StoryTime and Fast Programming

# - Despird Zhang

Programming Fast

For how many times have you heard that "a good programmer can code double/tripple/ten times faster than an average one"? Well, this does not seem to be a valid principle because the word "good" or "average" are very unsettling, not to mention the more unsettling multiple number, unlike the well-known Moore's law. However there is a definite point here stressing that there is a distinct gap between "good" and "average" in terms of productivity.

Quantification has been very difficult due to the diversity of programming tasks. Some developer are good at constructing from scratch, some excel at solving tough issues, and some others are expert in debugging code written by other people. Almost in every team there is such a guy who delivers his work really fast, but with significant number of bugs which later cost him triple effort to fix. The overall productivity is measured not based on the accomplishment of first edition, but the final artefact ready to be delivered to the client, or the module which has passed the final peer review or supervisor sign-off. We probably quantify from certain technical aspect and score it, then move to next index. Even during quantification over single index, we cannot rely on only a few samples to draw the conclusion. For example: Two senior developers are solving a tough issue without communication, instead, they are endeavouring to take their individual best practical approaches. The result of single instance is unpredictable, some time a lucky hunch takes the lead. But if I'm the team lead who has been working with both for many years and witnessed numerous similar situations, a ballpark judgement can be easily made.

Now that we know the individual productivity is indeed measurable, we must face the question how we can do better with it so that our bosses continue patting on our shoulders instead of desks.

**Is it about the attitude? Maybe**

What if people don't want to go very fast at all? There is a good reason for doing this when you are a senior developer, SENIOR enough to control your pace confidently and know how to complicate things up so that you easily convince your supervisor that you have actually made a very good progress on a very difficult assignment, or you are a contractor constantly being told by your agent to take your time so that the contract can be extended again and again. Usually, making things less complicated is usually much more difficult than making them more complicated, but ironically it makes you look naiver and less sophisticated in front of others. During a period when I worked in one of my previous teams, we each assign ourselves bugs to fix from the same Backlog management system. At the end of the development cycle I finished with the maximum number. With most of them only one or two lines of code change, I heard another "Senior" engineer complained to the manager that I only picked up simple ones. The truth was, I picked them up only by priority. It wasn't a coincidence that most of my fixes were simply made. Other people overlooked the fact that it took me actually ten times more effort to find out how to make a simple, elegant fix than the effort spent working on the fixes themselves.

**Is it about the tools used? Partially**

Tools like Resharper have made great success among Visual Studio community, interestingly I haven't seen a strong connection between utilisation of the these tools and the difference over productivity across people. It recurs that younger developers contribute the most of to the popularisation of a new tool, while on the other hand, veterans tend to advocate old-fashioned simple ways and maintain prudent toward these tools. In the history of software development, productivity tools prevailed and died at the turns of technology generations, only very few of them could survive. Some older developers, including myself, have accumulated experience in a good selection of tools which render our résumés looking so colourful. Senior developers are obnoxious creatures, they enjoy possessing knowledge of these tools and on the other hand they claim tools are just tools, a best weapon for an good assassin will be just a knife, or hands, or even just "minds". This is implying a fact, that generally speaking, a systematic methodology behind the tool will outlive those tools who are just tools, and even outlives the tool itself when a better tool supporting same methodology appears.

Back to young developers, tools are very helpful because they compensate the their lack of experience and the learning process is also a mind-polishing practice for them. Not to mention it also stands as a positive attitude of desiring to be fast.

**Then it's about methodologies? Not if it's a complex one**

Methodologies are invented every day, and yes by reading this document you are enduring another one. In long revolution of software development an interesting word has kept being mentioned and then forgotten again and again: simplicity. Like other theories, every new theory that aimed to help with productivity was simply presented by the inventor, elegantly formalised and standardised by some enterprise or organisation, and testified by the community, then evolved until it became a monster. My point is that if a methodology, or a OO pattern or just some trick you heard from community that is good has successfully made you think the use of it can make things much simpler rather than the opposite, it is suitable for you and will definitely speed you up. Believe it or not, I once know this guy who was more comfortable dealing with code in Structural languages rather than OO and he told me that OO is unnecessary and makes code lengthy and the most important, the runtime efficiency or performance is quite poor. If OO hadn't been invented, we might have developed something else, who knows? All we know is that simple theories got crucified much less than those complex ones did because ***people tend to make less mistake on things they highly understand.***

**Is OOP the Golden Compass? Not for everybody**

Object-Oriented is revolutionary, and pretty long-life. It still dominates among most of popular programming languages. It peaked into its golden period after the emergence of Design Patterns and UML. Despite all these flamboyant tricks what OO can do on the surface, we have forgotten completely all our original purpose was just for the reusability. I believe the original language did not have a methods/functions, which intended to make recurrent code reusable. As the embryo of OO, the Struct in C language introduced Encapsulation, while the C++ moved a further step to a more maintainable reusability by introducing Polymorphism (the other two features of OO, Inheritance and Abstraction are actually implied within Polymorphism), and it makes our code so elegant today because it provides a better simulation of our actual world. Anyway high reusability makes high maintainability, but what about the readability? From structural or static point of view, yes, OO has a stronger capability for modelling the system. But from dynamic point of view, reading OO code requires endless inevitable jumping-across. Although this disadvantage has been alleviated significantly by modern IDEs such as Visual Studio, during a reading of a large system which consists of numerous classes, after a couple of jumping you might probably have forgotten where you started from, and where you should head towards next. Ironically, as the essence of OO, the Polymorphism has the worst reputation in regard to readability. Wait a moment, you might ask what do all these have to do with coding speed? In fact, a software system is more like a dynamically functioning construction than a lifeless building without power. It is fully of parts which interact with each other and constantly happening events. Low readability in dynamic behaviours makes coding extremely difficult therefore increases the overall developing time. Experience shows us that the more classes you have, the less readable the code is.

Besides readability, another thing that can go wrong with more classes is that the code complexity is exponentially increased. OO community has encouraged the use of more classes all along. The Design Patterns by GoF, for example, teaches us how to manipulate interfaces, abstract classes and their heirs to help out with certain scenarios. But how many people have remembered that the one of original intentions of OO was to reduce repetition so that our code is more compact? Unfortunately, OO and Patterns have been exaggerated and misused by the advocators all along.

One of the examples is the use of MVP (Model-View-Presenter) pattern that helps with testability. Although examples have been provided, due to the management of large number of the interfaces and classes, people still implemented it quite variously. Unfortunately more than half of these mimics were a disastrous failure from every aspect. The reason mainly is because of the poorly built framework in which circumstance for each simple new feature the consumer himself has to add a couple of interfaces, classes and numerous methods. The maintainability and extendibility are quite poor, not to mention the horrible code readability. For how many times we appeal for simplicity but our solutions actually make things worse? Solutions per se are not incorrectly made but the complexity make it difficult to master. Think about the size of your target so you can decide to use a pistol or a machine gun, the latter can easily kill yourself rather than the target.

Having more classes sometimes entails tremendously more lines of code and therefore betrays the principle of simplicity. On the other hand, the high complexity of code requires high ability of manipulating these classes. I was once told by someone I admired so much, that '**Same code is different to different persons; for same person, all different codes look same**'. From this I can learn that I shall not follow the exact steps made by others, instead I shall keep the number of my classes as low as possible so that it won't turn into a runaway wild horse. Day by day as I progress in my understanding of this pattern, I will think about making this number higher. **A better developer must know how to maintain a bigger class diagram in mind and must also be capable of making structural alteration until it is fully functioning in mind**. Practise and learn the maximum number of classes you can hold in mind and deal with it. That's what makes you program faster because it will reduce the number of times you make alterations. Over use of OO is addictive, and is a waste of your valuable time.

**Why readability is so important?**

One of the other aspects that cause modern OO codes more implicit than they used to be is the use of call-back mechanism. This was originally introduced for GUI patterns like MVC, MVP or Document-View etc. These patterns were implemented as reusable frameworks following the principle named Hollywood (Don't call us, we will call you). The greatest benefit is that the library users actually don't have to know about the tricks of the pattern or manipulating these classes that formulate the pattern, instead they only need to know how to hook up their business-focused functions and know when they get called. Although such a framework usually conceals the implementation detail of a message dispatching for the purpose of assisting developer in their concentration on event-handling business, bus people curious like myself are still interested in what happen behind the curtain. I used to spend a lot of time reading the source code of MFC (Microsoft Foundation Class) because I really wanted to find out how exactly my code will be called.

DI (Dependency Injection) /IoC (Inversion of Control) has brought the implicitness of code execution to another level by encapsulating the details of object creations. All your classes are inherited from interfaces which have very obscure relationship only the designer will know exactly. For the first-time reader, not only he will learn the knowledge when a code start to execute, he is also bewildered when these objects are created and how they are related to each other.

On the one hand thanks to these frameworks, we have to admit that great simplicity has been achieved when it comes to the management of coupling, or the testability or other benefits like the performance. But on the other hand, the dynamic flow of the code has becomes further disconnected so that the it first-time reader are finding that the hardest task to undertake is understanding code and memorising the workflow. Another emerging risk is that this code disconnection will increase the chance of creating code duplication due to the fact the developer might have forgotten (or might even have not noticed) the existence of the functionalities.

* Maybe you could argue that code execution path is not important in modern web dev, like asp.net MVC, people are still glad to know which part of the code follows which other in sequence.
* That's why I have StoryTime ....

**Team working dilemma**

Unfortunately the majority of developers work in a team, which entails

**When should I write my unit tests?**

* TDD? Not necessary
* Unit tests should not reflect detailed design

**A example approach (OO language only)**

*Warning: the following rule only applies when you are in an Agile team where no formalised documentation is required and you are fully independent on the implementation of the full software module or component. It only provides a generic skeleton for typical scenarios and should be tailored for your particular case.*

1. Write your Stater Classes (with empty bodies). A Starter Class is a class that only contain Starter Methods. These methods are usually the ones that won't be directly called from your own code indicating a trigger of a system event, examples:
   1. Application Main entry
   2. UI event handlers
   3. Workflow/Service Starters or event handlers
   4. WebServices/WCF/Remoting methods
2. Add Starter Methods (with empty bodies) to your Starter Classes and put simple comments.
3. Optional: if your supervisor wants a high-level design, write a simple Use Case for each Starter Method.
4. Optional: if your supervisor wants Unit Tests, write a test method (with empty bodies) for each Starter Method.
5. For each Starter Method of each Starter Class, write code skeleton:
   1. Write comments to describe the workflow for this event
   2. Optional: if your supervisor wants a preliminary design, translate your workflows into Activity Diagrams(Flow Charts).
   3. Translate your comments into Skeleton code. Skeleton code includes following statements:
      * + Statements to declare or construct a new object.
        + Method invoking to your user classes.
        + Selection or Iteration such as if/else/switch/for/while etc.

It should not include regular statements such as assignment, calculation or API method invoking.

* 1. Add new local methods(with empty bodies) whenever you feel necessary
  2. Add new classes(with empty bodies) whenever you feel necessary
  3. Move your comment into your newly created methods whenever it is necessary.
  4. Optional: if your supervisor wants a detailed design, translate your workflows into Sequence Diagrams.

1. Revise your skeleton code until you feel good.
2. Add flesh to your skeleton code.
3. Add flesh to your unit tests
4. Debug your code until tests pass

This simple approach might or might not work as expected, but it is simple, so it will unlikely make things worse. A compromise between "might" and "might not" would be "probably" so it still offers a positive output.

**What speed you up?**

* DRY (Don't Repeat Yourself)
* Always make a design on what you are doing no matter how small it seems.
* Close your eyes, try designing in mind.
* Design your classes in code. Remember that modern IDE makes writing skeleton code much faster than drawing a diagram with a case tool. Unless your boss wants those diagrams.
* Write comment first, code second
* Don't bother compiling your code too until you finish your design stage
* Revaluate what the hardest part is in your current iteration
* Detail the hardest part first if you are not sure if it can be done
* Detail the hardest part last if you are quite sure it can be done
* Hesitating if it's time for refactoring because you are not quite positive (not because you hate to spend more time)
* Laziness over a positive refactoring
* Laziness over remembering all the shortcut keys of your tool
* Laziness over exploring features of your tool
* Hesitating over naming a class, methods or variables when you are using a tool with a powerful renaming functionality.
* Keep coding without thinking

**What slow you down?**

* Impulse of commencing an unnecessary documentation
* Commenting for each line of code
* Try compiling code each time you make a simple change
* Drawing a picture that can be pictured in mind
* Rushing into coding without a mature thinking
* Thinking about refactoring too early
* Over thinking
* Methodologies? Must be simple, complexity doesn't make things fast
  + Agile helps, but you need more, when you talk about Agile a lot, it's not agile anymore, if too little, nothing left at all.
  + Oo? Nothing related. It can make things worse or better but it won't guarantee the speed.

Ugly truth about Agile

* Why do we hate document
* Do we really not need document? Not true.

Comment-Driven Development (CDD) and pseudo-code driven development (PDD)

* History, background
* My practice:(how-to)

StoryTime - Let the source code tell the stories, or Story-Driven Development

* Agile dictates "source code is the design/documentation".
* StoryTeller: tell your stories to your source code
* Story Extraction: let your source code tell stories
* Let the source code record your stories so that it can tell more people

StoryTime and CDD

* Does agile

SOME and PDD

Final word: StoryTime is Agile

* The simplicity makes so, it is straightforward, plain, easy to understand. It's not sophisticated so it won't scare people off, people won't easily make mistake on things they highly understand.