Best practices for software coding.

Good coding practices help avoiding having a bad design.
There are some important characteristics of a good design that we should strive to achieve:

- Flexibility - It is easy to change code because it is possible to alter behaviour of a single part of the system, without affecting too many other parts of the system.
- Robustness - When you make a change, no unexpected parts of the system break.
- Mobility - It is easy to reuse in another application because it can be easily disentangled from the current application.
- Readability - It is easy to modify code that is easy to understand.

Coding in General

01 - Self documenting code

Improves: readability

"Any fool can write code that a computer can understand.Good programmers write code that humans can understand."
--- Martin Fowler, Refactoring: Improving the Design of Existing Code

Rather than trying to document how you perform a complex algorithm, try to make the algorithm easier to read by introducing more identifiers. Also, always try to make sure that you are using identifiers that makes the purpose very straightforward. This helps in the future in case the algorithm changes but someone forgets to change the documentation.

Here´s an example that should not be followed:

```
//Please do not do this
for ( i = 1; i <= num; i++ ) {
meetsCriteria[ i ] = True;
}
for ( i = 2; i <= num / 2; i++ ) {
j = i + i;
while ( j <= num ) {
meetsCriteria[ j ] = False;
j = j + i;
}
}
for ( i = 1; i <= num; i++ ) {
if ( meetsCriteria[ i ] ) {
System.out.println ( i + " meets criteria." );
}
}
```

Now, what do you think this routine does? It’s unnecessarily cryptic. It’s poorly documented not because it lacks comments, but because it lacks good programming style. The variable names are uninformative, and the layout is crude. Indentation also does not reflect the flow of code. Here’s the same code improved.

Just improving the programming style makes its meaning much clearer

```
//Yes, do it like this
for ( primeCandidate = 1; primeCandidate <= num; primeCandidate++ ) {
        isPrime[ primeCandidate ] = True;
}

for ( int factor = 2; factor < ( num / 2 ); factor++ ) {
        int factorableNumber = factor + factor;
        while ( factorableNumber <= num ) {
                isPrime[ factorableNumber ] = False;
                factorableNumber = factorableNumber + factor;
        }
}

for ( primeCandidate = 1; primeCandidate <= num; primeCandidate++ ) {
        if ( isPrime[ primeCandidate ] ) {
                System.out.println( primeCandidate + " is prime." );
        }
}
```

The difference between the two code fragments has nothing to with comments. Neither fragment has any. The second one is much more readable, however, and approaches the Holy Grail of legibility: self-documenting code. Such code relies on good programming style to carry the greater part of the documentation burden.
In code well written like this one, comments are often unneccessary.

02 - If statements

Improves: readability

When working on a piece of code that uses if statements, write the nominal path through the code first, then write the unusual cases. Write your code so that the normal path through the code is clear.
Make sure that the rare cases don't obscure the normal path of execution.
This is important for both readability and performance.

Here's a code example that does a lot of error processing, checking for errors along the way.

```
                    ‘ Please do not do this
                    OpenFile( inputFile, status )
                    If ( status = Status_Error ) Then
            error case          errorType = FileOpenError
                    Else
           nominal case         ReadFile( inputFile, fileData, status )
                                If ( status = Status_Success ) Then
           nominal case                 SummarizeFileData( fileData, summaryData, status )
                                        If ( status = Status_Error ) Then
             error case                         errorType = ErrorType_DataSummaryError
                                        Else
           nominal case                         PrintSummary( summaryData )
                                                SaveSummaryData( summaryData, status )
                                                If ( status = Status_Error ) Then
             error case                                 errorType = ErrorType_SummarySaveError
```

```
                                Else
nominal case                            UpdateAllAccounts()
                                        EraseUndoFile()
                                        errorType = ErrorType_None
                                End If
                        End If
                Else
 error case             errorType = ErrorType_FileReadError
                End If
End If
```

This code is hard to follow because the nominal cases and the error cases are all mixed together. It's hard to find the path that is normally taken through the code. In addition, because the error conditions are sometimes processed in the if clause rather than the else clause, it's hard to figure out which if test the normal case goes with. In the rewritten code below, the normal path is consistently coded first, and all the error cases are coded last.
This makes it easier to find and read the nominal case.

```
                ‘ Yes, do it like this
                OpenFile( inputFile, status )
                If status = Status_Success Then
nominal case            ReadFile( inputFile, fileData, status )
                        If status = Status_Success Then
nominal case                    SummarizeFileData( fileData, summaryData, status )
                                If status = Status_Success Then
nominal case                            PrintSummary( summaryData )
                                        SaveSummaryData( summaryData, status )
                                        If status = Status_Success Then
nominal case                                    UpdateAllAccounts()
                                                EraseUndoFile()
                                                errorType = ErrorType_None
                                        Else
  error case                                    errorType = ErrorType_SummarySaveError
                                        End If
                                Else
  error case                            errorType = ErrorType_DataSummaryError
                                End If
                        Else
  error case                    errorType = ErrorType_FileReadError
                        End If
                Else
  error case            errorType = ErrorType_FileOpenError
                End If
```

In the revised example, you can read the main flow of the if tests to find the normal case. The revision puts the focus on reading the main flow rather than on wading through the exceptional cases. The code is easier to read overall. The stack of error conditions at the bottom of the nest is a sign of well-written error processing code.

Exceptions

Exceptions are a specific means by which code can pass along errors or exceptional events to the code that called it. If code in one routine encounters an unexpected condition that it doesn't know how to handle, it throws an exception — essentially throwing up its hands and yelling, “I don't know what to do

about this; I sure hope somebody else knows how to handle it!”
Code that has no sense of the context of an error can return control to other parts of the system that might have a better ability to interpret the error and do something useful about it.

Languages that include support for exception handling are: Java, C++.

We present below a few good practices when working with exceptions.

01 - Always clean up after yourself

Improves: robustness

If you are using resources like database connections or network connections, make sure you clean them up. In Java, you should use the try-catch-finally mechanism.

```
public void dataAccessCode(){
    Connection conn = null;
    try{
        conn = getConnection();
        ..some code that throws SQLException
    }catch(SQLException ex){
        ex.printStacktrace();
    } finally{
        DBUtil.closeConnection(conn);
    }
}
```

The important point is the use of finally block, which executes whether or not an exception is caught. In this example, the finally closes the connection and throws a RuntimeException if there is problem with closing the connection.

02 - Never use exceptions for flow control

Improves: readability, robustness

Exceptions are not a good tool for flow control. This means that, for example: If you are going to divide something by something else, it's a much better idea to check the divisor for zero than to catch a DivideByZeroException.
An exception is meant as an instrument to signal that your application's flow has been broken. That's the one most important feature of the exception handling subsystem: it works across all boundaries of application flow. It's neither designed nor very useful for any situation where the state of the application is known.
If you expect that an error may occur in any specific line of code, and you are going to implement alternate handling for that particular case, don't use an exception. That's not an unexpected application state and shouldn't be handled with an exception.

Here’s another example you shuold not follow:

```
//Please do not do this
public void useExceptionsForFlowControl() {
    try {
        while (true) {
```

```
            increaseCount();
        }
    } catch (MaximumCountReachedException ex) {
    }
    //Continue execution
}

public void increaseCount()
    throws MaximumCountReachedException {
    if (count >= 5000)
        throw new MaximumCountReachedException();
}
```

03 - Avoid empty catch blocks

Improves: robustness

When a method from throws an exception, it is trying to tell the invoker that it should take some counter action. If the checked exception does not make sense in the invoker's context, do not hesitate to throw it again, just do not ignore it by catching it with an empty block and then continue as if nothing had happened.

```
//Please do not do this
public void someMethod() {
    try {
        someDangerousOperation();
    } catch (YouShouldHandleException ex) {
        //I´m not in the mood to write this kind of code right now
    }
    //Continue execution
}
```

04 - Avoid throwing exceptions in constructors and destructors

Improves: robustness

The rules for how exceptions are processed become very complicated very quickly when exceptions are thrown in constructors and destructors. In C++, for example, destructors aren't called unless an object is fully constructed, which means if code within a constructor throws an exception, the destructor won't be called, and that sets up a possible resource leak (Meyers 1996, Stroustrup 1997).
Similarly complicated rules apply to exceptions within destructors. It's better programming practice simply to avoid the extra complexity such code creates by not writing that kind of code in the first place.

05 - Throw exceptions at the right level of abstraction

Improves: readability, flexibility, mobility

A routine should present a consistent abstraction in its interface, and so should a class.
The exceptions thrown are part of the routine interface, just like specific data types are.
When you choose to pass an exception to the caller, make sure the exception's level of abstraction is consistent with the routine interface's abstraction. Here is an example of what not to do:

```
//Please, do not do this
class Employee {
        ...
```

```
        public TaxId getTaxId() throws IOException {
                ...
        }
        ...
}
```

The getTaxId() code passes the lower-level IOException back to its caller. It doesn't take ownership of the exception itself; it exposes some details about how it is implemented by passing the lower-level exception to its caller.
This effectively couples the routine's client's code not only to the Employee class's code, but to the code below the Employee class that throws the IOException. Encapsulation is broken, and intellectual manageability starts to decline. Instead, the getTaxId() code should pass back an exception that's consistent with the class interface of which it's a part, like this:

```
class Employee {
        ...
        public TaxId getTaxId() throws EmployeeDataNotAvailable {
                ...
        }
        ...
}
```

The exception-handling code inside getTaxId() will probably just map the IOException to theEmployeeDataNotAvailable exception, which is fine because that's sufficient to preserve the interface abstraction.

06 - Include all information that led to the exception in the exception message

Improves: robustness, flexibility, mobility

Every exception occurs in specific circumstances that are detected at the time the code throws the exception. This information is invaluable to the person who reads the exception message.
Be sure the message contains the information needed to understand why the exception was thrown.
For example, if the exception was thrown because of an array index error, be sure the exception message includes the upper and lower array limits and the value of the illegal index.

High Level Design Principles

In a higher level of abstraction, there are a few design principles that a programmer should have in mind at all times. Applying these principles correctly leads to software that is easy to change and evolve.

01 - Encapsulate what varies

Improves: flexibility

When designing software, look for the portions most likely to change and prepare them for future expansion by shielding the rest of the program from that change. Hide the potential variation behind an interface. Then, when the implementation changes, software written to the interface doesn't need to change. This is called encapsulating variation.
Let's look at an example. Let's say you are writing a paint program. For your first version, you choose to only handle .bmp files. You know that in the future, you'll want to load other files like .jpg, .gif, .png, etc.

The naive way to implement the loading of a bmp is to write some functions that do just that. They load

the bitmap file into your internal version of it. If you are using an api to load them, you might even be tempted to put the correct API calls directly in the Open button handler. Doing either will make life harder later.
Every place that has to load the files (the button handler, the file recovery routine, the recently-used menu selections, etc.) will have to change when you add support for JPEG and PNG.

A better solution would be to create an interface IImageLoader and implement it in a BMPLoader. Then all code handling loading files will call methods on IImageLoader and won't care (or know) about the specifics of the type of image being loaded. Adding JPEGLoader and PNGLoader will require changing much less code. If done right, changes will be isolated to just one place.

02 - Depend on abstractions, not concrete classes.

Improves: flexibility, mobility, robustness

When designing a module, try to isolate its dependencies and use them throug an abstraction. Do not depend on concrete classes. Let´s say you are writing a spam tool. Maybe your code would look like this:

```
class Spammer{
        public void sendSpam() {
                String[] mailAddresses = retrieveTwentyThousandAddressesFromDatabase();
                String message = getSpamMsg("enlarge_that.txt");
                MailSender sender = new SMTPMailSender("smtp.evilspammer.com");
                for(int i=0; i<mailAddresses.length; i++){
                        sender.send(mailAddresses[i], message);
                }
        }
}
```

A key point to take notice here is that our Spammer depends on a MailSender object. In this example it directly instantiates and uses a SMTPMailSender. There are a few reasons you should not do that.

For starters, this code is not testable. If you write a test case for it, it will actually send 20 thousand emails each time it is ran.
Second, it´s not reusable. Let's say there´s a new requirement that now you should be able to send spam using physical mail. Also, there is a PhysicalMailSender implementation of the MailSender interface that actually prints a letter that can be shipped afterwards. You would have to write a new PhysicalSpammer tool that instantiates a PhysicalMailSender instead of a SMTPMailSender. Wouldn´t it be better if you could reuse the same Spammer class and find a way to swap just the MailSender implementation?

So, the Spammer class could be redesigned to look like this:

```
class Spammer{
        private MailSender sender;

        public void setSender(MailSender sender) {
                this.sender = sender;
        }

        public void sendSpam() {
                String[] mailAddresses = retrieveTwentyThousandAddressesFromDatabase();
                String message = getSpamMsg("enlarge_that.txt");
                for(int i=0; i<mailAddresses.length; i++){
                        sender.send(mailAddresses[i], message);
```

```
                }
        }
}
```

The new improved Spammer class no longer knows the MailSender implementation it uses. Instead, a working MailSender must be provided to it.
It is now possible to build two instances of Spammer class, one for SMTP spam and another for physical spam. No Code duplication involved.
Also, you could write a test case that provides a dummy implementation of MailSender, just to test the Spammer class' logic.

This principle is also called the Dependency Inversion Principle (DIP) or The Holliwood Principle (You don´t call us, we call you).
The act of setting the MailSender in the Spammer class is called Dependency Injection.

03 - Low coupling and high cohesion.

Improves: flexibility, mobility, robustness
Much of software design involves the ongoing question, where should this code go? As a good programmer you should always be looking for the best way to organize your code to make it easier to write, easier to understand, and easier to change later. Structure your code well, and you''ll go on to fame and glory. Structure it badly, and the developers who follow you will curse your name for eternity.

In particular, you should strive to achieve three specific things with your code structure:

- Keep things that have to change together as close together in the code as possible.
- Allow unrelated things in the code to change independently (also known as orthogonality).
- Minimize duplication in the code.

By and large, these goals are closely related to the classic code qualities of cohesion and coupling. I achieve these goals by moving toward higher cohesion and looser coupling. Of course, first we need to understand what these qualities mean and why coupling and cohesion are helpful concepts.

Coupling:
A module (package, classe, method) X is said to be coupled to a module Y when changes in Y requires that X is also changed. Therefore, coupling is a measure of how interconnected those classes or subsystems are. Tight coupling means that related classes have to know internal details of each other, changes ripple through the system, and the system is potentially harder to understand. A good metaphore is this:

- Loose coupling: You and the guy at the convenience store. You communicate through a well-defined protocol to achieve your respective goals - you pay money, he lets you walk out with the bag of Cheetos. Either one of you can be replaced without disrupting the system.
- Tight coupling: You and your wife. You both know each other's "internal workings". Your resposibilities are mixed together, there's no clear rule as to who should do the dishes next time. It's impossible to replace either one of without having a great deal of impact in the other - and on those who depend on you.

Cohesion:
Cohesion is a measure of how closely related the members (classes, methods, functionality within a method) of a module are to the other members of the same module. It has to do with whether a class has a well-defined role within the system. A system consisting of cohesive classes and subsystems is like a well-designed online discussion group. Each area in the online group is narrowly focused on one specific topic so the discussion is easy to follow, and if you're looking for a dialog on a certain subject, there's only one room you have to visit. A good metaphore is:

- Low cohesion: The convenience store. You go there for everything from gas to milk to ATM banking. Products and services have little in common, and the convenience of having them all in one place may not be enough to offset the resulting increase in cost and decrease in quality.
- High cohesion: The cheese store. They sell cheese. Nothing else. Can't beat 'em when it comes to cheese though.

The question is: how do you decrease coupling and increase cohesion. Rather than trying to present a list of refactoring techniques, lets jump straight to an example. Here's the bad one:

```
public class InapropriatelyIntimateItemProcessor() {
	public void process(){
		string connectionString = getConnectionString();
		SqlConnection connection = new SqlConnection(connectionString);
		DataServer1 server = new DataServer1(connection);

		int daysOld = 5;
		using (SqlDataReader reader = server.GetWorkItemData(daysOld)) {
		while (reader.Read()) {
			string name = reader.GetString(0);
			string location = reader.GetString(1);

			processItem(name, location);
		}
	}
}
```

The class above must perform some data processing on items retrieved from a database. This is an example of bad (strong) coupling because of its inappropriate intimacy. In this case, the process method has to know a lot of the inner workings of DataServer1 and a bit about the SqlDataReader class. This method is a customer that greets the guy at the convenience store with a french kiss.

It is also an example of bad (low) cohesion because it does many things: configuring the DataServer connection, reading items from database and business logic.

Not let´s rewrite the code to fix those problems.

```
public class PoliteItemProcessor() {
	public void process(){
		DataServer2 server = new DataServer2();
		foreach (DataItem item in server.GetWorkItemData(5)) {
			processItem(item);
		}
	}
}
```

Now that´s much better. The SqlConnection and SqlDataReader object manipulation is encapsulated inside the DataServer2 class. DataServer2 is also assumed to be taking care of its own configuration, so the new Process method doesn't have to know anything about setting up DataServer2. The DataItem objects returned from GetWorkItemData are also strongly typed objects.

As a result:

- The new processor is easier to read and understand. There´s only one concern on the programmers mind: business logic.
- The new processor is also much less subject to impact on changes on the way the DataServer works.You could switch the data store to an Oracle database or to an XML file without any effect on the process method.

So remember: design your modules so that they provide a service as the cheese store, and consume services as the convenience store customer!

04 - Talk only to your immediate friends.

Improves: readability, flexibility, mobility, robustness

This is also known as "Principle of Least Knowledge" or "Law of Demeter".
This principle prevent us from creating designs with strong coupling.
When you build a lot of dependencies between many classes, the resulting system will hard to mantain and complex for others to understand.

The principle provide some guidelines: take an object and, for any method in that object, the principle tell us that we should only invoke methods that belong to:

- The object itself
- Members of the object
- Objects passed in as a parameter to the method
- Objects the method creates or instantiates

Of course, this is only a tool to help identify points of strong coupling. A violation of the principle is a “smell” of bad design. It doesn´t mean you can never takeAnObject.getAnotherOne().andCallAMethod(“violating the principle!”). There are cases when code like that is justifiable and does not add a great deal of coupling.
As it is also possible for a system to follow the principle 100% and still have strong coupling. When troubled by a design decision, talk to other fellow programmers, follow your instincts. Common sense is usually better for a specific situation than a generic rule.

05 - Single Responsibility Principle

Improves: readability, flexibility, mobility, robustness

This principle is about cohesion. It simply states that a class should have one and only one reason to change. For example, if a change to the business rules causes a class to change, then a change to the database schema, GUI, report format, or any other segment of the system should not force that class to change.
There is a clear reason for this: making sure your classes have a single purpose in life makes your system more receptive to change, more stable. On the other hand, if a class has more then one

responsibility, then these responsibilities become coupled. Changes to one responsibility may impair or inhibit the class' ability to meet the others. This kind of coupling leads to fragile designs that break in unexpected ways when
changed.
Changes can also ripple through the system as classes depending on our multi-responsibility class (which is more likely to change than a single responsibility class) have clients in other programs. It's worth considering your responsibilities and favoring more discrete classes with fewer methods.

06 - You Ain't Gonna Need It (YAGNI)

This principle is about avoiding overengineering.
Always implement things when you actually need them, never when you just foresee that you need them. Even if you're completely sure that you'll need a feature later on, don't implement it now. Usually, it'll turn out either a) you don't need it after all, or b) what you actually need is different from what you foresaw needing earlier.
This doesn't mean you should avoid building flexibility into your code. It means you shouldn't overengineer something based on what you think you might need later on.

There are two main reasons to practice YAGNI:
You save time, because you avoid writing code that you may not need.
Your code is better, because you avoid polluting it with 'guesses' that turn out to be more or less wrong but stick around anyway.

A good example of violating this principle is over-generalization.
Let´s say you are building a electronic health record system, which is supposed to be used by doctors. While designing the system, you realize that with a few changes, the same system might also be useful to veterinarians. So instead of creating a concrete "Patient" class, you create an "AbstractPatient" and a concrete HumanPatient subclass, thinking that maybe later you might add a DogPatient, CatPatient and so on.
Just don´t. You should really cross that bridge only when you get there. If you follow all the other principles, the new veterinarian-specific functionality should still be easy too add.

07 - Keep It Stupidly Simple (KISS)

This principle is not just about avoiding overengineering. It states that when facing a problem, we should ask ourselves the question: "What's the simplest thing that could possibly work?" - and do just that. But there´s a trick to that: you can't make anything simple (or, more accurately, simplest) until you fully understand it.

Therefore, to truly **do the simplest thing that could possibly work**, you need to know everything there is to know about the problem, which means **big research up front**.

However, what most people think or intend when they hear or say do the simplest thing that could possibly work is: take the simplest approach you can think (right now!) for solving the immediate problem, paying no attention to how a simple solution to the immediate problem might affect the overall complexity of the surrounding context.

So remember this: there´s a huge difference between simple and easy. Sometimes a software engineer must think long and hard before figuring out the simplest solution to a problem.

Also know that the natural process to **do the simplest thing that could possibly work** is to make small incremental refactorings, in other words, you usually don´t get it right the first time. Each refactoring must seek to simplify of remove an element of the architecture until the point you can´t remove anything else without violating a requirement of some sort.
Everything should be made as simple as possible, but not simpler.
~Albert Einstein

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.
~Antoine de Saint-Exupery
French writer (1900 - 1944)

Unit tests and TDD

Test Driven Development is a programming technique that combines: 1) test-first development - where you write a test before you write just enough production code to fulfill that test; and 2) refactoring. In TDD, the developer goes through a series of red/green/refactor cicles, following these steps:

- **Add a test**: Each new feature begins with writing a test, which will obviously fail (red!) because it was written before the feauture itself. To write a test, you must clearly understand the feature's specification and requirements.
- **Implement the feature**: Write some code that will cause the test to pass. The new code written at this stage will not be perfect and may, for example, pass the test in an inelegant way.
- **Run the automated tests and see them succeed**: If all test cases now pass (green!), you can be confident that the code meets all the tested requirements. That is enough for you to jump to the next step.
- **Refactor code**: Now the code can be refactored as necessary. By re-running the test cases, you can be confident that the changes do not break existing functionality.
- **Repeat**: Add more tests. The cycle is repeated to push forward the functionality, exercising different inputs and expected outputs.

There are many reasons why doing TDD is such a good idea. A few are listed below.

1) To write a test, you must clearly understand the feature's specification and requirements **before** you start coding. This is a subtle but crucial difference compared to writing tests after the code was written. Rather than simply validating correctness, TDD drives design by encouraging a “design by contract” approach.
2) By making it a rule to always write the test first, you ensure that every feature is tested. When writing tests after the code was written, developers tend to push (or be pushed) on to the next feature, neglecting tests entirely.
3) It gives you confidence to refactor code and add functionality. The test codebase works as a safety net that prevents functionality from silently breaking.