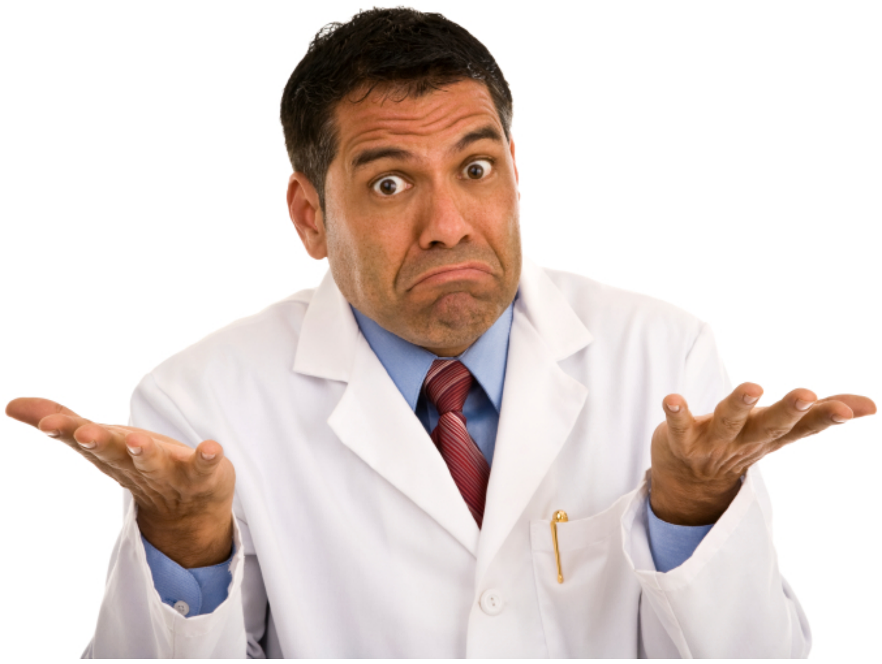
Why Your Algorithm Will Fail

Reason 4: The Optimization Approach

If you had a disease, you would go to the doctor's office, right? The nurse would take your vitals, you would talk about your symptoms with your doctor, and perhaps the doctor would run some tests based on those symptoms. You would then be much closer to getting a proper diagnosis and the right treatment -

But what if you couldn't communicate - at all? There would be some major issues indeed.



The human body is a very complex instrument. It would be nearly **impossible** for a doctor to know if you have a particular disease based solely on your appearance, right? It would be even harder to diagnose if you were unable to give the doctor any details of your symptoms. Sure, a doctor could take your vitals, run every test imaginable and possibly diagnose you that way, but that would be horribly **inefficient** in terms of both cost and time. It would also divert resources away from the patients who really need it, and without your feedback, it would be difficult to determine if treatments were even working. In other words -

Communication is **critical.**

Lucky for us, humans have brains and vocal chords that allow us to communicate. It's a pretty convenient design, right? -

Right. But how does this tie into algorithmic development?

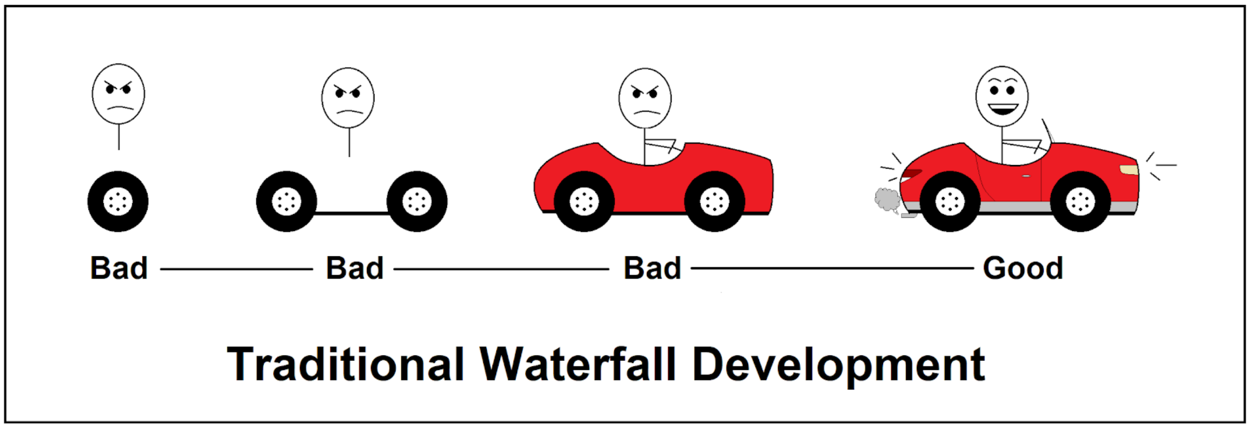
Well, when designing our systems, we need to make it easy for them to **"tell"** us their problems. Otherwise, if and when things go wrong, we won't know how to diagnose them. For instance, let's say we optimize 7 indicators and the SL/TP levels **all at once,** and somehow arrive at a seemingly robust system (don't do this). **We would have no idea how or why it works.**

There's a huge transparency issue here. Consequently, what do you do if a component fails? Spend a ton of time investigating the **entire** system - or even worse - go back to the drawing board? What if you actually have a good system, and it just needs a small tweak? You would have no idea.

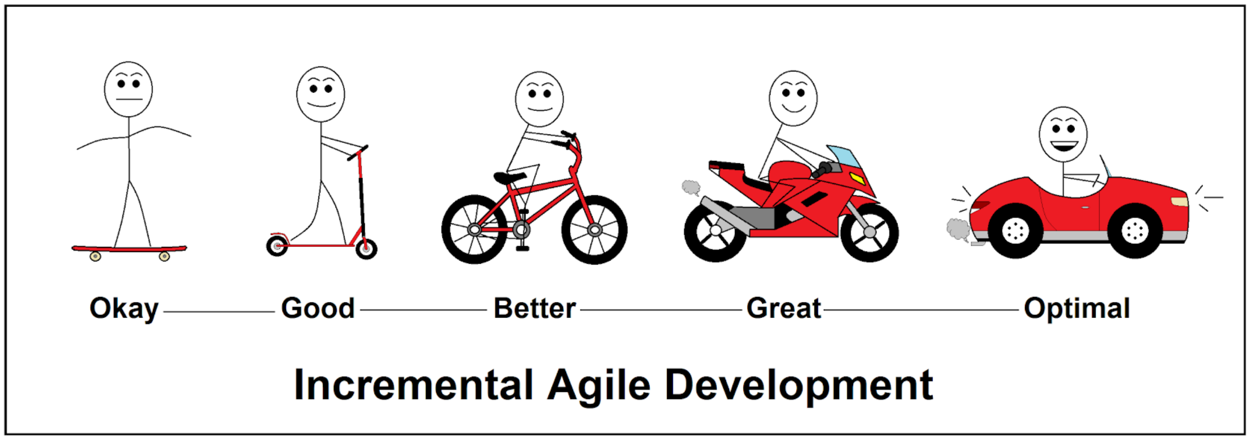
Yeah, big waste of time. So what's the alternative?

If we're serious about building professional trading systems, we need to look at how other industries approach their development processes, particularly the IT industry. In the past, IT developers used a method now known as the **Waterfall approach. Waterfall development** is when all stages of a project occur one after the other. In this approach, there is a planning phase, where product **requirements** are established, followed by a long development phase, and finally, a testing phase.

Unfortunately, with the **Waterfall approach,** testing could only occur once a product was finished. This is because there were no points throughout the development phase in which the product was **functional** enough to be tested. By the time a project was complete, it may not have even met the **requirements** that were established beforehand. This posed a big problem: due to lack of testing, they didn't know exactly which components were underperforming. It was very difficult to go back and make corrections.



This caused a lot of major IT project failures, which eventually led developers to invent a new approach known as the **Agile methodology. Agile development** is when stages of a project occur in **increments.** In each **increment, requirements** are defined for a **single component.** The component is then developed and tested **before** moving onto the next one. This ensures a fully functional product that can be used and tested at every stage of development. If any changes are required, it's relatively easy to identify the issues and quickly make corrections.



If you notice, the **Agile** and **Waterfall** methods are both trying to create the same product - in our case, a **robust** strategy - but the paths they take to get there are completely different. You can see for yourself which one is optimal - **the Agile approach.** So, what does this mean for algorithmic traders like us? Well, to develop an algorithm in line with the **Agile approach,** you need to break down the **components** of your algorithm in terms of **priority** and test them from the ground up by first **defining the requirement.**

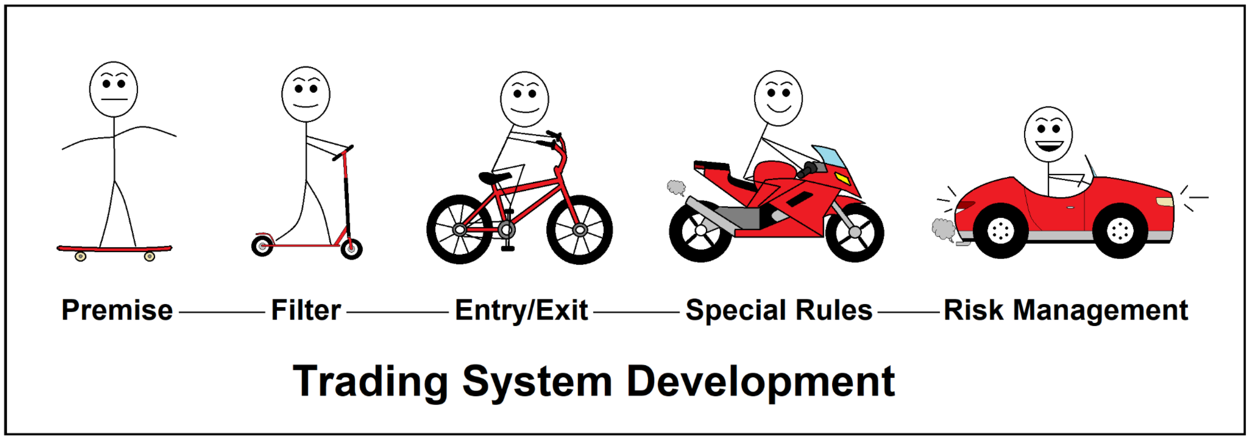
In a trading context, the **requirement** is also known as the **system premise. A system's premise is based on exploiting an observable market condition - in our case, trends.**

After defining the **requirement,** you should then implement a **development phase** which prioritizes **functionality,** in line with the **Agile approach.** Our focus will now be on defining the most **critical components** of a trading system, then **building them in order of priority.** So, what's the **first** most **critical component** of a trading system?

Is it the Entry? The SL? The TP? Nope -

It's the ability to identify a market condition. This is a simple, yet often overlooked fact.

This is where **Agile algorithm development** begins to deviate from the **Waterfall approach**. **Instead of relying solely on the entry, or beginning with predetermined components of a system** (such as ATR values, MACD settings, etc)**, for our systemic foundation, let's prioritize functionality to build an algorithm from scratch:**



1. <u>**Premise**</u>

    - **The first most critical component of a trading system is the <u>system premise</u>.** In step 1, you will not optimize anything. You will determine the market condition you want to exploit and design basic rules to do so. **Without a system premise, none of your trading system components mean anything.**

2. <u>Filter</u>
    - **The second most critical component of a trading system is the ability to <u>identify the market condition</u> you are targeting with your system premise.** In step 2, you will formulate a rule to identify when the condition is happening and when it is not happening. This ensures your system is in the market when it should be, and isn't in the market when it shouldn't be. You may use indicators or other means to accomplish this.

3. <u>Entry/Exit</u>

    - **The third most critical component of a trading system is the ability to both <u>open and close a trade</u> in line with the system premise.** In step 3, you will design entry and exit conditions with respect to your filter. This ensures the **most basic** trading functionality aligned to a system premise. You may use indicators or other means to accomplish this. Exit conditions can include an indicator, SL and/or TP.

4. <u>Special Rules</u>

    - **The fourth critical component of a trading system is <u>refined signal processing</u>.** This is where you can test and/or add complementary rules like: continuation entries, recovery entries, additional exits, the One Candle Rule, spread filters, session filters, trade timing, etc. You can think of this phase as taking your system to the lab for further study. If a rule doesn't enhance your system, refine it or scrap it.

5. <u>Risk Management</u>

    - **The final critical component of a trading system is the ability to <u>manage risk</u>.** I know what you're thinking: "How is risk management last, in terms of priority?" Hear me out: risk management itself is **critical** to trading your algorithm in a live setting, but it can distort **functionality** in the beginning. Remember, we are prioritizing **functionality** during **development,** which will ultimately determine the **transparency** of your system. Design your risk management in such a way that it extracts the maximum potential from your edge while protecting capital according to your risk appetite.

An important note on indicator functionality: The indicator types you may be familiar with are misnomers. Classifying indicators as "Two Lines Cross,"

“Zero Line Cross,” “Baseline,” or “On-Chart” gives zero context about the indicator’s function. This is not good.

You need to be able to differentiate between actual indicator types so you don’t end up with a bunch of indicators telling you the exact same information in different ways. The proper indicator classifications are Trend, Momentum, Volatility, Volume, Cycle and Support/Resistance.

Trend indicators measure the overall strength and direction of price while filtering volatile price action.

- Examples: Moving Average, Aroon Up/Down

Momentum indicators measure the direction and rate of price change.

- Examples: Stochastic Oscillator, Relative Strength Index

Volatility indicators measure price deviation.

- Examples: Average True Range, Standard Deviation

Volume indicators measure the number of price changes over a given timeframe or the number of shares/lots transacted over a given timeframe.

- Examples: On Balance Volume, Chaikin Money Flow

Cycle indicators measure periodic highs and lows.

- Examples: Commodity Channel Index, Detrended Price Oscillator

Support/Resistance indicators estimate supply and demand levels.

- Examples: Pivot Points, DeMarker

Many indicators overlap with each other, and thus understanding the inner workings of any indicator you are considering implementing in a live trading environment is crucial to the success or failure of your system.

End note.

By using the **Agile approach** to build an algorithm, any component of your system that may be lacking can be immediately diagnosed and treated accordingly. You have designed it to **communicate** with you. Pretty cool, right? Try it yourself. You may be pleasantly surprised at the increased transparency.

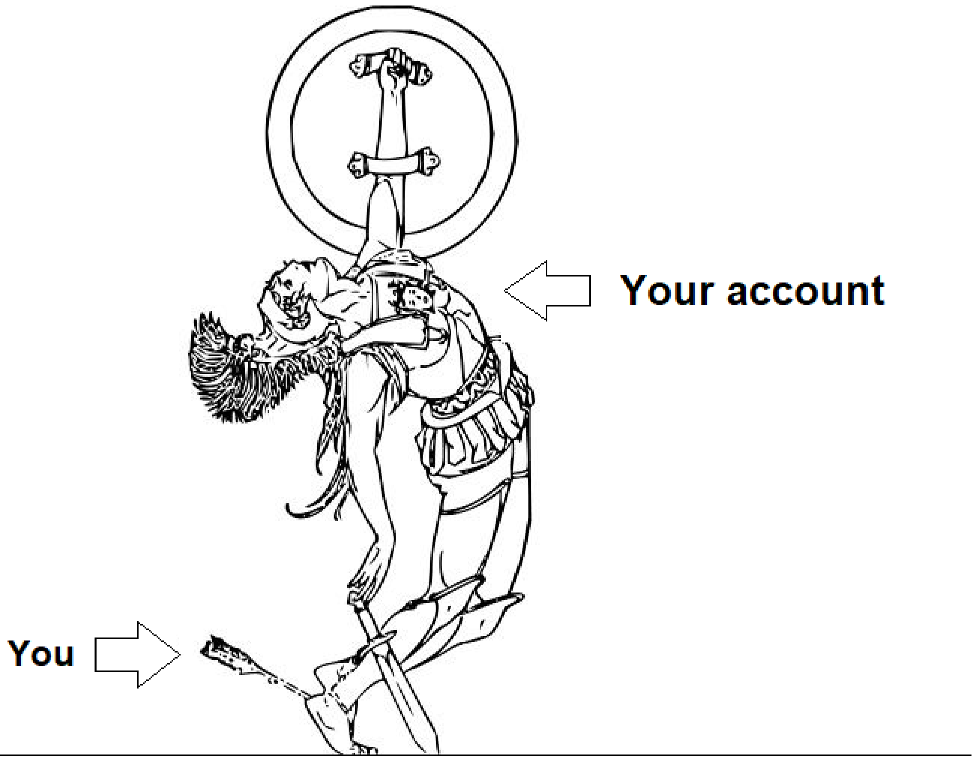
This transparency enables you to build, diagnose and rebuild systems quite quickly. Combined with the proper **optimization process** and your understanding of **overfitting,** spending months building ineffective trading systems will be a thing of the past.

Sounds perfect! Right? -

Too perfect…

The truth is, all of your knowledge can still be undone by one simple error. Remember: although algorithmic trading is a truly amazing science, it's only as good as your ability to find your weaknesses **before** the market does. In fact, your biggest shortcoming may be something that you think is a benefit. In the next article, we will explore an often overlooked, but horrible habit that many people have when developing algorithms -

Your Achilles Heel -



Doing things manually.