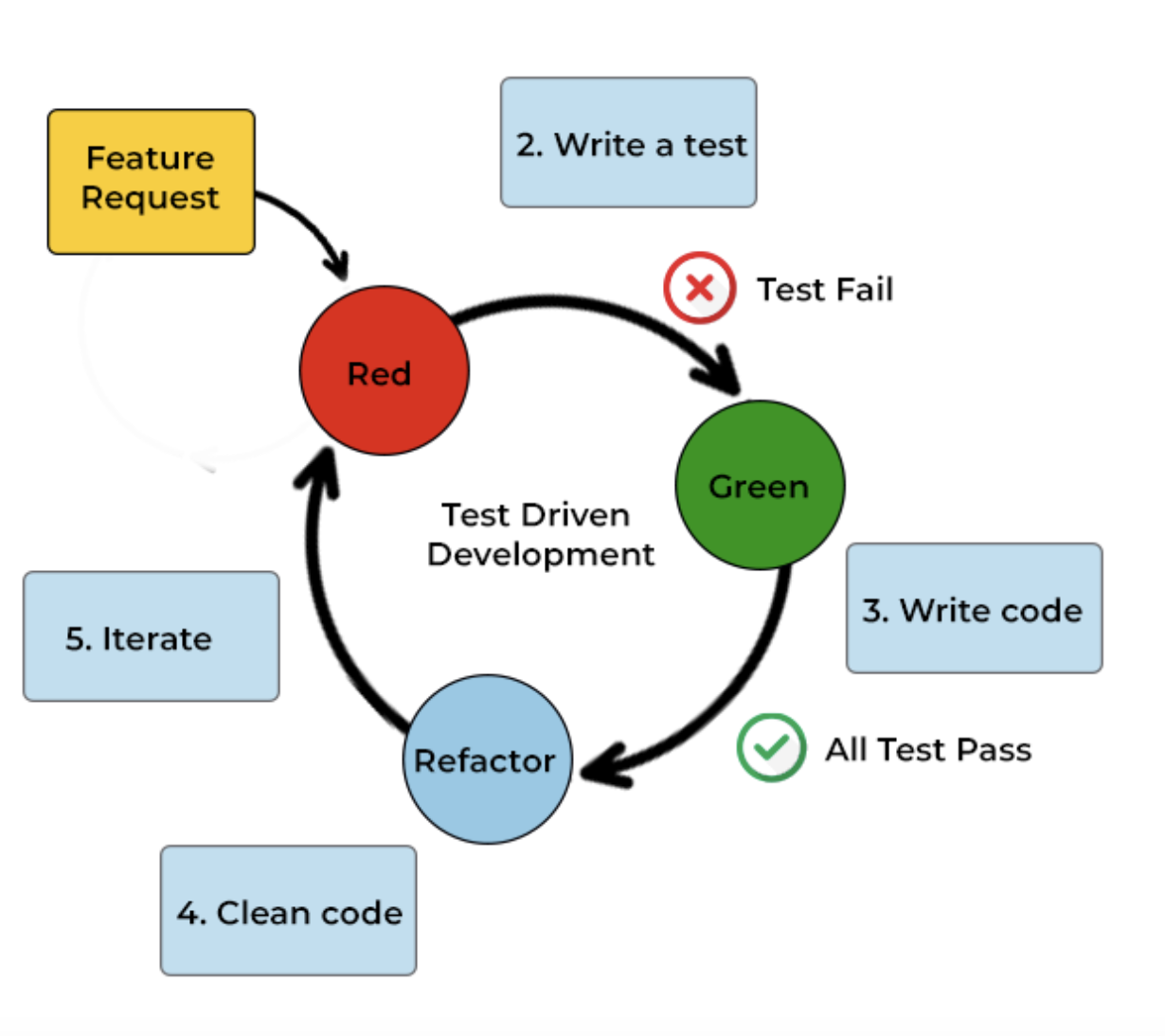
# **Development Framework**

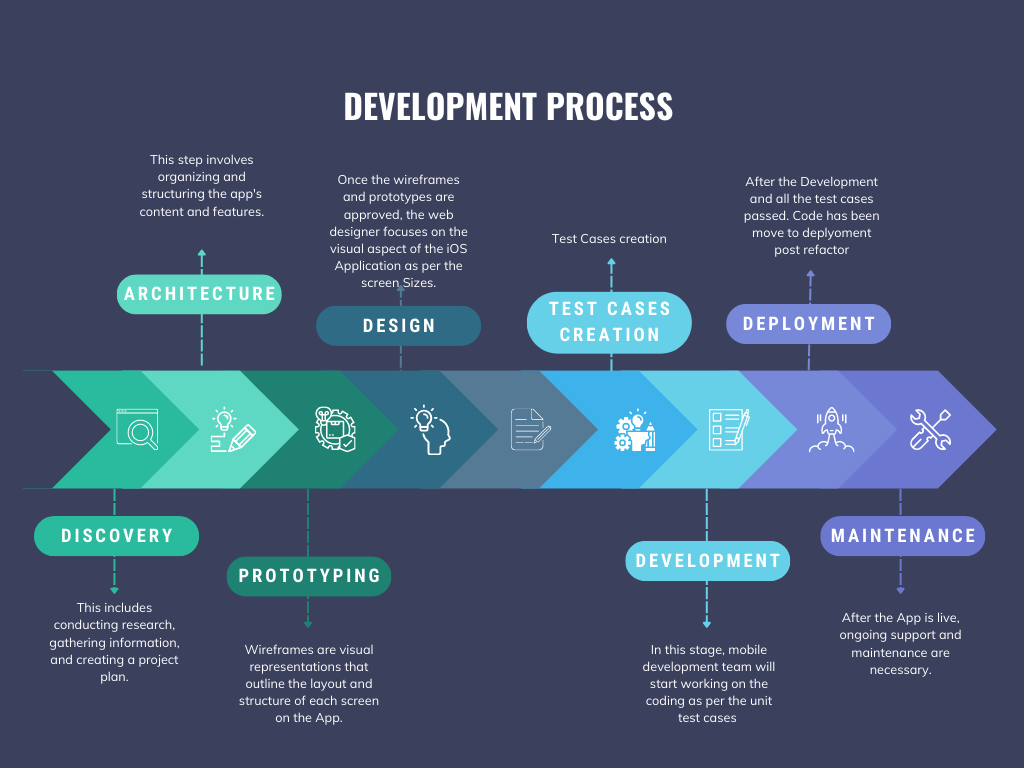
**Development Approach – TDD (Test Driven Development)**

Will follow the TDD approach with the following flow:

1. Getting Feature Request
2. Write Test cases.
3. Development
4. Code Cleaning
5. Distribute



So, the completed process will be like:



**Development Approach – Test Strategy**

Following will be the test strategy for the iOS app testing:

* XCode Accessibility Inspector
* UI Tests
* Unit Tests
* Performance Tests
* Snapshots tests for screen comparison testing
* Manual testing
* GitHub CI
* SonarQube/ Assessment tool scans

**Standards & Best Practices for code contribution**

* A concise description of the problem
* A reproducible test case.
* A description of the environment that reproduces the problem.
* Incremental Development
  + Should use small and incremental changes as preferred model so the branching should include small and frequent changes
  + Small branches are easy to review
  + The team feel more interactive in small and interactive works
  + Small branches make the code tested frequently by continuous integration infrastructure.
  + Some more considerations:
    - Decompose the remaining interrelated work into unrelated sets of changes.
    - Make each change in the set either stand-alone or part of a planned series of changes that work toward the development goal

**Commit Messages:**

* + Separate the commit message into a single-line title and a separate body that describes the change.
  + Make the title concise to be easily read within a commit log and to fit in the subject line of a commit email.
  + For text formatting and spelling, follow the same rules as documentation and in-code comments—for example, the use of capitalization and periods.
  + If the commit is a bug fix on top of another recently committed change, or a revert or reapplies of a patch, include the Git revision number of the prior related commit, e.g. “Reverted because it caused bug#”.
  + If the commit fixes an issue in the bug tracking system, include a link to the issue in the message.

**Attribution of Changes** - Developers should use the correct attribution of the contribution.

**Code Templates –** The licence and copyrights header should be filled out appropriately.

**Code Review –**

* + All significant changes, by all developers, must be reviewed before they are committed to the repository. Smaller changes can be reviewed after being committed.
  + The developer responsible for a code change is also responsible for making all necessary review-related changes.
  + Review other people’s changes.
  + Split your change into multiple smaller changes.
  + Ping the change

**Testing** - Developers are required to create test cases for any bugs fixed and any new features added, and to contribute them along with the changes.

* + All feature and regression test cases are added to the appropriate test directory
  + Write test cases at the abstraction level nearest to the actual feature
  + Reduce test cases as much as possible, especially for regressions.

**Quality** – Before committing to the main branch :

* + Code must compile without errors or warnings on at least one platform.
  + Bug fixes and new features must include a test case to pinpoint any future regressions.
  + Code must pass the appropriate test suites.

**Commit Access** - The following policies apply to users with commit access:

* You are granted commit-after-approval to all parts of Swift. To get approval, create a pull request. When the pull request is approved, you may merge it yourself.
* You may commit an obvious change without first getting approval. The community expects you to use good judgment.
* You are allowed to commit changes without approval to the portions of Swift to which you have contributed or for which you have been assigned responsibility. Such commits must not break the build.

# Source control branching strategy

#### **Branches**

##### Master

* Each commit in the master should reflect a stable release and be tagged with that release's version number.
* The latest commit in master reflects the build currently available in the App Store.
* Until the first version of the app is released, the master will usually reflect the initial state of the repository.

##### develop

* The main development branch reflects the most current non-shipped version of the app.
* Serves as the main integration point for new features/fixes as they are developed.
* The CI system points to develop by default, with QA performing the bulk of their testing on builds created from this branch.

##### feature/\*

* Represents a feature or fix actively in development
* Initially branched from develop to start feature work.
* As the feature is developed, develop should be merged into feature/\* periodically to keep the feature branch up-to-date with the latest changes.
* Once the feature is complete, the feature/\* branch is merged into develop (via pull request) and then removed.

##### release/\*

* Represents a release candidate.
* Initially branched from develop to start a code freeze process.
  + Only bug fixes should be merged into the release/\* branch (i.e. no merging from feature/\* branches).
* Merged into develop periodically so that fixes get incorporated into new feature development.
* Merged into master and tagged using [semantic versioning](https://semver.org/) once a release is complete (i.e. the app/update is live).
* Merged into develop one last time before being removed.

##### hotfix/\*

* Represents a "hotfix" update used to quickly address issues with the live app.
* Initially created from the corresponding version number tag on the master branch
* Allows for production issues to be resolved quickly without interrupting active feature development or having to manually cherry-pick or re-apply fixes from develop.
* Follows the same process as a release/\* branch to keep develop up-to-date with any fixes and finalize the hotfix release.

#### Key Stages

1. The repo is created with only a master branch, by default.
2. A develop branch is created from the master.
3. feature/\* branches are created from develop.
4. When a feature is complete, it's merged into develop and then removed.
5. To initiate a release, a release/\* branch is created from develop.
6. When a release is complete, release/\* is merged into develop and master, tagged, and then removed.
7. If an issue in master needs to be resolved, a hotfix/\* branch is created from master.
8. When the hotfix release is complete, hotfix/\* is merged into develop and master, tagged, and then removed.

**Commits:**

The developer should push any pending commits to the remote repo at the end of every day so that all in-progress work is backed up, and the commit message should be detailed and helpful.

## **Merging**

All merges into develop/trunk, release/\*, and hotfix/\* should happen via pull requests. This ensures that all code gets reviewed at some point before it's shipped.

## **Deleting Branches**

Branches should be deleted after they've been merged into develop or master. This keeps the repository clean and makes it clear where active development is occurring.

# Release Strategy from development to UAT and internal testing through to production to customers.

A diagram of a test

Description automatically generated

The release strategy should include the following checks:

* Code Review
* Release Planning
  + Feature Release
  + Scheduled release
* Release checklist
  + Tasks
  + Owner
  + Status
  + Comms to the end users
  + Backout/Rollback Plan
  + Review of completed design/Development work.
  + Unit testing
  + QA Planning and testing
  + Tracking/ fixing discovered bugs.
* Release configuration management
  + Reviewing official app submission guidelines
  + Checking for unapproved changes
  + Preparing the app for the release
  + Adding analytics to the app
* User Experience
  + Ensuring that all the new/ updated features are usable.
  + Updating user documentation
  + App localization
  + Providing feedback channels
* Legal aspects
  + Checking all the licenses
  + Checking compliance with laws/regulations.

The release strategy should start with the planning and the features team going to release. Once all the features have been completed at the developer level and developer testing has been completed, it should go to the code review stage before committing the code for the CI infrastructure to create a build. Once the code has been committed and the build has been created it should submit to the QA for testing and after successful completion of the testing (if there are bugs raised that should be properly raised/documented in the tool and fixed before resubmitting the build to the testers) build should be submitted to the business testers for the UAT approval.

Parallelly the mobile team should raise the SDLCs and change requests to complete the release management process to be ready for distribution. If the UAT is successfully completed, team can go to the CAB for the change approval and get the change approved for deployment, but parallelly should be ready with the backup/rollback plan in case of any issues with the distribution.

After the change management approval, the team can proceed to submit the build to iTunes Connect for Apple review to release it on the AppStore.