Test-Driven Development

with Swift

Agenda

- What is TDD?
- What are the benefits?
- Golden rules of TDD
- TDD best practices
- Reasons to avoid TDD?
- Tidbits
- Where to go from here
- Q&A

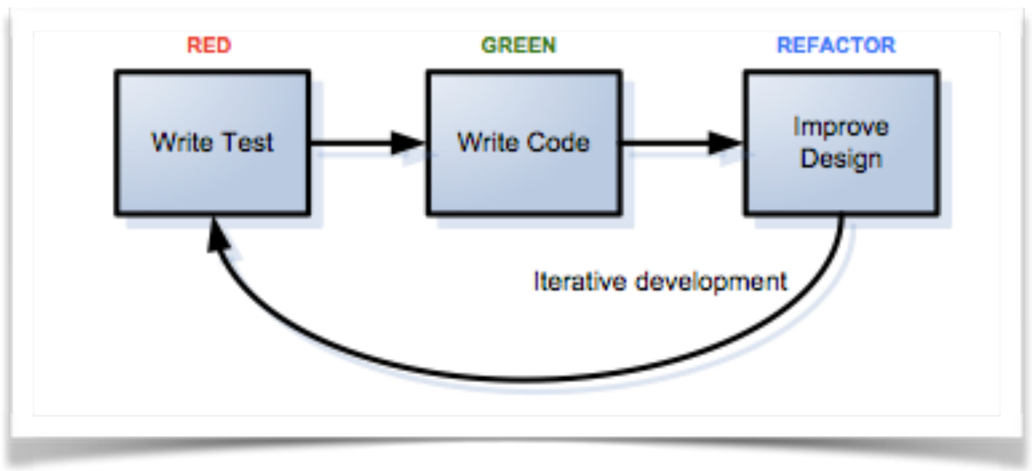
What is TDD?

What is TDD?

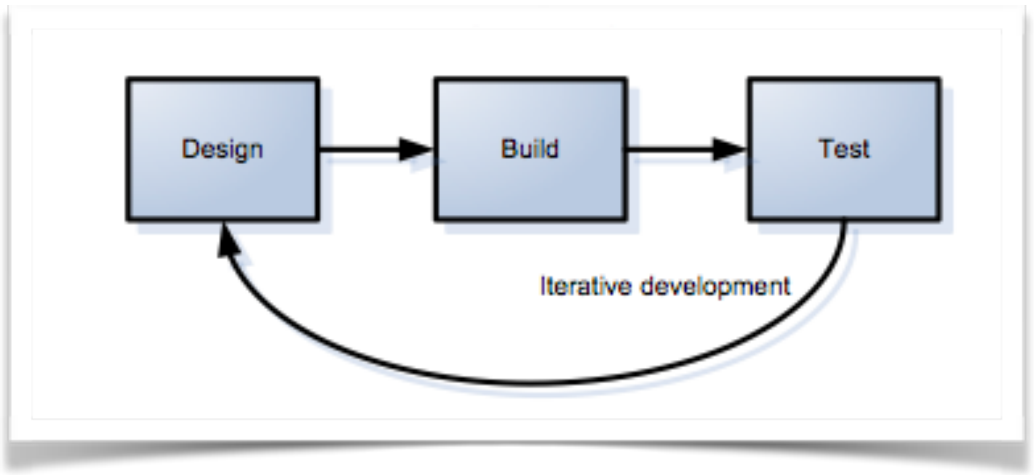
- Test-Driven Development is an approach to writing software.
- Red-green-refactor cycle.
- Test cases are written before the code itself; at that point, they are unpassable (‘red’). Code is written specifically to pass a given test case. When the written code successfully passes the test (‘green’), the passing code is **refactored.**

What is TDD?

Test-Driven Development



Traditional Development



What are the benefits?

Maintainable, flexible, Easily extensible

- Since testing in TDD is integrated into the development process at the most granular level, it is guaranteed that every standalone piece of logic can be tested – and therefore changed – confidently. At the end of the application development, there exist thousands of test cases. **When a change is made to the application, all that must be done is run the existing test cases to see if the change has adversely impacted any other piece of the application. This removes all roadblocks from updating legacy applications and making changes within the current development**.

Clean interface

- Because programmers write the test first, the APIs they produce are naturally written from an API-user perspective. **Resultantly, these APIs are far easier to use than those written by programmers more concerned with the internal workings of their packages.**

Refactoring encourages improvements

- The refactoring process central to TDD ensures that developers constantly shore up the strength of the codebase. This prevents applications from growing dated and monolithic.

- TDD gives programmers the confidence to change the larger architecture of an application when adding new functionality. **Without the flexibility of TDD, developers frequently add new functionality by virtually bolting it to the existing application without true integration – clearly, this can cause problems down the road.**

Self documentation

- Since TDD use-cases are written as tests, other programmers can view the tests as usage examples of how the code is intended to work.

Golden rules of TDD?

- Test first.
- Implement the simplest solution.
- Test once.
- Test in isolation.
- Test the interface, not the implementation.
- Refactor your code.

TDD best practices

- Organize your design.
- Create a test list.
- One faulting test at a time.
- Assert first.
- Simplest test data / Evident test data.
- Continuous integration.

Reasons to avoid TDD?

- My architecture and documentation are perfect.
- My team never changes and all members' memories are perfect.
- Unit tests don't help me, because my code works perfectly the first time.
- I never have enough time to write the tests.
- I don't like TDD, because I enjoy the hours I spend in my debugger.
- I'm paid to code, not test.
- I don't like to learn new things.

Tidbits

What not to test?

- Private methods (this is an implementation responsibility).
- UI Design (colors, positions, fonts, constraints).
- Auto-generated code.
- More tips here

Acquired skills after using TDD

- Loosely coupled objects.
- Well planned and designed public object APIs.
- Better usage of composition.
- Focus, patience, tenacity, discipline and dedication.

How to get started?

- Read this presentation.
- Check the demo project created specially for you.
- Start small.
- Think and plan long-term, investing in the future of your application.

Rewards of using TDD

- Making forward and consistent progress in small increments.
- Identify bugs early and avoid regressions.
- Reduces costs up-front.
- Add new features knowing you won’t break the existent ones.
- Deploy anything with confidence.
- Easier to understand the code base, not only for you, but your entire team.

Where to go from here

- Demo project
- Continuous integration
- Testing with Xcode

Q&A

?

The end

Tack så mycket

:)