A4 セッション

# モバイルのテスト

～モバイルプラットフォームに屈しないテストとは。～

# 特定モバイルプラットフォームで動作する

# アプリに対する効率的なテストを考える

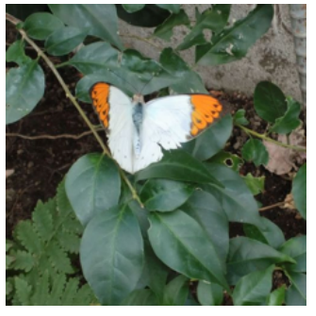
# Keyword

- モバイル
- スマートフォンアプリ
- Android, iOS
- テストの効率化

# 自己紹介

- **氏名**: 藤原 洋平 (FUJIWARA Yohei)
  @_mirer (Twitter)
- **所属**: 株式会社ACCESS
- **業務内容**:
  品質保証業務全般、品質管理
  # 組み込み系、Web・Service系、モバイル系
- **プライベート**:
  - WACATE 実行委員　http://wacate.jp
    - 2016 夏を6月に開催予定！コミュニティブース出店中
  - JaSST Tokyo 実行委員

# 自己紹介



- **氏名**: 外山 純生 (TOYAMA Sumio)
  @sumio_tym (Twitter) / @sumio (GitHub)
- **所属**: NTTソフトウェア株式会社
- **業務内容**:
  社内Android関連プロジェクトの技術支援
- **プライベート**:
  - STAR(テスト自動化研究会)
  - @IT連載「スマホ向け無料システムテスト自動化ツール」uiautomator/Appiumの回について記事を書きました。
    http://www.atmarkit.co.jp/ait/kw/smapho_testtool.html

# 自己紹介

- **氏名**: 長谷川 孝二 (HASEGAWA Koji)
  @nowsprinting (Twitter/GitHub/Hatena Blog)
- **所属**: フリーランス
- **業務内容**:
  iOS/Androidアプリ受託開発。最近はVRも。
- **プライベート**:
  - App『山吹色の茸疾走』『電エースQuiz』ほか
  - 著書『iOSアプリテスト自動化入門』ほか
  - 出演『電エース』シリーズ
  - STAR(テスト自動化研究会)、IPA(ITプロレス協会)

# モバイルの
# テストの効率化

# 背景：環境の変化

- スマートフォンの高性能化・より多機能化
- より複雑な仕様要求
- ビジネススピードの加速
  - 小型案件の増加
- 端末・OS による”分断”

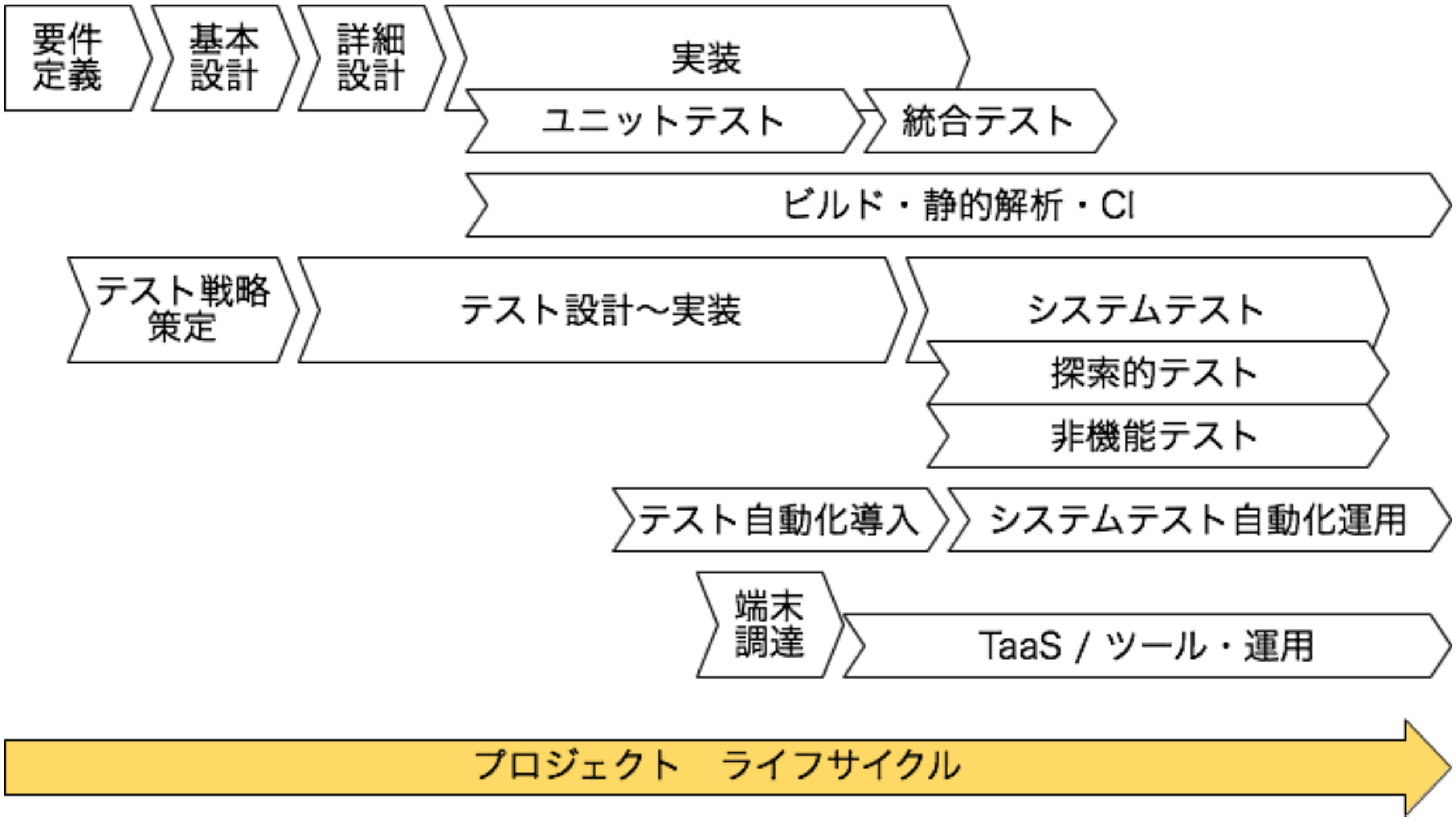
# 効率化が必要なパターン

- イテレーション（リリース間隔）が短い
  - 数日で出荷判定しなければいけない
- 期間はあるけど評価が大変

# 効率化が必要なパターン

- 本セッションでは**開発**と**QA**の視点それぞれから、
プロジェクト全体を通して

効率化できるポイントを紹介していきます


要件定義
基本設計
詳細設計
実装
ユニットテスト
統合テスト
ビルド・静的解析・CI
テスト戦略策定
テスト設計〜実装
システムテスト
探索的テスト
非機能テスト
テスト自動化導入
システムテスト自動化運用
端末調達
TaaS / ツール・運用
プロジェクト　ライフサイクル

# テスト戦略・計画
# （プロジェクト計画）

# テスト戦略

- 参考：

  JaSST'10 Hokkaido 基調講演

  湯本 剛「現場の力をメキメキ引き出すテスト戦略」

  http://jasst.jp/archives/jasst10s.html#keynote

- >「テスト戦略」は、テストをどう行なえば
  最も効率的で効果が高いテストが出来るかを示すもの

- 参考：ISO/IEC/IEEE 29119

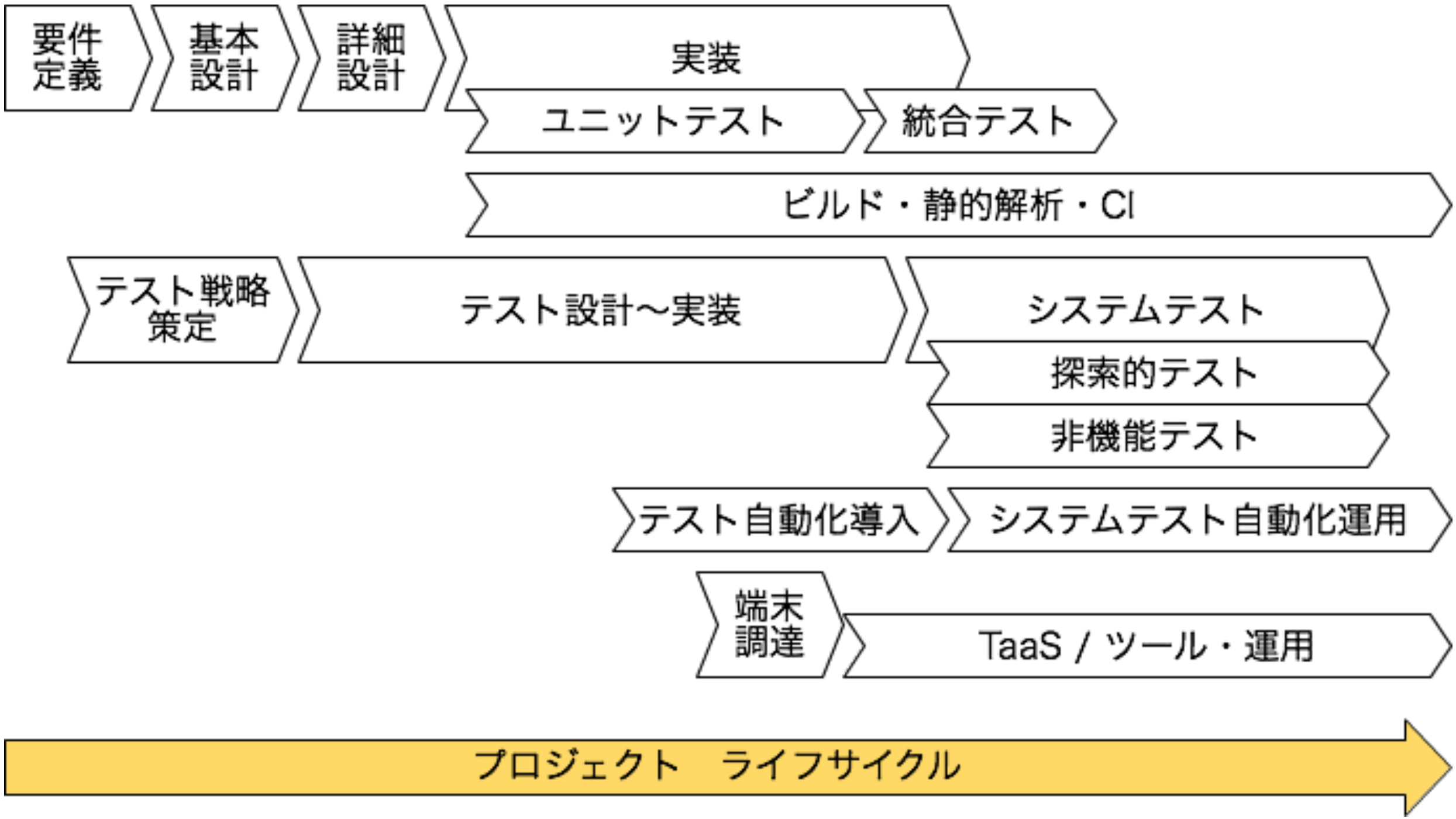
# テスト戦略

- >テスト戦略を立てるときに“人員が足りない”とか“納期が短 い”ということを出発点にしていませんか?
- >自分たちの力を最大限発揮するのに大事なことは、二つ
  - 本当にやりたいこと、やるべきことをテスト戦略に反映する
  - テスト戦略をテスト設計へつなぐ

# テスト戦略

- 考慮すべきことは、本当に沢山ある
- ここでは、ほんの一部の”要素”について紹介
  - 開発から
    - ユニットテスト自動化
    - Integration Testig, CI, CD（アプリ配布）
  - QAから
    - 探索的テスト
    - システムテスト自動化
    - 端末の調達、TaaS

# ユニットテスト


要件定義
基本設計
詳細設計
実装
ユニットテスト
統合テスト
ビルド・静的解析・CI
テスト戦略策定
テスト設計〜実装
システムテスト
探索的テスト
非機能テスト
テスト自動化導入
システムテスト自動化運用
端末調達
TaaS / ツール・運用
プロジェクト　ライフサイクル

# ユニットテストの位置付け

- 開発者が安心するためのテスト
  - プロダクトがデグレードしないようにするための
    セーフティーネット
  - 品質向上には寄与するが、品質保証が目的ではない
  - プロダクトコードを書きながらテストコードも書く
    **(プロダクトコード後にテストを書くのは難しい)**

# ユニットテストを書く文化の重要性

- 実行・確認の自動化の**容易さ**、実行**速度**:
  **ユニットテスト ＞ システムテスト**
- ユニットテストのカバー範囲を
  増やせば、それだけ後続のテストの
  負担が減る
- 回帰テストを確実、高速に実行できる
  (更新頻度の高いスマホアプリではより重要)

# ユニットテストを書くときの悩み (Android)

**Android上でのユニットテスト実行が遅い**
(エミュレータが遅い・実機転送が遅い)

- 開発者は実装中に何度もユニットテストを実行する
- 実行に時間がかかると開発のリズムが失われる
  (**生産性の低下に直結する**)

# 効率化のポイント(Android)

Androidで利用できるテスト実行方法

| | Instrumented<br>Unit Test | Local Unit Test |
|---|---|---|
| 実行環境 | 実機・エミュレータ | PC(JVM) |
| 実行スピード | 遅い | 速い |
| Android API呼び出し | 可 | 不可 |

# 効率化のポイント(Android)

Androidで利用できるテスト実行方法

| | Instrumented Unit Test | Local Unit Test |
|---|---|---|
| 実行環境 | 実機・エミュレータ | PC(JVM) |
| 実行スピード | 遅い | 速い |
| Android API呼び出し | 可 | 不可 |

- **実機を使う**
- **高速な商用エミュレータ「Genymotion」を使う**

  https://www.genymotion.com/

# 効率化のポイント(Android)

Androidで利用できるテスト実行方法

| | Instrumented Unit Test | Local Unit Test |
|---|---|---|
| 実行環境 | 実機・エミュレータ | PC(JVM) |
| 実行スピード | 遅い | **速い** |
| Android API呼び出し | 可 | 不可 |

**実行スピードの速い「Local Unit Test」の仕組みを使う**

# 効率化のポイント(Android)

ツールを駆使して**Local Unit Testで出来る範囲を増やす**

- **Mockito・PowerMock**など (汎用モックライブラリ)
  - Android APIのモック
  - エラー発生時のテスト
  - etc.

- **Robolectric** (JVM上でAndroid APIの一部を模倣)
  - UI部品の表示文字列を確認するテスト
  - データベース(SQLite)アクセスのテスト
  - トースト表示文字列を確認するテスト
  - ログ(logcat)出力内容を確認するテスト
  - etc.

# ユニットテストのツール紹介
# (Android)

# 汎用モックライブラリ

## Mockito

- http://mockito.org/
- 基本機能を提供

## PowerMock

- https://github.com/jayway/powermock
- Mockitoでは不可能なモックを生成できる
- Robolectricと併用する場合は以下のURL参照

  https://github.com/robolectric/robolectric/wiki/Using-PowerMock

# Robolectric

- http://robolectric.org/
- Android APIの一部をJVMで模倣
- 以下の確認程度であれば問題なし
  - UI部品の属性(色・テキストなど)の設定・取得
  - データベース(SQLite)アクセス
  - トースト表示文字列の確認
  - Logcat出力内容の確認
  - スケジューラ操作
- 内蔵モックHTTPサーバ機能はメンテされていないため、**MockWebServer**などのモックサーバと併用する
  https://github.com/robolectric/robolectric/issues/1156

# MockWebServer

- https://github.com/square/okhttp/tree/master/mockwebserver
- HTTPクライアントのテストに便利なWebサーバ
  - **テスト実行中専用**のHTTPサーバが立てられる
    - テストコード(テスト実行プロセス)から、
      ローカル(同一プロセス内)に、
      HTTPサーバを起動・終了できる
  - テスト内容に合った**任意のレスポンス**を返せる

# ユニットテストのツール紹介
# (iOS)

# モックライブラリ (iOS)

- Objective-C
- OCMock
- OCMockito
- Kiwi（BDDフレームワーク）に同梱

# モックライブラリ(iOS)

- Swiftでは言語仕様によりモックの実現に制約があるため、成熟したモックライブラリはまだない
- Swift（有望そうなもの）
    - Cuckoo
    - Dobby
    - MockFive
    - SwiftMock

# HTTPスタブサーバ (iOS)

- OHHTTPStubs (Obj-C/Swift)
- NLTHTTPStubServer (Obj-C)

# Spec BDD Framework(iOS)

- Kiwi (Obj-C)
- Quick (Swift)
- Spectre (Swift)

# Matcher Framework(iOS)

- OCHamcrest (Obj-C)
- Nimble (Swift)

# コードカバレッジ

# コードカバレッジ

- テストによって実行されるプロダクトコードのカバー率。
- ステートメントカバレッジ、デシジョンカバレッジ、条件カバレッジなど。
  ここではステートメントカバレッジについて。

- 高いカバレッジを目標にしてしまうと、無理・無駄なテストコードを生む。指標程度に利用する

# カバレッジ採取(Android)

- **IDEからの実行**: 「Run＞Run … with Coverage」
  - Instrumented/Local Unit Test両方で取得可能
- **CLIからの実行**:
  - Instrumented Unit Test: 標準でサポート
    - build.gradleでJaCoCoの有効化が必要
    - 「testCoverageEnabled = true」
  - Local Unit Test: Gradleに独自タスクを定義
    - http://stackoverflow.com/a/28648632/2925059 の後半参照
      ※「testDebug」を「testDebugUnitTest」に置換して使う(2箇所)

# カバレッジ採取(iOS)

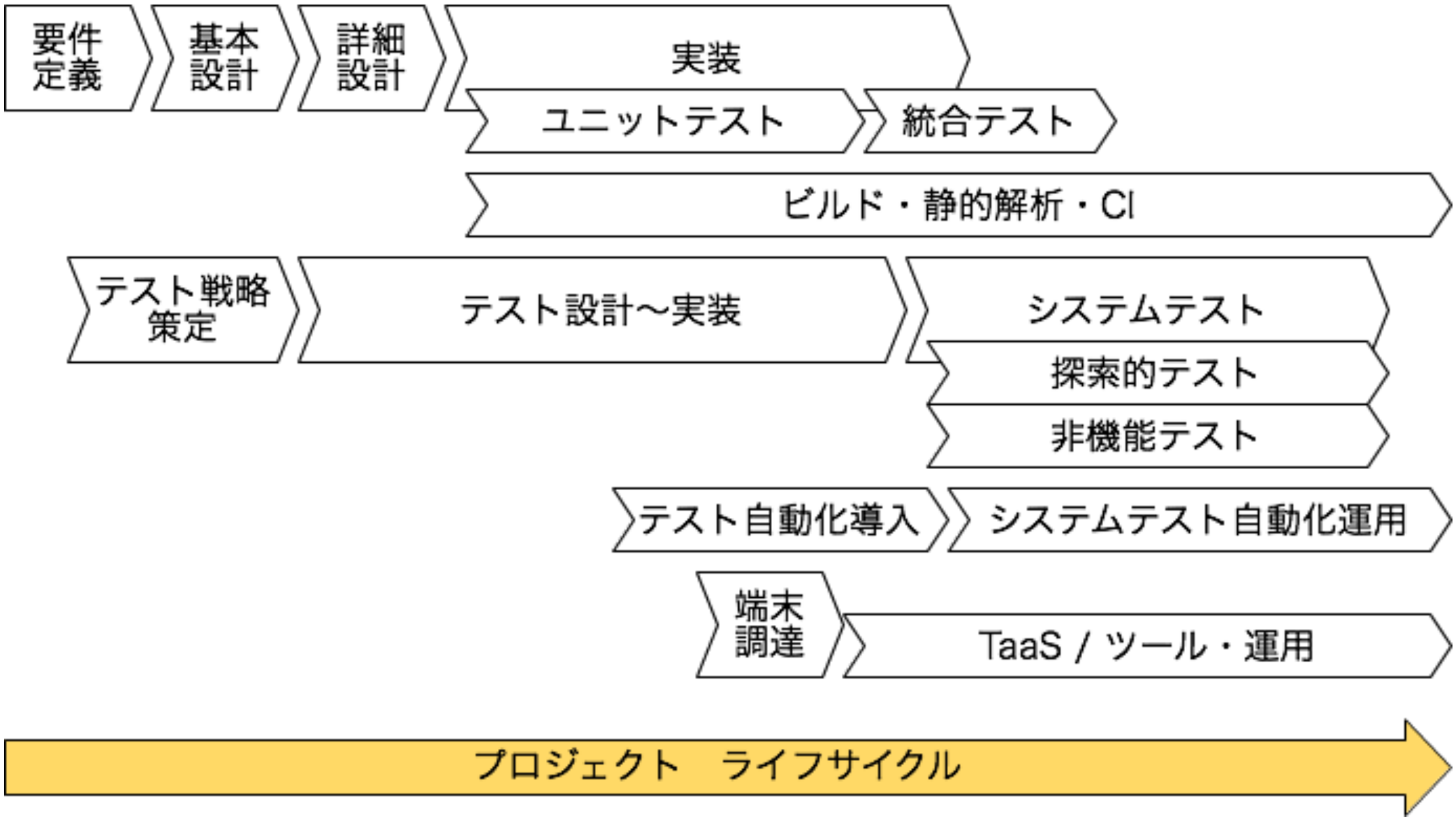
- スキームのTest Action設定で”Code Coverage/Gather coverage data”をonにする

- Xcode（IDE）のテストログビューに表示される

- OS X Server（CI）では”Coverage”タブに表示。75%で”Good”表示

- JenkinsのCoberturaプラグインで表示するには、gcovrを使用してフォーマット変換が必要

# 静的解析ツール


要件定義
基本設計
詳細設計
実装
ユニットテスト
統合テスト
ビルド・静的解析・CI
テスト戦略策定
テスト設計〜実装
システムテスト
探索的テスト
非機能テスト
テスト自動化導入
システムテスト自動化運用
端末調達
TaaS / ツール・運用
プロジェクト　ライフサイクル

# 静的解析ツールでできること

- コーディング規約違反の検出
- うっかり書いてしまいがちな潜在バグの検出
  (例: メモリリークの可能性)

静的解析ツールの指摘に対応しておけば、

**内部品質が向上する。**

# 静的解析ツール紹介
# (Android)

# 代表的な静的解析ツール(Android)

- Android Lint
- Android StudioのInspection機能

# Android Lint

- http://developer.android.com/intl/ja/tools/help/lint.html
- **Android特有の問題**を発見してくれる
  - 具体例は以下の発表資料に詳しい:

    中川幸哉 (@Nkzn)
    「Android Lintで正しさを学ぼう」at DroidKaigi 2016
    http://www.slideshare.net/Nkzn/androidlint-droidkaigi

# Android StudioのInspection機能

- https://www.jetbrains.com/idea/help/code-inspection.html
- Android Studio (IntelliJ IDEA)の**標準機能**
- コーディング中にリアルタイムに指摘
- **Javaのプログラミングで一般的な問題**の
  ほとんど(コーディングスタイル含む)をカバー
  - 具体例は以下の記事に詳しい:
    「Android Studio最速入門
    ～効率的にコーディングするための使い方
    第14回　エディタの話［その6］―インスペクション」
    http://gihyo.jp/dev/serial/01/android_studio/0014

# 代表的な静的解析ツール(Java)

- CheckStyle
- FindBugs

※「Android StudioのInspection機能」と
**検出範囲はほぼ重複**

# CheckStyle

- https://github.com/checkstyle/checkstyle
- 主に**コーディング規約上の問題**を指摘
- Android Studioで使うには**プラグインが必要**

# FindBugs

- http://findbugs.sourceforge.net/
- コンパイル後の**Javaバイトコードを解析**
- 主にJavaプログラムにおける**潜在バグ**を指摘
- Android Studioで使うには**プラグインが必要**

# CLIで実行するとき(1/2)
# (Android)

GradleタスクとしてCLIで実行できるもの

- Android Lint
- CheckStyle (※)
- FindBugs(※)

  ※追加Gradleプラグインが必要

  https://github.com/noveogroup/android-check

# CLIで実行するとき(2/2)
# (Android)

「Android StudioのInspection機能」もCLIは存在

https://www.jetbrains.com/ruby/help/running-inspections-offline.html

- **別環境(git clone直後)での実行が困難**
  - リポジトリに含め辛いIDE設定ファイル(*.iml)が必要
  - CLIで*.imlを生成する方法が無い
    (GUI版のAndroid Studioを起動→import操作が必要)
    http://stackoverflow.com/a/34635354
    https://code.google.com/p/android/issues/detail?id=181677

# 静的解析ツールの使いわけ (Android)

**IDE上での検出が目的なら以下の2つで十分**

- Android Lint
- Android StudioのInspection機能

**別環境(CIサーバーなど)でCLIから実行したい場合は…**

- Android Lint
- CheckStyle
- FindBugs

※IDE上ではAndroid StudioのInspectionも併用OK
(ただし重複が多い)

# 静的解析ツール紹介

# (iOS)

# 静的解析 (iOS)

- 解放済みメモリへのアクセスなど、実行時にエラーとなる箇所を検出
- Xcodeの”Analyze”機能
  - メニューから [Product]-[Analyze]
  - xcodebuildからも”analyze”アクションで実行できるが、ビルドエラーにはならない
  - X Serverではビルド結果とともに表示される

# インスペクション (iOS)

- コードの内部品質を測定。可読性の低いコード、サイクロマティック複雑度など

- OCLint (Obj-C)

- SwiftLint (Swift)

# 内部品質が低いコードへの対応

## テストコードの無いプロダクトコードへの対応

- 後からユニットテストを書くのはとても大変
- **新規機能追加/改造部分のテスト**を書くところから始める

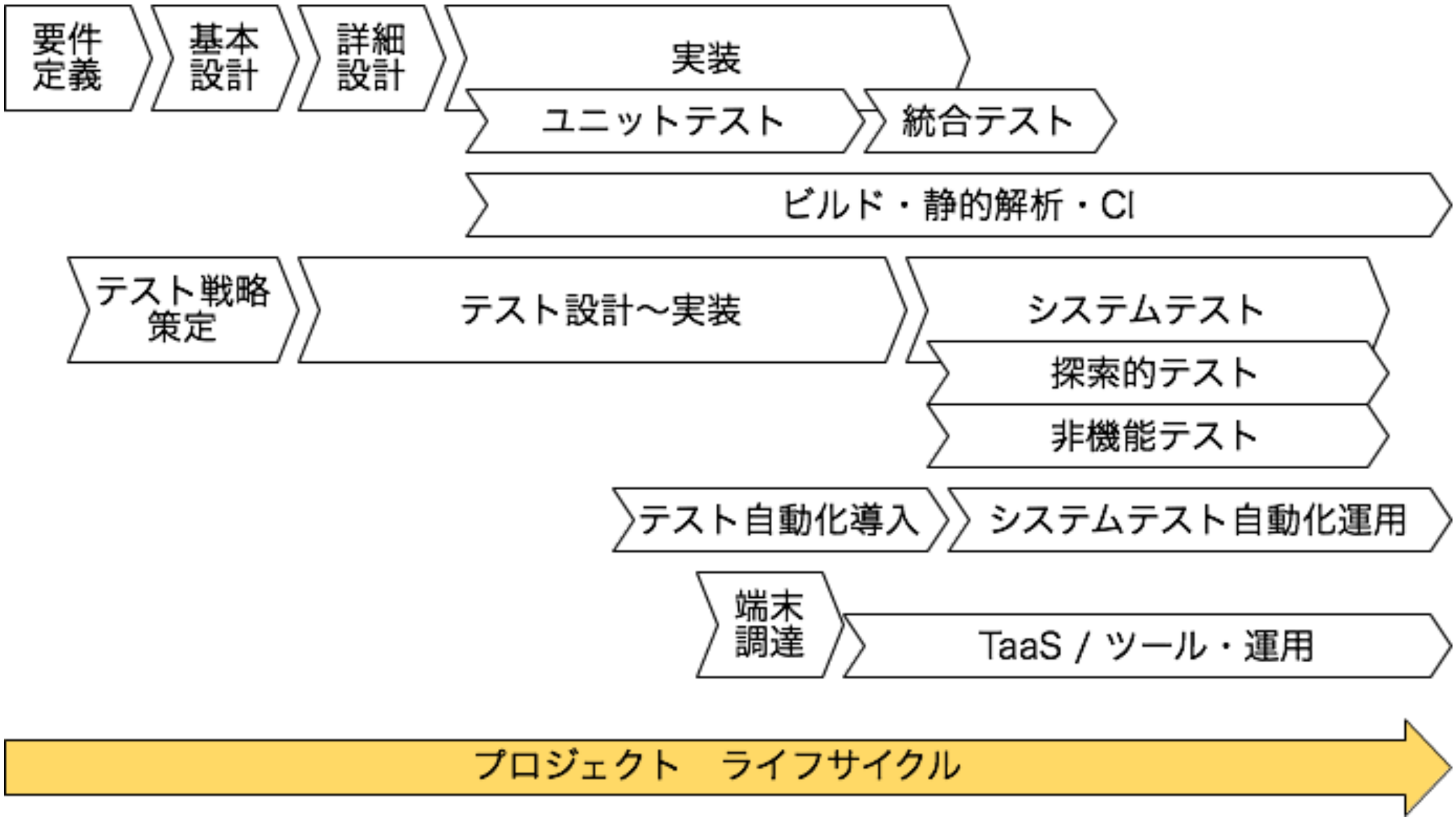
## 静的解析ツールからの大量の指摘への対応

- **重要度の高い指摘**への対応を優先する
  (無視して良い指摘は**チェック対象から外す**)
- ユニットテストが無い箇所を修正する場合は注意が必要
  - ユニットテストを書いてから修正する
  - IDEの自動修正機能を使う

## テストのためにプロダクトコードを書きかえるとき

- IDEの**リファクタ機能**を最大限に使う
  (極力手で書き変えない)

# 統合〜システムテスト
# の自動化


要件定義
基本設計
詳細設計
実装
ユニットテスト
統合テスト
ビルド・静的解析・CI
テスト戦略策定
テスト設計〜実装
システムテスト
探索的テスト
非機能テスト
テスト自動化導入
システムテスト自動化運用
端末調達
TaaS / ツール・運用
プロジェクト　ライフサイクル

# 統合テスト／システムテスト

- ここで「システムテスト」は、自動化する範囲（機能テストなど）を指します

- 「UIのテスト」「End to Endテスト」などと呼び、まとめて語られがち（特に開発者側から）

- 厳密に（工程として）分ける必要はないが、目的を考えてツールを使い分けたい

# テストレベル

- **システムテスト**
  - ビルドされたapk/ipaを使用する
  - ステージング以上の環境（ISTQBの定義による）
- **統合テスト**（結合テスト）
  - 開発者テストの範疇
  - モジュールの統合、プラットフォームとの統合

# テストレベル

- **システムテスト**
  - ビルドされ
  - ステージング以上の環境（ISTQBの定義による）

- **E2Eと呼べるのはこちら**
- **レスポンスに依存する→自動化困難**

- **統合テスト**（結合テスト）
  - 開発者テストの範疇
  - モジュールの統合、プラットフォームとの統合

# テストレベル

- **システムテスト**
  - ビルドされた
  - ステージング以上の環境（ISTQBの定義による）
- **統合テスト**（結合テスト）
  - 開発者テストの範
  - モジュールの統合、プラットフォームとの統合

・E2Eと呼べるのはこちら
・レスポンスに依存する→自動化困難

サーバ等をモック化する余地はある

# Hermetic Servers

- “密閉された”サーバ環境。バックエンドサーバへの通信を行わず、モックサーバに置き換える

- http://www.infoq.com/jp/news/2015/04/android-ui-testing

- http://googletesting.blogspot.jp/2012/10/hermetic-servers.html

# 他の自動化困難要因

- 日時、乱数、GPS、加速度センサなど
- DI（Dependency Injection: 依存性の注入）を使ってモック化する
  - Android: Dagger 2
  - iOS: Typhoon(Obj-C/Swift)、Swinject(Swift)

# 統合テストツール

- Android:  Espresso、UI Automator、Robotium
- iOS:  UI Testing Bundle、EarlGrey 、KIF
- いずれもプラットフォーム標準のテスト実行環境で動作し、Gradle/xcodebuildから実行可能

# システムテストの自動化

- でも、システムテスト（レベルの結合度）は必要
  - 統合テストでモックした範囲をカバー
  - ProGuard
  - ビルドのミス
- スモークテストの範囲で自動化
- (Scenario)BDD

# システムテストツール

- Android: Appium
- iOS: UIAutomation, Appium
- BDD: Calabash, Turnip

# システムテスト自動化

# システムテスト自動化

- 参考：

  JaSST’16 Tokyo

  太田 健一郎、永松 康能

  「いつどこをテスト自動化すべきか？」

＜一部抜粋＞

- テスト自動化の目的を明確にする
- 本当にテスト自動化が最善の策か利害関係者と確認
- 現在の手動テストで改善できないか確認
- トライアルの前に、利用するツールに慣れておく

# システムテスト自動化

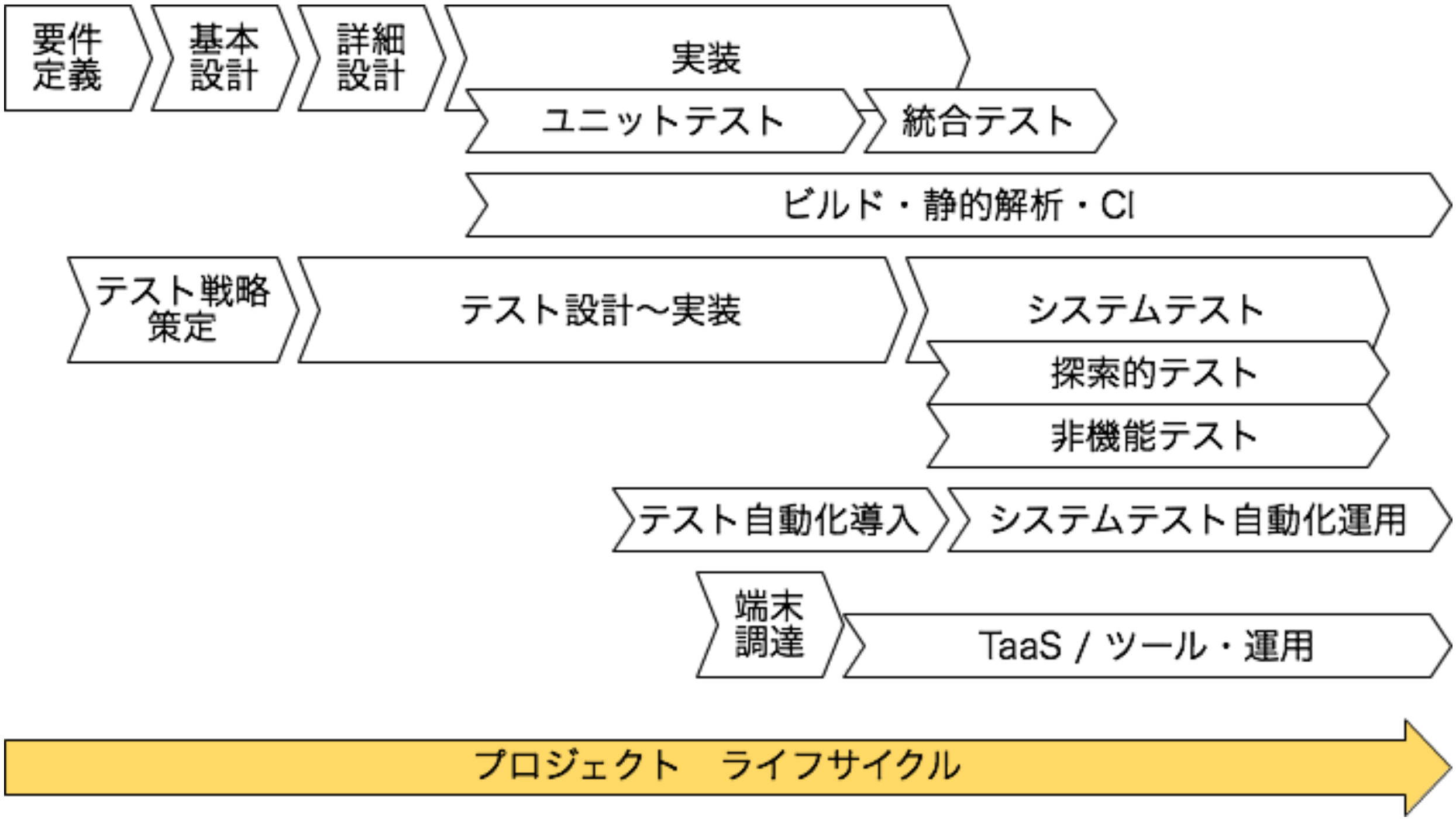
- 参考：第1回システムテスト自動化カンファレンス
  長谷川 孝二
  「スマートフォンアプリの
  テスト自動化をはじめよう」
  http://www.slideshare.net/nowsprinting/starcon2013-mobile-testautomationkeynote6

- 欲張らないROI
- 頑張らないテストスクリプト

# システムテスト自動化

- QAがいる場合の戦略：
  - テスト自動化の目的を明確にする
    - 自動化以前の問題・解決策がないか考える
    - 作成工数・運用工数・保守工数を検討する
      - 専任のひとをアサインする
    - 最初は効果が高い／できるところから着手

# ビルドの自動化


要件定義
基本設計
詳細設計
実装
ユニットテスト
統合テスト
ビルド・静的解析・CI
テスト戦略策定
テスト設計〜実装
システムテスト
探索的テスト
非機能テスト
テスト自動化導入
システムテスト自動化運用
端末調達
TaaS / ツール・運用
プロジェクト　ライフサイクル

# 手作業(GUI)でビルド？

- 異なる設定でビルドしてしまうオペミス
  - ステージングと通信するアプリをリリース
  - 無料版／有料版を取り違えてリリース
- 不具合が修正されたビルドを早く再テストしたい
  - ビルドが誰か（のマシン）に依存している

# コマンドラインでのビルド

- プラットフォーム標準の(CLI)ビルドツール
  - Androidでは：Gradle
  - iOSでは：xcodebuild
    （依存関係の解決に、CocoaPods, Carthage）
- テスト実行、ipa/apkへの署名まで可能

# スクリプトでラップ

- コマンドオプションが多いので、Make, Rake, Pythonなどのスクリプト言語でラップする
- CIツール（後述）の設定に直書きしないのが望ましい
  - Jenkinsの設定はリポジトリ管理外
  - .travis.ymlなど、手元のマシンで実行できない
- Walter（OSSのビルドパイプラインツール）

# CIの利用

- CI（Continuous Integration/継続的インテグレーション）

- リポジトリに変更をチェックインした契機で、自動的にビルド・テストを実行する

- Jenkinsでは、毎日や毎時などの定期実行も可能（UAT配布、時間のかかるシステムテスト等）

# CIツールの選定

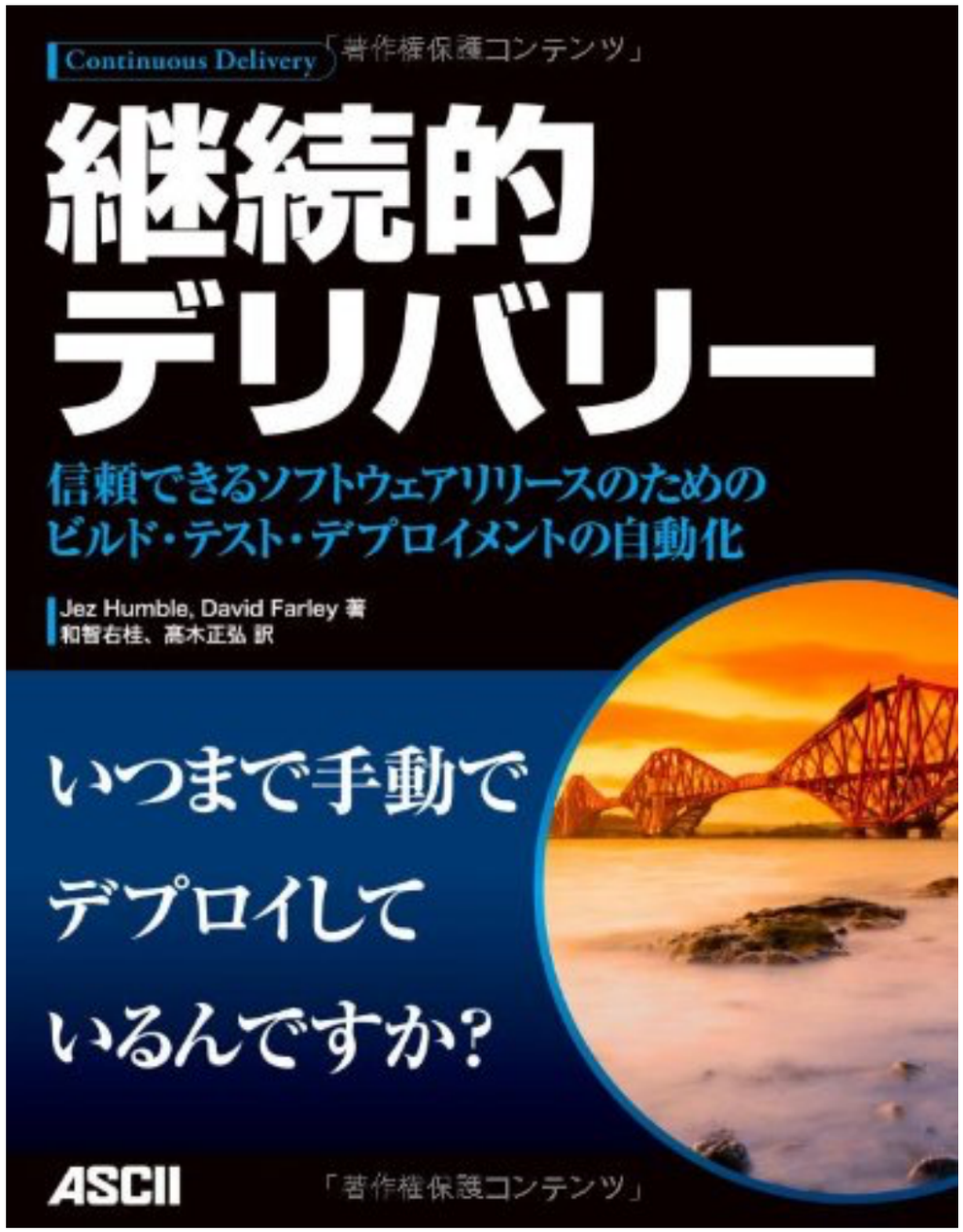
- ビルドマシンにインストール:
  Jenkins, OS X Server/Bots
- CI as a Service:
  Travis CI, Circle CI, CloudBees, etc……
  ※他にも存在するが、iOSに対応していないものもあるので注意

# テスターへの配布

- テスターやUATを実施する顧客にアプリを配信するサービスを利用し、ビルドの延長で自動実行
  - Beta by Crashlytics(Fabric)
  - DeployGate

# 参考文献 (1/2)


継続的
インテグレーション入門
Continuous Integration
開発プロセスを自動化する47の作法
ポール・M・デュバル、スティーブ・M・マティアス、アンドリュー・グローバー 著
ウルシステムズ 大塚雅史、丸山大輔、岡本裕二、亀村圭助 訳
日経BP社
「自動ビルド」(コンパイル、テスト、インスペクション、デプロイ)で
ソフトウエアを常に
正しく動作する状態に保つ
アジャイル開発を実践する必須プラクティス
マーチン・ファウラー絶賛! 2007年Jolt Awards大賞 Technical Books 受賞
日経BP社 定価:[本体3200円+税]

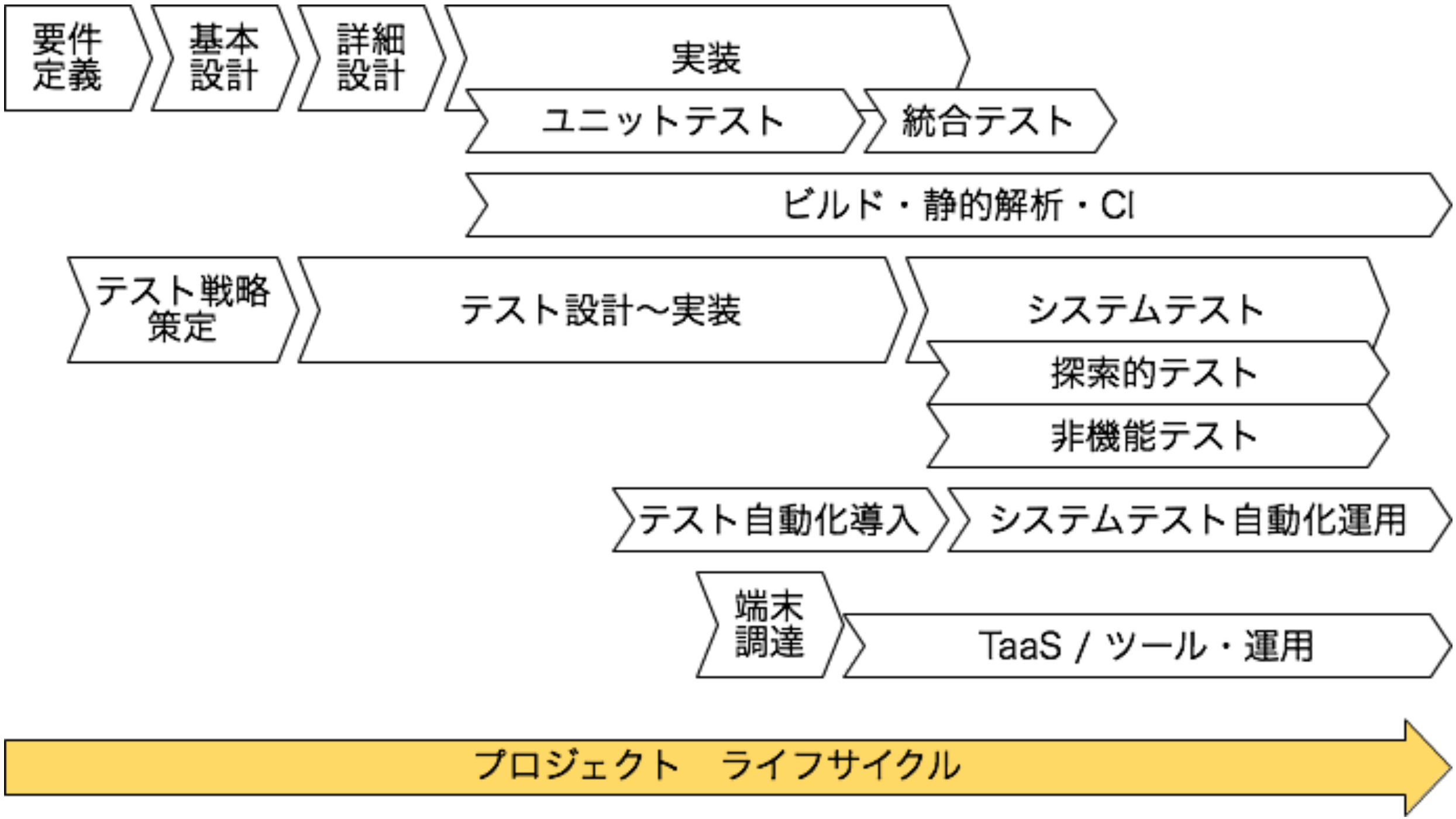

Continuous Delivery
「著作権保護コンテンツ」
継続的
デリバリー
信頼できるソフトウェアリリースのための
ビルド・テスト・デプロイメントの自動化
Jez Humble, David Farley 著
和智右桂、高木正弘 訳
いつまで手動で
デプロイして
いるんですか?
ASCII
「著作権保護コンテンツ」

# 参考文献 (2/2)


CodeZine
BOOKS
システムテスト自動化
標準ガイド
Mark Fewster、Dorothy Graham 著
テスト自動化研究会 訳
退屈なテストにさようなら!
システムテストを自動化するプロセスを
技術・組織の両面から実践的に解説
Seleniumなどの各種ツールを使った
システムテスト自動化の事例も収録
SE


iOS APPS TEST AUTOMATION MANUAL
iOSアプリ
テスト自動化
入門
長谷川孝二
KOJI HASEGAWA
TEST
「本気」で覚悟を決めた開発者へ!
「楽をするために」「開発に専念するために」
あなたを
サポートする
便利ツール!
●ユニットテスト=XCTest/GHUnit/Kiwi
●ユニットテスト補助=OCHamcrest/OCMock/OCMockito/NLTHTTPStubServer
●システムテスト=UI Automation/Frank/MonkeyTalk
●補助ツール=KIF(Keep It Functional)/Appium/Calabash/Zucchini
●ビルドと配布=xcodebuild/xctool/TestFlight
●CI=OS X Server+Bots/Jenkins/Travis CI
秀和システム

# テスト対象OS/機種の選定と調達


要件定義
基本設計
詳細設計
実装
ユニットテスト
統合テスト
ビルド・静的解析・CI
テスト戦略策定
テスト設計〜実装
システムテスト
探索的テスト
非機能テスト
テスト自動化導入
システムテスト自動化運用
端末調達
TaaS / ツール・運用
プロジェクト　ライフサイクル

# OSバージョンの差異

- UI部品の見た目・構造
  表示が崩れるだけでなく、コードによってはクラッシュする（部品の内部構造を触っている場合）

- APIの振る舞い

- Androidのパミッション（6.0からランタイム）

# OSバージョンの選定

- メジャーバージョンは網羅する。特定のマイナーバージョンやリビジョンに存在する不具合については個別対応（備えても仕方ない）

- UIのテストは、一通りの画面・Alertを表示させることで、最低限クラッシュは検知できる

# 機種の差異(1/2)

- 昔ほど機種依存、特に「機種固有のバグ」は無い
- 画面解像度（数パターンしか無い）、CPU（チップセット）が中心。
  アプリによっては、メモリ、カメラの差異。
- メーカー固有のUI部品（シャープ等）

# 機種の差異(2/2)

- Androidのシェアカバー率
  - 30機種でシェア率55.5%
  - 50機種で72%
  - 100機種で99%

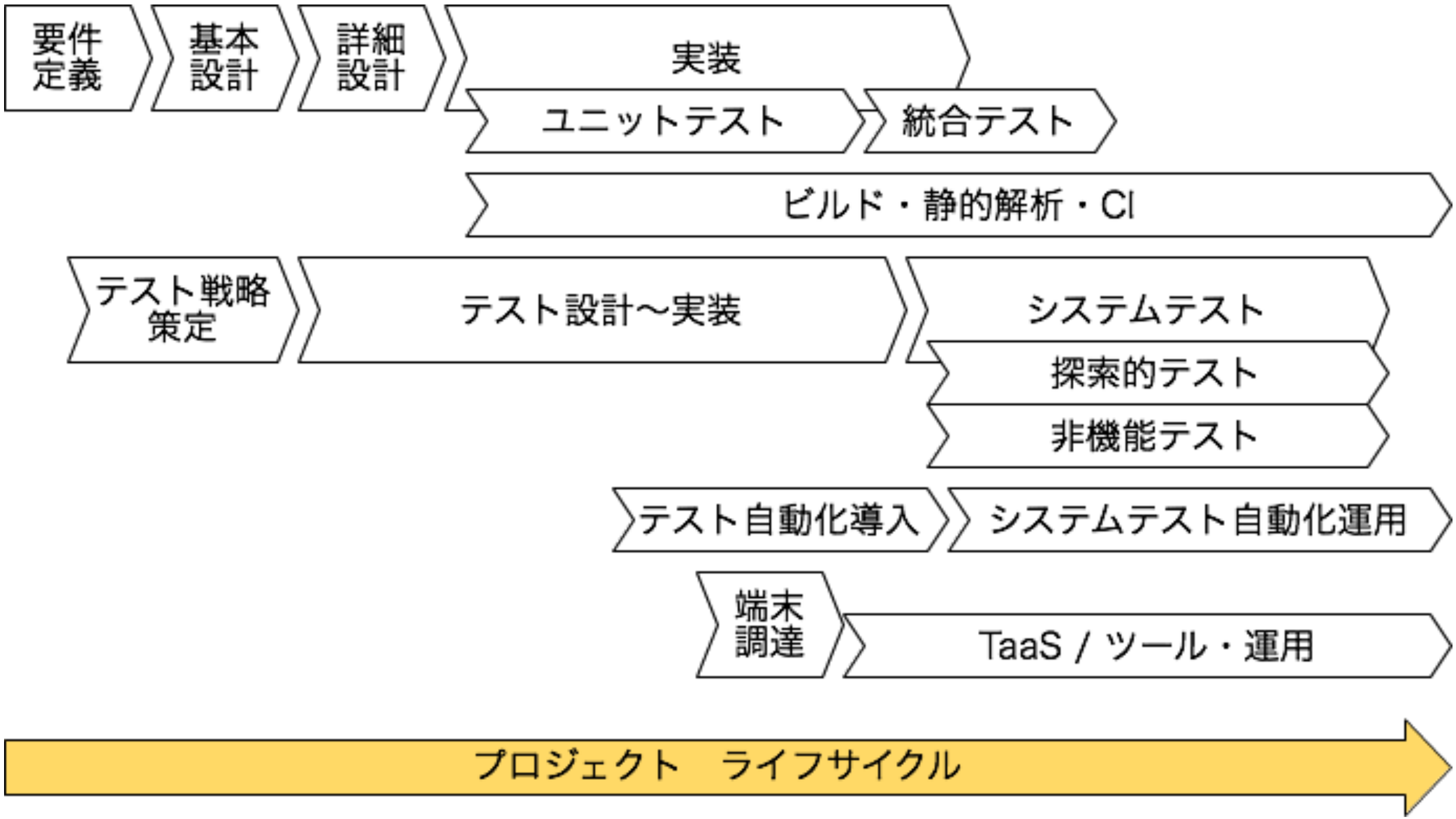
※出典：GREE Tech Talk #4 (2013/12)

# 機種の選定

- テスト目的（どのような問題を検知したいのか）によって要素を絞り込む

- 解像度、チップセットなどの要素から、ペアワイズ法でテスト対象機種の要件を決めて機種選定

# Testing as a Service


要件定義
基本設計
詳細設計
実装
ユニットテスト
統合テスト
ビルド・静的解析・CI
テスト戦略策定
テスト設計〜実装
システムテスト
探索的テスト
非機能テスト
テスト自動化導入
システムテスト自動化運用
端末調達
TaaS / ツール・運用
プロジェクト　ライフサイクル

# Testing as a Service

- クラウド上にある環境を利用してテストを行う
- ここ数年でサービスが増えてきた印象
  - Amazon: AWS Device Farm
    https://aws.amazon.com/jp/device-farm/
  - SAUCE LABS https://saucelabs.com/
  - Scirocco Cloud http://www.scirocco-cloud.com
  - ほんの一例

# Testing as a Service

- 端末の準備が不要、機種が多い
- Appium などによる、自動化に対応
- CI 連携
- テスト結果やログの可視化など

# Testing as a Service

- 相応の利用料金
- システムテスト自動化などの準備ができていないと費用対効果は落ちる
- 端末が用意できない環境では有用

# Smartphone Test Farm

- CyberAgent が開発した Android 遠隔操作ツール
- オープンソースとして公開
  http://openstf.io

# Smartphone Test Farm

- STF をインストールしたマシンに Android 実機をつないでおく
- 他のマシン（クライアント）からブラウザで接続
- ブラウザ上から、実機を操作できる

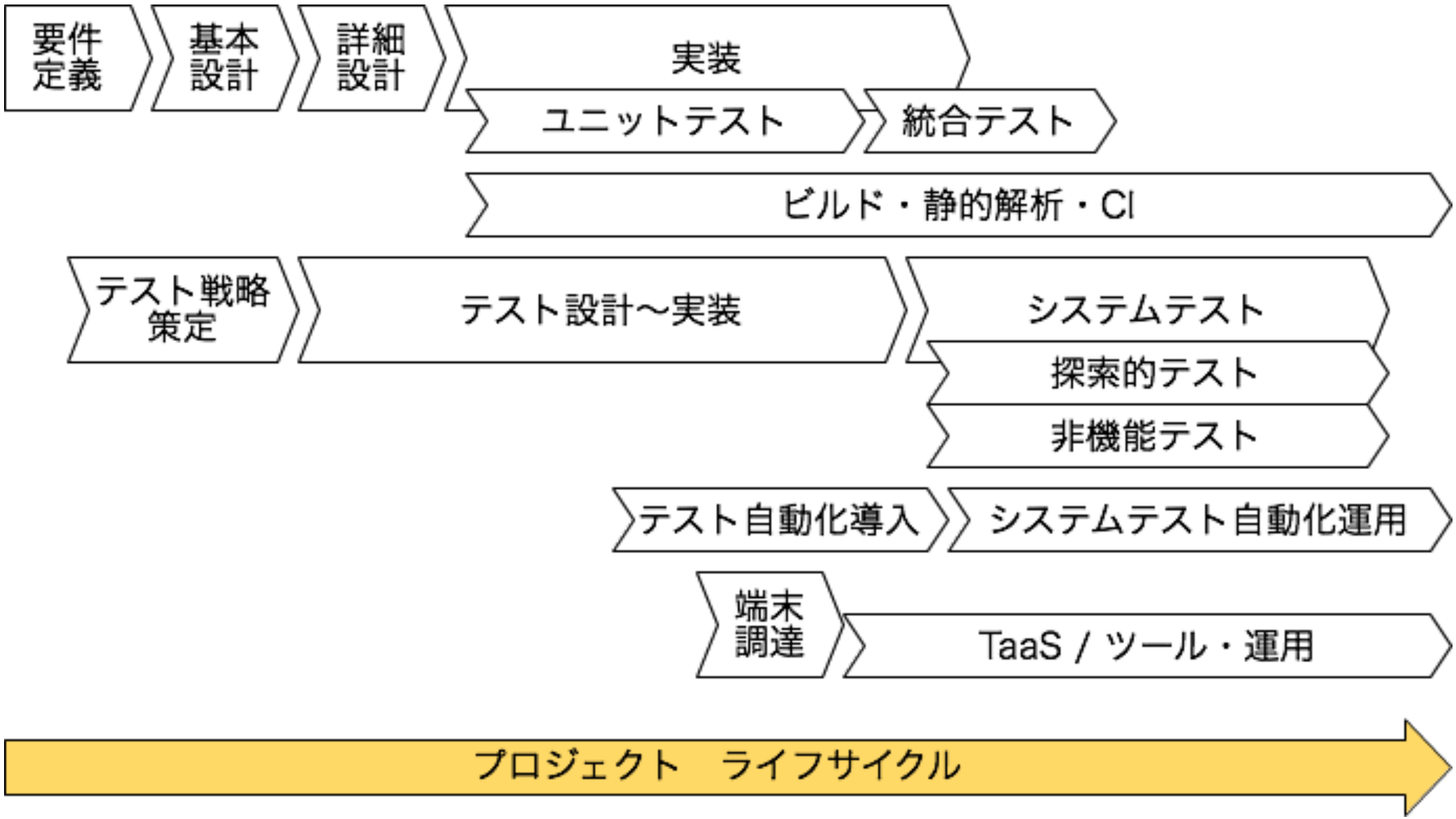
# Smartphone Test Farm

- ブラウザ上での描画速度が非常に早い
- ADB コマンドが簡単に流せるようになっている
- 自社で端末を複数台所有しているような環境では導入しやすい

# Smartphone Test Farm

- ”音”のフィードバックは分からない
- SDカードの抜き差しができない

# 探索的テスト


要件定義
基本設計
詳細設計
実装
ユニットテスト
統合テスト
ビルド・静的解析・CI
テスト戦略策定
テスト設計〜実装
システムテスト
探索的テスト
非機能テスト
テスト自動化導入
システムテスト自動化運用
端末調達
TaaS / ツール・運用
プロジェクト　ライフサイクル

# 探索的テスト

- 参考：

  JaSST'14 Kyushu

  高橋 寿一「探索的テストってなんですか？」

  http://jasst.jp/symposium/jasst14kyushu/report.html#invitation2

- 対象ソフトウェアの理解と試験設計とテストケースの実行を同時に行う手法

- 決して ad-hoc テストではない

# 探索的テスト

- 記述式（テストケースベース）
  - テスト要求分析 > テストアーキテクチャ設計 > 詳細設計 > テスト実装 > 実行

# 探索的テスト

- よく不具合が見つかるところ
  - ボタンの連打
  - 複数同時押し・マルチタップ操作
  - 画面遷移・回転・アニメーション中の操作
  - サスペンド＆レジューム
  - notification, 着信などの割り込み
  - ネットワーク環境　on/off
  - アンインストール・再インストール

# 探索的テスト

- 私の場合：
  - 100% 探索的テストには頼らない
    - テストケースベースと併用することが多い
      - 品質特性を用いた優先順位付けやリスクベースから抑えておきたいポイントは網羅する
- モバイルアプリの場合、非機能も視野に入れる

# 非機能テストの効率化

# 非機能テスト

- 非機能要件に対するテスト
- 性能テスト、ロードテスト、ストレステスト、ユーザビリティテスト、信頼性テスト、移植性テスト、etc……
- すべてのテストレベルで実行可能（システムテストだけではない）

# 非機能テスト

- 非機能要件に対するテスト
- 性能テスト、ロードテスト、ストレステスト、ユーザビリティテスト、信頼性テスト、移植性テスト、etc……
- すべてのテストレベルで実行可能（システムテストだけではない）

# 性能テスト(1/2)

- 速度に関しては、ユニットテストの中で実施できる
  - 対象メソッドの実行時間を計測し、しきい値を超えたらテストを失敗させる
  - iOSでは [measureBlock:]を利用。渡したブロックの処理時間を計測し、平均処理速度の劣化・ばらつきを検知するとテストが失敗する

# 性能テスト(2/2)

- CPU、メモリリークの計測
  - iOS: Xcodeのプロファイリング機能を使用（[Product]-[Profile]）
  - Android: DroidKaigi 2016講演『パフォーマンスを追求したAndroidアプリを作るには』参照（スライド/ビデオ）

# 可用性テスト

- 自動テストを流しながらランダムにテスト
  - ネットワーク
    電磁シールド袋（東陽テクニカ社製など）
  - サスペンド・レジューム、縦横回転
    ただし、描画回数やタイミングが変わるとスクリプトの実行が止まってしまうケースがある

# ユーザビリティテスト

- ユーザーテストのアウトソーシングサービス
  - uiscope.com

- NE比分析は、コンバージョン計測ツールのSDKを利用して計測
  Answers(Fabric), Google Analyticsなど

# さいごに

- 要件の複雑化、ビジネススピードの加速、
  スマートフォンの性能向上、OS のアップデート…

- これらに屈しないために各ポイント全てにおいて
  改善できるきっかけになれば幸いです。