

Tasting tests at Cookpad

by @Kazu_cocoa

for Try!SwiftTokyo

Hello, everyone.

I'm Kazu.

I'm so excited talking for many Swift and iOS developers.

Today, I talk about "**TEST**" since I'm a test engineer.

"**Tasting**" used in my title because my company is regarding with food.



Kazuaki Matsuo([@Kazu_cocoa](#))

 Cookpad Inc.

Test Engineer / Software Engineer in Quality

Lang:  /  /  / 

Maintainer: Ruby Appium binding 

At first, I introduce myself.

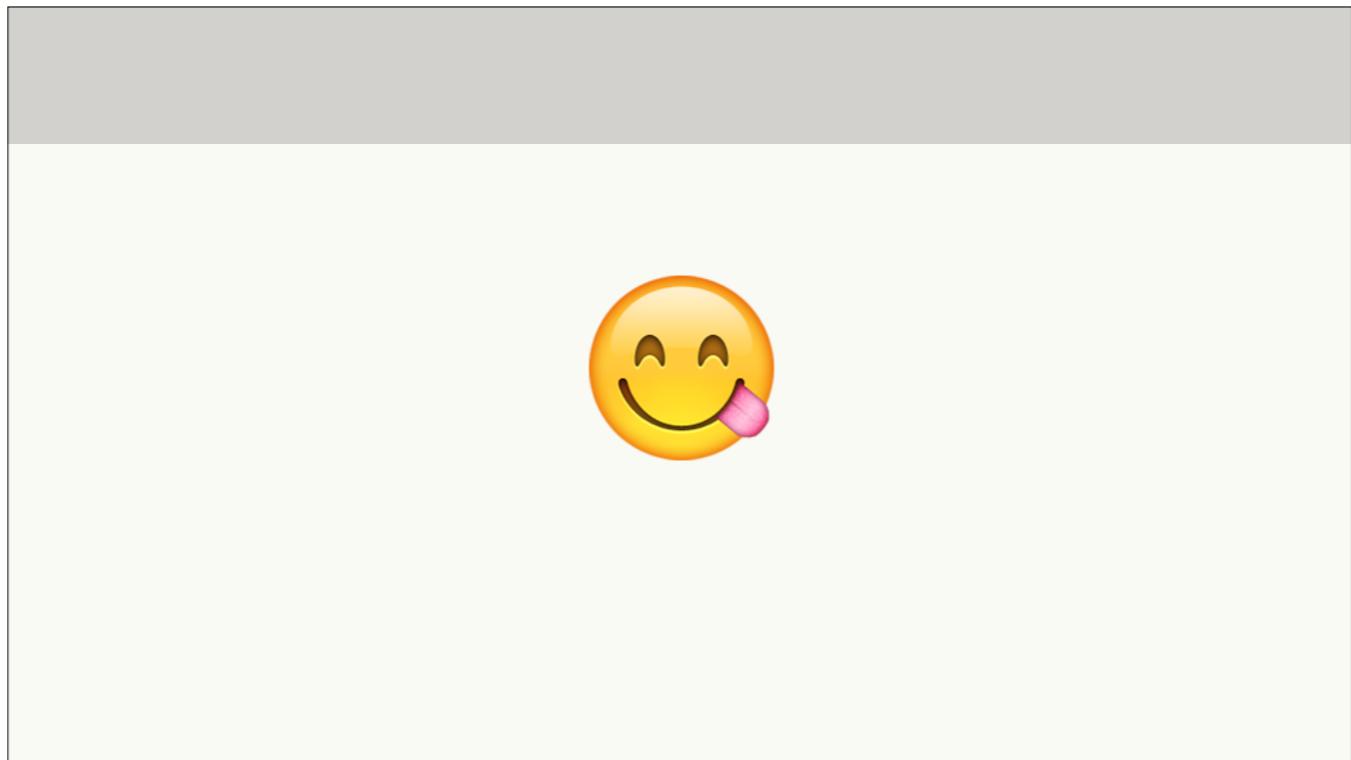
I'm Kazuaki Matsuo and working at Cookpad as a test engineer and Software Engineer in Quality.

I've tried test automation for mobile and improved development processes and other some roles to improve several qualities for our services.

I develop with some languages in my work such as swift, ruby, java for android and elixir.

Recently, I'm maintaining ruby appium binding.

Appium is one of the famous mobile test automation tools.



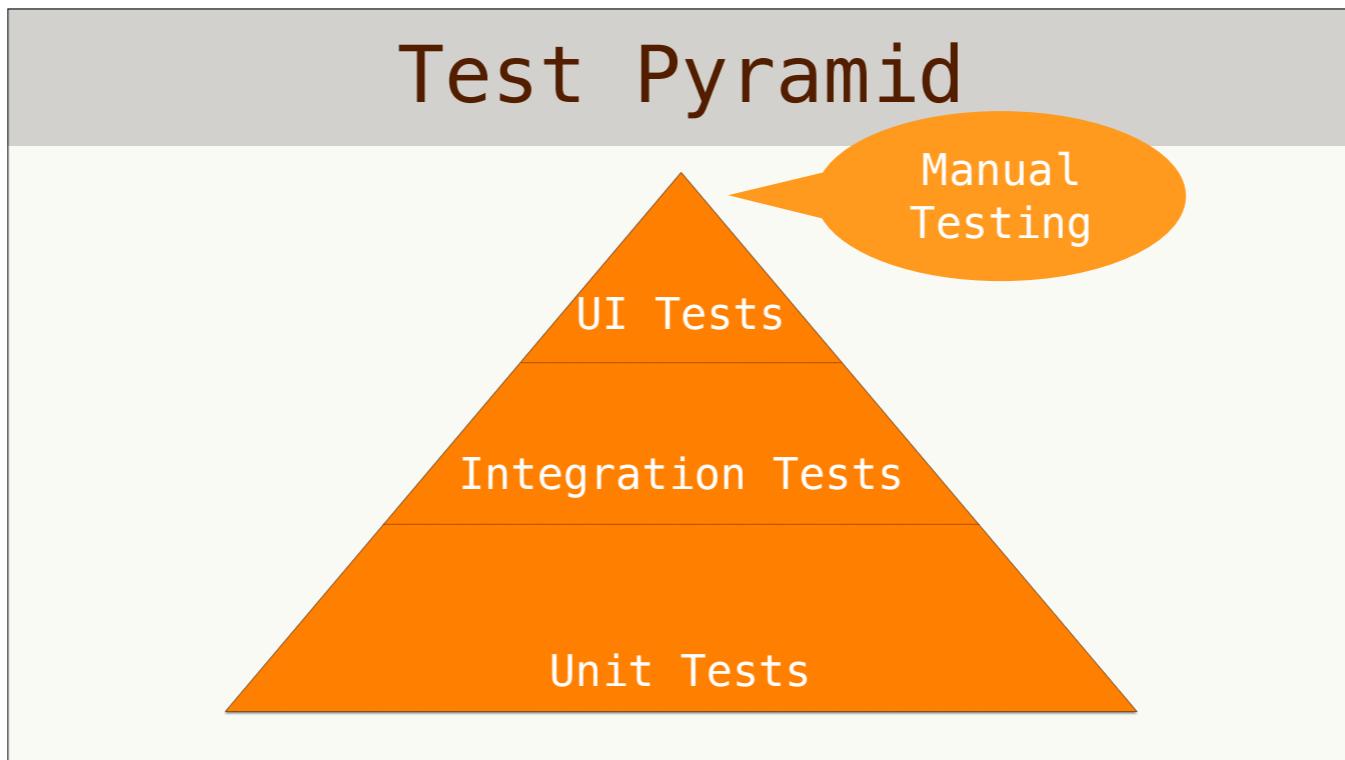
go ahead

A bunch of themes in tests

Even we use the word "**TEST**", it has a bunch of themes and various meanings.

For example, it has **categories** such as usability test and performance test, and test **level** such as unit test and integration test, and so on.

Today, I pick up test level and a test pyramid associated with test automation.



This pyramid is one of a famous diagram for test automation.

The pyramid has three layers, unit test, integration test and UI test.

The pyramid has also manual test outside of the pyramid.

This shows ideal **relationships** and **amounts** for each tests in development.

Unit tests check logic in method level and UI tests focus on checking GUI level simulated user behaviours.

Unit tests have the largest amount of test cases and UI tests have the smallest ones.

(<http://www.utest.com/articles/mobile-test-pyramid>)

How UI Tests support our development

I focus on UI tests and how the tests support our development.
I don't talk about unit/integration level tests today.

So, you can **taste** how UI tests support our development and the strategy in our app as a case study.

I'm happy if anyone gets the motivation to consider test strategy for your app and try UI tests, after my talk.

We should know about the test target



Before I start talking tests, we should learn a test target.

Because knowledge for the test target, how large, how long have been developed, release cycle and so on, help you understand test strategies and the case study.

So, I explain about Cookpad and its iOS app at first.



Cookpad is one of the most famous recipe sharing services in the world.

<https://www.similarweb.com/top-websites/category/food-and-drink/cooking-and-recipes>

The screenshot shows the 'Top Websites' section of the SimilarWeb website. The search filters are set to 'Cooking and Recipes' and 'Worldwide'. The results table displays three websites: Cookpad.com (rank 1), Allrecipes.com (rank 2), and Foodnetwork.com (rank 3). The table includes columns for Rank, Website, Category, Change, Avg. Visit Duration, Pages / Visit, and Bounce Rate.

Rank	Website	Category	Change	Avg. Visit Duration	Pages / Visit	Bounce Rate
1	cookpad.com	Food and Drink > Cooking and Recipes	=	00:04:20	4.63	36.34%
2	allrecipes.com	Food and Drink > Cooking and Recipes	=	00:03:07	3.08	52.67%
3	foodnetwork.com	Food and Drink > Cooking and Recipes	=	00:02:15	2.47	59.71%

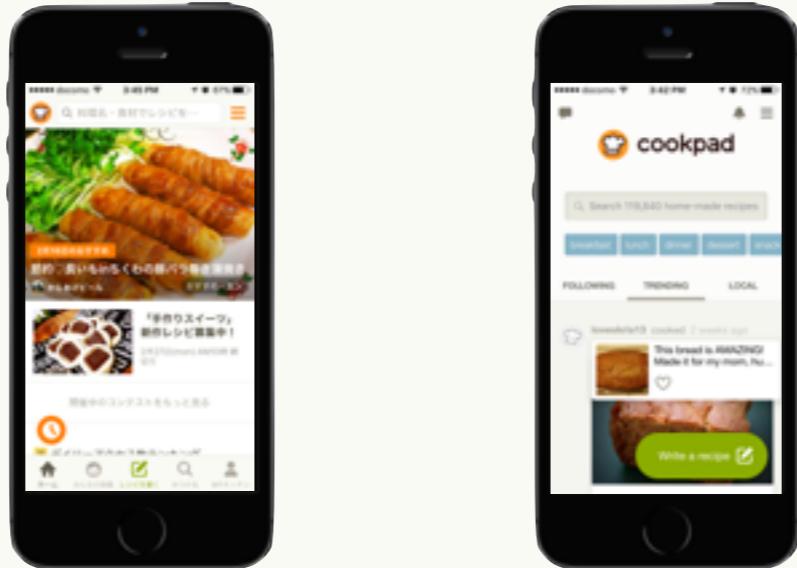
https://www.similarweb.com/top-websites/category/food-and-drink/cooking-and-recipes

According to the similarweb.com, Cookpad is the largest site for world wide in the food category.

We provide our service for two main areas, for Japan and for rest of the world right now.

<https://www.similarweb.com/top-websites/category/food-and-drink/cooking-and-recipes>

Cookpad for iOS(Japan and Global)



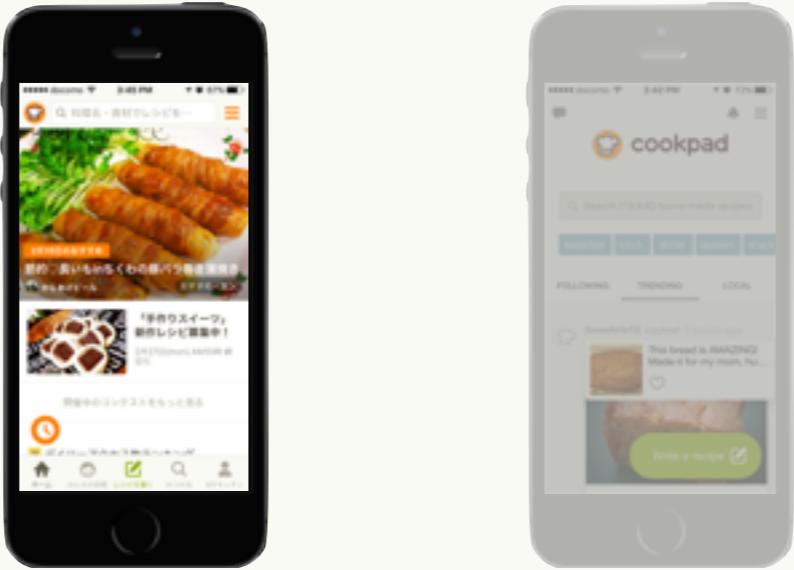
We also have two kinds of iOS app.

One is for Japan and another is for rest of the world.

Their service growth level is difference, so we haven't merged them yet.

Anyone comes from out of Japan, you can see global app.

Cookpad for iOS(Japan)



Today's test target is for Japan Edition.

The app has been developed for a long time



The Japan edition has grown for around 5 years.

I attached some screenshots to show the growth and the changes.

The app changed UI component, added or deleted any features or re-write/refactor implementations during the period.

The production code also has grown and it is around one hundred thousand lines except for comment, blank and new lines right now.

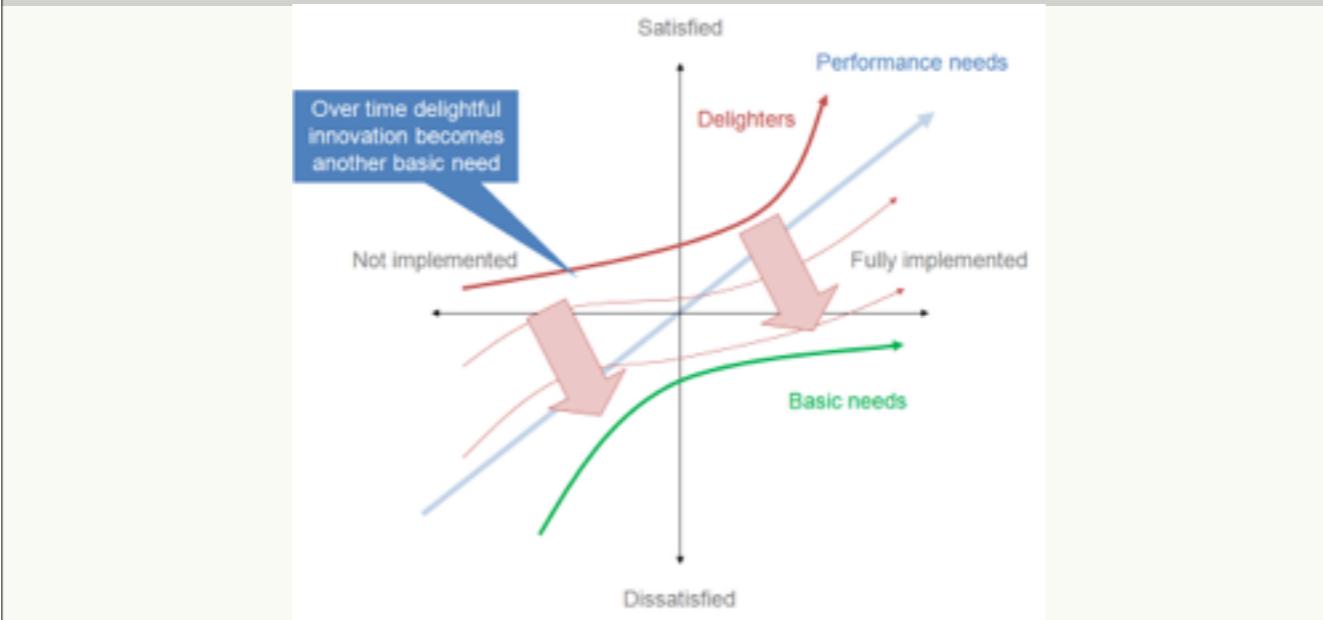


In this period, we've released the app many times.

Recent our release cycle is around one month, and we released around 2weeks cycle before.
We change source code from around 5 thousand to 10 thousand per a release.

The changes have some new features, GUI update, refactor/re-wrote internal logic and so on.

kano-model and Japan market



by the way, I also explain a bit about quality.

kano-model is one of the famous models to explain about quality.

The model has two main quality.

One is **Must-be** Quality and another is **Attractive** Quality.

Must-be quality is a criteria you must satisfied with basic needs.

Must-be quality in Japan have required crash-free app for almost views and no degrade features for a long time.

So, we had needed to keep crash-free conditions in our development.

It is not attractive for them.

(https://en.wikipedia.org/wiki/Kano_model)

Diachronic Quality for Mobile

Mobile app's environment have been changed quickly through time.
OS versions, UI, design, user experience, required quality by market and so on.

Recently, I sometimes mention such a movement as diachronic quality.
This word is influenced by linguistics.
Diachronic means something has developed and evolved **through time**.

A bunch of mobile environments have changed through time.

take a break



Cookpad iOS app has been developed so long time.

Must-be quality in Japan is high, especially
crash free rate.



So far, I talked histories of Cookpad iOS app and how long the app has been developed for a long time.
In addition, I talked a bit about quality for japan market and mobile.

~ 7:00

Tasting how re-engineering the app with UI Tests 😊

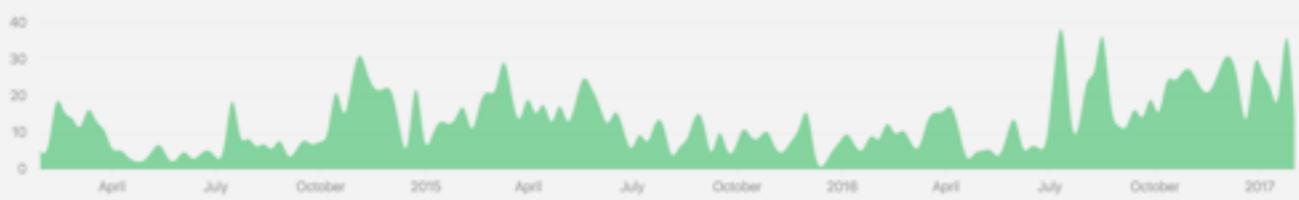
I start talking UI Test what I've challenged for the environment.

A History for UI Tests for Cookpad Apps

Feb 9, 2014 – Feb 16, 2017

Contributions to master, excluding merge commits

Contributions: Commits ▾



This graph shows commit log for UI tests I've implemented at Cookpad.
I've developed the environment since 2014.

Why have we implemented this UI tests?🤔

In 2014, We expected our service need to develop continuously and we should evolve apps in the future. So, we needed to proceed re-engineering for our mobile apps.

Re-engineering means re-structured the target and make it **testable** to be able to develop stuff continuously without degrade features and keep development speed and so on.

Should we taste from?

Writing unit tests before refactoring is sometimes impossible and often pointless.

Re-Engineering Legacy Software

> Writing unit tests before refactoring is sometimes impossible and often pointless.

This sentence is quoted from "Re-Engineering Legacy Software".

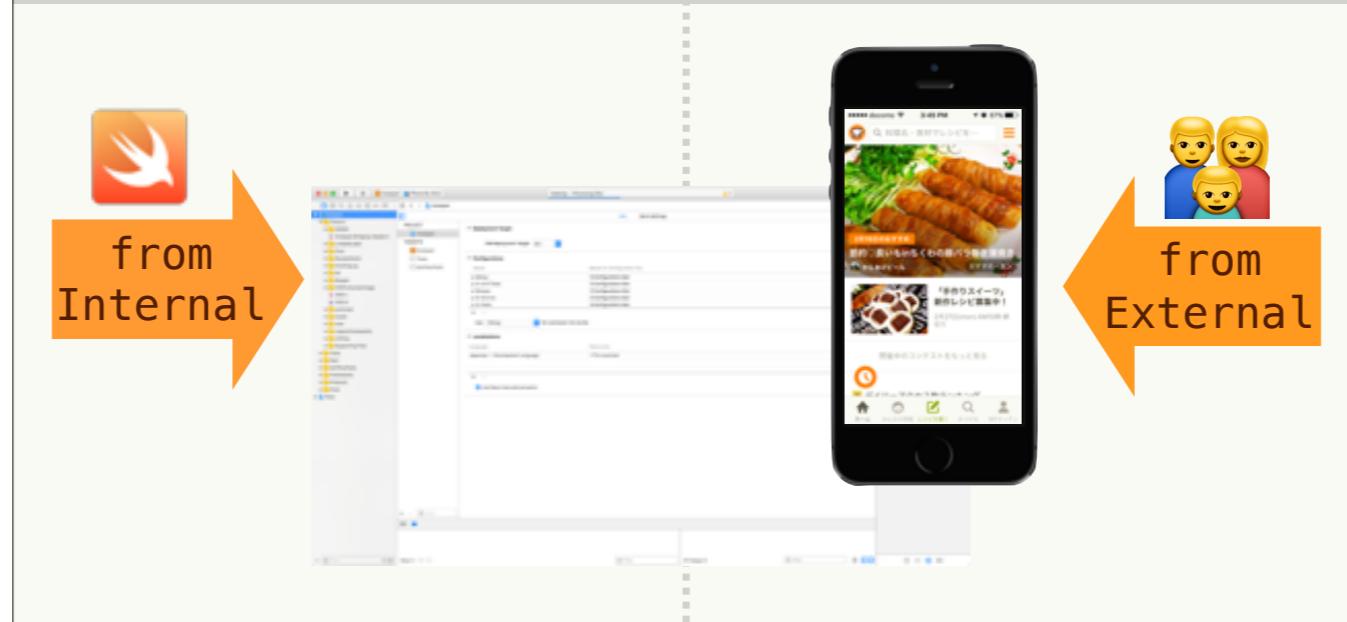
It is true since we can't check behaviour without tests.

The most of the developers may agree with re-write/refactor features without tests lead unexpected broken stuff.
In addition, to make the target testable, we should consider architectures for the app and other many things if target app isn't testable.

On the other hand, without CI environment, it is difficult to iterate development cycle quickly without tests.

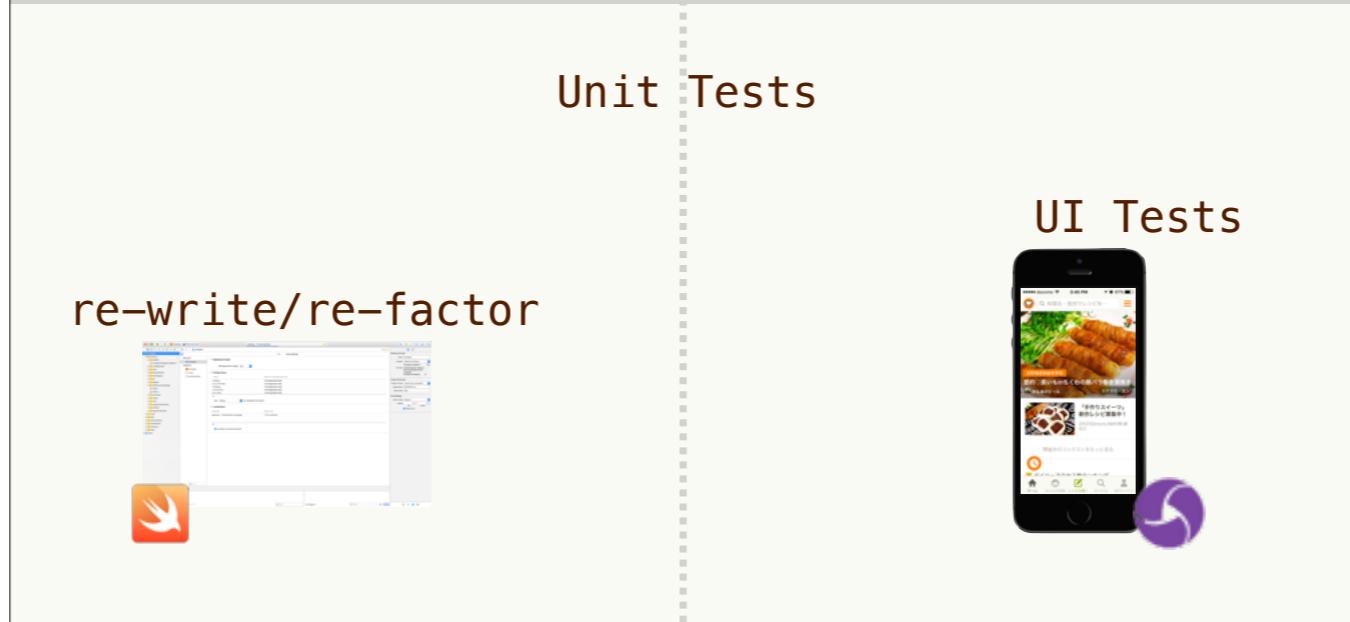
<https://www.manning.com/books/re-engineering-legacy-software>

Basic strategy



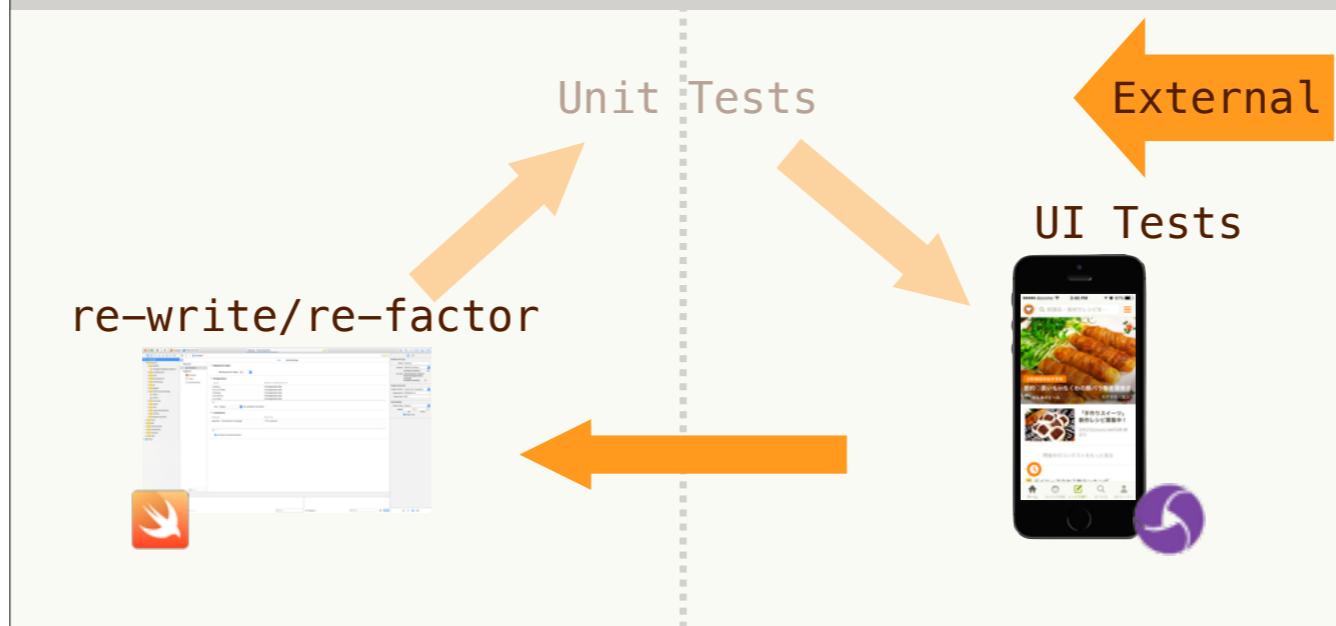
We approached to proceed re-engineering the app from two aspects, internal and external.
"Internal" means production code side.
"External" means end-user and GUI side.

Make checkable from external to internal



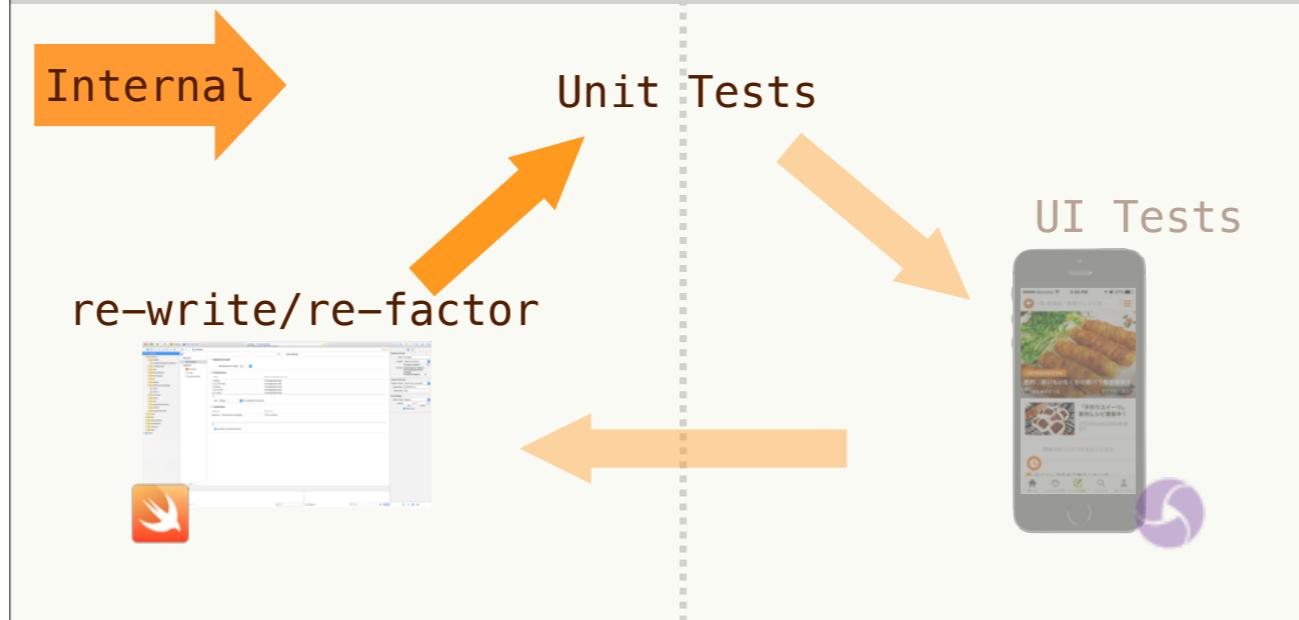
I explain a cycle.

Make checkable from external to internal



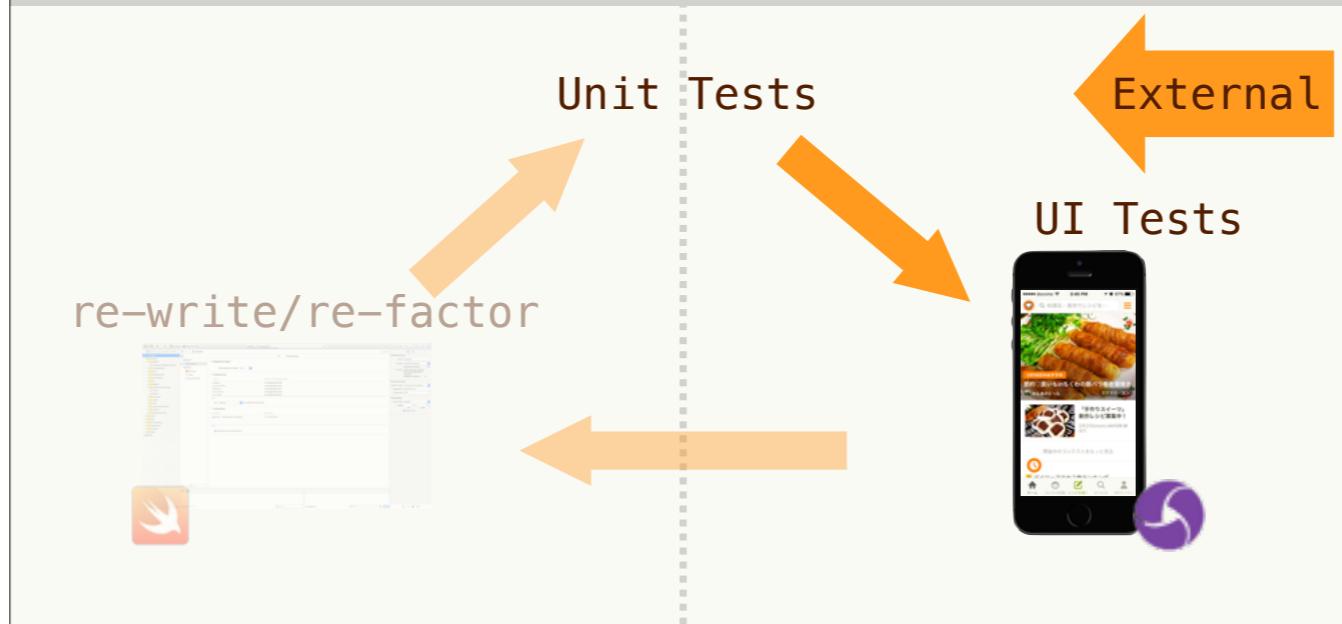
Check whether GUI, behaviour are broken from external.

Make checkable from external to internal



And re-write/refactor/implement some features, and implement and conduct unit tests.

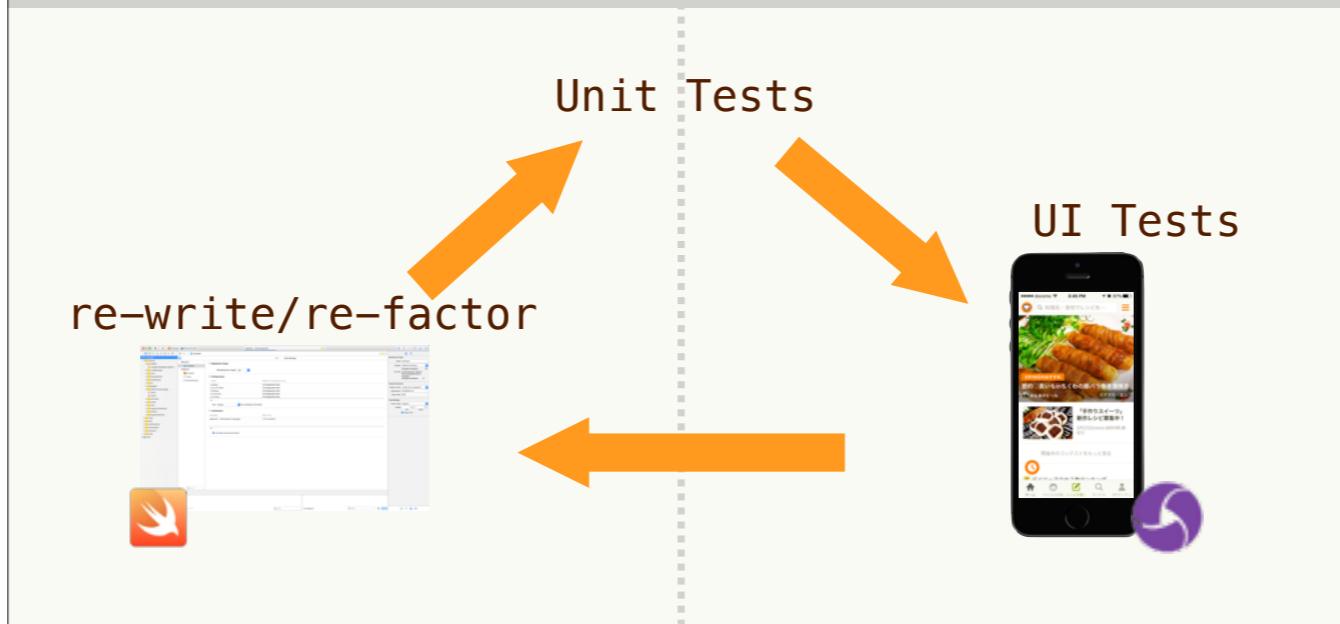
Make checkable from external to internal



Check the changes from UI side to uncover errors.

Developers can find some bugs if their implementations break some other features unexpected.

Make checkable from external to internal



If tests uncover some bugs, we can fix the issue before release.
We'll be able to proceed re-engineering with this cycle step by step.

This kind of UI level checking is not only automated tests but also manual checking.

Unit tests for Re-Engineering

*Most developers would agree that unit test
should be fully automated,*

Re-Engineering Legacy Software

The re-engineering also describes the following quotation.

> Most developers would agree that unit test should be fully automated,

I think most of the developers agree with this.

Unit tests are not a silver bullet

but the level of automation for other kind of tests(such as integration tests) is often much lower.

Re-Engineering Legacy Software

But, it also describes as...

> but the level of automation for other kind of tests(such as integration tests) is often much lower.

Certainly, implementing automated tests for integration and UI layer is difficult than unit layer especially mobile app.

UI Test should be automated

*One area that cries out for automation is
UI testing.*

Re-Engineering Legacy Software

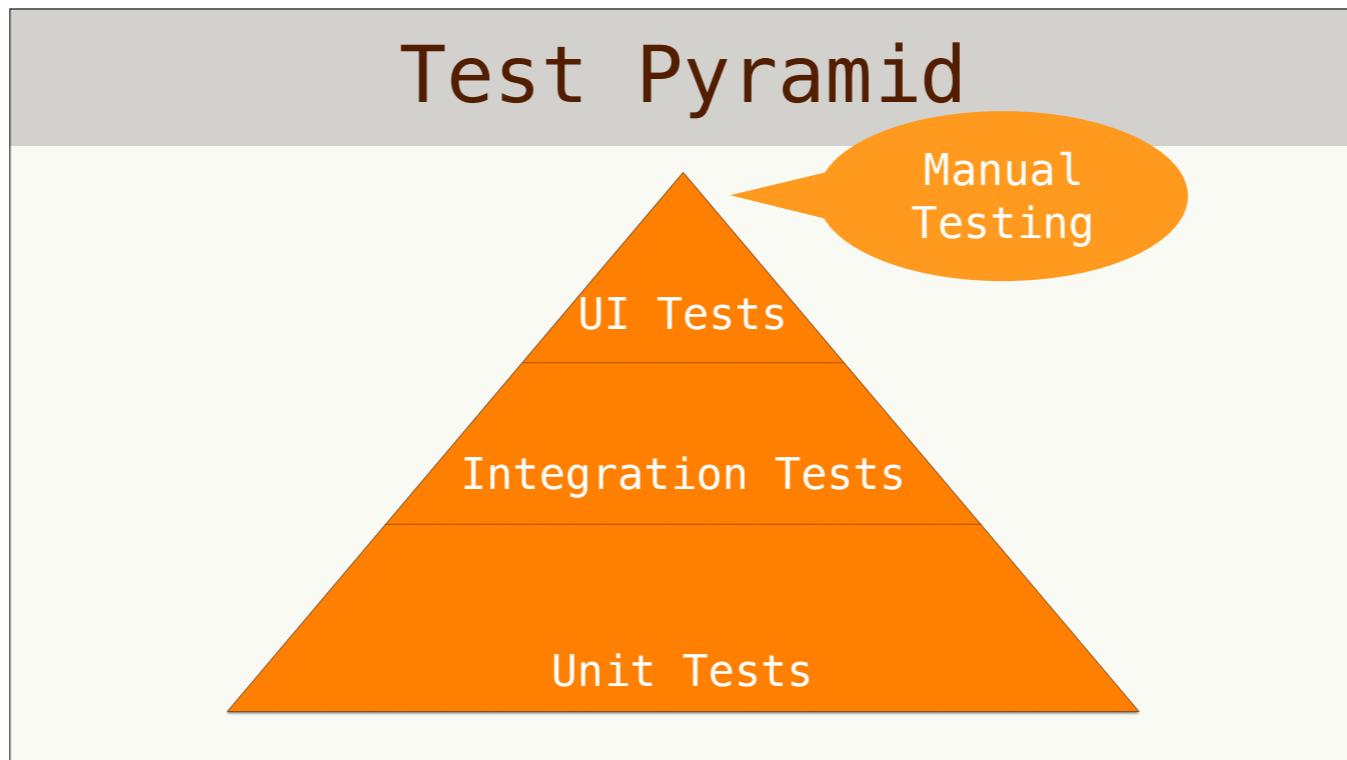
But...

> One area that cries out for automation is UI testing.

yes, automated UI testing is very important.

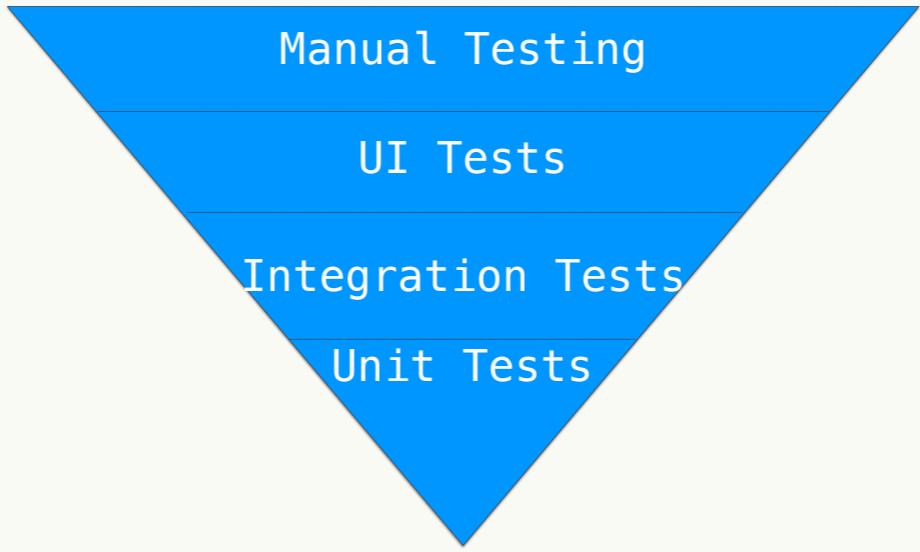
In many cases, UI tests take too many times and too many human resources in mobile because test automation for mobile is difficult.

UI tests for mobile tend to conducted by manual.



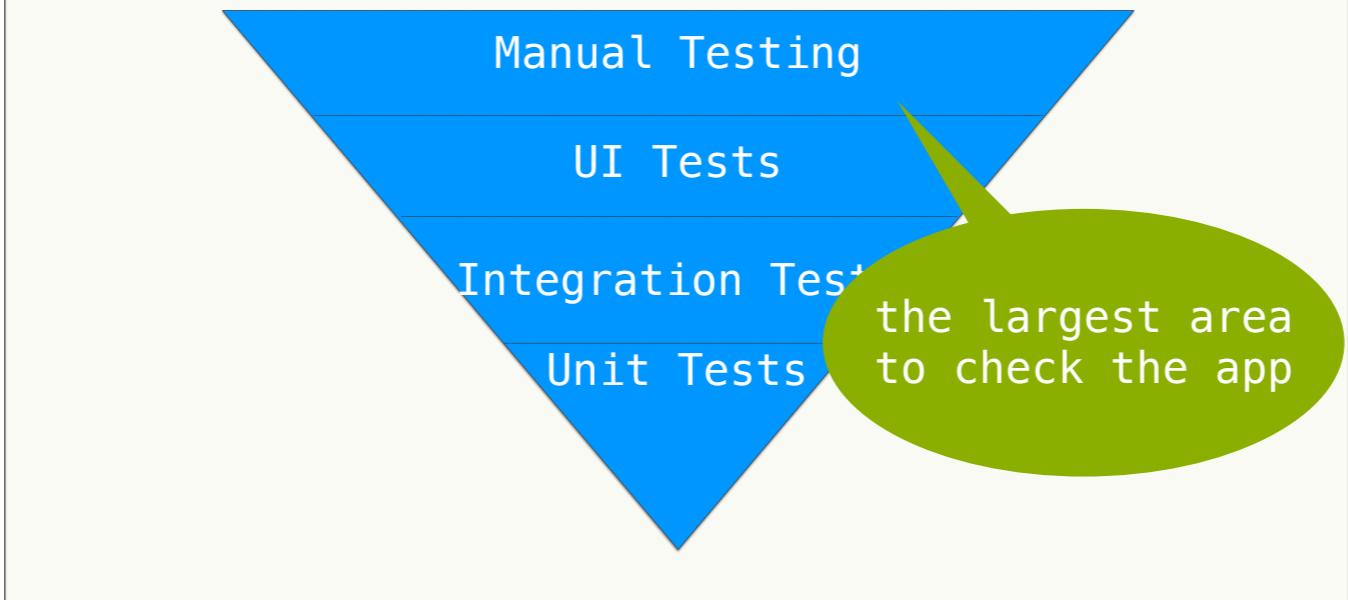
I talked the ideal test pyramid for test automation before.
This shows unit tests are the largest and UI tests are the smallest.

Flipped pyramid make development cycle slow



But in the mobile context, it is easy to make it flip.

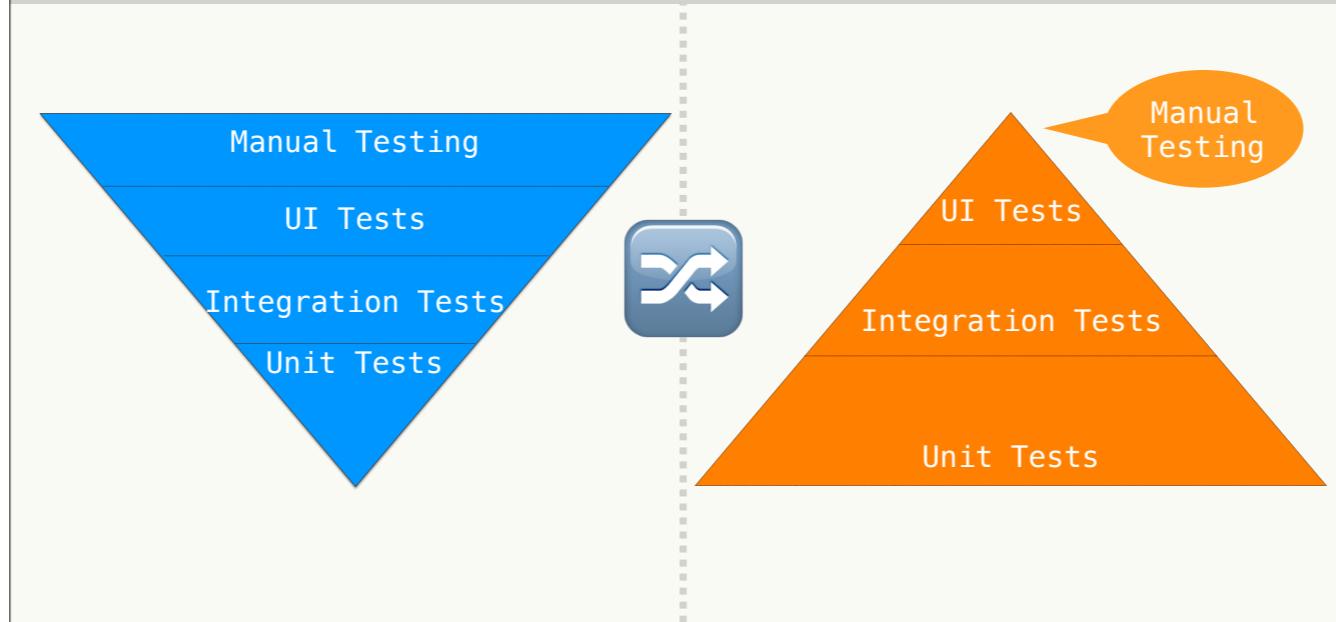
Flipped pyramid make development cycle slow



Checking UIs manually is easier than test automation.
But large manual tests block swift development cycle in the future.

<http://www.utest.com/articles/mobile-test-pyramid>

Flipped again



So, converting manual checking to automated UI tests and increase unit tests is very important.

If you succeed increasing automated UI tests, you can check **automatically** most of the layouts, screen transactions and so on.

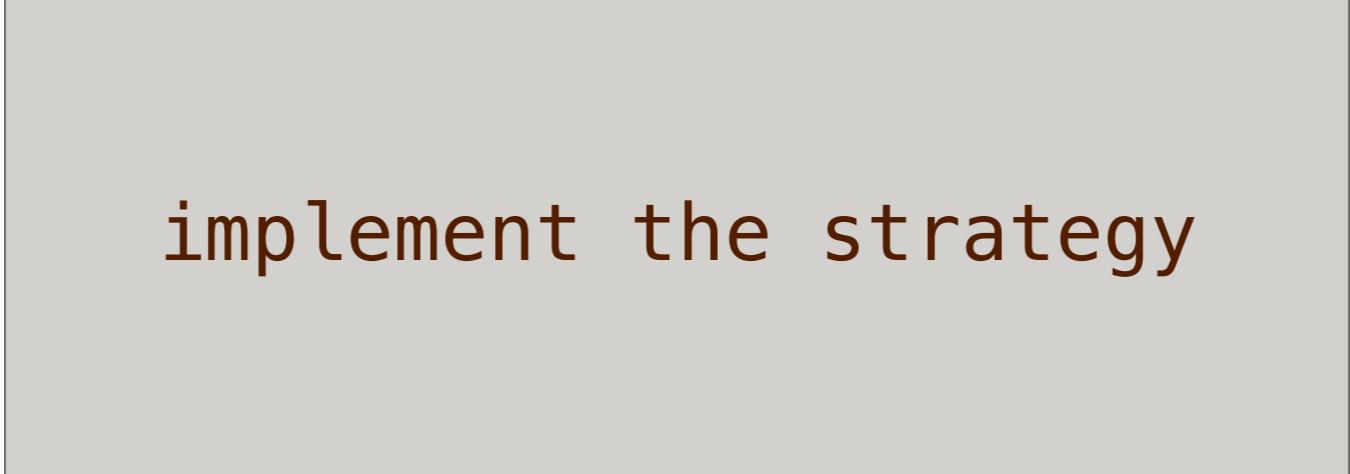
Conduct tests for combinations

	8.0	8.1	8.2	8.3	8.4	9.0	9.1	9.2	9.3	10.0
iPhone	●			●	●	●	●	●	●	●
iPad		●		●			●			
iPad Pro									●	

You can also check combinations of various OS versions and resolutions without additional human resources.

Certainly, designing test architecture for test automation is also important.

Don't convert all manual tests to dirty automated tests.



implement the strategy

--

~ 11:00

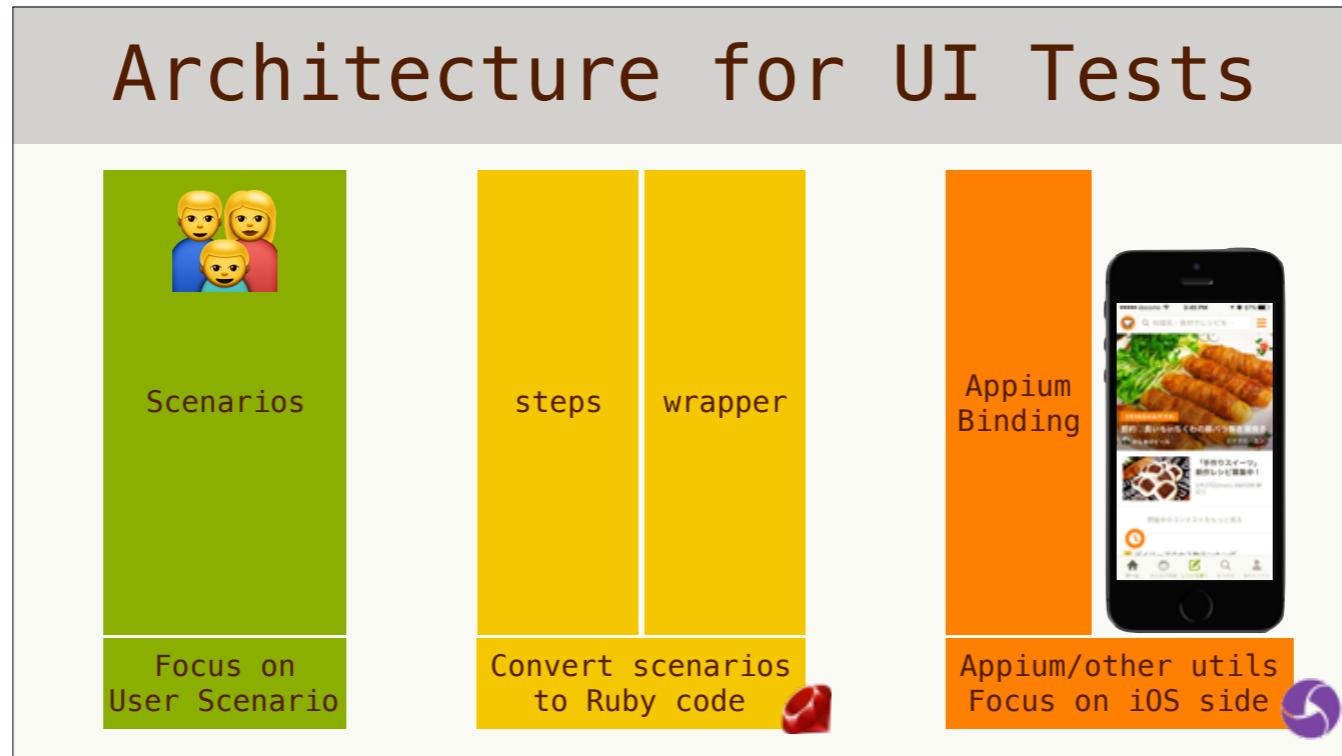
--

Automated UI Test with Appium from 2014



I've tried to implement UI Test since 2014, I've joined cookpad.
While conducting manual tests, I've been working on implementing test automation step by step.

Architecture for UI Tests



Our architecture for the automated UI tests is like this.

We separate some layer to divide responsibilities for scenarios associated with end-users, and product code associated with the iOS framework.

This architecture keeps independence between end-user scenarios and concrete implementations depends on iOS framework.

Thus, we can decrease maintenance costs for scenarios side and iOS side.

I think separating responsibility is familiar to developers.

(<http://www.slideshare.net/KazuMatsu/20141018-selenium-appiumcookpad>)

Scenarios with data-driven testing

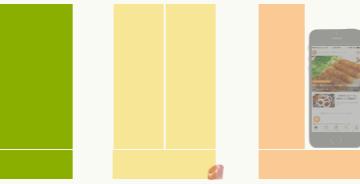
```
feature: A user can search several words via search field
background:
  given Conduct tests with 'iPhone' device

  scenario_outline: Users can see search results

    when I login with <user_status>
    when I search <search_words> via search fields
    then I swipe down '3' times
    then I can see 'xxx' on the display
```

example:

user_status	search_words
'ps'	'sushi'
'non-ps'	'🍣'
'guest'	'sukiyaki'



This is an example to describe scenarios.

I implement scenarios with data-driven testing style with Turnip.

Turnip allow us to describe tests with national languages.

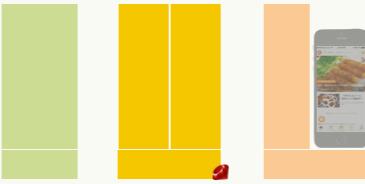
This scenario is close to end-user.

So, I chose the national language to describe the scenarios.

steps/wrapper/bindings

```
step "Conduct tests with :device device" do |device|
  start_device(device)
end

def start_device(device)
  driver_start_with(desired_capabilities: des_app_caps,
                     server_caps: des_server_caps)
  set_location(DEFAULT_LOCATION)
  close_initial_information(device)
end
```



This is steps and wrappers to convert national language to programming language.
This example is implemented by Ruby.

These two layer is foundations for UI tests.

_seasoning

I share some tips to season the tests.

Reduce dependency from internal product code



```
find_element :xpath,  
    "//UIAApplication[1]/UIAWindow[1]/UIATableView[1]/UIATableCell[1]"
```

It is important to describe test scenarios independent of internal logic for production code.
For example.

This `find_elements` method try to find out elements with XPath.

XPath indicate view hierarchy(háI(ə)rà:rki).

But this finding elements will be broken easily if OS version is updated.

Reduce dependency from internal product code



```
find_element :xpath,  
    "//UIAApplication[1]/UIAWindow[1]/UIATableView[1]/UIATableCell[1]"
```

Strongly depends on view structures

Because the path strongly depends on the iOS framework's logic.

Reduce dependency from internal product code



```
find_element :xpath,  
    "//UIAApplication[1]/UIAWindow[1]/UIATableView[1]/UIATableCell[1]"
```



```
find_element :accessibility_id, "an arbitrary identifier"
```

So, use accessibilityIdentifier or label to avoid this kind of cause.

Don't conduct tests for all boundaries in UI Tests

Test for all boundaries is better to implement in the unit test.
For example, validations for text field.

Because UI test is too slower than the unit test.
So, if you implement all boundaries in UI side, it is better to move tests from UI level to method level, and remove it from UI side in the future.

more 

More spice to extend UI tests we've tried.

I just explain about scenarios and its implementations for UI tests.

But test automation is not only describing scenarios but also judging results and reporting it.

image diff

e.g. Broken the order of Allow/Cancel button



for example, we also implement image diff to judge the results and **feedback to designers**.
Designers can know the differences between the previous version and the new version easily.
This helps layout checking.

image diff

e.g. Broken the order of Allow/Cancel button



This example shows the diff on systems alert.

The diff was caused only for iOS8.1 when we refactored some internal code before.

Request counts

```
{  
  "total_request": 484,  
  "requests": {  
    "private_github.com": 27,  
    "private_github.com": 20,  
    "private_github.com": 20,  
    "private_github.com": 20,  
    "private_github.com": 20,  
    "private_github.com": 16,  
    "private_github.com": 16,  
    "private_github.com": 15,  
    "private_github.com": 15,  
    "private_github.com": 12,  
    "private_github.com": 12,  
    "private_github.com": 11,  
    "private_github.com": 8,  
    "private_github.com": 8,  
    "private_github.com": 8,  
    "private_github.com": 5,  
    "private_github.com": 4,  
    "private_github.com": 4,  
    "private_github.com": 4,  
    "private_github.com": 3,  
    "private_github.com": 3,  
    "private_github.com": 3,  
    "private_github.com": 3  
  }  
}
```

And we also capture network traffic for particular scenarios to uncover unexpected requests in them.

Request counts

We can uncover unexpected burst requests and server loads.

Re-Engineering re-write/re-factor without fear for developers

Keeping this kind of UI tests and other spices encourage developers re-write ,refactor and introduce new tests.

If their changes broke some features, UI tests can uncover the errors because our UI tests cover around 80% screen transactions.

introduce Swift



We start to introducing Swift into the app a few months ago.
Near 30% code already implemented by Swift.

Also in this cases, tests help fearless changes for developers in UI/request level.



faster and more stable

Current UI tests are enough for current our development process.

But our team will become bigger than now in the future.

So, I'll introduce alternative tools such as XCUITest or EarlGrey for UI tests partially.

But we don't stop using Appium because their responsibility area and test framework's lifecycle is different.

Conclusion

Quality, Environment and Services change
frequently in Mobile

UI Tests support Re-Engineering



I talked about histories of cookpad app and how we've tried to re-engineering the big app with automated UI tests.
UI tests need to separate responsibility from production code to catch up with some changes painless.

Do you get motivations
to challenge
automated UI Tests? 🙋

Thanks for listening 😊

Thanks for listening my talk.
And our co-workers.