AGIL-O-NOMICS

ENGINEERING PRACTICES FOR TEAM EFFECTIVENESS

Scrum and Engineering practices

Scrum does not talk about any Engineering practices

So, how do we get the work done?

Assumption 1:

Become a Jedi master, use the force to start the sprint, commit to the work and magically make it appear

Assumption 2:

'As a team' begin work in an agile manner

Come to realize that good Engineering practices are essential to becoming a high performing team

Solution (What to do?)

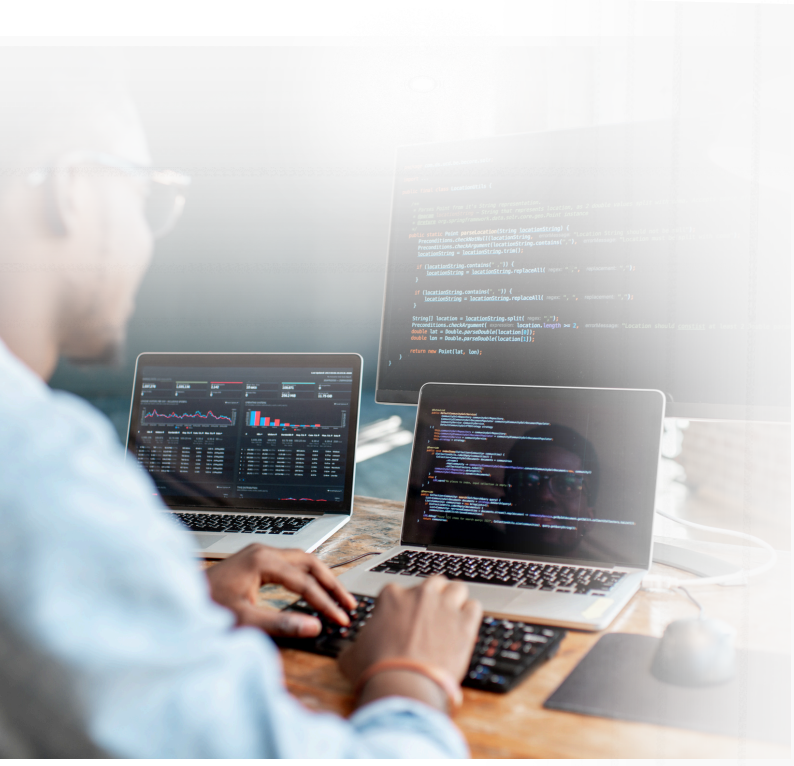
Introduce key engineering practices to scrum

- Test Driven Development (TDD)
- Continuous Integration and more frequent check-ins (CI)
- Refactoring
- Pair Programming (PP)
- Automated Integration and Acceptance tests

Test Driven Development (TDD)

What is TDD?

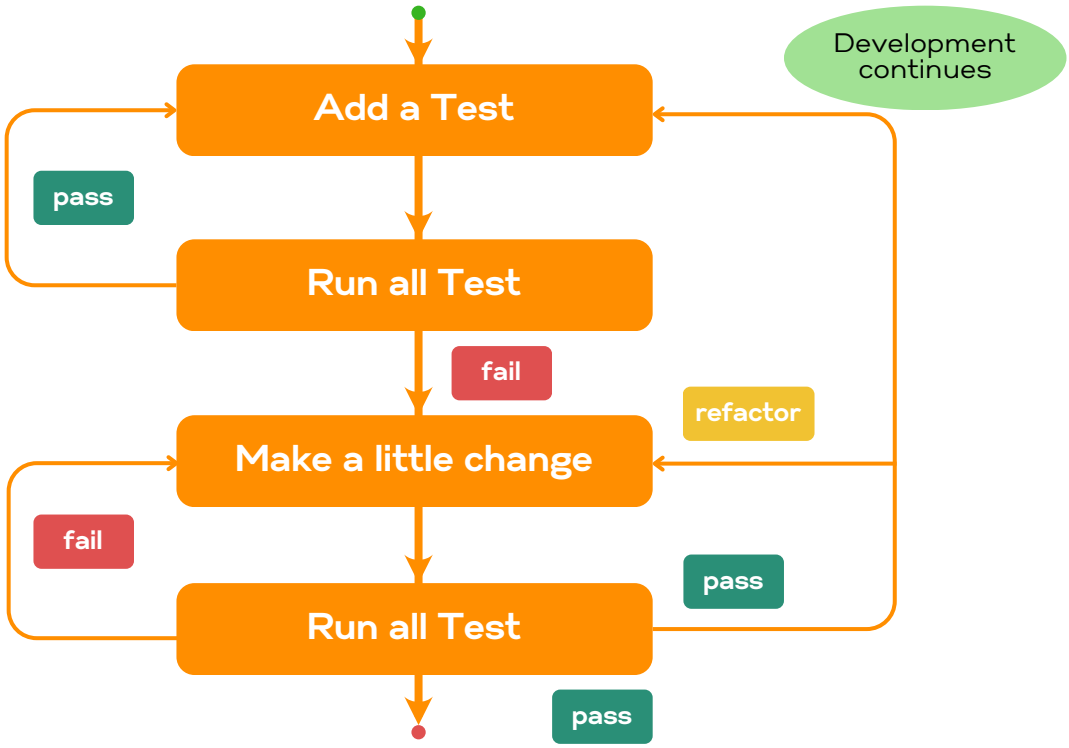
- Not a testing process
- A s/w design technique where the code is developed in short cycles
- 3 state process – Red, Green, Refactor



What are the Advantages of TDD?

- Best choice for meeting sprint goals at a high velocity
- Each automated test builds on the next, forming a safety net of automated unit tests

TDD – How it works



Test Driven Development (TDD)

Advantages (contd.)

- A team choosing to use the high wire act with TDD safety net can recover easily/fast from failures. Otherwise, it could take days, maybe weeks, to recover
- Team feels confident even with requests for last minute (emergency) changes
- More code is written when using TDD, but, total reduction in debugging time makes up more than enough for the extra code/time
- Gives the team peace of mind (priceless)

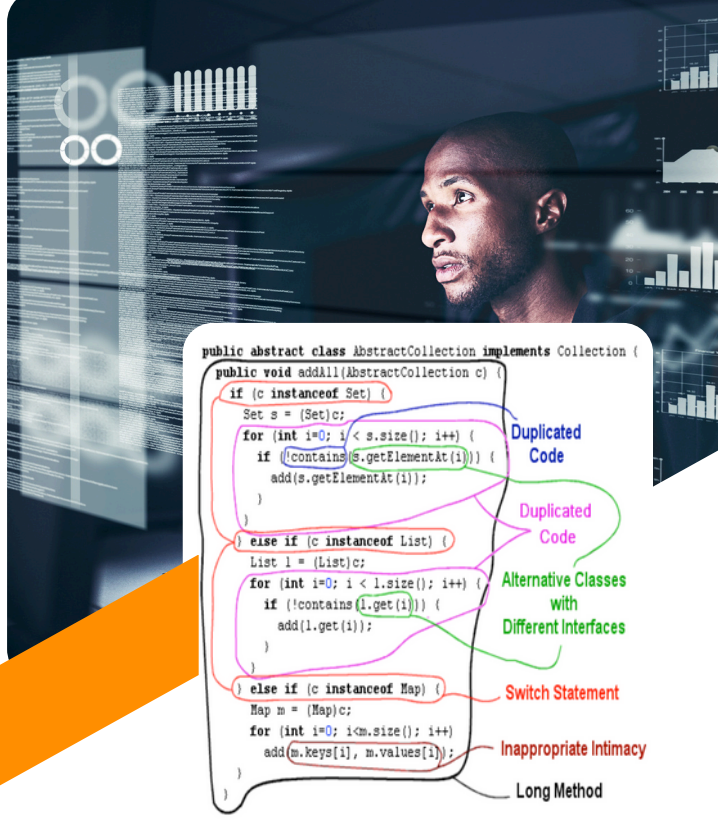
Refactoring

The act of enhancing or improving the design of existing code without changing its intent or behavior.

The external behavior remains the same, while under the hood, things are more streamlined.

When is Refactoring needed?

- When the code smells
- If it stinks, change it ☺
- A code smell is seemingly small issue but has symptoms of becoming big



Jeff Atwood lists some common smells in a 2006 blog post

SYMPTOMS	PROBLEM
Comments	There's a fine line between comments that illuminate and comments that obscure. Are the comments necessary? Do they explain "why" and not "what"? Can you refactor the code, so the comments aren't required? And remember, you're writing comments for people, not machines.
Long Method	All other things being equal, a shorter method is easier to read, easier to understand, and easier to trouble-shoot. Refactor long methods into smaller methods if you can.
Inconsistent Names	Pick a set of standard terminology and stick to it throughout your methods. For example, if you have open(), you should probably have closec().

Robert Martin's 'SOLID' principle for OOD

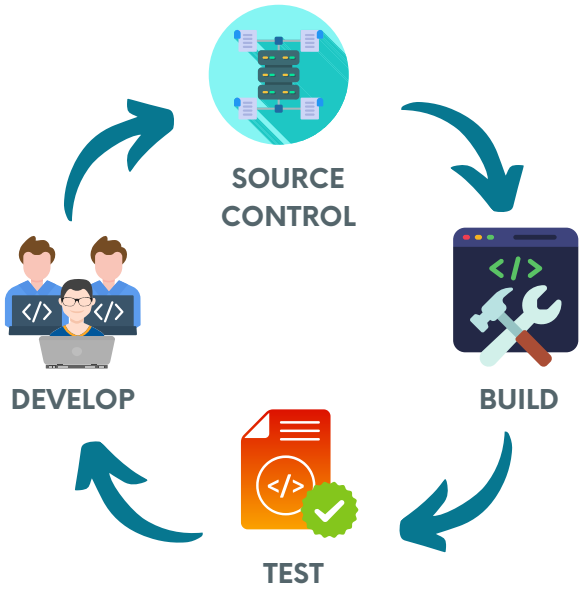
PRINCIPLES	DESCRIPTION
SRP: Single Responsibility Principle	A class should have one, and only one, reason to change.
OCP: the Open Closed Principle	You should be able to extend a class's behavior, without modifying it.
LSP: the Liskov Substitution Principle	Derived classes must be substitutable for their base classes.
ISP: Interface Segregation Principle	Make fine-grained interfaces that are client specific.
DIP: The Dependency Inversion Principle	Depend on abstractions, not on concretions.

When can Refactoring be done?

- Anytime during the code lifecycle
  - When someone sees a piece of code that can be improved
  - During legacy work
  - When bugs pop out from a section of code
- Bottom-line, anytime the code smells
- Refactor does not mean to re-write the code, but just to optimize to good coding practices

Continuous Integration & frequent check-ins (CI)

- CI is a software development practice which requires team members to integrate their work frequently, with usually each member integrating at least once/day
- This results in multiple integrations every day
- Every integration is verified by an automatic build with test to detect any integration errors
- Provides quicker feedback loops on the code
- Team members are encouraged to check in the code of 100 lines or less (way different than traditional check in of 1000 lines or more)



CONTINUOUS INTEGRATION & FREQUENT CHECK-INS (CI)

ADVANTAGES

CI	DAY TO DAY ANALOGY
CI shines a light down the path of code development so that you can know if you'll hit something (warns about blockages)	Car Headlights at night
CI reduces all (or most) manual processes. Not only does this free up time and resources, it also helps improve quality.	Washing Machines
CI provides visibility into the system under development – Current / next locations. Enables the team and the PO to react in real time.	GPS

CONCLUSION

- Scrum holds the expectation – at sprint end, all code should be potentially shippable (Teams struggle with what that really means)
- Continuous integration enables teams to build and release at any time
- With discipline, continuous integration can take a Scrum team from somewhat good to just plain awesome