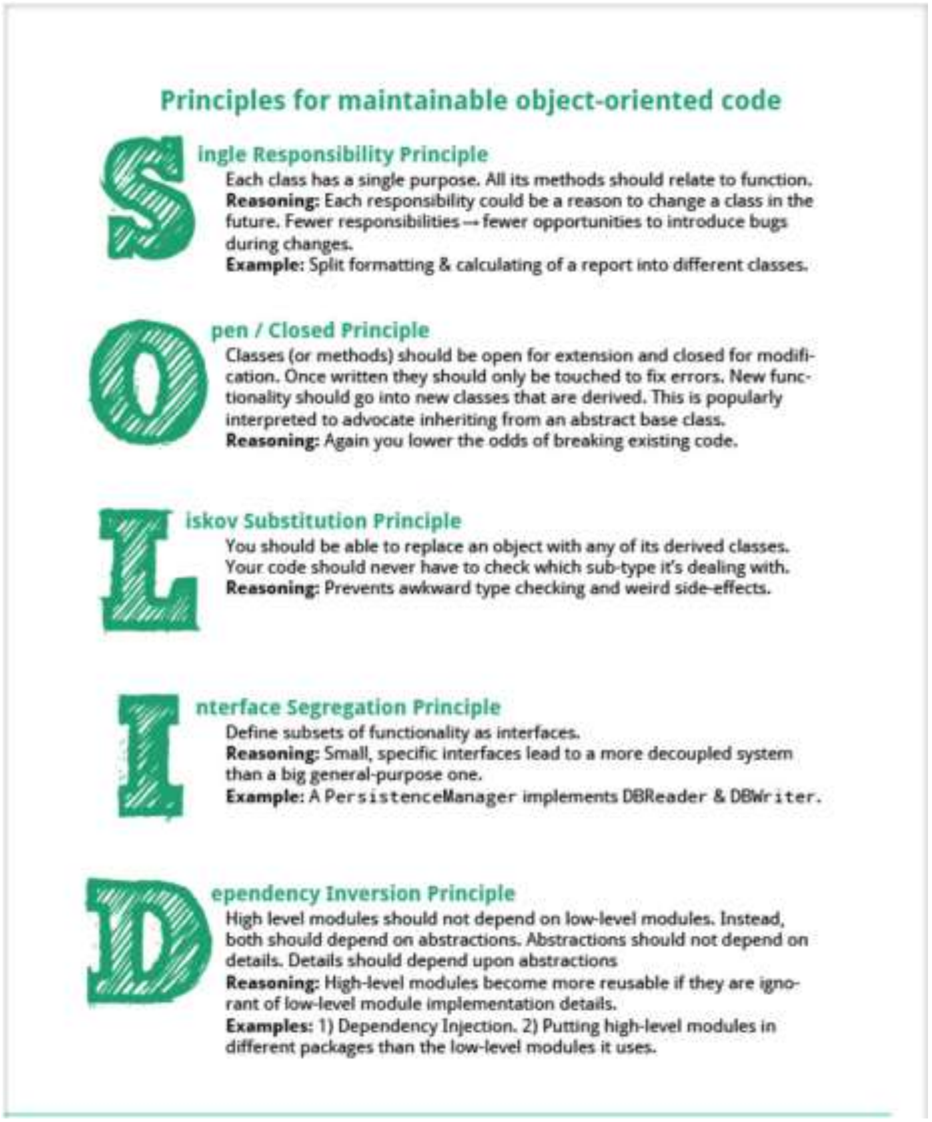
Developer Guidelines

- Developers must be familiar with and demonstrate most of the SOLID principles. (See Image Below)
- Developers must have a demonstrable competency in Test Driven Development (TDD). (See Image Below)
- Developers must have a strong understanding of at least one Mobile oriented language (Java or Swift) and a good understanding of at least one Mobile platform (iOS or Android).
- Should be able to successfully complete a supplied Kata in their choice of language and platform
- Should demonstrate good organization skills, along with consistent and well-formed code within the supplied Kata
- Developers must have a working knowledge of the Agile Process including; Sprints, Grooming, Demos and Retros.
- Developers must be willing to pair program with another developer and be willing to switch pair partners often throughout a sprint.



Principles for maintainable object-oriented code

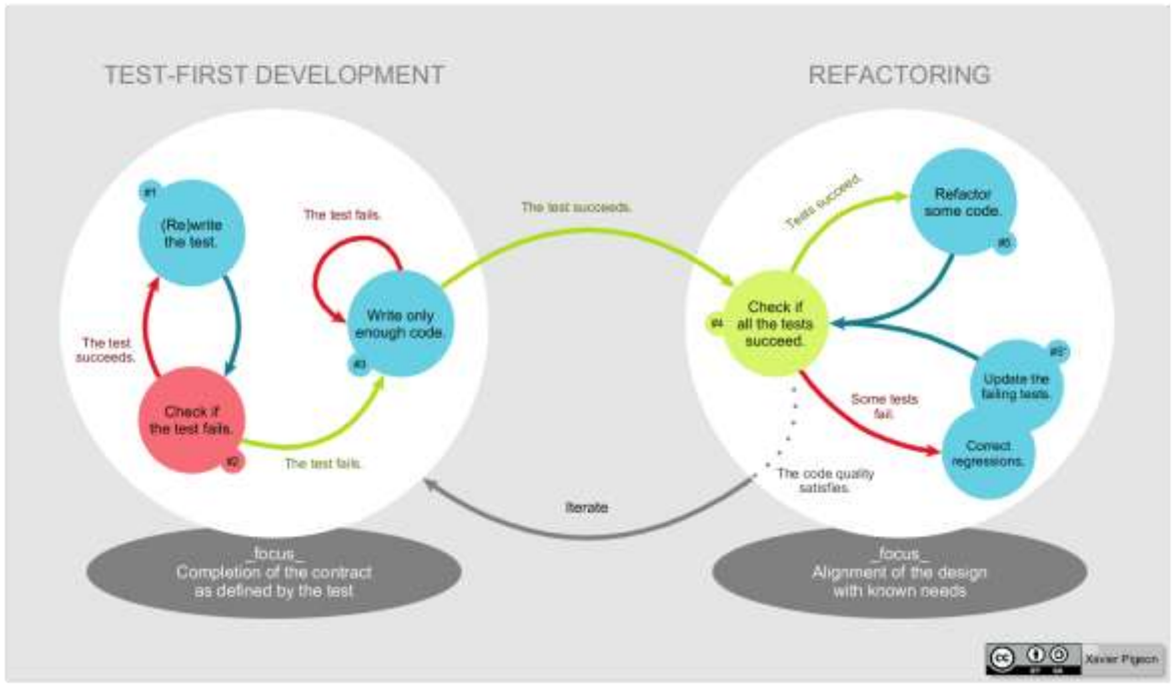
Single Responsibility Principle
Each class has a single purpose. All its methods should relate to function.
Reasoning: Each responsibility could be a reason to change a class in the future. Fewer responsibilities → fewer opportunities to introduce bugs during changes.
Example: Split formatting & calculating of a report into different classes.

Open / Closed Principle
Classes (or methods) should be open for extension and closed for modification. Once written they should only be touched to fix errors. New functionality should go into new classes that are derived. This is popularly interpreted to advocate inheriting from an abstract base class.
Reasoning: Again you lower the odds of breaking existing code.

Liskov Substitution Principle
You should be able to replace an object with any of its derived classes. Your code should never have to check which sub-type it's dealing with.
Reasoning: Prevents awkward type checking and weird side-effects.

Interface Segregation Principle
Define subsets of functionality as interfaces.
Reasoning: Small, specific interfaces lead to a more decoupled system than a big general-purpose one.
Example: A `PersistenceManager` implements `DBReader` & `DBWriter`.

Dependency Inversion Principle
High level modules should not depend on low-level modules. Instead, both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions
Reasoning: High-level modules become more reusable if they are ignorant of low-level module implementation details.
Examples: 1) Dependency Injection. 2) Putting high-level modules in different packages than the low-level modules it uses.


TEST-FIRST DEVELOPMENT
REFACTORING
#1
(Re)write the test.
The test fails.
The test succeeds.
Write only enough code.
#3
The test succeeds.
Check if the test fails.
#2
The test fails.
Tests succeed.
Refactor some code.
#5
Check if all the tests succeed.
#4
Update the failing tests.
#5'
Some tests fail.
Correct regressions.
The code quality satisfies.
Iterate
focus
Completion of the contract as defined by the test
focus
Alignment of the design with known needs
Xavier Pigeon