私が経験したソフトウェアテスト

～ ワークステーション、組み込みシステム、ウェブサービス ～

柴田　芳樹

JaSST 2019 Tokai

2019年10月4日

予稿版（一部のページは意図的に空白です）

本日のテーマ

- 技術的変遷
  - テストファースト/テスト駆動開発
  - 継続的インテグレーション
- 私の経験
  - HOLENET開発（九州工業大学）
  - ワークステーション開発
  - デジタル複合機コントローラソフトウェア開発（Take 1〜Take 4）
  - 初めてのウェブサービス開発
- ソフトウェア開発とQA

自己紹介

- 柴田 芳樹（Yoshiki Shibata)　1959年生まれ
- 九州工業大学情報工学科＆大学院（情報工学）
- 職務経歴
  - 富士ゼロックス（株）（パロアルト研究所を含め米国ゼロックス社に4年半駐在）
  - 日本オラクル（株）
  - （株）ジャストシステム
  - 富士ゼロックス情報システム（株）（米国ゼロックス社に半年駐在）
  - （株）リコー
  - ソラミツ（株）
  - （株）**メルペイ**（2018年6月～現在）
- 主な著書および翻訳本
  - 『プログラマー”まだまだ”現役続行』
  - 『Effective Java 第3版』
  - 『プログラミング言語Go』
  - 『ベタープログラマ』

テスト駆動開発と継続的インテグレーション

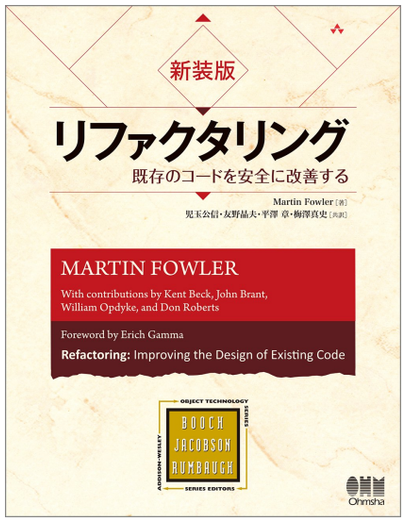
パラダイムシフトは2000年前後に起きている

1980年代・1990年代までは、自動化されたテストというのは常識ではなく、テストは手作業で実行され、結果の確認は目視が主流であった。

パラダイムシフトは2000年前後に起きている

1980年代・1990年代までは、自動化されたテストというのは常識ではなく、テストは手作業で実行され、結果の確認は目視が主流であった。

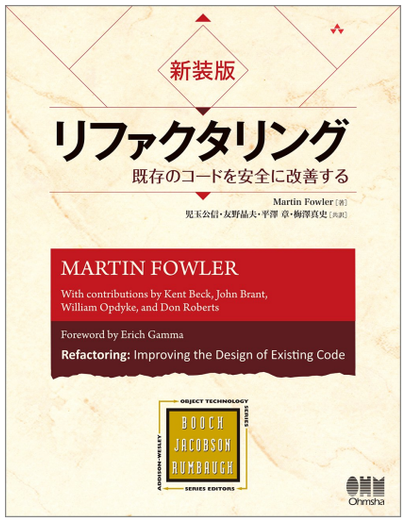
> テストの実行は確かに簡単になりました。しかし、テストの実行が簡単になっても、テストは依然として極めて退屈なものでした。これは、コンソールに出力されるテスト結果を**私がチェックしなければならない**ためです。



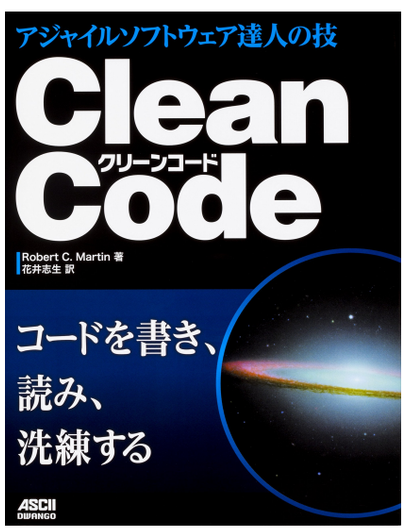
パラダイムシフトは2000年前後に起きている

1980年代・1990年代までは、自動化されたテストというのは常識ではなく、テストは手作業で実行され、結果の確認は目視が主流であった。

テストの実行は確かに簡単になりました。しかし、テストの実行が簡単になっても、テストは依然として極めて退屈なものでした。これは、コンソールに出力されるテスト結果を**私がチェックしなければならない**ためです。



この10年の間に、この業界では多くのことがありました。1997年当時、テスト駆動開発などという言葉は、誰も聞いたことがありませんでした。ほとんどの人にとって、**単体テストというのは動作をひとたび「確認」したら捨ててしまうものでした**。苦労してクラス、メソッドを書き上げ、それらをテストするための、その場しのぎのコードをでっちあげていたのです。



パラダイムシフトは2000年前後に起きている

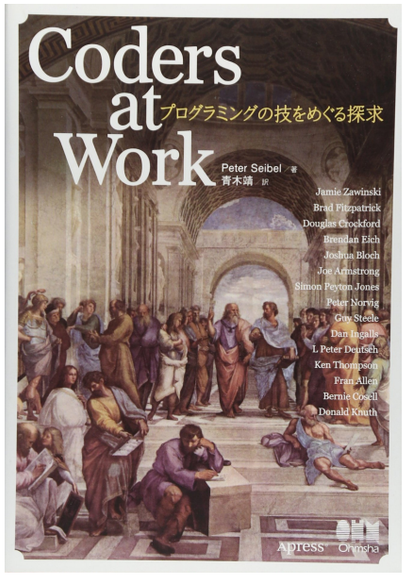
— デバッグの話をしましょう。あなたが追いかけた最悪のバグはどんなものでしたか

ブロック: 思いつくものの一つは、 …

・ ・ ・

— つまりバグはあなたのコードにはなかったのに、自分のコードに詳細なユニットテストを書いてテストし、自分のコード以外に目を向ける以外なくなったわけですね。**そのミューテックスの作者がテストを書いていればバグは見つかったはずで、自分が一週間半デバッグすることはなかったのにと思いますか？**

ブロック: あのミューテックス機構に良い自動化ユニットテストがあれば、あの苦痛は避けられたとは思いますが、**頭に入れておく必要があるのは、これが90年代初期の話だということです。十分なユニットテストを書いていないということでそのエンジニアを非難しようという気は全く起きませんでした。**



テストファースト開発

> 僕たちの厳密さの一つの例が、**自動化したユニットテストをテスト対象のコードより先に書く**というルールだ。たいていのプログラマーはすぐにコードを書きたがる。書き上がれば出来映えに自己満足して、そのコードのために自動テストを書こうという気にはなかなかならないものだ。こういった習慣をやめてしまうのは簡単だろうが、僕たちがコードに期待する品質には極めて重大なものなのだ。テストコードを対象コードの前に書くという規律により、常に守れるようになるわけだ。
>
> リチャード・シェリダン著『Joy, Inc.』（日本語版、p.212)

テストファースト・テスト駆動開発（Test First / Test Driven Development）

ソフトウェア開発では、1990年代までの手作業によるテストから、2000年以降はテスト駆動開発による自動化されたテストがソフトウェア開発の基本

自動化されたテスト

コンピュータによる自動実行可能な**テストコードを資産として残す**ことにより、**手作業によるテスト**から**コンピュータによる自動テスト**への移行

- ソフトウェア修正に伴う**デグレードの早期検出**
- 負荷テストによる不具合の発見（特にマルチスレッドプログラミング）
- コードの品質を良くするためのリファクタリング（技術的負債の返済）

継続的インテグレーション

終盤での結合の問題を避けるために、メンローでは各自の作っているものが全体として動くかどうか、**継続して結合し続けることで確認する**。終盤で驚かされることはない。結合で問題があっても、対応する時間も予算も十分にあるタイミングで発見できる。結合を最後にしか行わないと、大炎上してチームも生き残れないことが多い。

リチャード・シェリダン著『Joy, Inc.』（日本語版、p.215)

あるべき姿

ソフトウェア開発では、1990年代までの手作業によるテストから、2000年以降はテスト駆動開発によるテストの自動化が行われ、ビルドの自動化は1990年代から行われている

テストの自動化

コンピュータによる自動実行可能な**テストコードを資産として残す**ことにより、**手作業によるテスト**から**コンピュータによる自動テスト**への移行

- ➢ ソフトウェア修正に伴う**デグレードの早期検出**
- ➢ **負荷テスト**による不具合の発見（特に**マルチスレッドプログラミング**）
- ➢ **コードの品質を良くする**ための**リファクタリング**

ビルドの自動化

- ◆ 1990年代までの**夜間ビルド**※（*Nightly Build*）から、2000年以降は**継続的インテグレーション**（*Continuous Integration*）の時代
  - ➢ ソースコードの修正がコミットされると、**コンパイルから単体テストの自動実行まで**、即時にサーバで実施し、その結果をフィードバック
  - ➢ 早期に問題を発見する
  - ➢ **ビッグバン・インテグレーション**を避けて、常に動作するソフトウェアを保つ
- ◆ 2010年以降は、**継続的インテグレーション**から**継続的デリバリー**（*Continuous Delivery*）の時代
  - ➢ ソースコードの修正をコミットすることで、コンパイルから始まって、テストの自動実行（最終成果物のテストも含む）、リリースソフトウェアの作成までをサーバで自動で実施

※ 有名な例は、Windows NT開発で行われた夜間ビルド。その内容は、書籍『闘うプログラマー』に書かれている

私のソフトウェア開発経験

HOLENET開発（九州工業大学）

九州工業大学時代（1978年4月～1984年3月）

教育・研究用マイクロコンピュータネットワークHOLENET開発

（1981年4月～1984年3月）

- 通信コントローラボードの組み立て（ラッピング等）とボードのHWデバッグ
- 通信ソフトウェアの開発（ROM 2KB, Z-80アセンブリ言語）
- HOLENETを使った学生実験演習のテキスト作成、指導、レポート採点

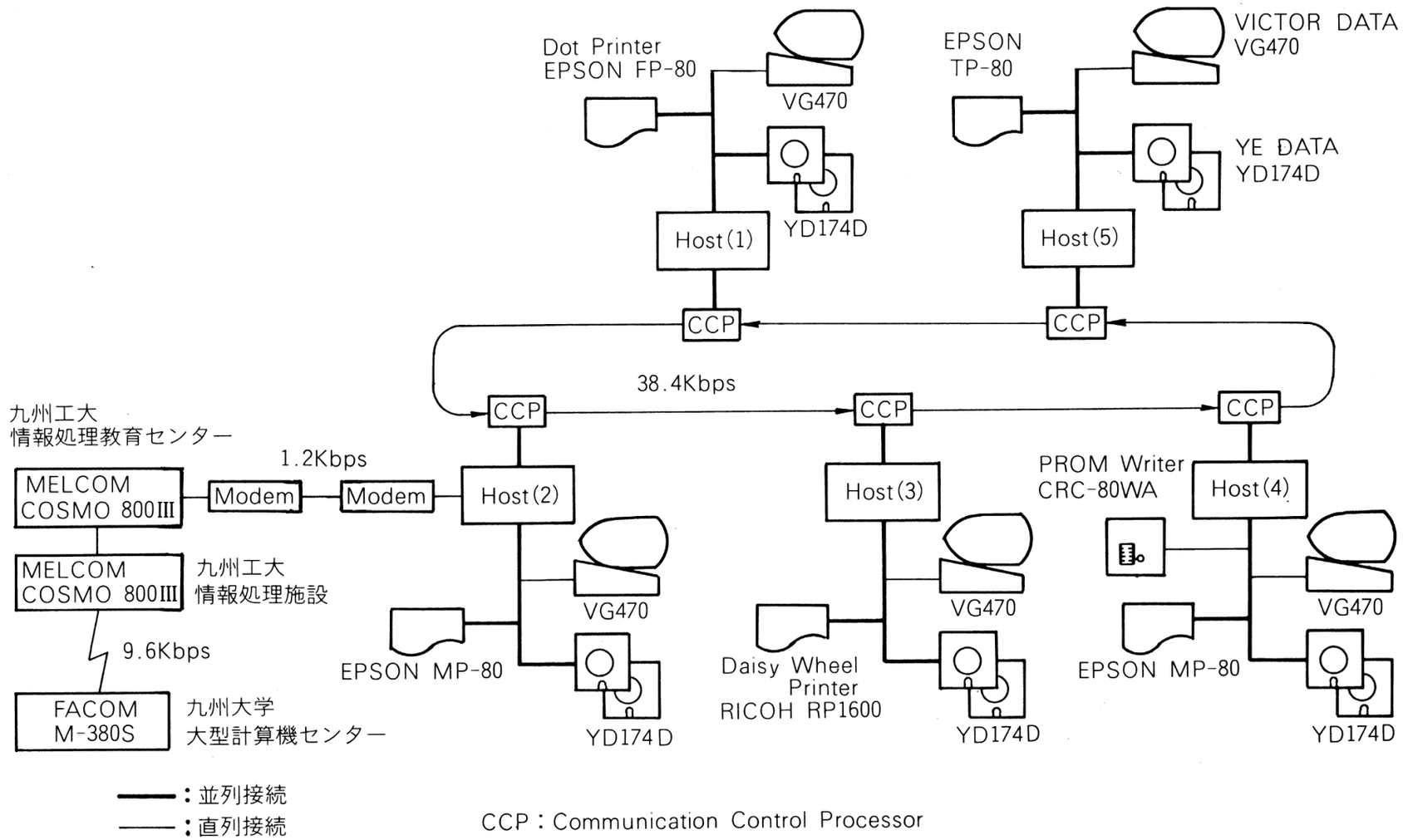


図 4.1　HOLENET とその環境

重松保弘著『マイクロコンピュータネットワークアーキテクチャの基礎』より

九州工業大学時代（1978年4月～1984年3月）

教育・研究用マイクロコンピュータネットワークHOLENET開発

（1981年4月～1984年3月）

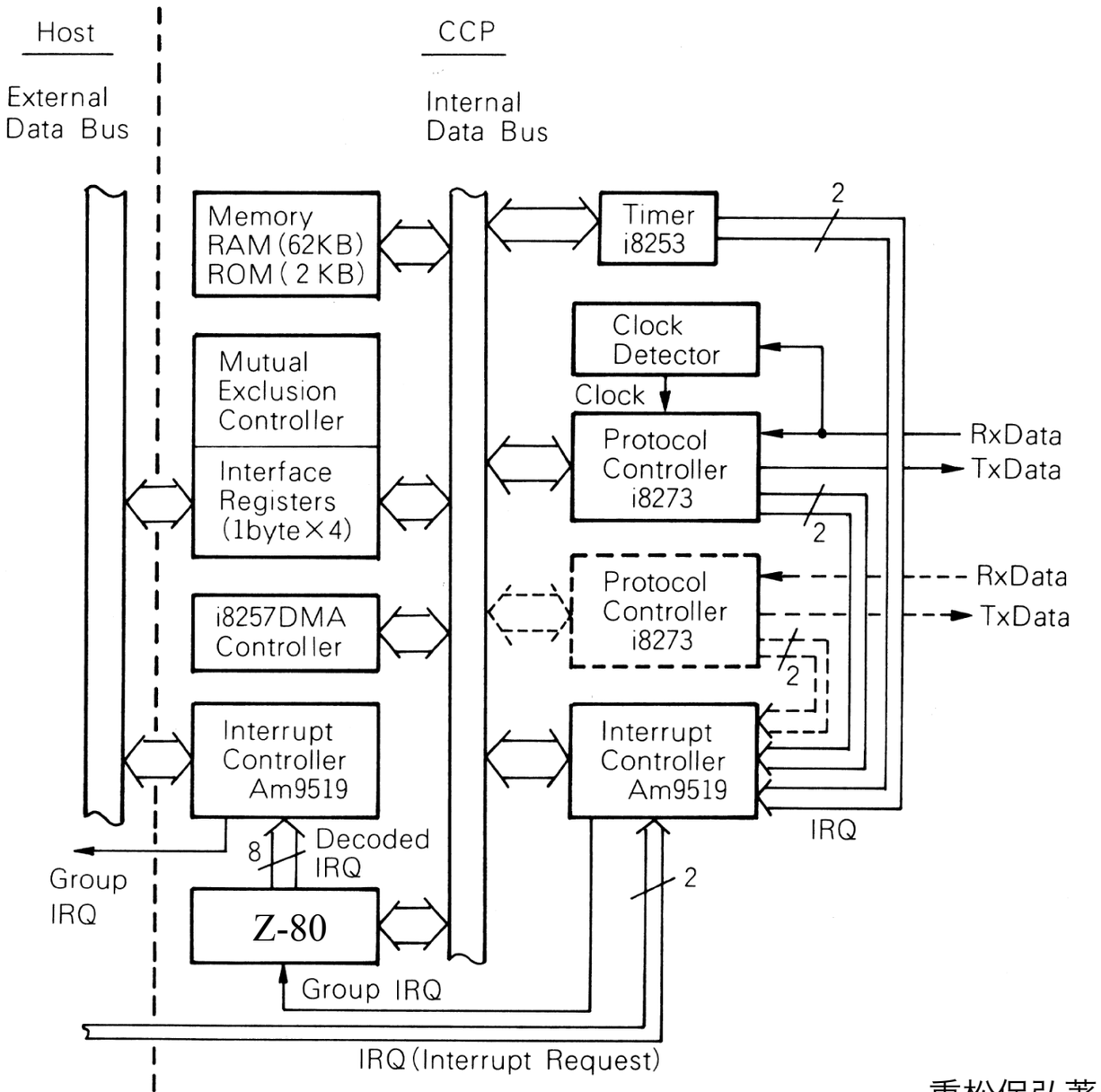


図 4.2　CCP のブロック構成図

重松保弘著『マイクロコンピュータネットワークアーキテクチャの基礎』より

九州工業大学時代（1978年4月～1984年3月）

バージョンコントロールシステム

- 何も使っていない
- HDもない頃で、すべて8インチフロッピーディスクで管理

通信コントローラボードのデバッグ（テスト）

- 5セットの通信コントローラボード（CCP）を一人で組み上げてデバッグした（半田付けとラッピング）
- ハードウェアのデバッグは、オシロスコープ一台で行った

通信ソフトウェアのデバッグ

- 2KB ROM用の通信ソフトウェアをインクリメンタルに開発
  1. I/Oポートへの出力をループで行うROMプログラムを作成して、HWのデバッグ
  2. HWが動作するようになったら、ホストコンピュータ（MP/M）からRAMへプログラムをダウンロードするROMプログラムを開発・デバッグ
  3. ROMプログラムの次の機能追加バージョンをRAMへダウンロードして、デバッグ。デバッグが完了したら、ROMへ焼き付ける
  4. ステップ3を繰り返す
  5. 2KBのROMが一杯になったら、さらなる拡張機能は起動時にホストコンピュータからRAMへダウンロードするようにして開発

私の経験

ワークステーション開発（富士ゼロックス）

富士ゼロックス時代（1984年4月〜1996年8月）

Fuji Xerox 6060 Workstation （1986年5月発売）

- ビットマップディスプレイ、マルチウィンドウ、文書作成、ネットワーク機能、etc
- CPU: 68000, OS: Idris (Unixクローン）、開発言語：C言語
- プロセス間通信によるプログラミング
- Ethernetドライバー、XNS（Xerox Network Systems）のプロトコルスタック・アプリケーションの設計・実装を担当

出典：http://www.fujixerox.co.jp/company/profile/history/product.html

当時のCM：https://youtu.be/KGL1PadFa8M

Fuji Xerox 1161 AI Workstation （1988年発売?)

- Smalltalkを搭載したワークステーション
- OSはUnix System V(?)
- TCP/IPプロトコルスタックの移植の責任者（移植そのものはやっていない）

Fuji Xerox GlobalView Workstation（1992年発売）

- Mesa言語で書かれたStarワークステーションのソフトウェアを、Xerox社のハードウェアからSunワークステーションへ移植し、カラー化
- 日米で約200人のエンジニアが従事した大規模プロジェクト
- 1988年11月〜1991年3月まで米国El Segundo, CAに駐在して従事
- VP Document Editorの移植、ビルド担当、Mesa PreProcessorの開発

富士ゼロックス時代（1984年4月〜1996年8月）

Fuji Xerox GlobalView Workstation（1988年11月〜1991年4月）

バージョンコントロールシステム

- SunOSを使っていたので、たぶん何かVCSを使っていたはず（cvs?）

ビルド／インテグレーション

- ビルド担当者がビッグバン・インテグレーションをアプリケーションごとに行う
- VP Document Editorのビルドも担当していたので、**分散コンパイルのプログラムを開発して、コンパイルを高速化**もしていた。

デバッグ

- バイナリーを実行して手作業でテスト
- ソースコードを見て怪しいところにデバッグ文を入れて、ビルドし直して、動作させてデバッグする

私の経験

デジタル複合機コントローラソフトウェア開発

Take 1（富士ゼロックス）

Take 2（富士ゼロックス情報システム）

Take 3（富士ゼロックス情報システム）

Take 4（リコー）

デジタル複合機コントローラソフトウェア開発 Take 1

米国ゼロックスPARCでのPageMillプロジェクトの商品化

（PARC駐在：1991年4月～1993年5月）

Fuji Xerox DocuStation IM 200（1996年発売）

- コピーとFaxの基本機能に加えて**ペーパーユーザーインタフェース**搭載
- CPU: SPARC, OS: Solaris 2.3、開発言語：C++言語
- **マルチスレッドプログラミング**
- アプリケーションランタイムの設計・実装、ビルド担当
- メモリ管理ライブラリの設計・実装

出典：http://www.fujixerox.co.jp/company/profile/history/product.html

Fuji Xerox DocuStation AS 200（1995年発売）

- コピーとFaxの基本機能に加えて、
- 自治体窓口証明（戸籍）発行システム
- **プレズマディスプレイによるタッチパネル操作**
- CPU: SPARC, OS: Solaris 2.3、開発言語：C++言語
- **マルチスレッドプログラミング**
- 1998年にリコーへOEM（「イマジオ 市町村窓口証明システム SG2000」）

出典：https://web.archive.org/web/19980119153537/http://www.fujixerox.co.jp/product/hukugo/dsas200-2.html

デジタル複合機コントローラソフトウェア開発 Take 1

Fuji Xerox DocuStation IM 200（1993年5月〜1996年8月）

バージョンコントロールシステム

- sccsか何かをベースとした独自のシステム

ビルド／インテグレーション

- 当初は**2週間ごとのビッグバン・インテグレーション（必ず失敗）**
- 必ず失敗するので、頭にきて**夜間ビルドを導入**

デバッグ

- バイナリーを実行して手作業でテスト
- ソースコードを見て怪しいところにデバッグ文を入れて、ビルドし直して、動作させてデバッグする
- **デッドロックが発生していれば、デバッガーを起動して、プロセスへアタッチして個々のスレッドの状態を調査してデバッグ**

デジタル複合機コントローラソフトウェア開発 Take 1

Fuji Xerox DocuStation IM 200（1993年5月〜1996年8月）

QAによるテストが開始されると2週間ごとにリリース

- 計画：日曜日の夜の夜間ビルド版を、月曜日にリリース

リリース予定の月曜日

① 各エンジニアは、自分が担当した過去2週間に修正したはずのバグが、（a）**リリースビルドで本当に修正されているかの確認**と（a）**担当部分の基本機能の確認**

② デグレードも含めて確認結果をリーダー達へ報告

③ 全員の確認結果が集まれば、リーダー達が集まって協議

④ エンジニアは、指示があるまで自分の担当以外も含めて機能テストを継続

⑤ QAへリリースする前に修正すべきと判断されたバグを担当者が修正

⑥ 再度ビルドして、修正が反映されたかの確認。**修正がないエンジニアは、④を継続する。その際に、新たな問題が発見されたら、②へ戻る。**

⑦ リリースできると判断されるまで②から⑥を繰り返す。**リリースできると判断されるまで、エンジニアは次の開発に着手してはいけない。**

デジタル複合機コントローラソフトウェア開発 Take 1

Fuji Xerox DocuStation IM 200（1993年5月〜1996年8月）

QAによるテストが開始されると2週間ごとにリリース

- 計画：日曜日の夜の夜間ビルド版を、月曜日にリリース

実際に計画通りに月曜日にQAへリリース出来たのか？

デジタル複合機コントローラソフトウェア開発 Take 1

Fuji Xerox DocuStation IM 200（1993年5月〜1996年8月）

QAによるテストが開始されると2週間ごとにリリース

- 計画：日曜日の夜の夜間ビルド版を、月曜日にリリース

出来ませんでした

実際のリリースは、水曜日が多かった

デジタル複合機コントローラソフトウェア開発 Take 1

Fuji Xerox DocuStation IM 200（1993年5月〜1996年8月）

隔週の月曜にリリースできなかた要因

- 各エンジニアは、2週間、QAテストから報告される**バグの調査・修正に忙殺**
  - **テストは手作業**であったため、誰も**機能を日々テストしなかった**
  - **リリース予定日の月曜日に、機能のデグレードや重要な障害に気付く**

教訓：フィードバックループが2週間と長すぎた

デジタル複合機コントローラソフトウェア開発 Take 1

Fuji Xerox DocuStation IM 200（1993年5月〜1996年8月）

メモリーリークとメモリ破壊のデバッグの困難さ

- 最初のコピーができるようになったときに、**一枚コピーするごとに2MBのメモリーがリーク**していた。
- 24時間動作するFaxの機能にとっては**メモリーリークは致命的**
- 登場したばかりのpurifyを試してみた
- purifyを適用すると使い物にならないほど遅くなった
- 当初、purifyはマルチスレッドにきちんと対応していなかった

メモリー管理機構をどうにかする必要に迫られた

- **日々の開発で常に組み込んでいても性能に影響を与えない**
- **メモリーリークを容易に調査できること**
- **メモリー破壊を早期に検出できること**

デジタル複合機コントローラソフトウェア開発 Take 1

Fuji Xerox DocuStation IM 200（1993年5月〜1996年8月）

C++用の独自のメモリー管理機構を実装した

- new/deleteオペレータを置き換える実装
- deleteオペレータが呼ばれたときに、解放されるメモリーの破壊を検出
- 獲得メモリーへのマーキング機構（未解放メモリーの一覧表示機構）
- スレッド単位のメモリー管理機構

「clibと呼ばれるライブラリー開発の思い出」（https://yshibata.blog.so-net.ne.jp/archive/c2306124827-1）

デジタル複合機コントローラソフトウェア開発 Take 1

Fuji Xerox DocuStation IM 200（1993年5月〜1996年8月）

マルチスレッドプログラミングの困難さ

- QAからハングしたという電話連絡があれば、即調査に入る
- 調査は、**Debugger of the Day**に割り当てられたその日の担当エンジニアが行う。
- 調査担当エンジニアは、ハングした機器にリモートログインして、デバッガを立ち上げて原因が分かるまで調査

デジタル複合機コントローラソフトウェア開発 Take 2

新たなコントローラソフトウェア開発（2000年1月〜2002年8月）

- コントローラソフトウェアを再設計・再実装する大規模プロジェクト
- オブジェクト指向設計の教育・コンサルティングを担当
- 2001年はほぼ毎日設計レビューやコードレビューを実施（自分では実装せず）

各種ガイドの制定

- 『Code Review Guide』
- 『C++ Coding Standard』

C++用メモリ管理ライブラリーの再設計

- Fuji Xerox IM 200の経験を踏まえて、C++用メモリ管理ライブラリーを再設計
  - メモリ破壊報告機構
  - メモリリーク調査機構
  - メモリ使用量報告機構
- そのメモリ管理ライブラリーをベースにスレッド関連のライブラリーを設計
  - スレッド生成
  - 再帰的ロック、Reader/Writerロック、通知機構（NotifyAll/Wait）
  - スレッドセーフなコレクションライブラリー

テストはすべて手作業

空白ページ

空白ページ

学んだこと

デジタル複合機コントローラソフトウェア開発

Take 1（富士ゼロックス）

Take 2（富士ゼロックス情報システム）

Take 3（富士ゼロックス情報システム）

Take 4（リコー）

マルチスレッドプログラミングの困難さ

マルチスレッドプログラミングにおける重要な４要件

1. きちんとしたレビュー

- マルチコアおよびマルチスレッドプログラミングのきちんとした経験および知識を持つ人が設計やコードをレビューしていること
  - マルチスレッドプログラミングは、逐次的プログラムと比較して、経験の浅い人にとっては直感に反する動作をする

2. 自動テストによるテスト

- 自動テストが整備されていて、いつでもテストを自動実行可能であること
  - 手作業で一回動作したとか100回動作したからと言って、プログラムの正しさは保証されない
  - 機能を確認するためのテスト設計がきちんとされていること

3. 様々な環境での長時間ランニングテスト

- 製品がシングルコアであっても、マルチコア環境で長時間ランニングテストを繰り返す（開発者のPCを総動員して、平日夜間・週末も）
- 同時にシステムにさまざまな負荷（CPU負荷、HD負荷、メモリ負荷）をかけて動作させる

4. ハングしたらその場で徹底して調査

- ログを入れてもう一度再現させるという対応は避けて、その場で徹底的に調査する

マルチスレッドプログラムのテスト

継続的インテグレーションは必要条件ではあるが、十分条件ではない

- CIサーバでの1回のテストによる合格は、プログラムの正しさを保証しない
- システム全体を自動でテストできるように最大限の工夫をする

開発者の開発用PCを最大限に活用する

- 夜間、週末に開発用PCを遊ばせるのではなく、自動テストを様々な環境で動作させる
- 開発用PCのスペックをケチらない。開発しながら仮想環境で自動テストを実行できるぐらいのスペック

朝、出社してテストが停止していたら、調査を最優先業務とする

- 二度と発生しないかもしれないので、最優先で調査させる
- 原因が判明したら必ず「再現テスト」を作成してから、修正を行わせる

発生した障害をきちんと記録して、分析する

- 開発と並行して、原因分析・対策の検討を開発グループ内で開発活動の一環として実施する
- プロジェクトが終わってから分析するのでは、開発中に改善に取り組めない

初めてのウェブサービス開発

初めてのウェブサービス開発

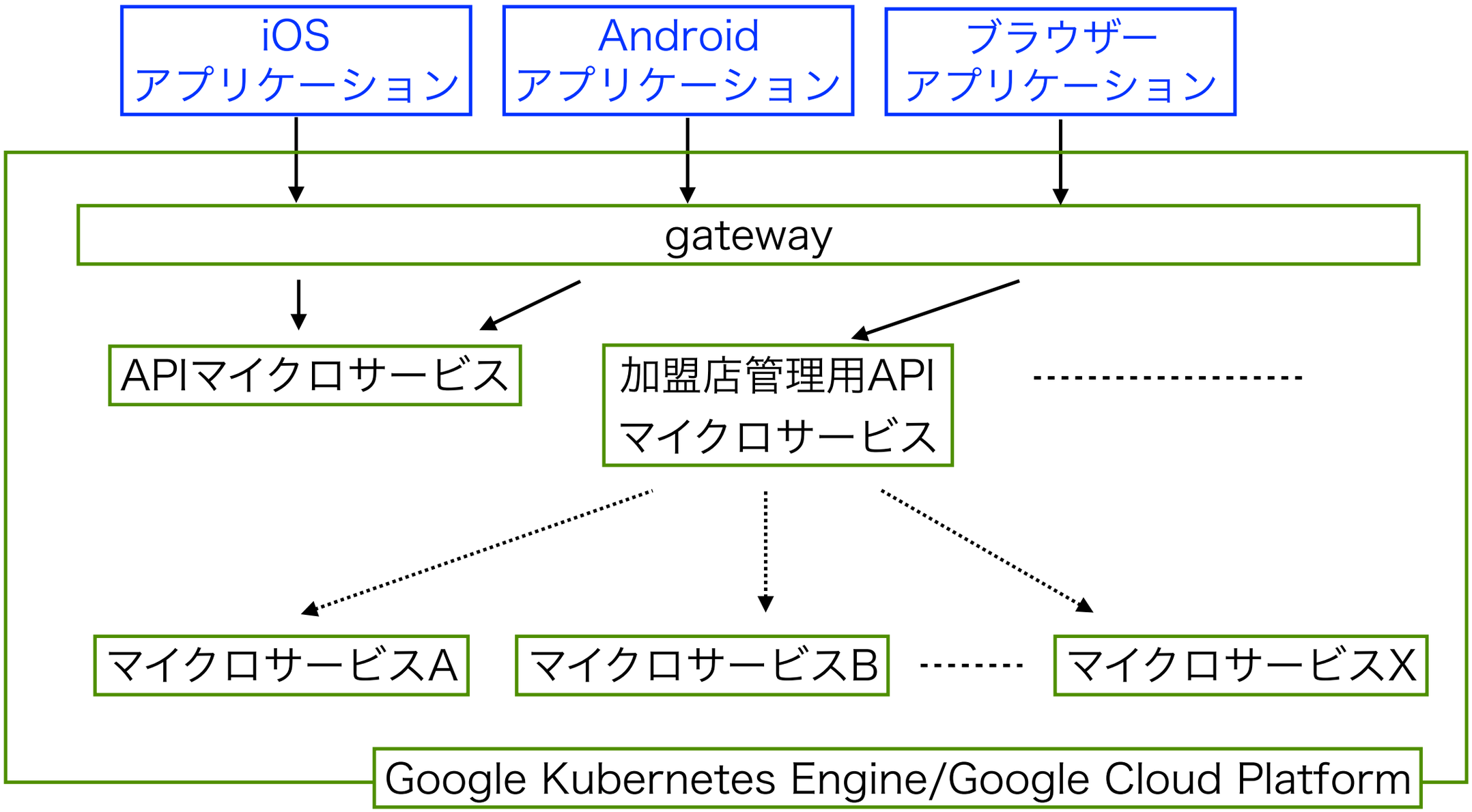
カンボジア国立銀行プロジェクト（ソラミツ株式会社）（2017年9月〜2018年5月）

- ブロックチェーン技術Irohaを用いたカンボジア国立銀行との共同プロジェクト
- プロトタイプされていたウェブサーバ側の実装の追加開発
- gRPCを使ってクライアントがウェブサーバへアクセスするシステム
- ウェブサーバはすべて**Go言語**で開発

加盟店管理画面用のAPIマイクロサービス（株式会社メルペイ）（2018年6月〜2019年4月）

- Google Cloud Platform / Google Kubernetes Engineを使用
- 加盟店が使う管理画面用のAPIマイクロサービスの開発
- マイクロサービスが提供するAPIを設計し、その実装・テストまでを担当
- マイクロサービスは**Go言語**で開発

マイクロサービスの構成（簡略版）



- クライアント（iOSアプリやAndroidアプリ）はgatewayを通して、APIマイクロサービスと通信する
- APIマイクロサービスは、他のマイクロサービスを使って機能を実現する

開発手順

1. API仕様の作成

- APIマイクロサービスは、gRPCで通信するので仕様を.protoファイルに記述。詳しくは、「gRPCを用いたマイクロサービスのAPI仕様の記述」（https://tech.mercari.com/entry/2019/05/31/040000）を参照。
- gatewayがJSON⇔gRPC変換を行い、クライアントからはPOSTする。

2. 実装・単体テスト

- APIの実装を行い、単独で単体テストが可能な機能は単体テストを作成

3. e2eテスト

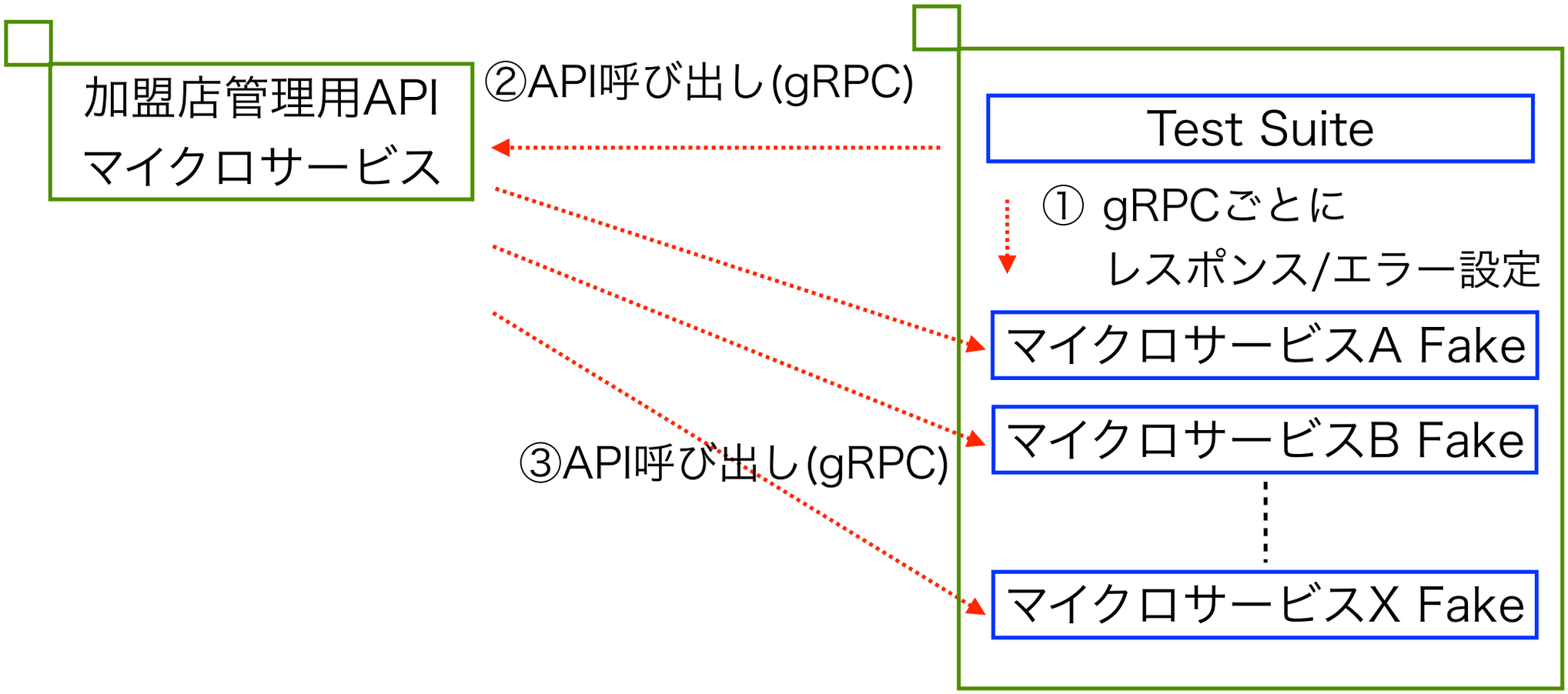
- 依存しているマイクロサービスをすべてFakeして、APIマイクロサービスのgRPCを呼び出して、すべての機能をテスト

4. サービステスト

- 開発用のGCP環境にAPIマイクロサービスをデプロイして、実際に依存しているサービスと通信するテスト

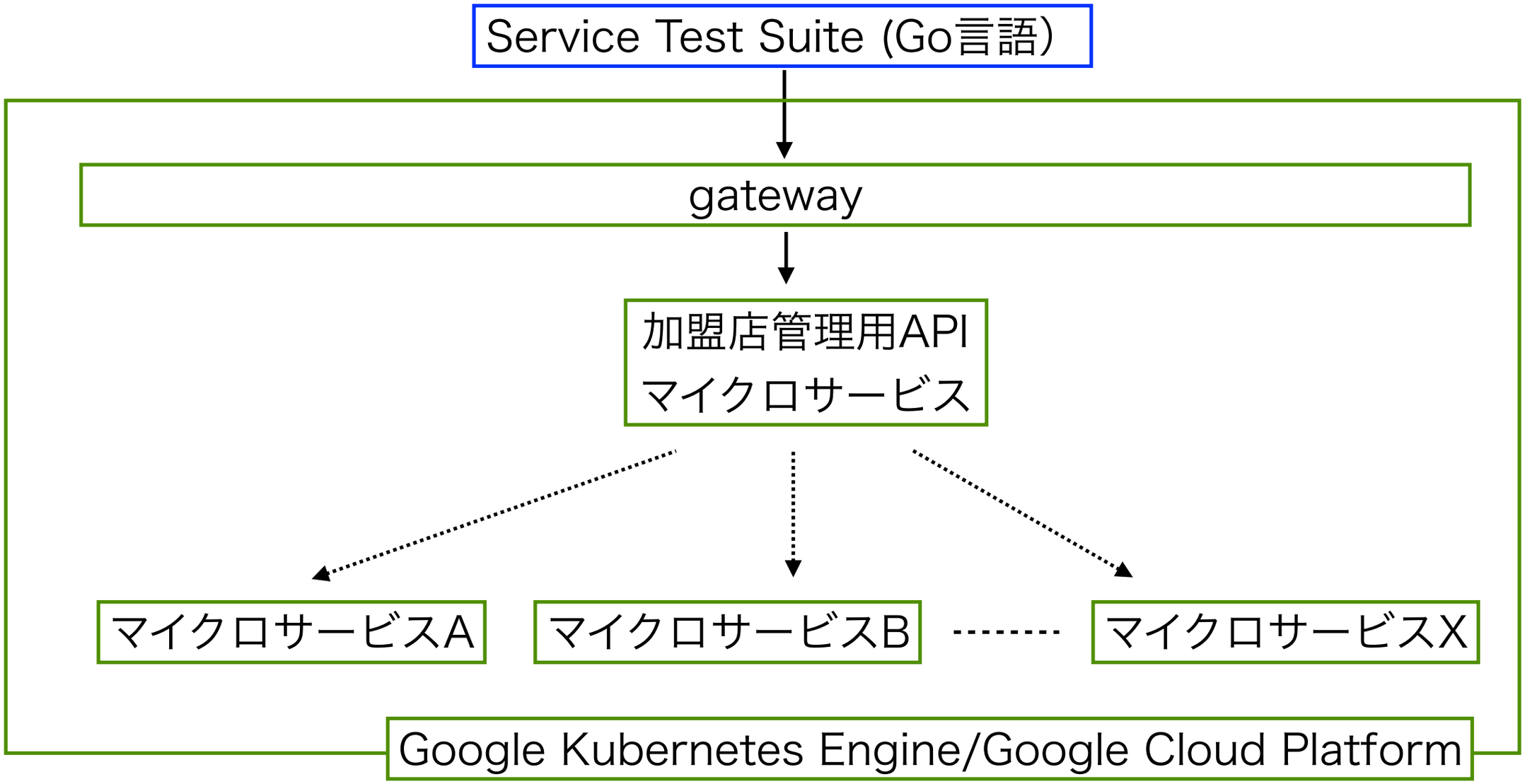
課題：依存しているマイクロサービスも含めて、多くのマイクロサービスの並行開発が行われるなかで、e2eテストまでを完了させなければならなかった。

e2eテスト



- **依存しているマイクロサービスのFake版をgRPCから自動生成**
  - RPCごとに返すべきレスポンス/エラーを設定できる
  - テストコードと同じプロセス内でgRPCのサーバーとして動作する
- **テスト対象のマイクロサービスは、別プロセスとして起動**
  - 依存するマイクロサービスは、起動時の環境変数の設定によりTest Suiteのプロセスへ接続するようになっている

サービステスト



- **開発用GCP/GKE環境**で動作しているマイクロサービスをテスト
- HTTP POSTを使ってgateway経由でAPIをテストする
- APIの**正常使用**、**不正使用**、**同時呼び出し**などのテストを行う

初めてのウェブサービス開発で示したかったこと

- **APIマイクロサービスをきちんと記述する**
  - APIマイクロサービスも含めて、マイクロサービス間はgRPCで通信しており、その仕様をきちんと.protoファイルに記述すること
  - 詳しくは、「gRPCを用いたマイクロサービスのAPI仕様の記述」（https://tech.mercari.com/entry/2019/05/31/040000）を参照。
- **開発チームは、自分達が作ったマイクロサービスのテストをきちんと行う**
  - 特にフロント（iOS アプリやAndroidアプリ）と通信するAPIマイクロサービスは、フロントと接続することなく、そのAPIをきちんとテストしている TestSuiteを開発チームが開発すること

組み込みシステム開発 v.s. ウェブサービス開発

ウェブサービス

- 組み込みシステムとは異なり多くのプラットフォームが整備されている
  - Google Cloud Platform
  - Google Kubenetes Engine
  - Spinnaker
  - Docker
  - Circle CI
  - etc
- プラットフォームの使い方や設定を学ばなければならない（プラットフォーム自身も発展し続けている）
- 自分達のサービスのロジックに専念できる
- 複雑なマルチスレッドプログラミングはしない
- ハードウェアやOSに関する基礎知識がなくても開発できる

組み込みシステム

- OSなどの最低限の環境上に独自にプラットフォームを構築する
- ハードウェアやOSに関する基礎知識が必須
- 複雑なマルチスレッドプログラミングやマルチタスクプログラミングをすることがある

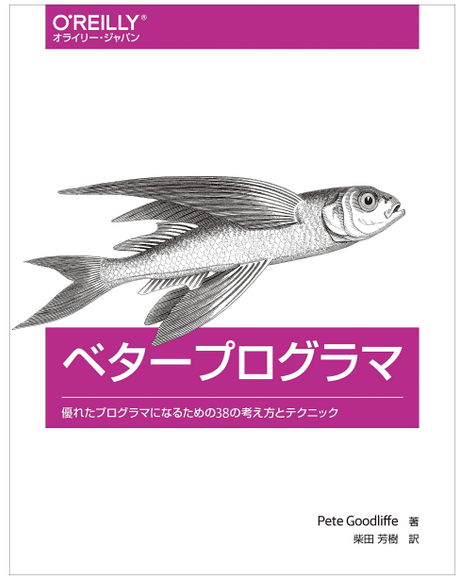
ソフトウェア開発とQA

QAとは（『ベタープログラマ』より）

彼らの名前は、理由があって「テスト部門」ではなく「QA（品質保証）」なのです。彼らの役割は、ロボットのようにボタンを押すことではありません。**品質を製品に作り込むこと**なのです。

　そのために、**QAは開発プロセスの最後ではなく、プロセス全体を通して深く関与していなければなりません。**

- 作られるものを理解して形づくるために、ソフトウェアの仕様書に関与します。
- 作られるものをテスト可能にするために設計と開発に貢献します。
- 当然ですが、テストフェーズでは深く関与します。
- そして、最終の物理的なリリースにも深く関与します。つまり、テストされたものが実際にリリースされて配置されるようにします。



ソフトウェア開発部門によくある問題

- **テスト設計がうまくできない**ソフトウェアエンジニアが多い
  - プログラミングの教育や研修は受けても、テスト設計の教育や研修を受けていないので、自動テストでもテストケースが漏れていることが多い（パラメータの組み合わせ網羅率って何？）

- **API設計がうまくできない**ソフトウェアエンジニアが多い
  - 誤った呼び出しなどのエラーケースの動作の記述を含むAPI仕様をうまく書けない（APIの契約をきちんと設計できない）
    - テストファースト開発がうまくできない
    - 防御的プログラミングができない

- **コードカバレッジを品質基準にしている**ソフトウェア開発組織が多い
  - コードカバレッジはソフトウェア品質を担保しない
    - バグがあっても、コードカバレッジ100%は容易に達成できる

ソフトウェア開発部門とQA

「テストファースト・テスト駆動開発」と「継続的インテグレーション」の時代には、早い段階からソフトウェア品質の作り込みが重要

「**QAは開発プロセスの最後ではなく、プロセス全体を通して深く関与していなければなりません。**」（『ベタープログラマ』）

- 自動化された各種テストがきちんと開発部門でテスト設計されるように協業する
  - テスト設計の学習と実践を一緒に行う（レビューする）
  - テスト品質の向上させるために一緒に改善に取り組む
- 開発部門で行われている継続的インテグレーションの状況に関心を持ち、QAへリリースされる前のソフトウェア品質を確認する
- QA向けのリリースビルドが、開発の終盤ではなく、開発の初めから自動化されているよう推進する

まとめ

ソフトウェア開発は複雑な活動です。その中で1990年代までは常識ではなかった活動が、今日のソフトウェア開発では必須となっている。

テスト駆動開発および**継続的インテグレーション**は、今日では必須であり、それにより開発者は、テストやインテグレーションといった作業をコンピュータに実行させることにより、1990年代と比べて多くの時間を創造的な活動へ費やすことができるようになっている。

ソフトウェア開発部門とQA部門は、開発の早い段階から、ソフトウェアの品質を作り込むために協業すべき時代となっている。

作業はコンピュータに、人は創造的な活動を

ご清聴ありがとうございました

Thank you for your time and attention.

yoshiki.shibata@gmail.com
http://yshibata.blog.so-net.ne.jp/