

}

interface EmployeeService{

int getLeavesAvailed(Employee employee);

int getCurrentFinancialYearsEarningTillDate(Employee employee);

int getSerialNumberOfLaptopUsedByEmployee(Employee employee);

}

This code reveals full information about the Employee Object and also helps in getting an idea about the operations EmployeeService performs on the employee object. We can clearly see that the basic rule of encapsulation is violated here. We should not be exposing all the data of the employee, but should try to expose the functionality by using functions.

Program that is written in the above fashion is not able to make full use of inheritance and polymorphism. This code resembles to the code written using procedural languages.

Lets see the object oriented version of the above code

interface Employee {

int getFullName();

int getEmailAddress();

String getDesignation();

int getLeavesAvailed();

int getCurrentFinancialYearsEarningTillDate();

int getSerialNumberOfLaptopUsedByEmployee();

}

Referring to a blog of coldfusion which explains both the programming styles

<http://objectorientedcoldfusion.org/procedural-vs-object-oriented.html>

**Procedural programming**

These functions typically take some input, do something, then produce some output. Ideally your functions would behave as "black boxes" where input data goes in and output data comes out.

The key idea here is that our functions have no intrinsic relationship with the data they operate on. As long as you provide the correct number and type of arguments, the function will do its work and faithfully return its output.

**Object oriented programming**

In object oriented programming, the data and related functions are bundled together into an "object". Ideally, the data inside an object can only be manipulated by calling the object's functions. This means that your data is locked away inside your objects and your functions provide the only means of doing something with that data. In a well designed object oriented system objects never access shared or global data, they are only permitted to use the data they have, or data they are given.

Both the programming styles has its own advantages.

Object Oriented Programming is better if

1.) There was a lot of code that could be shared and reused between the forms.

2.) The data entry forms were anticipated to change often over time.

3.) Many new data entry forms were anticipated to be added to the project over time.

Procedural Style Programming is better if

1.) The forms were very unique and few elements were shared.

2.) The forms were static and not expected to change much over time.

3.) None or only a few forms were expected to be added to the project over time.

Looking at the above comparison, it is most beneficial to program in Object Oriented Style

**Data Transfer Objects**

We use these simple POJOs to save objects to database or for communicating with third party web-services. These objects just contains properties and getter setters for accessing those.

**Code Smells**

**Comments**

**Inappropriate Comments**

We don’t need to add unnecessary information in the comments which can be fetched through version control eg author name, date of creation of class, ticket/story number corresponding to which the code has been added. We should just be adding technical details to the comments which cannot be be inferred straight away.

**Obsolete Comment**

Comments that give irrelevant information. A common example of this is comments which give more info about the methods and its arguments. With time some of these comments become obsolete as they are not changed when we change the method signature or functionality.

**Redundant Comment**

One example of this is just adding of comments for Java docs which don’t tell anything extra about the methods or its arguments.

**Commented-Out Code**

We also see this smell all the time. There is no use of commenting out code and keeping it as backup. The previous version i.e. the commented out code can always be retrieved from Version Control. I have never seen commented out code being reused. Its just adds more noise to the code and helps out with nothing.

**Environment Setup**

If you have to do more than two three steps to have the complete setup done for a project. I have seen a project which had a document of three pages for the setup and it used to take developers at least one full day to do the setup. Do you think that we should have scripts that do the job rather than describing it in documents ?

**Functions**

**Too many arguments :** having a lot of arguments in function signature

**Output Arguments :** asking to method to add the result in the variable that has been provided as an argument

**Flag Arguments :** passing of a flag which tells if a condition was met or not.

**Dead Function :** Functions that are no more used

The details of all these smells have been discussed in the “Function Arguments” section.

**General**

**Obvious Behavior Is Unimplemented**

Some of the things are expected to be implemented in the code and it should be added to the functionality whenever we are implementing it for the first time. We should not wait for future for implementing simple functionality which we know makes the code(classes & methods) more usable. an example for this is.

DistnceUnits.StringToDistanceUnits(String untits)

Whoever used this method will be expecting or would like to get distance units for “km” and “mi” , “Kilometer” and “Mile”, and case insensitive i.e. “kilometers” and “miles”.

We should be implementing these the very first time we write the code.

**Duplication**

This is the most common and most serious code smell that is seen in the code. NEVER EVER REPEAT THE CODE.

**Code at Wrong Level of Abstraction**

Have been explained very nicely above for both functions and classes

**Prefer Polymorphism to If/Else or Switch/Case**

There are a lot of times when we can get away with the repeated if-else or switch statements by using polymorphism.