修士論文

ウェブアプリケーション開発における要求獲得のための テスト記述支援環境の提案

中地 祥剛

2019 年 3 月 14 日

奈良先端科学技術大学院大学
情報科学研究科 情報科学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
修士（工学）授与の要件として提出した修士論文である．

中地 祥剛

審査委員：

飯田元 教授	（主指導教員）
井上 美智子 教授	（副指導教員）
市川 昊平 准教授	（副指導教員）
崔 恩瀞 助教授	（副指導教員）

ウェブアプリケーション開発における要求獲得のためのテスト記述支援環境の提案*

中地 祥剛

内容梗概

近年，ゲームやウェブサービスなどの開発現場では，厳密な要求仕様書が作成されずプランナー（プログラミング知識はないがソフトウェアの機能の仕様を提案する人）からプログラマーへ仕様変更が口頭で伝えられる事が多い．従って，コミュニケーション齟齬（用語やニュアンスなどの誤解）によって誤った開発が行われ，開発が最初からやり直しになる可能性がある．実際に行った予備実験において，口頭だけの仕様伝達で齟齬が発生することを確認した．またその結果から，この問題の解決案としてプランナーがテストコードを通じて仕様変更をプログラマーへ伝えることにより，齟齬を防止するというアプローチを考案し，実際にその有効性を評価した．

キーワード

ウェブアプリケーション，テストコード自動生成，要求獲得

*奈良先端科学技術大学院大学 情報科学研究科 情報科学専攻 修士論文,
2019 年 3 月 14 日.

Proposal of a test description support environment for request acquisition in web application development*

Nakaji Yoshitake

Abstract

In the development of web application, requirements changes are frequently delivered from planners (i.e., people who suggest new specifications of software functions but do not have knowledge of programming) to developers without using the required documents. Therefore, there is a high possibility that development will be redone from the beginning due to communication discrepancy (i.e., misunderstanding of terms, nuances) between a planner and a developer. I assume that this problem can be solved by presenting the input values of the web application and corresponding output values when the planner tells the specification change. To prove this assumption, this study conducts preliminary experiments to confirm whether communication discrepancies occur during the development of web application. Based on the results of these preliminary experiments, it also proposes a planner's test description support environment and conducts a case study to show the effectiveness of the environment.

Keywords:

Web application, Test code generation, Request acquisition

*Master's Thesis, Department of Information Science, Graduate School of Information Science, Nara Institute of Science and Technology, March 14, 2019.

目次

図目次

第 1 章　まえがき

近頃，ネットワークの発展によりウェブブラウザを用いてインターネットを手軽に使えるようになった．それに従い， スマートフォンやウェブブラウザ向けのゲームやウェブサービスなどのソフトウェアが盛んに開発されている．

またソフトウェアの開発における問題の 1 つとして， 開発の手戻り (開発のプロセスが一部逆行してしまうこと）があり、その主要な原因として口頭などのコミュニケーションによって発生する齟齬が挙げられる．特に，ウェブアプリケーション開発の現場は， より高速に開発サイクルを回転しサービスの機能などを発展していく必要がある．そのため， ソフトウェアの仕様が変更された際にも従来のソフトウェア開発のように仕様書などを作成することなく， 口頭， または数行程度の簡単な文書などでプランナー（プログラミング知識はないがソフトウェアの機能の仕様を提案する人）からプログラマーへの要求の伝達が行われる事が多い．しかし，自然言語によるコミュニケーションにはあいまいさが多いため， 要求を受け取ったプログラマーが疑問を抱くことなく誤って理解した仕様の通り実装し， プランナーによる成果物の確認のときに初めて誤解が発覚し， 開発の手戻りが発生する可能性がある． Ferrari らは口頭インタビューを用いた要求獲得における曖昧さ発生のきっかけについて調査し， 仕様書に記載されている文書とは異なる種類の曖昧さが発生していることを明らかにした [5].

本研究ではこのようなコミュニケーション齟齬が実際に発生するのかということを確認するために， 予備実験を行った．予備実験では被験者に対して競技プログラミングサイトの簡単な問題を口頭で説明した上で，問題の要件を満たすプログラムを被験者に作成してもらった．また， 正解に至るまでの誤提出数や質問数，その内容を記録した．この予備実験の結果から， 実際に口頭のみの要求伝達で齟齬が発生することが確認できた． さらに， 追加予備実験として口頭での伝達に加えて，実際の問題文に存在するいくつかの入出力例を与えたところ， 被験者の正解までの提出回数に大きな減少が見られた． この予備実験と追加予備実験の結果から， コミュニケーション齟齬による誤った開発を防ぐ解決策として，プランナーが直接ソフトウェアの期待する入出力例を与えるためにテストコードを作成するということが考えられる．

本研究と同様にプログラミング知識がない人間のテスト記述支援というアプローチを採用したツールとしては，プログラミング知識を持たない人間でもブラウザの操作を記録，再生することによってテストコード作成をするアプローチを提案した中島ら [16] の研究や，アプリケーションに対する操作と自然言語フレーズの対応表を作成した上で自然言語を用いてテストを作成するツール Gherkin [1] がある．しかし，いずれもテスト保守性の問題やプログラマーの保守工数増加などそれぞれの課題がある．

そこで本研究では，既存研究の課題の解決のため，自然言語対応表の自動生成や，自然言語フレーズで記述されたテストコードとプログラミング言語で記述されたテストコードの両方の内容を常に同期するというアプローチによって既存研究の課題の解決を提案する．以降，2 章ではテスト記述支援技術および，その関連研究について述べる．3 章では実際に口頭による仕様伝達で齟齬が発生するのか，また，齟齬を回避するアプローチとして入出力の例示が効果を発揮するのかという事実を確認するために行った予備実験および追加予備実験の概要と結果を述べる．4 章では本研究において提案するテスト記述支援技術システムについての詳しい説明を行い，5 章で想定している実際の利用シナリオについて述べる．最後に 6 章でまとめと今後の課題について述べる．

第 2 章　背景

本章では，本研究の背景としてウェブアプリケーション開発およびソフトウェアテストついて説明する．はじめに 2.1 節で，ウェブアプリケーション開発について説明し，その特徴とウェブアプリケーション開発で用いられているアジャイル開発手法の特徴と関連研究を紹介する．その後，2.2 節ではソフトウェアテストやソフトウェアテストの分類について説明し，最後に 2.2.3 節では既存のウェブアプリケーションテスト支援に用いられている技術や関連研究を紹介する．

2.1　ウェブアプリケーション開発の特徴

ウェブアプリケーションとは，インターネットを通して利用することができるアプリケーション，ソフトウェアのことを指す．多くのウェブアプリケーションはウェブブラウザ上での表示を管理する HTML 言語や JavaScript 言語で作成されたプログラムと，バックエンドのサーバー側で動作するプログラムによって構成される．

それに対してインターネット接続を必要せず，ローカルの計算機上のみで動作するソフトウェアをデスクトップアプリケーション，またはスタンドアロンアプリケーションなどと呼称する．ウェブアプリケーションのメリットとしてインターネットとウェブブラウザ以外の OS などの環境に依存しないということが挙げられる．デメリットはネットワーク環境がない状況では利用できないことである．近年技術の発展によりインターネット回線速度が大幅に向上したことに伴い，ゲームなど従来はスタンドアローンアプリケーションが主流であったドメインにおいても、ウェブアプリケーションへの移行が進んでいる．

スタンドアローンアプリケーションと比較してウェブアプリケーション開発では，開発の区切り方が異なる傾向がある．ウェブアプリケーションは常にサービスが稼働しており，またそこに対して断続的に仕様変更や機能追加が発生するという特徴がある．最近ではスタンドアローンアプリケーションについてもいくつかの変更などをまとめたパッチなどをインターネット経由で配布し，機能更新を行うケースもあるが，それでもある程度の変更がまとめられて区切りの良いところでのリ

リースとなることが多い．それに比べウェブアプリケーションの開発では変更のテストが済み次第すぐにでも最新のコードがデプロイ（各サーバーでアプリケーションを動作させること）される．例えば，世界で最も著名かつ巨大なウェブアプリケーションサービスの 1 つである Amazon.com は，2012 年時点で 11 秒ごとに新しいコードがデプロイされていると発表している [13].

このことからも分かるようにウェブアプリケーション開発では設計，開発を含めた 1 つのサイクルがスタンドアローンアプリケーション開発と比べ，高速である．そのことに対応するために生まれた開発手法がアジャイル開発である．

アジャイル開発とは，従来のウォーターフォール型の開発と比較される，小規模な開発サイクルを複数回設けることにより，顧客またはプログラマー自身によるフィードバックを獲得し，製品の質を少しずつ改善していくような開発方式である．アジャイル開発の明確な定義は存在しないが，一般的にはアジャイルソフトウェア開発宣言 [3] に書かれている

- プロセスとツールより個人との対話
- ドキュメントよりも動くソフトウェア
- 契約交渉よりも顧客との交渉
- 計画に従うよりも変化への対応

の 4 項目に従うような開発がアジャイル開発として望ましいと考えられている．

従来のウォーターフォール型の開発の場合，テストや受け入れ時に不具合が発覚したり仕様の変更が発生した場合，手戻り工数が大きくなってしまう問題があるが，アジャイル開発の場合，小規模の実装，テストの組み合わせを積み重ねていくような開発方式になっているため，それらの問題に対して柔軟に対応し，手戻り工数を最小限に抑えることができる．また，その開発方式の特徴は最終的に製品として出荷されるようなパッケージソフトウェアよりも，順次仕様を追加していけるウェブアプリケーション開発と相性がよいため，現在は多くのウェブアプリケーション開発の現場において，この開発形態もしくはそれに準じた開発方式が採用されており [17]，本研究もそのような環境を想定している．

表 2.1　テスト時に使用する分類軸による分類

テスト種類	テスト時に使用する情報
ホワイトボックステスト	設計，仕様，実装
ブラックボックステスト	仕様のみ
グレーボックステスト	設計，仕様，実装

2.2　ソフトウェアテスト

次に本研究でも使用するソフトウェアテストやそれに関連する概念について説明する．ソフトウェアテストとは，開発したソフトウェアが意図した動作を満たすかどうかを確認するために行われる開発工程の 1 つである．ソフトウェアに対して直接操作を行うものから，ソースコードの一部分を実行して入出力を確認するものまで，多様な種類が存在する．ソフトウェアテストは，いくつかの分類軸によって分類される [15]．以降は各軸に基づく分類およびそれぞれのテスト手法について説明する．

2.2.1　テスト時に使用する情報によるソフトウェアテストの分類

まずもっとも大きな分類軸である，テスト時に使用する情報によるソフトウェアテストの分類と，各テストについての説明を述べる．表 2.1 に各テストの分類を示す．

ホワイトボックステスト

ホワイトボックステストとは，テスト対象のソフトウェアの内部パス，構造，実装に基づいて実施されるテストのことを指す．ホワイトボックステストの一般的な手順は以下のようなものである．

1. テスト対象ソフトウェアの実装を分析する
2. テスト対象ソフトウェアのパスを識別する
3. テスト対象ソフトウェアの特定のパスを実行するような入力を求める
4. テストを実行する

表 2.2　テストカバレッジの分類

カバレッジ種類	説明
ステートメントカバレッジ	ステートメント (文) レベルの網羅
ブランチカバレッジ	条件判定レベルの網羅
コンデイションカバレッジ	条件式レベルの網羅
パスカバレッジ	プログラムの通りうるすべてのパスの網羅

5. 実際の出力結果と期待する出力値を比較する
6. テスト対象ソフトウェアが正常に機能していることを確認する

制御構造を網羅するように作成されたテストケースを実行する事が多く，処理経路の網羅度合いについての基準をカバレッジという言葉で表現する．カバレッジの分類を表 2.2 に示す．

ステートメントカバレッジ

ステートメントカバレッジとはステートメントの実行の網羅率を表現する指数であり，ステートメントが 100% ならテスト対象のステートメントがテスト実施中に少なくとも一回実行されるということを意味する．最も単純なカバレッジとして挙げられる事が多いが，多くの実行パスを省略してしまうため，バグを見逃す可能性もあるカバレッジである．通常の操作外の事態，たとえばディスクの容量不足やネットワークの切断などの例外発生時だけ実行されるようなステートメントをシミュレーションするためのツールなども存在する．

ブランチカバレッジ

すべての条件判定に対して True/false を少なくとも一回ずつ評価するのに十分なテストケースを用意するカバレッジをブランチカバレッジ，あるいは 100% デシジョンカバレッジと呼ぶ．

コンデションカバレッジ

ブランチカバレッジよりさらに条件判定を網羅するカバレッジをコンディションカバレッジと呼ぶ．ブランチカバレッジとの差異として，ブランチカバレッジは条件判定全体の真偽の結果のみを網羅するカバレッジなのに対し，コンディションカバレッジでは条件判定の部分集合の条件判定に対してのすべての真偽の組み合わせ

を網羅するカバレッジになる．

パスカバレッジ

可能なパスすべてを通るカバレッジのことをパスカパレッジと呼ぶ．

ホワイトボックステストの難点として，複雑なソフトウェアになるほどテストパスの組み合わせ数が急激に増えてしまい，実質的に網羅することが困難になること，存在しないパス (パス自体の実装漏れ) に対してアプローチできないこと，テスト実行者にテスト対象ソフトウェアの実装を理解できるだけのプログラミング知識が求められるなどがある．実際には今紹介したカバレッジの中で機能自体の重要度，テストケース作成コストを加味して現実的なテスト内容を作成していくことになる．

ブラックボックステスト

実装の情報に基づいて行われるホワイトボックステストに対して，要件の情報のみを元に行われるテストをブラックボックステストという．ブラックボックステストの一般的な手順は以下の通りである．

- 要件や仕様書を分析する
- 仕様を元に有効，または無効な入力値を選択する．
- それぞれの入力値に対して，期待する出力値は何かを決定する．
- 選択した入力値を使ったテストを作成する
- テストを実行する
- 実際の出力と期待する出力を比較する
- テスト対象ソフトウェアが正常に機能しているかどうかを判定する

ブラックボックステストは内部実装に基づかないテストケースによって実行されるので，ホワイトボックステストのようなカバレッジ的概念が存在しない．可能性のある全ての入力値の組み合わせでテストを実行することは現実的ではなく，そのためテスト対象のソフトウェアがどれだけテストできたかを明確に知る方法はない．しかしながら巨大なソフトウェアでホワイトボックステストで実行するべきパス数が多くなりすぎて手に負えなくなってしまった場合などに，無作為に作成したブラックボックステストで最低限の動作を確認する場合などがある．

グレーボックステスト

ブラックボックステストとホワイトボックステストの間の概念としてグレーボックステストというものがある．このアプローチはテスト対象ソフトウェアがどのように実装されているかどうかをある程度調査し，その知識をブラックボックステストの効率的なテストケース作成に利用するというものである．

2.2.2　テスト対象による分類

前章ではどのようにテストを行うかという分類軸でテストを分類した．ここではどのような対象をテストするかという軸で分類されたいくつかのテストについて説明する．

単体テスト

単体とソフトウェアにおける最も小さな部品のことを指す．プログラミング言語や開発環境によって単体の定義は異なり，関数（メソッド）を単体とする場合もあれば，クラス，ファイルを単体とみなす場合もある．その単体に対して行われるテストが単体テストである．

結合テスト

結合テストは単体テストと比較していくつかの機能が統合して正しく動作することを確認するためのテストである．ソフトウェア全体としての機能テストも一応ここに含まれるが，その場合は後述のシステムテストと呼ばれることが多い．単体テストと結合テストの区別は難しく，機能の側面から見ると単体テストでも，実装の側面から見ると結合テストと考える事もできるということもありうる．

システムテスト，受け入れテスト

これが実質的なすべての機能を統合して行われる全体としてのテストである．システムテストは開発側が行うテストで，受け入れテストは顧客側がそのソフトウェアを受け入れられるかを確認するために行うテストである．ただし，現在のウェブアプリケーション開発においては受注開発ではなく自社開発であることも多く，そ

の場合この 2 つのテストは区別されない.

2.2.3 実行方法による分類

ここではどのような形でテストを行うかという軸で分類されたいくつかのテストについて説明する

動的テスト

動的テストとは，コンパイルされたソフトウェアに対して実際にある操作を行い，その結果を確認することによって行うテストのことである．自動化スクリプトを用いることもあれば，実際に人間に操作させることもある．

静的テスト

静的テストとは，主にソフトウェアのソースコードに対してテストを行うことである．たとえば，単体テストなどの一部の機能のみをテストする場合，対応するクラスのメソッドを呼び出して，予め仕様から導き出した入出力のペアをもとにメソッドに対して引数として渡し，返り値が正しいものであるか確認するというものである．また仕様書などから状態遷移図を作成し，動作を確認するのみのテストのことを同様に静的テストと表現する場合もある．

2.2.4 テスト駆動開発とビヘイビア駆動開発

テストの分類とはまた別に，特に本研究と深い関わりのある，TDD 及び BDD という用語について説明する

テスト駆動開発 (TDD)

通常，上で紹介した通りソフトウェアテストはあくまで実装が終了したソフトウェアに対して行われ，そこで不具合が発見された場合にまた修正のプロセスが発生するが，主にアジャイル開発の現場で用いられている開発のやり方として，先に仕様に基づくテストの方を記述して，そのテストがパスするようなコードを実装し

ていくという開発形態がある．このようなやり方を TDD(Test Driven Develop:テスト駆動開発) と呼ぶ.

このやり方のメリットとして，テストを先に書くことで実装者としての視点ではなく，そのコードの利用者 (ここでの利用者はそのインターフェースを利用する他のコードを記述するプログラマー，という意味でシステム自体のユーザーという意味ではないことに注意) としての視点から効率的な実装を行うことができる，実装のフェーズで満たすべき仕様がソースコードレベルで明らかになっていることにより，設計を考えることなく処理の記述に集中できるといった点が挙げられる.

ビヘイビア駆動開発 (BDD)

上記のテスト駆動開発から更に派生したビヘイビア駆動開発という手法がある．テスト駆動開発で記述されるテストケースは作成したプログラムの動作が正しいかを検証するためのテストであるのに対し，BDD は作成しようとするプログラムに対して期待する「振る舞い:Behavior」を自然言語に近い形で記述するという形の開発形態である．テストコードの可読性が高まることにより，要求仕様を表現するドキュメントとしての側面も期待できるようになるというメリットがある.

2.3 ウェブアプリケーションのテスト記述支援技術

この節では，実際にウェブアプリケーションにおけるテスト記述支援を目的としたいくつかの関連技術について述べる.

ウェブアプリケーションのエンドツーエンドテストと呼ばれる内部動作を確かめることなく表示画面のみを通じて入力に対する最終的な出力を確認するテストに必要技術は大きく分けて Capture&Replay 手法と Programmable 手法に分類できる．ここでこれらの 2 手法について詳しく解説する.

2.3.1 Capture&Replay 手法

Capture&Replay 手法とは専用のツールを用いてウェブブラウザ上の操作を記録 (Captrue) し，それを再び再生 (Replay) することによって記録した操作を再現し，

ウェブアプリケーションの挙動をテストする手法のことである．代表的なツールとしては Selenium IDE [4] がある．しかし，Selenium IDE はブラウザ動作を記録することのみなので，最終的な出力が正しいことを確認するアサーションなどについてはツール使用者が書き加える必要がある．

この手法のメリットとしては最後のアサーション部分を別としてプログラミング言語を記述する必要がないため Programmable 手法よりテストケースを作成する初期コストが低い．その反面デメリットとしては，テストケース自体を保守することができず，HTML 要素などが変更されるたびに再度ブラウザ操作を記録する必要があるため，テストケースの保守コストが大きくなることが挙げられる．

2.3.2 Programmable 手法

Programmable 手法とは，プログラミング言語のソースコードを用いたテスト手法のことを指す．プログラミング知識がない人間がテストコードを記述することは困難であるので，基本的にはプログラマーがテストコードを記述することが前提になる．Programmable 手法は，テストケースの作成コストが Capture&Relay 手法と比較して大きくなる代わりに，テストケースの保守がテストコードの修正のみで済むので保守コストが大きく抑えられるという特徴がある．Maurizio ら [11] は Programmable 手法と Capture&Replay 手法の中で，3 回以上のリリースが行われるソフトウェアでは Programmable 手法を採用するほうがテストにかかる総コストを抑えられると明らかにした．従ってプランナーがテストを書くという本研究の条件を付け加える場合には，Programmable 手法の初期コストにプランナーがプログラミング知識を取得するための時間なども加える必要があると考えられる．

DePoT:ウェブアプリケーションテストコード自動生成フレームワーク

青井ら [18] は頻繁に変更されるウェブアプリケーションの HTML のコードのテスト保守コストを減らすために，ウェブアプリケーションの頻繁な変更に対応可能なデザインパターンを利用した保守性の高い内部 DSL(Domain Specific Language) [12] 及びそれらを用いテストコードの自動生成ツール DePoT を開発した．

DePoT はページオブジェクトパターン [6] に基づいてウェブアプリケーションテストコードを自動生成するフレームワークである．ページオブジェクトパターンとはウェブアプリケーションのテストを行うツール Selenium[4] のプログラマーが考案したウェブページ単位でページ操作をモジュール化し，ページ操作とテストシナリオを分離するというテストコードのデザインパターンである．DePoT では各 HTML ページから HTML 要素を変数に，リンクを遷移を表現するメソッドに変換したクラスファイルを生成する．メソッドチェーン的にページ遷移を表現する内部 DSL によって容易にテストコードを作成することができる．

DePoT で用いられている DSL とはプログラミング言語と比較してより特定のタスクを処理するために特化し，その代わりとして構造がより簡単になった言語のことを指す．代表的な DSL としてハードウェア記述言語の Verilog HDL や，データベースの問い合わせ言語である SQL などがある．表現力が汎用プログラミング言語に劣る代わりに，文法が単純な言語であるため習得が比較的容易であるという特徴がある．

DePoT はあくまでプログラマーがテストコードを記述することを前提にそのコストを削減するという目的のために，テストコード自体も JUnit[9] という Java のテストフレームワークに従ったものになっているなど，プログラミング知識が無い人間が記述するには若干複雑なものになっており，本研究の目的であるプランナーのテストの記述の支援は難しいと思われる．

Gherkin:テスト記述フォーマット

次に自然言語を用いたテスト記述 DSL ライブラリである Gherkin[1] について説明する．Gherkin は大きく分けると自然言語フレーズとそれに対応するウェブアプリケーションに対する操作をまとめた step ファイルと呼ばれるファイルと，その自然言語フレーズを用いてテストシナリオを記述した feature ファイルと呼ばれるファイルで構成される．図 2.1 は step ファイルの例を示し，図 2.2 は feature ファイルの例を示す．図 2.1 の例ではテスト対象のウェブアプリケーションに対する一部の動作の step 及び，その自然言語表現を定義している．図 2.2 の例では事前に作成した自然言語フレーズを組み合わせてテストシナリオを表現している．

step ファイルで定義した自然言語フレーズとライブラリが定義し

```
step 'hoge␣サイトにアクセスする' do
  Capybara.app_host = "http://www.hoge.jp/"
end

step 'トップページを表示する' do
  visit '/'
end

step '画面にようこそと表示されていること' do
# page.should have_content('ようこそ') # should はもう古い
  expect(page).to have_content('ようこそ')
end

step 'id␣と␣password␣を入力する' do
  fill_in 'session_login', :with => 'testuser'
  fill_in 'session_password', :with => 'password'
end
```

図 2.1　step ファイルの例

```
機能: ポータル画面からログイン
  シナリオ: トップページにアクセスしてログインする
    前提 hoge サイトにアクセスする
    もしトップページを表示する
    ならば 画面にようこそと表示されていること
    かつ id と password を入力する
    かつ サインインボタンをクリックする
    ならば 画面にユーザ名 testuser が表示されること
```

図 2.2　feature ファイルの例

た"When"，"Given"，"And"などのキーワードを組み合わせることによって一見自然言語文章のような形でテストコードを作成することができる．これにより，プログラミング知識がない人間でもテストシナリオを理解，記述することができる．しかしその一方で，アプリケーションのソースコードやテストコードに加えて新たに step ファイルの作成，保守コストがプログラマーの負荷になる課題や，自然言語で書かれたテストコードはプログラマーにとっては逆に理解コストが上がってしまうなどの課題がある．

Tellurium

中島らは Capture&Replay 手法の初期コストの低さと Programmable 手法の保守コストの低さという双方の手法のメリットを組み合わせたツール Tellurium を開発した [16]．このツールはウェブブラウザ上で操作を行うたび，その操作に対応するテストコードを挿入する．これにより，プログラミング知識がない人間でもブラウザ上の操作でテストコードを作成でき，かつテストはソースコードの形になっているのでプログラマーが保守できる．

また，Selenium IDE [4] などの Capture&Replay 手法を支援するツールの中にはテストケースをソースコードの形でエクスポートできる機能をもつものもある．この機能との差別化としてはテストコードの生成単位であると筆者らは述べている．Selenium IDE などのテストケースエクスポート機能はテストケースに相当するソースコード以外にも種々の必要な動作（例:モジュールや設定の読み込みなど）が含まれており無駄があるが，提案手法はあくまでブラウザ操作に対応するコード片のみであり無駄がないという点を挙げている．

この手法は Programmbale 手法と Capture&Replay 手法のメリットを両取りしているが，最終的なテストシナリオがソースコードの形になるため，あくまでテストコードの保守はプログラマーの役割となり，本研究が目指すところのプランナーのテスト保守という要求は満たされない可能性がある．

第 3 章　予備実験

ゲームやウェブサービスの開発現場では，プランナーとプログラマーの口頭によるコミュニケーション齟齬によって開発の手戻りが発生する可能性がある．本研究では実際にそのような問題が起こるのかを確認するための予備実験を行った．また予備実験のフィードバックインタビューから得られた仮説を検証するために追加の比較実験として口頭での仕様説明に加えて入出力のサンプルを与えた場合の実験も行った．本章ではこれらの実験の概要及び結果について述べる．

3.1　予備実験の概要

予備実験では，被験者としてはソフトウェア工学を研究している修士課程の学生 3 名を対象とした．被験者のプログラミング知識については若干のばらつきがあったが，事前に簡単なチュートリアルを受講してもらい被験者が本実験に参加するこための十分なプログラムの知識があることを確認した．

実験内容は次の手順で実施された．まず，実験実施者は実験対象の問題として，競技プログラミングサイト AtcorderBeginnerContest(以下 ABC)[2] 第四回の問題 A を実験で使用する仕様として選択した．ABC は初心者向けのプログラミングコンテストとして有名なサイトであり，提出するプログラムの言語も多くの種類が認められている．また，問題 A は簡単な四則演算程度を実装できれば解答できる程度の難易度の問題であり，予備実験で確認するべきである「仕様が正確に伝達されたか」を確認するのには適当な難易度であると考えられる．次に，被験者に馴染みがあるプログラミング言語について簡単な標準入出力についてのチュートリアルを行った後，口頭とホワイトボードによる簡単な図を用いた問題の説明を行い，解答とするプログラムを作成してもらった．完成したと思った段階で被験者から実験実施者にプログラムを提出してもらい，実験実施者が ABC の判定ページに該当プログラムを提出する．プログラムが正解であればそこで実験を終了し，フィードバックインタビューを行った．プログラムが間違っていた際には，誤提出のカウントを 1 つ追加した上でサンプルとなる入力と出力の例を 1 つ示し，提出されたプログラムの該当入力に対する出力との違いのみフィードバックを行い，再び開発に戻って

もらった．実験実施者はプログラミング知識が無いプランナーという設定であるため，実装についてのフィードバックは一切行わなかった．なお，実験中に質問があった場合には，質問に解答した．

3.2 実験に使用した問題

本実験で使用した元のサイトの問題文は以下の通りである．

> AtCoder 社の社員である青木さんの給料は以下のように決められます．ある月に，青木さんがタスクをこなした数を x とします．この月の給料は，1 から x までの整数が 1 面ずつに書かれた x 面ダイスを振って出た目 × 1 万円がもらえます．ただし，このダイスは，どの面が出る確率も等しく 1/x です．青木くんは，暮らしていくのに十分な給料が得られるかどうかが心配で，平均いくら程度給料がもらえるか調べたいです．毎月，青木くんはちょうど N 個のタスクをこなすこととし，毎月の給料の平均値を求めるプログラムを書いてください．

入力

入力は以下の形式で標準入力から与えられる．

N

1 行目には，整数で，青木くんが毎月こなすタスクの数 N (4 ≦ N ≦ 100) が与えられる．

出力

青木くんがもらえる毎月の給料（単位は円）の平均値を 1 行で出力せよ．絶対誤差，または，相対誤差が 10 − 6 以下であれば許容される．また，出力の末尾には改行を入れること．

実験実施者はこの問題をウェブサービスの給料の計算システムの要求として想定し，被験者に説明を行った．

表 3.1　予備実験結果

	被験者 1	被験者 2	被験者 3
正答までの提出回数	3 回	2 回	2 回
質問回数	1 回	0 回	2 回

3.3　予備実験の結果と考察

表 3.1 は予備実験に参加した被験者Ａ，Ｂ，Ｃの予備実験中の正答までの提出回数と質問回数を示す．予備実験で全被験者は 1 回以上の誤提出を行い，1 回目の提出で正解をした被験者はいなかった．

予備実験で，いずれの被験者も 1 回目の提出では正解しなかったことから，口頭での要求伝達において齟齬が発生することが確認できた．また，実験後に行ったフィードバックインタビューで実際の問題文に記述されているサンプルを被験者に見せたところ，「これを見ながらなら最初にしたような仕様の誤解をしなかったと思う」という意見が一部の被験者から得られたので，プランナーが要求と同時に入出力の組み合わせのサンプルを提示することが要求伝達の齟齬を防ぐ方法になるという仮説を立て，その仮説を検証するために追加の予備実験を行った．

3.4　追加予備実験概要

予備実験と同じ実験条件（口頭での問題説明，ホワイトボード）に加えて実際の問題文の入出力の例を与える追加の予備実験を新たにソフトウェア工学を研究している修士課程の学生 3 名（最初の実験を行った学生とは異なる人間）に対して行った．

3.5　追加予備実験の結果と考察

表 3.2 は追加で行われた予備実験に参加した被験者 D，E，F の実験中の正答までの提出回数と質問回数を示す．この表に示す通り，すべての追加予備実験の被験者が誤提出を行うことなく，1 回目の提出で正解のプログラムを提出した．

表 3.2　入出力を加えた追加予備実験結果

	被験者 4	被験者 5	被験者 6
正答までの提出回数	1 回	1 回	1 回
質問回数	0 回	0 回	2 回

この追加で行われた予備実験の結果から，入出力例の提示が仕様の伝達の齟齬の防止について一定の効果を発揮することがわかった．また，追加予備実験のフィードバックインタビューで以下のような意見が得られた．

- 「入出力のサンプルは仕様の伝達に効果はあったか」
  「あった．問題文の説明時には一部の意図が不明瞭だったが，サンプルを与えられたことで意図がつかめた」
- 「与えた入出力のサンプルを用いて自らのプログラムを提出前にデバッグしたか」
  – 「デバッグを行った」
- 「入出力のサンプルを与えていなかった場合，自ら入出力の組を考案し，デバッグしたかと思うか」
  – 「面倒なので，やらなかったと思う」

この結果より，口頭での要求伝達に齟齬が容易に発生しうること，またその齟齬はプログラムに対しての理想の入出力例の提示によってある程度防ぐことができるということがわかった．

第 4 章　提案環境

本章では本研究で提案するプランナーのテスト記述支援システムの概要について解説する．

4.1　提案環境の概要

提案環境の利用者はシステムについての要求を所持しているがプログラミング知識を持っていないプランナーと，要求を受け取った後に，ウェブアプリケーションの追加機能を実装するプログラマーである．本章で提案システム全体の概要と構成モジュールの役割を説明する．図 4.1 にモジュール全体の概要を示す．赤い矢印は本研究で新しく実装する予定部分を表す．

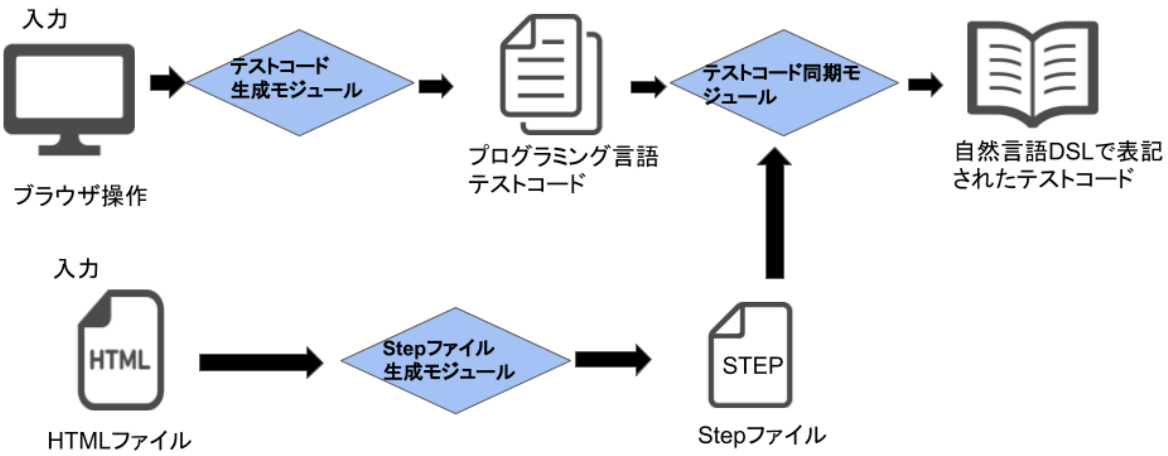


図 4.1　提案手法の概要

- Capture&Replay 機能を持つブラウザ操作を入力としてテストコードを出力とするモジュール
- HTML ファイルを入力として自然言語テストコード生成に必要な Step ファイルを生成するモジュール
- Step ファイルとテストコード (または自然言語テストコード) を入力として自然言語テストコード (またはテストコード) を生成するモジュール

以降，各モジュールの動作について詳しく解説する．

4.2.1 Capture&Replay 手法によるテストコード出力モジュール

Capture&Replay 手法によるテストコード出力モジュールはプランナーのブラウザ操作を入力としてそのブラウザ操作にを再現するテストコードを出力する．このモジュールには既存の Capture&Replay 手法によるテストコード出力ツールの 1 つである Katalon Recorder [10] を使用する．このモジュールの大まかな挙動を図 4.2 に示す．

Katalon Recorder とは Capture&Replay 手法によってテストコードを生成できるツールの 1 つであり，現在はブラウザ拡張機能として提供されている．Capture&Replay 手法の代表的なツールとして挙げた Selenium IDE を用いなかった理由として，現在 Selenium IDE はテストケースをソースコードの形で出力することができず，今回の目的には適さなかったからである．Katalon Recorder の実際の画面を図 4.3 に示す．

4.2.2 HTML 解析による step ファイル作成モジュール

HTML 解析による 2.2 節で説明した Gherkin の step ファイル作成モジュールは本研究で新たに実装するものであり，プランナーが保守する自然言語テストコード作成のための step ファイルを作成するモジュールである．このモジュールは入力として HTML ファイルを受け取り，受け取った HTML ファイルをパースし，中のリンクなどの要素を読み取り 2.2 節で説明した Gherkins の step ファイルを生成する．このモジュールの大まかな挙動を図 4.4 に示す．実装としては，HTML ファ

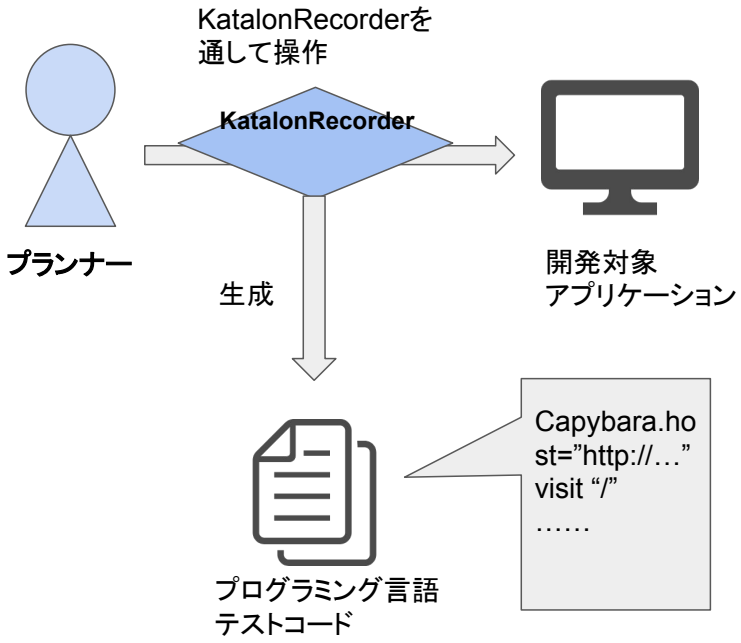


図 4.2　モジュール 1 の図

イルの中からタイトル，内部のテキスト (HTML タグを無視したもの)，リンクやフォームなどの情報を種類分けして，テンプレートとして用意してある step ファイルの形に整形し直したものになる．実際のページとそこから生成された step ファイルの例を図 4.5 記す．

4.2.3　自然言語テストコードとプログラミング言語同期モジュール

自然言語テストコードとプログラミング同期モジュールも本研究で新たに実装するものであり，step ファイルで定義された自然言語フレーズを用いてプランナーに記述された自然言語テストコードとプログラマーが保守するためのプログラミング言語のためのテストコードの内容を同期するモジュールである．このモジュール

の入力として自然言語テストコードを受け取った場合にはプログラミング言語のテストコードを出力し，プログラミング言語のテストコードを入力した場合には自然言語テストコードを出力する．このモジュールの大まかな挙動を図 4.6 に示す．本ツールの GherkinDSL で表記されたテストを実行する部分については Turnip というライブラリを使用しているが，Turnip は step ファイル上の動作を Ruby の Proc オブジェクトとして保持するため，対応関係を別途保存する必要があった．例として，GherkinDSL で表記されたテストとそこから生成された対応するテストコードを図 4.7 と図 4.8 に示す．

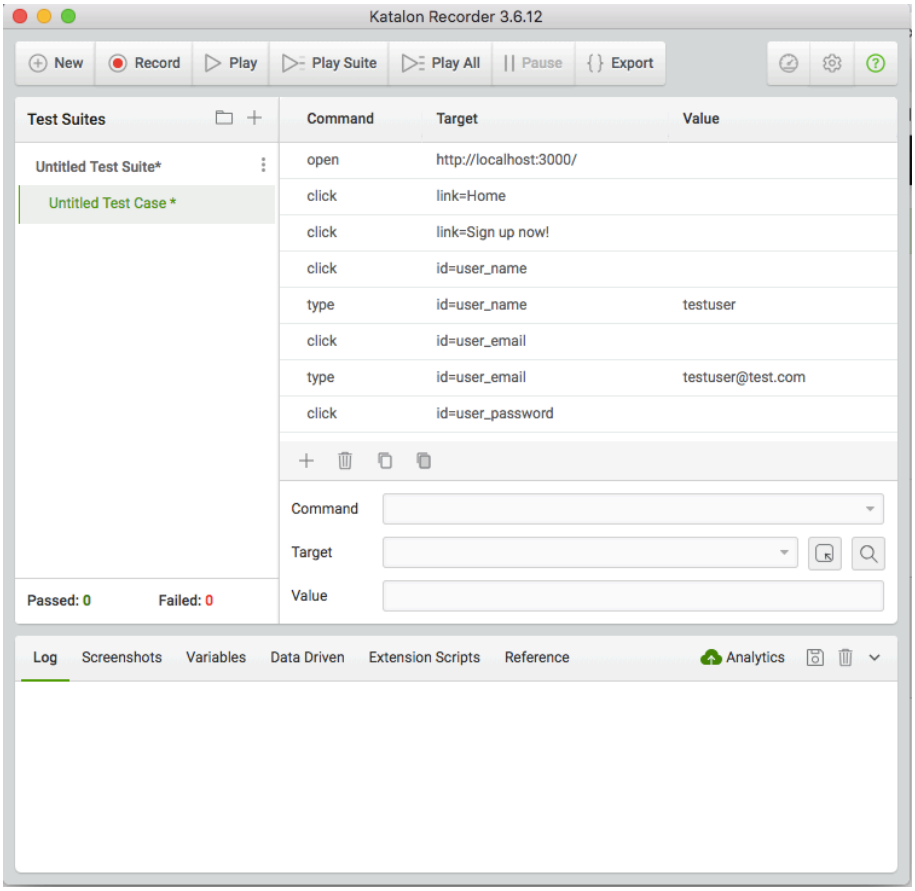
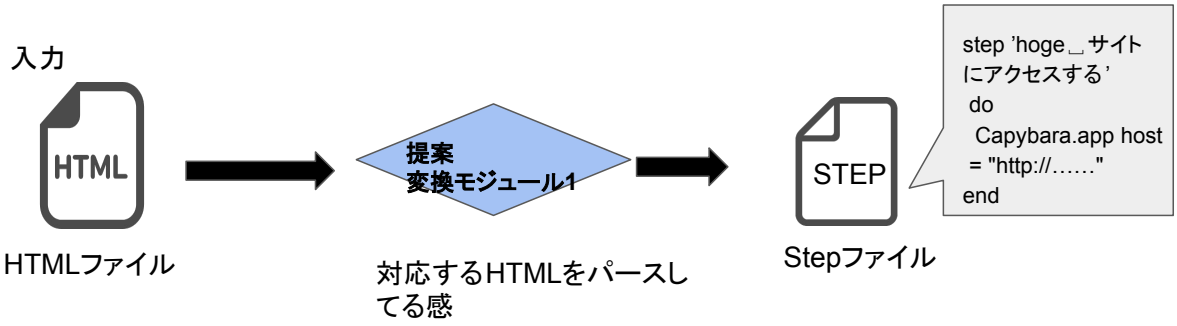


図 4.3　Katalon Recorder の画面



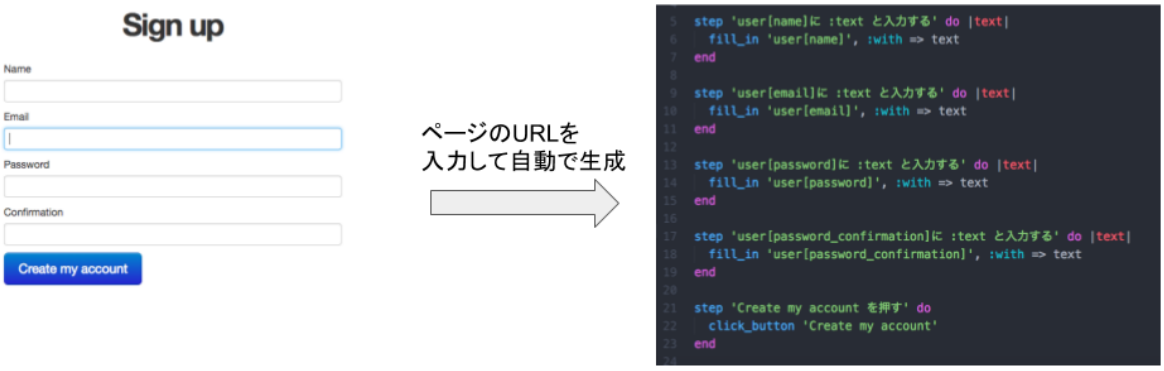


図 4.5　HTML の入力フォームから step ファイルの生成例

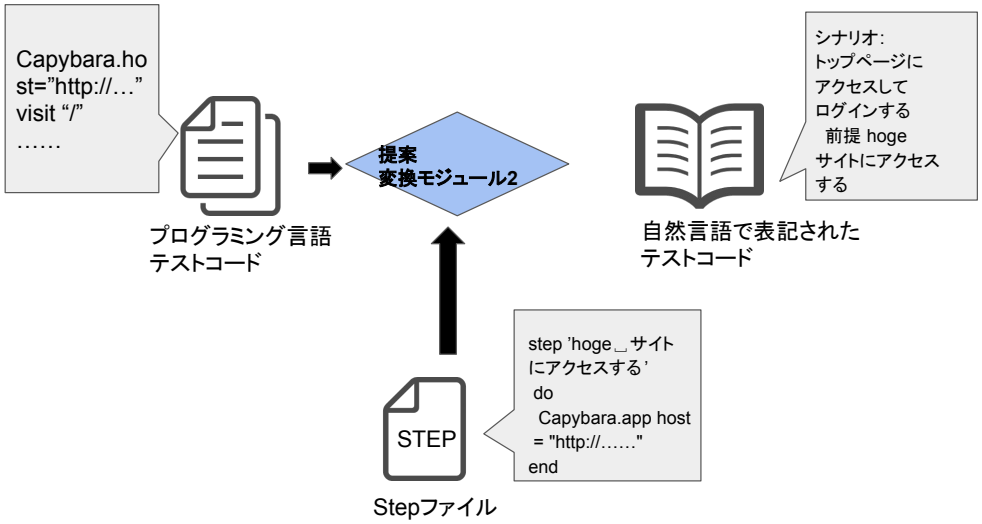


図 4.6　モジュール 3 の図

```ruby
#機能: サインアップ機能の確認;サインアップしてみる
describe 'サインアップしてみる' do
  it 'サインアップページにアクセスする -> user[name]に %testuser% と入力する ->
  user[email]に %testuser@hoge.com% と入力する -> user[password]に %testpassword% と入
  力する -> user[password_confirmation]に %testpassword% と入力する -> Create my
  account を押す -> testuserと表示されている' do
      visit signup_path
      fill_in 'user[name]', :with => 'testuser'
      fill_in 'user[email]', :with => 'testuser@hoge.com'
      fill_in 'user[password]', :with => 'testpassword'
      fill_in 'user[password_confirmation]', :with => 'testpassword'
      click_button 'Create my account'
      expect(page).to have_content("#{'testuser'}")
  end
end
```

図 4.7　対応するテストコードの例

図 4.8　対応する GherkinDSL の例

第 5 章　実験

5.1　実験概要

本章では提案環境の有用性を評価するための実験とその考察について述べる．

提案環境の主要な目的の 1 つとしてプランナーからプログラマーへの仕様伝達の際に発生する齟齬を防止することにより，開発全体の効率に貢献することがある．本ツールの開発現場への貢献度を計測するために，実際のウェブアプリケーション開発を模した実験用ウェブアプリケーションに対して仕様変更を伝達して実装してもらうという実験を，口頭のみで仕様を伝達するケースと，口頭に加えて提案環境によって作成したテストの結果によって仕様を伝達するケースのそれぞれに分けて行った．

評価についてはプランナーがプログラマーに対して仕様をプランナーの意図通りにアプリケーションを仕様変更を実装し終えるまでの作業所要時間及び齟齬の発生によってプランナーを拘束した時間を計測し，作業効率における貢献を評価した．

実験用のウェブアプリケーションは Rails Tutorial[7] という Ruby 言語の著名なウェブアプリケーション開発フレームワーク Ruby on Rails を学ぶためのサイトにおいて開発題材とされているアプリケーションを用いたが，実験にあたって Ruby on Rails やその他のウェブアプリケーション開発関連技術の経験，知識を必要としないようなタスク内容にし，また実験開始前にチュートリアルを行った．これは，実験の評価軸として所要時間を用いているため，Ruby on Rails や，ウェブ開発の経験，知識による差が出ないようにするためである．チュートリアルに用いた画面を図 5.1 に示す．

その他の環境についても実験に用いた PC の環境はすべて被験者間で統一されるよう，同一の PC 環境を用いて実験を行った．行った変更を確認するために開発対象アプリケーションを操作した場合の動作速度，またテストの実行速度は，実験の評価基準である作業時間に影響を与え，また使用 PC の性能に依存すると考えられる．そのため，被験者には同一の PC の上で用意された同一の仮想マシン上で作業を行ってもらい，環境の差によって結果に影響が出ないように配慮した．表 5.1 は実験環境をまとめたものである．

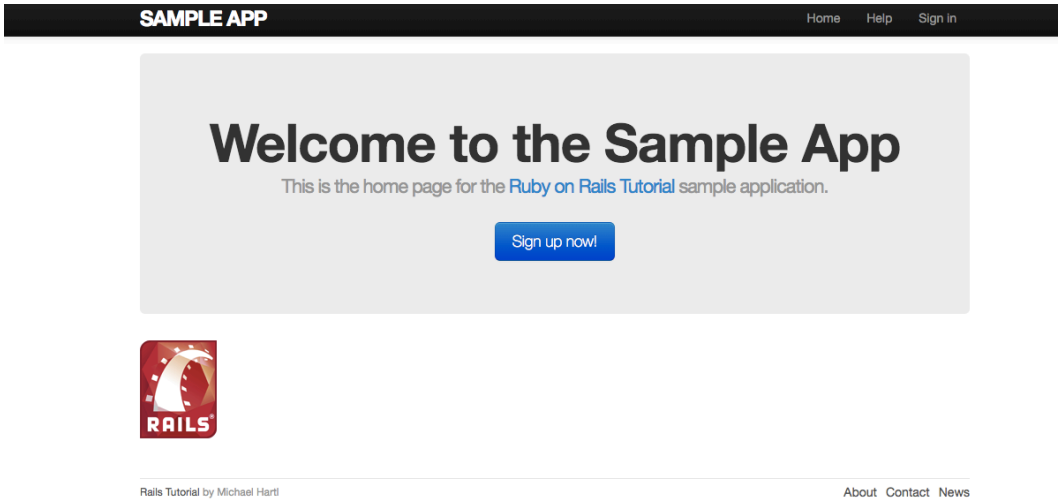


図 5.1　実験対象アプリケーション

また，提案環境の有用性を評価するため，以下のような評価軸を設けた．

1. プランナーへのフィードバック時間を含めた総合的な合計所要時間
2. プランナーのフィードバックの所要時間

1 については，提案環境の総合的な作業効率への貢献を評価するために設けた．2 については，提案環境の齟齬発生防止への貢献を評価するために設けた．

表 5.1　実験環境

名称	内容
OS	Windows10 Enterprise
CPU	Intel(R) Core(TM) i7-5600U CPU2.600GHz 2.59GHz
メモリ	8.00GB(7.88GB 使用可能)
仮想マシン OS	Ubuntu 16.04
仮想マシン CPU コア	1
CPU 使用率制限	100%
仮想マシンメモリ	1024MB

5.2 実験詳細

提案環境による仕様伝達の際の齟齬防止，開発効率上昇における貢献を評価するために，ウェブアプリケーションに対する仕様変更を対象とした評価実験を行った．

実験は口頭のみで仕様を伝達する場合と，口頭に加えて提案環境を用いて作成したテスト結果を与えた場合の 2 つのケースに分け，最終的に実験対象アプリケーションが実験者の意図通りとなるまでの所要時間および，プランナーのフィードバック時間を比較した．仕様変更の題材には，Rails Tutorial（引用）という Ruby 言語の著名なウェブアプリケーション開発フレームワーク Ruby on Rails を学ぶためのサイトにおいて開発題材とされているアプリケーションを用いた．実験は口頭のみで伝達するケース，口頭とテストを用いるケースの順で行った．

5.2.1 実験設定

ここでは，実験時に行った手順について詳しく説明する．まず，実験および研究の意義を理解してもらうために被験者に対して簡単な研究の説明を行った．その後，実験対象アプリケーションの開発に関して，どのファイルを変更すればどのページに変更が反映されるか，実際に変更を行ってもらいその内容がアプリケーションに反映されるか確認してもらうというチュートリアルを行い，被験者間の認識及び，実験対象ウェブアプリケーションについての知識を統一した．

その後，実験の開始前に，実験の重要な趣旨である，今回の実験では実験中のプランナーへの質問を認めていないということを説明し，あくまでプランナーから口頭で与えられた仕様に対して自分なりの 1 つの理解を実装してもらった上でチェックを申請することのみを認めているということを説明した．

チェックの際に，もしプログラマーの作業内容がプランナーの意図と異なっていた場合には，修正内容をフィードバックした後再び開発に戻ってもらうというのを最終的にプランナーの意図通りに仕様変更が実装されるまで繰り返し，最初に仕様を伝えてから開発が終了するまでの時間，およびプランナーをフィードバックのために拘束した時間を計測した．実験に使用した修正内容は，以下のようなものである．

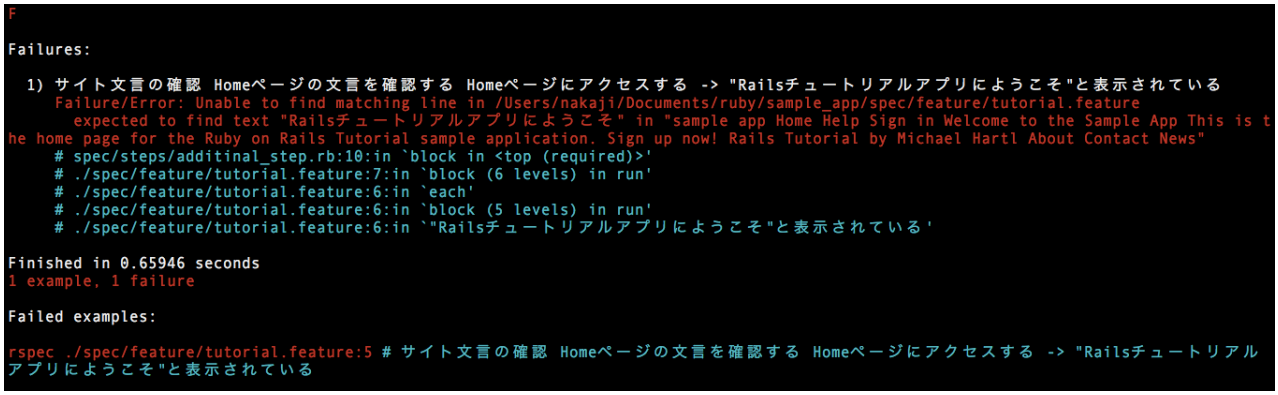


図 5.2　テストに落ちた場合のエラーメッセージ

アバウトページに「ここはアバウトページです.」と書いておいてください.

正解意図としては対象ページの本文の最初に，上の文章のように「アバウト」とカタカナ表記にした上で追加するというものにし，それ以外のケースにチェックを要請された場合には，その正解意図との差分を説明した.

口頭のみのケースが終了した後，テスト情報を与えるケースを実施する前に，テストの実行方法，およびテスト結果の解釈のやり方について改めてチュートリアルを行った．このタイミングで説明とチュートリアルを行ったのは，口頭のみのケースを行っている間にテストの実行方法や解釈のやり方についての記憶が消失しないようにするためである．チュートリアルの内容として，まず予め用意してあった実験対象アプリケーションのホーム画面に対するチュートリアル用のテストを実行してもらい，それが落ちることを確認してもらった後，テストのエラーメッセージの解釈方法を伝え，実際にそのテストにパスするような変更を行ってもらった．エラーメッセージのスクリーンショットを図5.2 に示す.

なお，実験で実際に使用した修正内容は以下のようなものである.

ヘルプページに「ここは Help ページです.」と書いておいてください.

実験 1 とほとんど同じ内容の文言だが，こちらは対象ページの本文の末尾に，上の文章のように「Help」と英語表記にした上で追加するというものにし，実験 1 の影響を極力除くようにした.

これにより，２つ目の実験の開始時点で被験者はテストが失敗した場合のエラーメッセージの解釈方法が理解できているということになる．テストの実行チュートリアルが終わった後，残りの条件は 1 つ目の実験と同じと伝え，同様に仕様を口頭で伝え，実装を開始してもらった．

２つの実験が終了した後，被験者にアンケートに回答してもらった．回答項目を以下に示す．

- Ruby on Rails を用いた開発経験について
- プログラミング経験年数 (言語を問わない)
- チームの共同開発経験
- 提案環境を使用しなかった場合 (実験 1) の要求の理解度
- 提案環境を使用した場合 (実験 2) の要求の理解度
- 自由記述

5.2.2 実験結果

各被験者に仕様を伝達してからプランナーの意図通りに実装が完了するまでの所要時間，およびプランナーの拘束時間を表 5.2 に示す．また，それらの平均時間も改めて表 5.3 に示す．

5.2.3 考察

口頭のみで仕様を伝達した場合と口頭に加えてテストを与えた場合の齟齬防止への貢献，また開発効率への貢献について考察する．またいくつかのアンケートの結果にも触れる．

今回の実験結果の有意差についての確認については, K-S 検定*, F 検定†, Student の t 検定‡,Welch の t 検定§，Wilcoxon の順位和検定を用いた．検定のワークフ

*R の ks.test() を使用
†R の var.test() を使用
‡R の f.test() を使用
§R の wilcox.test() を使用

ローは藤原 [8] の論文で解説されているワークフローに従い，まず K-S 検定にて分析対象が正規分布に従っているかを検定したのち，F 検定で 2 郡の分散の有意差を検定し，Student の t 検定にて平均値に有意差があるかの検定を行った．実際の検定結果については実験結果の節で解説する．検定のワークフローを図 5.3 に示す．

最初にまず所要時間の統計的有意差について確認した．口頭のみのケースとツールを用いた場合で，平均所要時間については口頭のみのケースは 2 分 43 秒，テストを与えたケースについては 2 分 51 秒となり，ほぼ変わらない結果となった．K-S 検定の結果，両方のケースについて正規分布に従うことがわかった ($p = 0.9249, 0.9294 > 0.05$)．したがって次に F 検定を行い等分散性を確認したところ，$p = 0.8517 > 0.05$ となり，等分散であることが確認できた．

次に Student の T 検定を用いて統計的有意差を確認する．Student の t 検定の結果，$p = 0.8001 > 0.05$ となり，所要時間の平均値に統計的有意差はなく，有意傾向 ($p < 0.10$) も存在しないという結果になった．

次にプランナーの拘束時間についても平均値を比較した．口頭のみのケースに

表 5.2　実験結果

被験者名	口頭	口頭 (プランナー)	口頭 + テスト	口頭 + テスト (プランナー)
A	5 分 10 秒	1 分 6 秒	4 分 39 秒	9 秒
B	5 分 2 秒	47 秒	3 分 5 秒	12 秒
C	2 分 13 秒	43 秒	3 分 21 秒	7 秒
D	3 分 45 秒	42 秒	2 分 53 秒	4 秒
E	2 分 56 秒	40 秒	2 分 15 秒	6 秒
F	1 分 13 秒	20 秒	2 分 19 秒	5 秒
G	1 分 44 秒	29 秒	2 分 25 秒	8 秒
H	1 分 40 秒	30 秒	1 分 39 秒	4 秒
I	2 分 57 秒	31 秒	1 分 4 秒	2 秒
J	2 分 34 秒	39 秒	1 分	3 秒
K	3 分 52 秒	48 秒	5 分 2 秒	5 秒
L	36 秒	5 秒	2 分 32 秒	4 秒
M	1 分 39 秒	36 秒	4 分 55 秒	9 秒

表 5.3　実験結果

	口頭	口頭 (プランナー)	口頭 + テスト	口頭 + テスト (プランナー)
平均	2 分 43 秒	36 秒	2 分 51 秒	6 秒

おいてプランナーの拘束時間の平均は 36 秒，テストを与えた場合の平均は 6 秒となった．これについても同様に統計的有意差を確認したところ，p= 6.932e-07 < 0.05 となり，こちらは明らかに統計的に有意差が存在する結果となった．

まず，検定結果の考察として，プログラマーの開発効率に対しては口頭だけのケースもテストケースを利用したケースについても変わらないということが考えられる．これは，最初に口頭で仕様を伝達したときに発生したいくつかの齟齬に対して，試行錯誤を行い，最終的にプランナーの意図した通りの仕様に到達するというプロセスについては変化しないため，単純にその作業に必要な時間が所要時間として表れていると考えられる．

また，今回の実験は用意した PC の上の仮想環境上で行ったため，テストの実行に数秒程度の時間を必要とし，それが若干の実験 2 の所要時間の増加につながっていると考えられる．全体の所要時間と比較して，プランナーの所要時間については有意に削減傾向が見られた．基本的にテストを与えたケースについてはどれも数秒，つまり最終確認にしかプランナーの時間を使用していない結果となっている．

実際の開発現場では開発チーム内にエンジニアは複数いることが普通であり，それらの確認の時間を有意に削減できることは開発効率に大幅な貢献ができると考える．アンケートの結果として，提案環境の開発における貢献は概ね認められたものの，エラーメッセージの読みづらさなどの指摘がいくつかあった．実際実験中の被験者の様子を観察していても，エラーメッセージの大まかな内容は解釈できていても細かな表記の違い，また綴りミスなどの問題に時間を取られてしまい，スムーズに実装を終える事ができないケースが観測された．

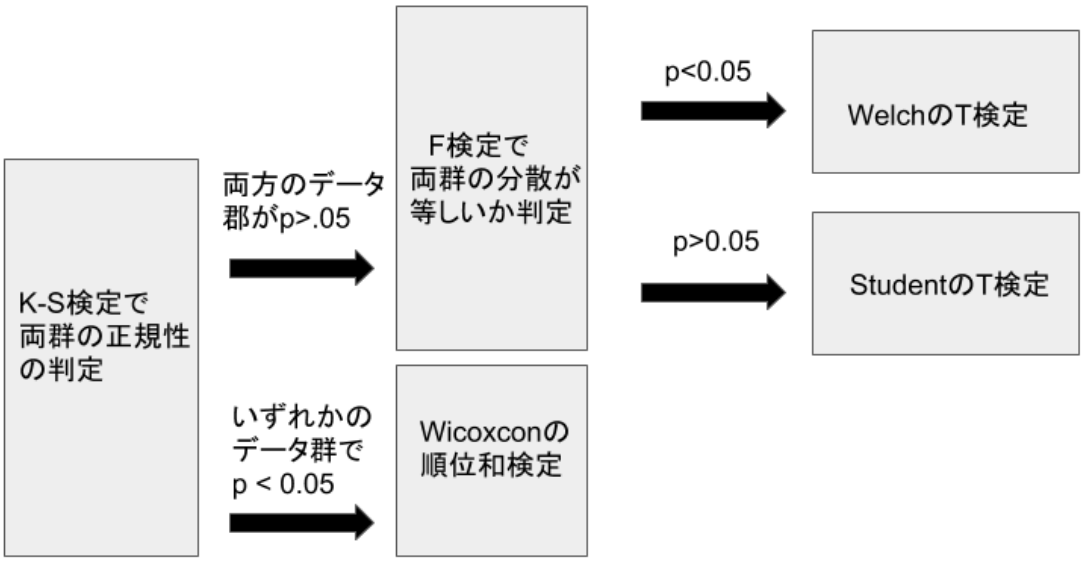


図 5.3　検定のワークフロー図

第 6 章　まとめ

本論文ではまず，ウェブアプリケーション開発において説明し，その早い開発サイクルの中で支持されているアジャイル開発という開発手法の中で発生している，プランナー・プログラマー間の仕様伝達時の齟齬発生問題について説明し，実際に予備実験を行いそのような問題が発生しうることを確認した．また，その問題に対してプランナーがテストを記述し，テストコードを通じて仕様を伝達することにより齟齬を防止するアプローチを考案し，既存のテスト記述支援ツールの課題を解決した要求獲得支援環境を提案，その中のいくつかのモジュールを開発した．実際に開発環境が齟齬を防止することができるか，開発全体の効率に貢献できるかについて評価実験を行い，その結果提案環境が手戻りの指摘によるプランナーの工数を有意に削減できることを確認した．

今後の課題として，テストの表現力の向上が考えられる．現在提案環境で利用している Gherkin ツールは HTML 構造に対してそこまで対応しておらず，テストにパスすることが必ずしもプランナーの意図通りとならないケースが存在し，実際実験でもそのような提出が行われた事があった．これは利用しているライブラリ自体の課題でもあるが，簡易な記述を維持しつつより表現力を向上させることによって，さらなる齟齬の防止に貢献できると考える．

謝辞

本研究を進めるにあたり，多くの方々にご指導，ご協力，ご支援を頂きました．奈良先端科学技術大学院大学情報科学領域 ソフトウェア設計学研究室 飯田 元 教授には熱心なご指導を賜りました．研究に関すること以外にも研究室生活の中で色々とご助言を頂きました．ありがとうございました．

奈良先端科学技術大学院大学情報科学領域 ソフトウェア設計学研究室 市川 昊平 准教授には，様々な場面で本研究に対し貴重なご指導を賜りました．ありがとうございました．

奈良先端科学技術大学院大学情報科学領域 ソフトウェア設計学研究室 崔 恩瀞 助教授には，特に研究内容や論文執筆，発表練習にあたり，非常に丁寧かつ親切なご指導を多く頂きました．ありがとうございました．

名古屋大学大学院情報学研究科 附属組込みシステム研究センター 吉田 則裕 准教授には，研究の方針や，発表資料について多くのご助言を頂きました．ありがとうございました．

小川 暁子 氏，仲田 綾奈 氏には，研究室秘書として，研究の遂行に必要な事務処理など，多岐にわたり御助力頂きました. 心より感謝申し上げます.

また奈良先端科学技術大学院大学情報科学領域 ソフトウェア工学研究室，ディペンダブルシステム学研究室，大阪大学大学院情報学研究科ソフトウェア工学講座の学生の皆様には実験について多大なる協力を頂きました．ありがとうございました．

奈良先端科学技術大学院大学情報科学領域ソフトウェア設計学研究室の皆様には日頃より多大なご協力，ご助言ならびに楽しい交流をいただきました．皆様に支えられて充実した研究生活を送ることが出来ました．ありがとうございました．

またその他研究生活の間で話した友人や大学時代のサークル関連の皆様，その他知人の皆様についても研究についての相談や，日常の気分転換などに付き合って頂き，ありがとうございました．

参考文献

[1] Gherkin. https://docs.cucumber.io/gherkin/.

[2] AtCoder Inc. Atcorder. https://atcoder.jp/?lang=ja.

[3] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Agile manifest. https://agilemanifesto.org/iso/ja/manifesto.html.

[4] A. Bruns, A. Kornstadt, and D. Wichmann. Web application tests with selenium. *IEEE Software*, Vol. 26, No. 5, pp. 88–91, 2009.

[5] Alessio Ferrari, Paola Spoletini, and Stefania Gnesi. Ambiguity cues in requirements elicitation interview. In *Proceedings of the IEEE 24th International Requirements Engineering Conference*, pp. 56–65, 2016.

[6] Martin Fowler. Pageobject. https://martinfowler.com/bliki/PageObject.html.

[7] Michael Hartl. Rails tutorial. https://railstutorial.jp/.

[8] 藤原雄介. 分散型版管理システムにおける コードクローンの一貫した修正支援ツール. Master's thesis, 奈良先端科学技術大学院大学情報科学研究科, 2014.

[9] JUnit Team (2018). Junit: Junit 5. https://junit.org/junit5/.

[10] Katalon. Katalon studio. https://www.katalon.com/.

[11] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proceedings of the 20th Working Conference on Reverse Engineering*, pp. 272–281, 2013.

[12] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, Vol. 37, No. 4, pp. 316–344, 2005.

[13] Publickey. Amazon は 1 時間に最大 1000 回もデプロイする。クラウドネイティブなデプロイとはどういうものか？ aws re:invent 基調講演

(day2 am). https://www.publickey1.jp/blog/12/amazon11000_aws_reinventday2_am.html.

[14] Wikipedia. ウェブアプリケーション. https://ja.wikipedia.org/wiki/%E3%82%A6%E3%82%A7%E3%83%96%E3%82%A2%E3%83%97%E3%83%AA%E3%82%B1%E3%83%BC%E3%82%B7%E3%83%A7%E3%83%B3.

[15] 宗雅彦ソープ・コートランド. はじめて学ぶソフトウェアのテスト技法. 2005.

[16] 中嶋学, 高田喜朗. Web アプリケーションを対象としたユーザー操作の補足による gui テストコードの部分補完システム. 電子情報通信学会技術研究報告 SS2016-45, 電子情報通信学会, 2017.

[17] 大月美佳ほか. ソフトウェアテストの最新動向: 6. test-driven development (テスト駆動開発) 開発手法としてのテスト. 情報処理, Vol. 49, No. 2, pp. 162–167, 2008.

[18] 青井翔平, 坂本一憲, 鷲崎弘宜, 深澤良彰. Depot：web アプリケーションテストにおけるテストコード自動生成テスティングフレームワーク. 情報処理学会論文誌, Vol. 56, No. 3, pp. 835–846, 2015.