**It is often heard that one of the most important goals in Object-Oriented Design (and code design in general) is to have High Cohesion and Loose Coupling. What does it mean? Why is it that important and how is it achieved?**

*What I believe is this:*

*Cohesion refers to the degree to which the elements of a module/class belong together, it is suggested that the related code should be close to each other, so we should strive for high cohesion and bind all related code together as close as possible. It has to do with the elements within the module/class.*

*Coupling refers to the degree to which the different modules/classes depend on each other, it is suggested that all modules should be independent as far as possible, that's why low coupling. It has to do with the elements among different modules/classes.*

*To visualize the whole picture will be helpful*

[*https://stackoverflow.com/questions/14000762/what-does-low-in-coupling-and-high-in-cohesion-mean*](https://stackoverflow.com/questions/14000762/what-does-low-in-coupling-and-high-in-cohesion-mean)

[*https://stackoverflow.com/questions/3085285/difference-between-cohesion-and-coupling*](https://stackoverflow.com/questions/3085285/difference-between-cohesion-and-coupling)

**Why does array index start with '0' in most of languages?**

*In C, the name of an array is essentially a pointer, a reference to a memory location, and so the expression array[n] refers to a memory location n-elements away from the starting element. This means that the index is used as an offset. The first element of the array is exactly contained in the memory location that array refers (0 elements away), so it should be denoted as array[0].*

**How do tests and TDD influence code design?**

*Most people think that Test-Driven Development is tool to write code with lesser number of bugs. But ,in reality, that is the by-product of TDD. TDD is more of a tool for code designing.*

*Overall, TDD helps in quality code development in following ways:-*

*It makes you think about your code design and requirements at every stage, thereby ensuring that you are actually implementing what is required.*

*You are forced to write testable code, thereby ensuring that your code has loose coupling and high cohesion.*

*If your code is getting difficult to test, mostly is signifies that there is some issue with your design(Your code is too coupled or not isolated enough)*

*With that said, I tend to disagree with people that think if you follow TDD blindly you'd always end up with good code design(because that depends more on you and your knowledge of Sofware Design), but I believe there is a good chance you would.*

**Write a snippet of code violating the Don't Repeat Yourself (DRY) principle. Then, explain why is it a bad design, and fix it.**

*Short of a full framework, what I tend to do for content (even if it contains logic) is separate it out into files and use another logical evaluation to merge them together (mangle them) and then evaluate the templating logic after that. This chunkifies your content and makes chunks sharable / reusable on common state.*

*This way each final template buffer is a flattened tree of discrete re-usable content nuggets that you can store on disk or a database. Even something as simple as a little parser that replaces:*

*<h1>{{insert:shared\_page\_header}}</h1>*

*With shared\_page\_header.txt helps keep things separate. It also forces you to look at separation on concerns even in the logic that is embedded in your templates. Manageable, reusable chunks of anything (dynamic or not) are always the way to go. Your templates are just strings until evaluated, so treat them as shared components merged into a big-dirty-string(TM) and then evaluated.*

**What's the difference between cohesion and coupling?**

*Cohesion refers to what the class (or module) can do. Low cohesion would mean that the class does a great variety of actions - it is broad, unfocused on what it should do. High cohesion means that the class is focused on what it should be doing, i.e. only methods relating to the intention of the class.*

*Example of Low Cohesion:*

*-------------------*

*| Staff |*

*-------------------*

*| checkEmail() |*

*| sendEmail() |*

*| emailValidate() |*

*| PrintLetter() |*

*-------------------*

*Example of High Cohesion:*

*----------------------------*

*| Staff |*

*----------------------------*

*| -salary |*

*| -emailAddr |*

*----------------------------*

*| setSalary(newSalary) |*

*| getSalary() |*

*| setEmailAddr(newEmail) |*

*| getEmailAddr() |*

*----------------------------*

*As for coupling, it refers to how related or dependent two classes/modules are toward each other. For low coupled classes, changing something major in one class should not affect the other. High coupling would make it difficult to change and maintain your code; since classes are closely knit together, making a change could require an entire system revamp.*

*Good software design has high cohesion and low coupling.*

**What is refactoring useful for?**

*Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying a series of small behavior-preserving transformations, each of which "too small to be worth doing". However the cumulative effect of each of these transformations is quite significant.Jun 22, 2009*

*Scisr is a simple, standalone refactoring tool for PHP.*

[*http://iangreenleaf.github.io/Scisr/*](http://iangreenleaf.github.io/Scisr/)

*I know this is an old question, but since it's one of the best rundowns of PHP refactoring tools, I thought I would add my new project to the list.*

*A lot of my design goals sprung out of the inadequacies of other items mentioned here - they are tied into certain IDEs, or try to dictate your testing practices, or require a whole bunch of rote manual labor. I am hoping to create an overall better experience in Scisr. It's simple to install and run. It tries to be clever, but not too clever. It does the tasks that I have found to lend themselves best to automation. Enjoy!*

**Are comments in code useful? Some say they should be avoided as much as possible, and hopefully made unnecessary. Do you agree?**

*Only if the comment describes what the code is doing.*

*If I wanted to know what was happening in a method or block, I would read the code. I would hope, anyway, that any developers working on a given project were at least familiar enough with the development language to read what is written and understand what it is doing.*

*In some cases of extreme optimization, you might be using techniques that makes it difficult for someone to follow what your code is doing. In these cases, comments can and should be used to not only explain why you have such optimizations, but what the code is doing. A good rule of thumb would be to have someone else (or multiple other people) familiar with the implementation language and project look at your code - if they can't understand both the why and the how, then you should comment both the why and the how.*

*However, what's not clear in the code is why you have done something. If you take an approach that might not be obvious to others, you should have a comment that explains why you made the decisions that you did. I would suspect that you might not even realize that a comment is needed until after something like a code review, where people want to know why you did X instead of Y - you can capture your answer in the code for everyone else who looks at it in the future.*

*The most important thing, though, is to change your comments when you change your code. If you change an algorithm, be sure to update the comments with why you went with algorithm X over Y. Stale comments are an even bigger code smell.*

**What is the difference between design and architecture?**

*You're right yes. The architecture of a system is its 'skeleton'. It's the highest level of abstraction of a system. What kind of data storage is present, how do modules interact with each other, what recovery systems are in place. Just like design patterns, there are architectural patterns: MVC, 3-tier layered design, etc.*

*Software design is about designing the individual modules / components. What are the responsibilities, functions, of module x? Of class Y? What can it do, and what not? What design patterns can be used?*

*So in short, Software architecture is more about the design of the entire system, while software design emphasizes on module / component / class level.*

**Why in TDD are tests written before code?**

*Some good comments here, but I think that one thing is getting ignored.*

*writing tests first drives your design. This is an important step. If you write the tests "at the same time" or "soon after" you might be missing some design benefits of doing TDD in micro steps.*

*It feels really cheesy at first, but it's amazing to watch things unfold before your eyes into a design that you didn't think of originally. I've seen it happen.*

*TDD is hard, and it's not for everybody. But if you already embrace unit testing, then try it out for a month and see what it does to your design and productivity.*

*You spend less time in the debugger and more time thinking about outside-in design. Those are two gigantic pluses in my book.*

**Is there a difference between using delegation and using inheritance? [This question is from The Pragmatic Programmer, by Andrew Hunt and David Thomas]**

**Pros and cons of holding domain logic in Stored Procedures.**

*In theory, the pros and cons are as so:*

*Pros:*

*One place to contain all of the business logic*

*Possibly faster applications because multiple SQL queries and such can be performed in one "round trip" to the database*

*Trivial to make use of the stored procedures from multiple applications*

*Cons:*

*A DBA will be required for performance tuning*

*All developers will have to be very well versed in your particular SQL dialect(T-SQL, Pl/SQL, etc)*

*SQL code isn't as expressive and thus harder to write when covering higher level concepts that aren't really related to data*

*A lot more unnecessary load on the database*

**In your opinion, why have Object-Oriented Design dominated the market for so many years?**

*Object Oriented Programming has great advantages over other programming styles: Code Reuse and Recycling: Objects created for Object Oriented Programs can easily be reused in other programs. Encapsulation (part 1): Once an Object is created, knowledge of its implementation is not necessary for its use.*

**What would you do to understand if your code has a bad design?**

*The biggest clue for me is:*

*When you have to go back and add/modify a feature, is it difficult? Do you constantly break existing functionality when making changes?*

*If the answer to the above is "yes" then you probably have a poor overall design.*

*It is (for me at least) a bit difficult to judge a design until it is required to respond to change (within reason of course; some code is just bad and you can tell right away, but even that comes with experience.)*