Not quite what you are looking for? You may want to try:

- Test Driven / First Development by Example
- Designing Application Using Test Driven Development

highlights off

"Design, **Test**, Developement, **Test**" in-stead of "**Test** **Driven** **Development**" (DTDT over TDD)

Mahmud Hasan, 27 Nov 2013 CPOL

★★★★★ 4.60 (19 votes)

Rate this: ☆☆☆☆☆

The only good thing of **Test** **Driven** **Development** is, it ensures the maximum **test** coverage but TDD puts you in risk of a week design. DTDT tries to solve this while ensuring maximum **test** coverage.

Introduction

In modern software industry **Test** **Driven** **Development** or TDD is a big buzz word. In most of the job advertisement about software **development**, you will see knowledge or experience on TDD is must or a plus. But, in reality how many applications are being developed in a so called TDD model? I am in no way talking against writing **test**s. I have no doubt that having maximum possible **test**s can improve the quality of the application in-terms of reducing the number of bugs, reducing the number of bugs to recur and reducing the ultimate time to develop and maintain an application. But, if you practice **test** **driven** **development** according to its definition how much realistic is that? Let's 1st see what actually TDD means in theory.

What is TDD?

> *[1]* **Test**-**Driven** **Development** (TDD) is a technique for building software that guides software **development** by writing **test**s. It was developed by Kent Beck in the late 1990's as part of Extreme Programming. In essence you follow three simple steps repeatedly:
>
> 1. Write a **test** for the next bit of functionality you want to add.
> 2. Write the functional code until the **test** passes.
> 3. Refactor both new and old code to make it well structured.

Let's try to see the process with a picture:

Here you can see a traditional diagram (Figure 1) of **Test** **Driven** **Development**:

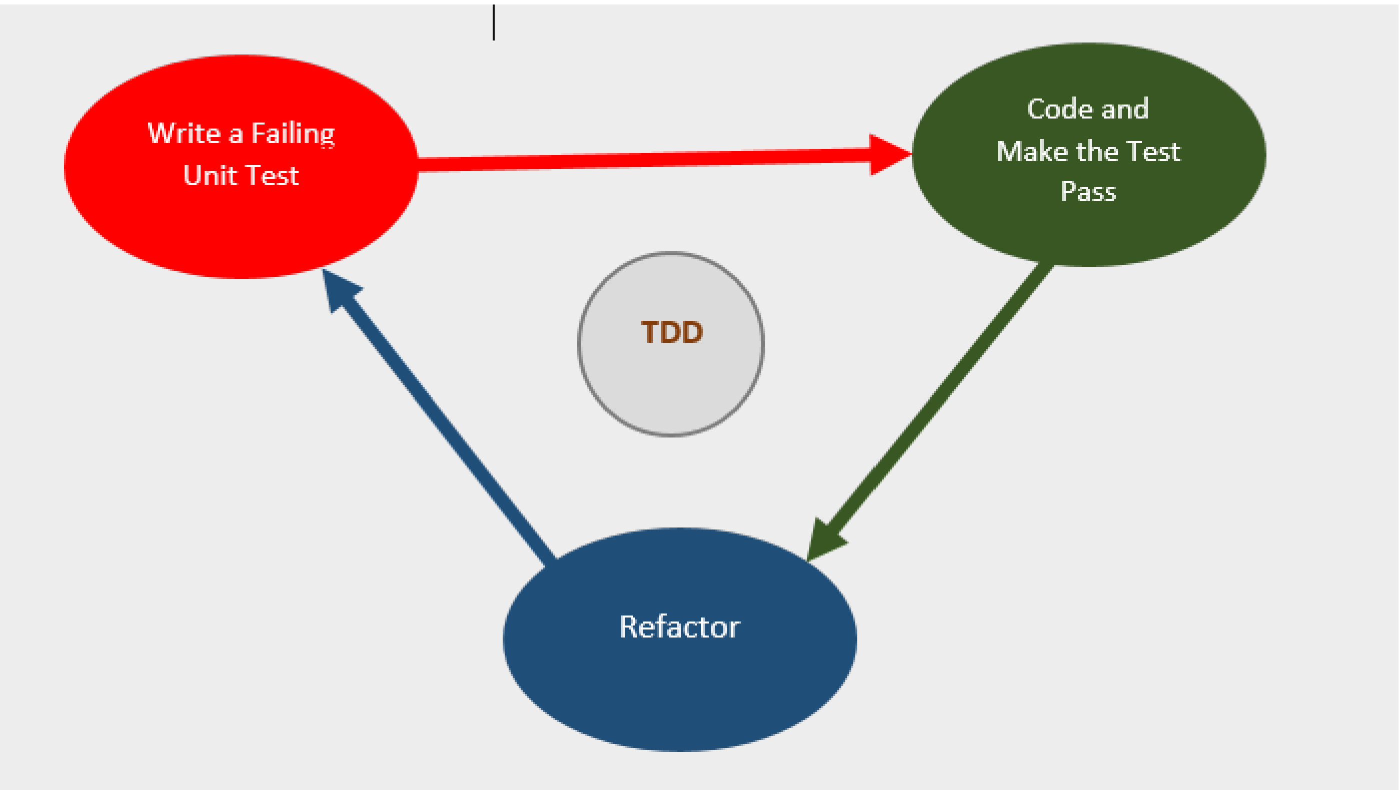


Figure 1 : Traditional Diagram of TDD

You can see in **Test Driven Development** you basically have a 3 step cycle. First you write a **test** for the functionality you desire to develop. As you have not developed the functionality yet, the **test** will fail. Then you write minimum code to make the **test** pass. Then you re-factor your code to have the desired functionality implemented in well-structured format. When you re-factor your code you must ensure, at the end of the re-factoring, the **test** still passes. In addition you also have to ensure all other **test**s that may exist in the project also still pass.

Let us try to have a more detail diagram in a flowchart format:

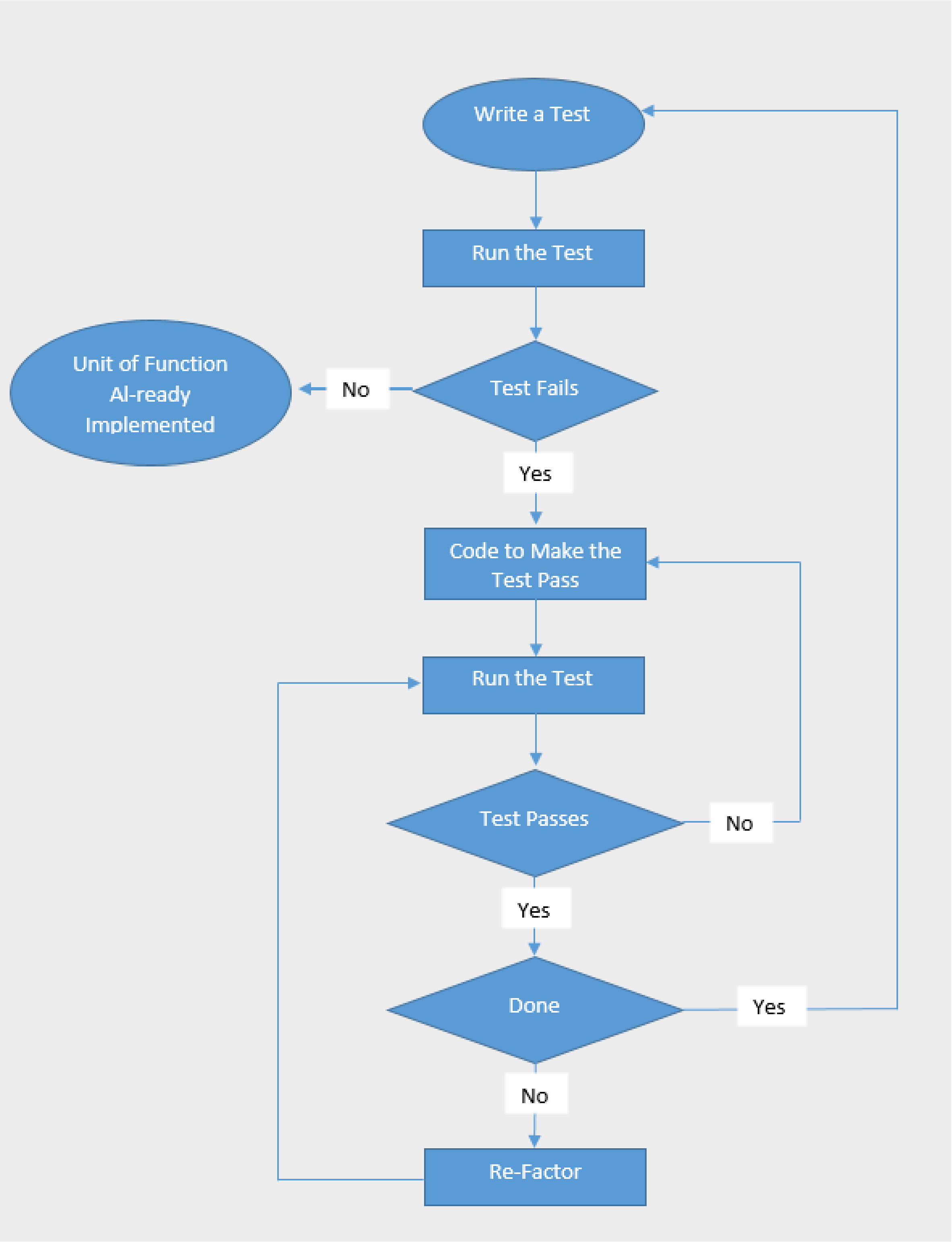


Figure 2: TDD A Detail Diagram

In Figure 2 you can see in a TDD cycle you always check if your desired implementation of a unit of function is done. Now, How do you define if it is Done! What is the definition of "Done"? The definition really differs developer to developer. Some lazy developers have attitude to get the job done with no real focus on extensibility, maintainability or scalability. Some average developers have some focus on structured **development** but perhaps they have lack of skill. Some developers could be too much tempted on design. So, every developer will have his own decision taking pattern for the word "Done". Yes, at one point every developer will be in the same position. That is, the unit of functionality will be developed and it will work. But the difference arises when the question of "Structured" code comes. If you are not very lucky perhaps you will come out with a BBOM.

Yes, theoretically when you re-factor during a TDD cycle, you will look at the design and architecture, and you will re-factor everything perfectly to make your code extensible and maintainable. But in reality how much of that you can achieve? When you concentrate only to the unit of function and become rigid to "Keep It Simple and Stupid (KISS principle)", Every possibility is that, you do not get an up-front domain knowledge of the application and never do anything for future extension. In many cases you will develop a function even without knowing who will use it and how will it be used. On the other hand as you are a lover of YAGNI (You aren't gonna need it) principle, You do not add anything that you don't need at that specific moment. So, KISS and YAGNI are making your life easy for that moment but as the project continues to grow the amount of re-factor will increase, will become difficult and also will become risky. In many cases during the re-factor phase you also require to re-factor your already written **test**s that passed previously. This is because you are trying to achieve a good design through re-factoring! If that is the case what is the point of writing **test** before you code? One argument is, writing **test** first ensures every line of code is covered by a **test**. Well, perhaps that is true but should we accept the trade-off of trying to increase the **test** coverage over sacrificing a good architecture of the application? I would say NO! I don't want to sacrifice the software's architecture because, this may kill my software pre-maturely. So, what about **test** coverage? Yes, I want to achieve maximum **test** coverage too. I will try to tell my way of thinking to solve this problem in later part of this article.

Why TDD might not output a good design?

Let us summarize the reasons why, in reality, whatever the theory says, TDD is not a good way to achieve a great design or architecture for the application:

1. I do not see a role named architect in a TDD team. Perhaps it is my ignorance but do you see that?
2. Achieving structured code through re-factoring is not very practical. Because it largely varies developer to developer and even for the same developer it varies time to time.
3. TDD does not impose any design goal to the developers. So, violation of standard design principles is most likely to happen.
4. If you want to achieve a good design following TDD, perhaps you need to re-factor the infrastructure of your application every now and then which is always risky.
5. When your project grows, re-factoring becomes expensive and your client may not agree to pay for that.

A Soccer Team Analogy:

Think of a soccer team. The soccer team has only two objectives. To score goal and to restrict the opposition from scoring goal. So, from this objective every player can start playing and run behind the ball. Because getting the ball means you can try to score a goal and also the opposition has no way to score a goal. Now, when suddenly the opposition gets the ball and passes it through to a forward you discovered no body is there to defend. Because everybody was running behind the ball. At this point few player starts running and by the time they reach near the forward of the opposition, the goal is already scored. That is why soccer teams always have a plan of playing and they design the strategy keeping the plan as a goal . The most successful soccer teams always have a great coach who design the game and also great players who can execute the design.When you have a great coach all the players become motivated and they also become capable of taking decisions instantly on the field so that the ultimate plan is successful.

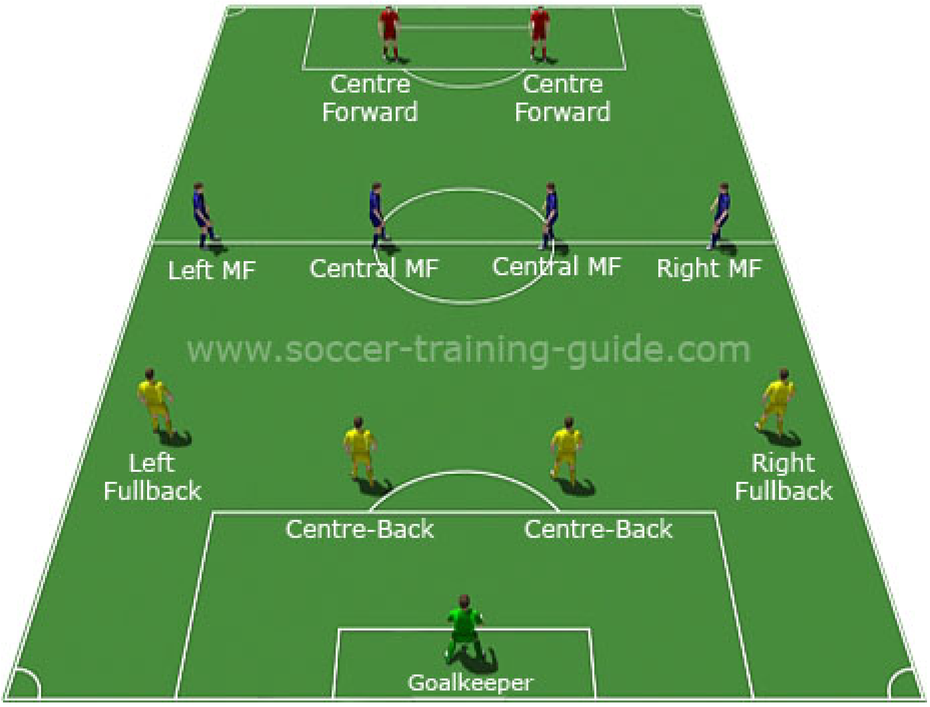


Figure 3: 4-4-2 format for soccer

The soccer analogy is just an example to convince you for an upfront design. You see if you do not have any plan in the game and suddenly you lose the ball even by running as Usain Bolt will not save you. In the same way I believe if you do not have a planned architecture of your project whatever re-factor effort you give, at some-point you are in risk to fail.

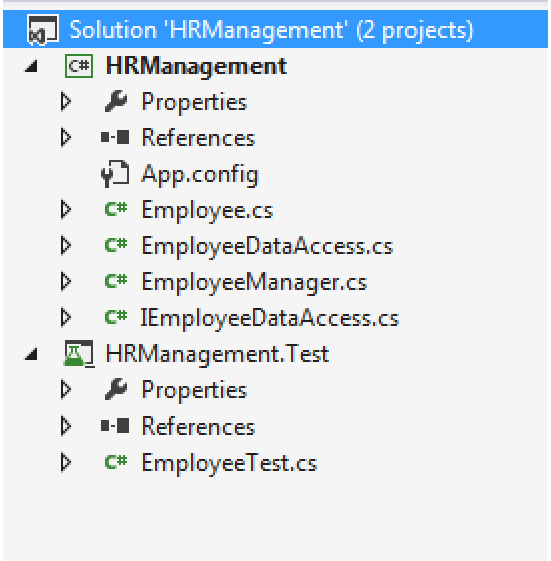
Software **Development** Analogy:

Now I will try to present an analogy from software **development** world. Say you are going to work in a HR management application. The specification is written and the whole application is broken down in some user stories. You start the project and you are assigned to develop an user story where you need to save an employee. So according to the extremists of TDD you start writing the **test** first. Note that, you have nothing in your project upfront. You just have a blank solution. So what will you do now? As you need to write a **test** first. You must create a **Test** project. Now the question comes what will you **test**? Will you **test** the "Employee" object creation to check if any business rule or validation is violated? Will you check if a valid employee can be saved from the business layer? Well, You do not have any business layer infrastructure for your project yet. In your **test** first you will check the object creation. So, write the **test**, make it fail and then create the employee class that make the **test** eventually pass. In the same way you will create another **test** for save method of the employee and thus you create an EmployeeManager class, implement the save method and it will make your **test** pass. During creating the **test** for Save method you also decided you need a data-access method to save the object to database. And at this point you found you need to Fake the DataAccess method. So, you created an EmployeeDataAccess class that is derived from an interface named IEmployeeDataAccess.

Now I have some questions:

1. Where did you write your **test**? Did you write both the **test** in the same class?
2. Where did you create the Employee class?
3. Where did you create the EmployeeManager class?
4. Where did you create the IEmployeeDataAccess interface and EmployeeDataAccess class?

If your answer shows the following project structure (Figure 4) I don't see any reason to blame you. Because you are keeping it simple (KISS) and you are not doing anything that you don't need (YAGNI) at this moment.



```
Solution 'HRManagement' (2 projects)
◢  HRManagement
     ▷  Properties
     ▷  References
        App.config
     ▷  Employee.cs
     ▷  EmployeeDataAccess.cs
     ▷  EmployeeManager.cs
     ▷  IEmployeeDataAccess.cs
◢  HRManagement.Test
     ▷  Properties
     ▷  References
     ▷  EmployeeTest.cs
```

Figure 4: HR Management Project Structure

But yes, perhaps you will not come up with this because you already have the concept of multilayer project architecture and you know you should have different layers for your Models, DataAccess and Business. So, when you start developing following extreme TDD, that means, without any up-front architecture, the structure of your code depends on your skill and thinking capability. If you have no idea about layered architecture you may come out with the above project structure. If you have no idea of software design principles whatever re-factor you do you never come out with an optimum design. Do you see the reason? The reason is you are not obliged to follow a pattern of **development** as no up-front architecture is created.

As it is natural that in a team there will be different kind of developers. Fresher or Experienced, Talented or Un-Smart, Expert or Novice, Hard Working or Lazy and so on. Should you really take the risk by allowing every developer to re-factor as TDD suggests?

See, If 5 person starts running from different place and they all know the single destination, with whatever speed they run and in whatever path they choose all of them will reach the destination. And if there are milestones defined before the destination that they must touch, all of them will eventually run with correct path from the beginning. But if they had no destination defined, no milestone defined, where they will reach? At no-where. So, isn't it better to set your destination and milestones first?

Let us find a solution

The solution I will be talking about is nothing new. Whatever slogan people give in favor of TDD I don't believe any project can be successful with the extreme practice of **Test Driven Development**. So , we must do something where we can build a great architecture for the project and still we have great deal of unit **test** coverage.

In summary you need to follow the following steps:

1. You Do the Design!
2. You Write the minimum **Test** Code! You make it fail forcefully!
3. You Develop!
4. You implement the **test** completely. Now, if you had developed correctly, your **test** will pass.

So, we can call it as DTDT (Design à**Test**à**Development**à**Test**) in short. It should be a simple cycle as the following picture:

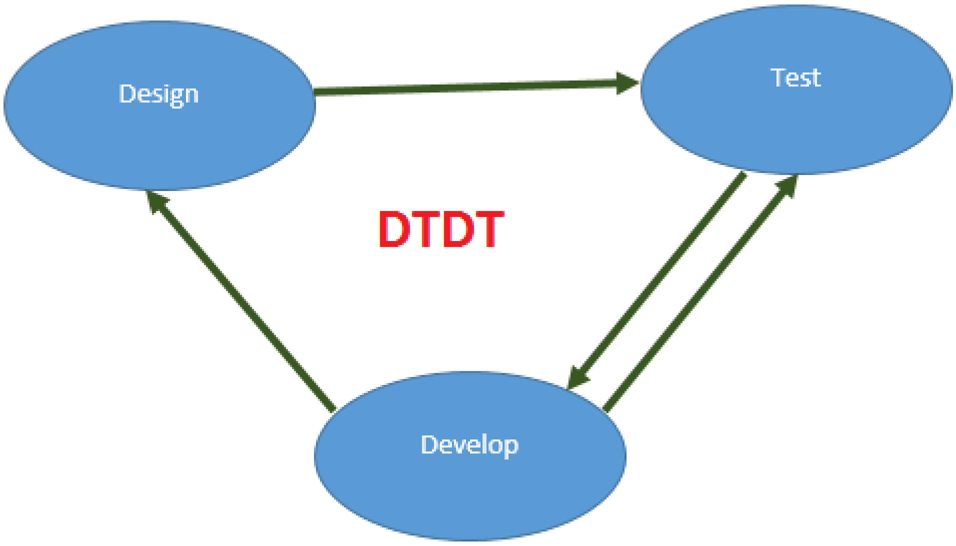


Figure 5: Design > Test > Development > Test (DTDT)

Now, I will try to elaborate the process.

Design, **Test**, Develop, **Test** (DTDT)

According to my opinion your 1st sprint should never implement even a single user story. It should be dedicated towards designing the architecture of the project. Yes, I agree architecture evolves. I agree you should not create anything based on an assumption that is not supported by evidence. I also agree in an agile framework you cannot create a big infrastructure for your project in the 1st sprint. But, you must agree, the project definitely does not start from the 1st sprint of **development**. It starts way back. It goes through information analysis, business analysis, requirement analysis etc. So, at the first sprint when you start designing and start questioning yourself about the project you definitely will have lot of knowledge's from the engagement phase. So, you can off-course create a minimum domain model for the application and also you can create a minimum infrastructure of the application**.** Yes you will keep it simple but I don't agree to keep it stupid. So, better you remove one "S" from the KISS principle. Yes, you follow YAGNI but do not become too rigid. Because, the interface that you know you must need in few days, off-course you create that. There is no point of leaving it behind because "You aren't Gonna Need It" just now.

So, in the start of your project you create the minimum infrastructure and create a minimum model of the domain. That means, the 1st sprint is used for architecture and you already have a minimum design to start developing the user stories.

Now, you start working in the 1st user story using **DTDT** in the following way:

Step 1 - Design:

1. You make sure you understand the user story perfectly.
2. You make sure you know how and why this functionally will be used.
3. You find all the objects that you need to implement the functionality.
4. You look at your solution if any of the object is already created.
5. You try to find if the objects that you will work with have any relation with other objects in the system.
6. You decide what are the interfaces you need to create. And also what are the methods and attributes those interfaces should

have.
7. You ensure you honor SOLID design principle when you create your interfaces, classes and method definitions.
8. Remember, In this step you do not implement any method. You just find or create the required interfaces, classes and just define the signature of the methods.

So, you are done with the design step. Now you are more confident and you know:

1. What exactly you are going to do.
2. What classes and interfaces you need to work in.
3. Which code you should write in which place.
4. What extension point you are keeping open for future.
5. How the functionality fits with the overall system.

As you already had created the infrastructure in the 1st iteration, you are now forced to keep yourself with-in a minimum boundary and you no more can develop in your own pattern.

For example if now you want to build the “Employee Save” user story you will do the following:

1. You check if an entity is already created for Employee.
2. If not, you think if there is any other type of human resource that can exist.
3. Perhaps you may come up with another type of human resource called "Consultant".
4. So, You realize you need one interface named "`IHumanResource`".
5. As you already have a minimum up-front infrastructure in place you will be forced to derive this `IHumanResource` interface from something like `IEntity` interface.
6. Now, You create the `IEmployee` interface derived from `IHumanResource`.
7. In future when you need `IConsultant` interface you can derive it from `IHumanResource` too.
8. So, when you create the `Employee` class derived from `IEmployee` interface it will have its hierarchy correct.
9. Later You create the `IEmployeeManager` (Business Layer) interface for Employee.
10. But when you create the `IEmployeemanager` interface, your infrastructure will enforce you to derive it from `IManager` interface.
11. In the same way when you will create `IEmployeeRepository` interface you must derive it from `IRepository`.
12. You have no way to violate the design. Because, your up-front infrastructure will ensure a Manager interface derived from `IManager` only can work with a repository that is derived from `IRepository`.
13. In the same way, the Repository that is derived from `IRepository` only can work with entities that are derived from `IEntity`.
14. So, the up-front design is forcing you to design and develop your work in standard way.

Next, **DTDT** has two steps to write the **test**s. Now, I will talk about the 1st **Test** writing step:

Step 2 - **Test**:

1. You already did a good amount of analysis of your user story during design. So you already know the maximum detail of your work.
2. In this step you find out the atomic units of works that some up to the full user story.
3. You write unit **test**s for all the atomic units.
4. No. I am not telling that you have to write the full functional **test**s at this step. You at-least create the definition of the **test**s and make all the **test** methods fail forcefully by possibly creating a false assertion.

So, now you have your design ready. You also have defined the **test**s that you must pass after you finish your work. Currently all the **test**s are failing. That means, you can do your **development** now, as you are not going with any risk of missing any **test**.

Step 3 - **Development**

1. At this step you start implementing your user story. You have the design in place and you also have the atomic unit **test**s defined.
2. The design and the unit **test**s that you already have defined, will guide you to develop your functionality in proper way.
3. Once you are done with the **development**, you enter in the 2nd **Test** writing step.

Step 4 - **Test**

1. In this step you will implement all the **test** functions one by one. As now you already have all the methods for your user story implemented, Once you finish the implementation of a **test** method perfectly, it will pass, unless your implementation is not wrong.
2. Once you finish coding all the **test** functions and all of them passes you are done with your implementation.

So, here I have tried to find a solution where you can find the following benefits:

1. You do the analysis to come up with a good design. So, most likely you will be able to raise important questions on your work that you may miss when you analyze to find unit of works for the sake of writing **test**s.
2. When you analyze from broader perspective before you go for finding atomic units of work, the chance of missing is less.
3. In this approach you 1st write all the **test**s that fails initially. So, there is no chance of missing any **test**.
4. When you develop you give more concentration on the design and requirements, so your code looks way better.
5. At the end when you implement all the **test** methods and after they pass you achieve 100% **test** coverage.

That means, in **DTDT** you achieve all of the following:

1. Good Design!
2. Great Code!!
3. **Test** Coverage!!!

Conclusion

As I said before, in this article I actually did not tell anything new. What I tried, is to formalize the practice what most of the software projects go through if the project targets to include unit **test**. Unit **Test** is actually a great weapon to fight against bugs and recurring bugs. So, there is no question that the more coverage of unit **test** you have in your project, the quality of the output is greater. But really the extreme TDD might not be the best way to proceed. Because your **development** has to be design **driven**, not anything else **driven**. The unit **test** will exist in your code, so that it confirms your work never breaks. We should not be extreme TDD practitioner. We should find a compromising point, so that the design never suffers. It could be DTDT or it could be any other way that you think is right.

References

1 : **Test Driven Development** by Martin Fowler

License

Share

@ EMAIL

About the Author

Mahmud Hasan

Software Developer (Senior)
Netherlands

Software Engineer | Software Architect | System Designer | System Analyst | Team Leader | Consultant (.Net)

12 Years of Experience in the Industry.

Currently working as System Designer at CIMSOLUTIONS, Netherlands

View My Profile in LinkedIn

You may also be interested in...

Test Driven / First Development by Example

Developing Factorial Application Using Test Driven Development

Designing Application Using Test Driven Development

Microsoft's Guide to Modern Dev/Test

Learning Test Driven Development with TDD Katas

The Best Angular 2 Data Grid: FlexGrid

Comments and Discussions

You must Sign In to use this message board.

Search Comments Go

First Prev Next

My vote of 5

Gaston Verelst 30-Jan-16 22:43

I wrote about the same, in other words. So I like your article 🙂 Look at http://www.codeproject.com/Articles/1074253/Test-Driven-Development

Sign In · View Thread · Permalink

My vote of 5

Monjurul Habib 17-Dec-13 5:16

thanks for sharing your thoughts!

Sign In · View Thread · Permalink

vote 5

Andrew Rafas 26-Nov-13 11:44

This was also written by Mark Seemann who is the author of the book "Dependency Injection in .NET".
http://blog.ploeh.dk/2010/12/22/TheTDDApostate/

Simply TDD leads to bad abstractions. Well said!

Sign In · View Thread · Permalink

Re: vote 5

Mahmud Hasan 26-Nov-13 21:11

Thanks for sharing the article. It was a nice reading. I think you understood my point nicely. Our development must be covered by Test but Design should be the first thing we should think about.

Mahmud Hasan
System Designer from Netherlands

Sign In · View Thread · Permalink

My vote of 5

S. M. Ahasan Habib 19-Nov-13 5:44

I personally prefer Behavior driven development(BDD) over TDD and think BDD is more realistic approach. Anyway you nicely explain the concept of DTDT.

Sign In · View Thread · Permalink

Re: My vote of 5

Mahmud Hasan 19-Nov-13 23:47

BDD is certainly a good way to write test that gives you more readable and logical testing opportunity.

Thanks
Mahmud

Mahmud Hasan
System Designer from Netherlands

Sign In · View Thread · Permalink

Re: My vote of 5

Jesús Utrera 12-Feb-15 22:58

I agree with BDD is more adjusted to reality. In fact, I think we should test the core business with UnitTests trying to cover all possible ways and exceptions regardless of the frontend. And finally, test with BBD the frontend functionallity.

Sign In · View Thread · Permalink

My vote of 2

Aescleal 18-Nov-13 7:48

You pose several questions in the introduction but don't answer them. In several places you use the phrase "I believe" which considering you're advocating against using TDD is a bit weak.

The whole thrust seems to be "What if I don't trust the other members of my team to do the right thing?" I'd suggest changing jobs if that's the way you feel about them and communicating with them isn't going to help things along.

Sign In · View Thread · Permalink 1.00/5 (1 vote)

Re: My vote of 2

Mahmud Hasan 18-Nov-13 8:13

I am sorry but it seems you did not understand my points and logics. Perhaps my igonrance.

Mahmud Hasan
System Designer from Netherlands

Sign In · View Thread · Permalink

My vote of 4

Paulo Zemek 18-Nov-13 3:24

I really liked your article. In fact, I am not writing about TDD, but I am actually writing an article about architecture that talks about "no architecture" and some kinds of architecture and what I see is that many of those development strategies hurt the architecture of applications.
I am not voting 5 exactly by the fact that you don't talk how TDD may (or may not) interact with other architectures. For example, do you think that you can apply TDD on projects that already use an architecture? Or do you really believe TDD avoids the use of a better architecture? Can't your DTDT be seen as TDD over another architecture?

Sign In · View Thread · Permalink

Re: My vote of 4

Mahmud Hasan 18-Nov-13 3:45

Hi Thanks for your feedback.

I would say DTDT is not exactly another form of TDD. You can think it as a process of Design and Development with Test. I think whenever you try to implement too much of TDD that means nothing before a test, your concentration will be deviated from the broader view of the project. That is why, I said, Design should be the 1st step where you can always analyze what you are going to do and how it fits with your software. Once your design is set, you can go for test, development and test cycle.

Regards

Mahmud

Mahmud Hasan
System Designer from Netherlands

Sign In · View Thread · Permalink

Re: My vote of 4

Paulo Zemek 18-Nov-13 7:38

In fact I know that many methodologies have 1 of those two problems:
* They don't accept other methodologies, as they try to do everything;
* They require another methodology, but that's never explained.

I really don't have a strong opinion on TDD, but what I see is more something like: When you finally decide to implement, start by tests, instead of "your project starts by tests". I really don't see TDD saying that we must start by tests and then refactor code later... I see the refactoring part as something that "happens" when we see anything wrong.

Yet, I am not a fan of TDD for another reason: I think it focuses too much on obvious tests and this may create the illusion that everything is working, when the more important tests may still be missing (and the problem is that "confidence" that everything is OK). Also, in some cases, the tests become much bigger than the functionality itself, which is terrible when people simply want to apply the methodology and keep the dead-lines untouched.

Sign In · View Thread · Permalink

Focus

JarmoM 17-Nov-13 23:20

All patterns focus on one thing, so does TDD. It focus on Development. It's just a pattern. Of course you have to follow other patterns to achieve the needed functionality and design. Therefore you should have some sort of design before you start developing.

Sign In · View Thread · Permalink

Re: Focus

Mahmud Hasan 17-Nov-13 23:33

Thanks for your feedback. I think we are in the same page.

Regards
Mahmud

Mahmud Hasan
System Designer from Netherlands

Sign In · View Thread · Permalink

My vote of 3

Gustavorhm 16-Nov-13 23:35

I don't agree 😊

Sign In · View Thread · Permalink

I don't agree with all of what you say

Gustavorhm   16-Nov-13 23:33

Hi Mahmud,

I have to disagree with you on this one.
You should definitely design upfront, but not in code. Just design an overview of the components you may need and their interaction on a whiteboard. This is enough to get the team going on the same direction. And if you code in a TDD manner you may identify flaws in that overall design and it's much easier to fix them, because if you had a basic architecture already coded you would be much more reluctant to change it.

You also say that people in a team have different skill levels and they will move at different speeds and in different directions if you don't lay an architecture upfront. This is a problem that can be overcome by implementing code reviews as part of the development process. And communication is also part of it, if the devs communicate with each other they will never let the code base drift so far apart.

My 2 cents 😉

Sign In · View Thread · Permalink

Re: I don't agree with all of what you say

Mahmud Hasan   17-Nov-13 1:01

Hi,

Thanks for your argument.

But, a design in white board is not enough to ensure everybody follows that design. An up-front overall infrastructure will ensure developers will have proper direction when they design and develop their small small unit of works.

Yes, code review is a nice way of ensuring code quality, but if code review results to too much of re-factoring then you must have some measure to improve the process.

Thank you again for giving your feedback. Besides, it will be great to know some cons of DTDT from you so that I can think of improving the process.

Regards
Mahmud

Mahmud Hasan
Software Engineer from Bangladesh

Sign In · View Thread · Permalink   5.00/5 (1 vote)

Re: I don't agree with all of what you say

Gustavorhm   17-Nov-13 1:31

Hi again.

Some cons of your DTDT that pop to my mind:
- If you have a team with less experienced developers, they will follow your architecture without questioning it. This means they won't learn why you designed it that way and their chances of learning are smaller. If you do it the TDD way, they will experience much more and learn throughout the process. They will make mistakes, but the code reviews will catch them and the devs will learn from the process. It may not appear optimal but it will develop your team much faster;
- As an architect you should involve your team in your architecture design. It *feels* to me that when you create an architecture up front you are excluding your team from the design thinking effort because they don't know what's best, as you are the one with the vision. If this is the case, it's also a problem;
- When you have an architecture layed out up front, if you discover problems during development you will be more likely to work around the problem rather than addressing them. Refactorings will be bigger. With TDD, you identify design problems much faster and refactoring is smaller.

- If you work Agile requirements may change between sprints and you will have to take the project in a new direction. That means you invest time in an architecture that may be deprecated when it's time to do the concrete development.
- Creating the architectural infrastructure up front that respects the SOLID principles is not that easy in my experience and requires a very experienced person. These are the kind of problems that TDD will help solving, by bringing composition into a system in little bits.

What do you think?

Sign In · View Thread · Permalink

Re: I don't agree with all of what you say

Mahmud Hasan 17-Nov-13 21:42

Hi,
Thanks for your feedback again.

1st of all in DTDT you create minimum up-front architecture, In most of the cases it should not be wrong if the engagement phase get's minimum knowledge on the application. And an architect of-course will work along with his team to create the architecture. In agile framework there is no way to have the design created and delivered by a single person without doing any discussion with team. So, the 1st point you mentioned is very important but I think in DTDT you do not have that risk.

I do not agree that TDD will have less amount of re-factor. Look I am talking only about the initial project infrastructure. Rather you will have a design phase before starting every user story. So, before starting every user story you always have a look on the overall project and the direction where the project is moving. So, you always can design keeping the extension points open. It must help you to honor SOLID principles.

When requirement changes in agile methodology, DTDT is more powerful in that situation. Because you design your work before you write test and code. So, it will be easier to change and extend.

I hope I could answer all of your concern. And I loved your points.

Regards
Mahmud

Mahmud Hasan
Software Engineer from Bangladesh

Sign In · View Thread · Permalink

5.00/5 (1 vote)

Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin