Gradleビルドスクリプトの検証を目的としたテストライブラリの提案

藪下 友[1]　柗本 真佑[1]　楠本 真二[1]

概要：ソフトウェア開発においてビルドツールが広く利用されている．ビルドツールは，ソースコードから実行ファイルや配布パッケージなどを生成するビルド工程を自動化する．Gradle は Java で広く用いられるビルドツールの 1 つであり，ビルドスクリプトと呼ばれるファイルに基づきビルドを実行する．ビルドスクリプトは Groovy DSL で記述されたソースコードであり，一般的なプログラミング言語のソースコードと同様に継続的な自動テストを実施すべきである．しかし，Java ソースコード内に存在し得るビルド失敗要因や，ビルドスクリプトの作用がメモリ上の変数にとどまらず多岐に渡ることから，ビルドスクリプト自体のテストは容易ではない．本研究では，ビルドスクリプトの検証に特化したテストライブラリを提案する．本ライブラリは，検証のためのスタブとアサーションメソッドを提供する．スタブは，Java ソースコードの代替としてビルド失敗要因を隠蔽する．またアサーションメソッドは，ビルドスクリプトの作用に特化し検証を行う．評価実験及びケーススタディの結果，対象プロジェクトの 8 割以上の適用に成功し，テストによりビルドスクリプトに実在するバグを検出可能であることを確認した．

1. はじめに

ソフトウェア開発においてビルドツールが広く利用されている [1]．ビルドツールは，ソースコードから実行ファイルや配布パッケージなどを生成するビルド工程を自動化する．Java で広く用いられるビルドツールの 1 つして Gradle が挙げられる．Gradle は，`bulid.gradle` と呼ばれるビルドスクリプトに基づきビルドを実行する．ビルドスクリプトを用いることで，依存関係の解決やコンパイル，テストなどのビルド工程全体の自動化が可能となる [2]．

ソフトウェア開発におけるビルド失敗の原因を調査する研究が数多く行われている [3][4]．また，ビルドスクリプトの記述に基づきビルド工程における影響やバグを検出する研究も多く存在する [5][6]．これらの研究では，外部依存ライブラリの宣言不足やビルド結果であるファイルの欠如，タスク実行順序の誤りなど，ビルドスクリプトが原因となるビルド失敗やバグの存在が指摘されている．したがって，ビルド工程の保守においてビルドスクリプトの動作が期待通りであるかを検証することが重要である．

Gradle のビルドスクリプトは Groovy DSL で記述されたソースコードである．さらに，ビルドスクリプトのコードレビューでは一般的なプログラミング言語のソースコードに比べて，欠陥に関するコメントの割合が高い [7]．よって我々は，一般的なプログラミング言語のソースコードと同様に，ビルドスクリプトに対する継続的な自動テストを実施すべきだと考える [8]．

しかし，ビルドスクリプトのテストは通常の単体テストなどと違い容易ではない．ビルド対象となる Java ソースコード内にランタイムエラーなどの実行時エラーが含まれると，ビルドスクリプト自体に問題がなくてもビルド全体が失敗する．そのため，ビルドスクリプト以外のビルド失敗要因を隠蔽したうえで，純粋にビルドスクリプトのみを検証できるテスト手法が必要となる． また，ビルドスクリプトの作用はメモリ上の変数にとどまらず，ファイルシステムや標準出力，エラー出力，リターンコードなど多岐にわたる．そのため，様々なビルドスクリプトの作用に応じた検証手法が必要となる．

本研究の目的は，Gradle のビルドスクリプトに対する自動テストの実現である．そのために，ビルドスクリプトの検証に特化したテストライブラリを提案する．本ライブラリは，検証のためのスタブとアサーションメソッドを提供する．スタブは，Java ソースコードの代替としてビルド失敗要因を隠蔽し，開発者の期待通りのビルド結果を維持する．これにより，純粋なビルドスクリプトの検証が可能となる．一方アサーションメソッドは，ビルドスクリプトの作用に特化しファイルの取得やログの抽出などを行う．これにより，ビルドスクリプトの作用を容易に検証できる．

[1] 大阪大学大学院情報科学研究科 大阪府吹田市

提案手法の有効性を評価するために評価実験及びケーススタディを実施した．その結果，対象プロジェクトのおよそ 8 割以上に適用可能であることを確認した．またケーススタディを通じて，テストによりビルドスクリプトに実在するバグを検出可能であることを示した．

2. 関連研究

ビルド失敗に関する原因を調査した研究として，Lou らは [9]，Stack Overflow のビルド関連質問 1,080 件を解析し，ビルドエラーの原因を 50 のカテゴリに分類した．エラー症状として，ビルドスクリプトが起因となる依存関係エラー（19.6%）や構文エラー（15.5%）などが特に多く，大半は依存関係の追加などのビルドスクリプトの修正によって解決可能であると示した．また Rausch らは [10]，14 の OSS Java プロジェクトの CI ログを調査し，分類した 14 の失敗原因においてビルドスクリプトが起因となるビルド失敗が 4 番目に多いことを示した．

ビルド工程に存在する欠陥を検出する研究として，Sotiropoulos らは [11]，増分ビルド及び並列ビルドを実行する上でのビルドスクリプトの欠陥の自動検出ツールを提案した．提案手法は，ビルドスクリプトの記述内容とビルドツールによる実際のファイル操作を比較することにより，ビルドスクリプトの入出力や依存に関する宣言誤りを検出できる．これを用いて，47 の OSS プロジェクトに存在する 247 件の欠陥を報告した．また Hassan らは [12]，ビルドスクリプトに存在する欠陥の自動修正ツールを提案した．提案手法は，Gradle ビルドスクリプトに対する修正履歴から修正パターンを解析し，新たなビルド失敗に対しても類似する修正パターンを適用することで自動修正を可能にする．これを用いて，24 件の再現可能なビルド失敗のうち 11 件（45%）を自動修正できた．

これらの既存研究では，ビルドスクリプトに起因するビルド工程の欠陥の分類や検出が行われている．しかし，これらは増分・並列ビルド特有の欠陥や明示的なビルドエラーの検出を目的としており，ビルドスクリプトの作用が開発者の期待通りでない欠陥は検出できない．Gradle のビルドスクリプトはソースコードであるため，ソースコードの動作が開発者の期待通りであるか検証するためには，開発者自身が検証項目を定義したテストを行うべきである [8]．そのため，本研究ではビルドスクリプトの検証に特化したテストライブラリを提案し自動テストを可能にする．

3. 提案手法

本稿では，Gradle のビルドスクリプトに対する自動テストの実現を目的として，ビルドスクリプトの検証に特化したテストライブラリを提案する．提案ライブラリは，以下の 2 点から構成される．

スタブ
: Java ソースコードの代替となり，ビルド失敗要因を隠蔽し開発者の期待するビルド成果物を維持する．

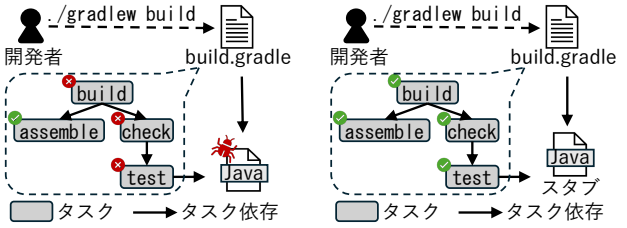
アサーションメソッド
: ビルド成果物に特化して検証を行う．

なおここでのビルド成果物には，Java のコンパイル結果であるクラスファイルや，それらをまとめた JAR ファイル，Maven ローカルに公開されたファイル，ビルド時のログ，及びリターンコードが含まれる．

3.1 Java ソースコードのスタブ

Java ソースコード内のビルド失敗要因を隠蔽し，開発者の期待するビルド成果物を維持するために，Java ソースコードの代替となるスタブを提案する．図 1 に提案スタブを用いた `build` タスクの検証例を示す．図の通りビルド工程は複数のタスクから構成され，各タスクは依存関係を持つ．そのため図 1(a) のように，Java ソースコードにランタイムエラーなどのビルド失敗要因が存在した場合に，それに依存する `test` タスクが実行時エラーにより失敗し，依存関係を通じて `build` タスクも失敗してしまう．これを防ぐために `test` タスクなどの一部のタスクでは，ビルドスクリプトのタスク宣言部に `ignoreFailures=true` を追記することにより，`test` タスクが失敗したとしてもそれに依存する `build` タスクを強制的に続行できる．しかし，Java ソースコードへ依存する全てのタスクで利用できるわけではなく，テスト対象であるビルドスクリプトにビルド失敗要因が存在している場合でも失敗せず続行してしまう．そのため，直接的に Java ソースコード内に存在し得るビルド失敗要因を隠蔽する必要がある．そこで，ビルドスクリプト以外の要因によるビルド失敗を防ぐため Java ソースコード全体をスタブ化する．スタブは通常の結合テストにおいて，未完成や複雑な下位モジュールを擬似的に再現し，上位モジュールの検証に焦点を置くために用いられる．提案手法のスタブでは，ビルド工程でタスクの入力となる Java ソースコードを擬似的に再現する．このスタブを入力にとることにより図 1(b) の通り，ソースコード中の欠陥を隠蔽化しビルドを成功させる．これにより，ビルドスクリプトの検証に焦点を置くことができる．また，スタブを利用する副次的な利点としてビルド実行時間の短縮が挙げられる．後述するが，スタブは元の Java ソースコードと比べてビルドスクリプトそのものの検証に不要な一部の構文要素（メソッドやコンストラクタなど）が除去され簡略化される．このような簡略化されたプログラムを用いてビルドを実行することにより，元のプログラム比べてビルド実行時間の短縮が見込まれる [13]．

スタブ作成の方針は，ビルドスクリプトを検証するために，Java ソースコード中のエラー要因を隠蔽することに



(a) Java ソースコードの欠陥により build タスクが失敗　(b) スタブにより Java ソースコードの欠陥を隠蔽化

図 1: スタブを用いた build タスク検証例

ある．しかし，ビルドスクリプトの作用はクラスファイル等のビルド成果物に反映されるため，これに影響を与えない範囲で隠蔽化を行う必要がある．具体的には，Java Language Specification（JLS）における構文記法（Chapter 19. Syntax[*1]）を参考に，Java ソースコード内の成果物に影響する構文要素を保持し，それら内部の影響のない要素を全て除去した新たな Java ソースコードをスタブとして作成する．特に，参照不整合によるコンパイルエラーを防ぐため，可能な限りクラスや外部ライブラリの参照を除去し隠蔽する．ここで，この処理によりソースコード内に含まれるコンパイルエラーが隠蔽され検出されない可能性がある．しかしテスト対象はビルドスクリプトであるため，ソースコードに起因するエラーは検出対象でない．そのため実行時エラー同様に，ビルド失敗要因であるコンパイルエラーも隠蔽されて問題はない．この方針に基づき，スタブ作成における主要な構文要素ごとの処理規則を表 1 に示す．これらの処理規則は対象要素がボディを持つ場合再帰的に適用される．したがって，クラス宣言（`class`，`enum`，`record`）などは，成果物としてクラスファイルを生成するため保持されるが，その内部のフィールドや初期化ブロック，コンストラクタなどは余計な参照やエラーの原因となるため一律で除去する．また，メソッドについては，実行可能 JAR のエントリポイントである `main` メソッドと，テストレポート生成に関わる`@Test` が付与されたテストケースのみを保持する．

3.2　ビルド成果物を検証するためのアサーションメソッド

ビルドスクリプトの成果物に対する検証を容易にするため，それに特化したアサーションメソッドを提案する．アサーションメソッドの実現方法は様々だが，本稿では従来よりも可読性の高い記述を可能とする AssertJ[*2]を基盤として実装する [14]．さらに，本ライブラリでは成果物を抽象化した `BuildArtifact` オブジェクトを提供する．図 2 は `BuildArtifact` が持つ成果物例を示しており，階層構造で成果物を保持する．そのため，`BuildArtifact` 直下

[*1] https://docs.oracle.com/javase/specs/jls/se24/html/jls-19.html
[*2] https://joel-costigliola.github.io/assertj/

表 1: スタブ作成における主要な構文要素ごとの処理規則

構文要素	処理
`package`	保持
`import`	`@Test` に必要なもののみ保持
クラス宣言	`extends`・`implements` を除去し保持
クラスボディ	他構文要素に基づき再帰処理
インターフェース宣言	`extends` を除去し保持
インターフェースボディ	他構文要素に基づき再帰処理
フィールド	除去
メソッド	`main` と`@Test` のテストケースのみ保持
初期化ブロック	除去
コンストラクタ	除去
コメント	保持
Javadoc	除去
修飾子	`annotation`，`(non-)sealed` のみ除去

のファイル群や，ローカルリポジトリへ公開されたコンテンツ，リターンコードはアサーションメソッドを用いて直接検証でき，特定の JAR 内のファイルや，コンパイルバージョン，実行されたタスクなどの詳細な情報については，`BuildArtifact` から該当情報を抽出しアサーションメソッドを適用できる．AssertJ を基盤としたアサーションメソッドと成果物を保持する `BuildArtifact` により，開発者はビルド結果の具体的な出力先を意識せず直感的かつ宣言的にテストを記述できる．

アサーションメソッドの実装にあたり，Stack Overflow[*3]及び Gradle Forums[*4]を対象とし，検出対象であるビルドスクリプトに起因するバグ事例を調査した．Stack Overflow では `build.gradle` と `gradle` タグ付き投稿を対象に，Gradle Forums では「Help/Discuss」と「Old Forum Archive/Bugs」カテゴリを対象としてバグを収集した．その結果，ビルドスクリプトに起因する 23 件のバグ事例を収集した．そのカテゴリと収集件数を表 2 に示す．例えば B1 カテゴリには，`jar` タスクの設定ミスにより推移的依存関係を含まない JAR が生成される問題などが含まれる．また B4 カテゴリには，`doLast` 記述漏れにより，実行フェーズでタスクが動作しない問題などが含まれる．

表 2 に示したバグの検出を目標に，アサーションメソッ

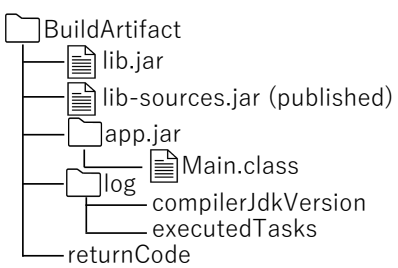


図 2: `BuildArtifact` が保持する成果物例

[*3] https://stackoverflow.com/
[*4] https://discuss.gradle.org/

表 2: 収集したバグカテゴリ

ID	バグカテゴリ	件数
B1	Jar に特定のファイルが含まれない	6
B2	実行する Jar のエントリが存在しない	1
B3	期待通りでないタスクの実行順序	4
B4	タスクが実行されない	5
B5	成果物が公開されない	3
B6	期待通りでないクラスファイル Java バージョン	1
B7	テスト失敗無視（ignoreFailures）されない	2
B8	成果物が生成されない	1

表 3: BuildArtifact 直下に対するアサーションメソッド

メソッド	検証項目	対象バグ
contains()	JAR や WAR などの存在	B1,B8
containsExecutedTestReport()	テストレポートの存在	B7
containsPublication()	公開物の存在	B5
containsBuildOutcome()	ビルド成否	B7
containsAnyJars()	JAR の生成有無	B8
containsAnyWars()	WAR の生成有無	B8

表 4: 抽出されたファイルに対するアサーションメソッド

メソッド	検証項目	対象バグ
contains()	JAR 内のコンテンツ存在	B1
runsSuccessfully()	JAR 実行のリターンコードが 0 か	B1,B2
isNotEmptyJar()	JAR が META-INF 以外を含むか	B1
targetsJavaVersion()	クラスファイルの Java バージョン	B6

表 5: 抽出されたログに対するアサーションメソッド

メソッド	検証項目	対象バグ
containsExecutedTask()	タスクが実行されたか	B3,B4
containsExecutedTasksSequence()	タスクの実行順	B3
containsTaskOutcome()	タスク成否	B7
containsCompilerJdkVersion()	コンパイラの JDK	B6

ドの実装を行なった．実装したアサーションメソッド一覧を表 3，表 4，表 5 に示す．各表にはアサーションメソッドとその詳細，そして検出対象とする主なバグカテゴリを表 2 中の ID を用いて示している．図 2 における lib.jar などの BuildArtifact 直下の成果物については，表 3 に示すアサーションメソッドを用いて，生成有無を直接検証できる．また，図 2 における app.jar などのファイルについては，BuildArtifact から抽出し表 4 に示すアサーションメソッドを用いて，実行成否や Main.class などの JAR 内ファイルの存在を検証できる．さらにログ情報についても同様に抽出し，表 5 に示すアサーションメソッドを用いてタスクの実行成否や実行順を検証できる．

図 3 にテストケース例を示す．対象プロジェクトのパスを指定してスタブを生成し（2-3 行目），build タスクを指定しビルドを実行する（4-6 行目）．ビルドの結果は BuildArtifact オブジェクトの artifact が保持する（4 行目）．これを引数にとりアサーションを記述できる．図 2 の階層構造に従い，BuildArtifact 直下の成果物 lib.jar については，表 3 の contains() で存在を確認できる（7 行目）．

```
1  @Test void test() {
2    StubProject project = StubGenerator.from("android-app")
3                                       .generate();
4    BuildArtifact artifact = GradleRunner.from(project)
5                                         .task("build")
6                                         .run();
7    assertThat(artifact).contains("lib.jar");
8    assertThat(artifact).extractingFile("app.jar")
9                        .contains("Main.class");
10   assertThat(artifact).extractingLog()
11                       .containsExecutedTask("test");
12 }
```

図 3: テストケース例

また，app.jar の内容については，extractingFile() でファイル情報を抽出し（8 行目），表 4 の contains() で JAR 内のファイル有無を検証できる（9 行目）．さらに，ログ情報は extractingLog() を用いて抽出し（10 行目），表 5 の containsExecutedTask() で，指定した test タスクがビルド工程で実行されているか検証できる（11 行目）．

4. 実験

4.1 実験概要

ビルドスクリプトに対するテストにおける，提案手法の有効性を確認するために以下の評価実験を行う．

実験 1：適用可能性の検証

実験 2：エラー隠蔽効果の検証

実験 3：ビルド実行時間短縮効果の検証

実験 1 では，複数のプロジェクトに対して提案手法が適用可能であるか検証する．提案手法の適用に必要な最小限のテストを定義し，それを複数のプロジェクトに対し実行することで，提案手法の適用可能性を評価する．実験 2 では，実験 1 で最小限のテストが成功したプロジェクトの一部を対象に，スタブの利用により Java ソースコードに起因するビルド失敗要因が適切に隠蔽され，ビルド工程に影響を及ぼさないか検証する．実験 3 では，実験 1 で最小限のテストが成功した全てのプロジェクトを対象に，スタブを利用する副次的な効果としてビルド実行時間を短縮できたか検証する．

4.2 実験 1：適用可能性の検証

4.2.1 実験設計

本実験では，複数のプロジェクトに対して提案手法が適用可能であるか検証する．まず提案手法の適用に必要な最小限のテストを定義する（以降，最小限テストと呼ぶ）．最小限テストは，検証対象であるビルド成果物が生成されるまでの，スタブの作成と指定されたタスクの実行から構成される（図 3: 2-6 行目）．実行するタスクには，ビルド時の最上位タスクである build タスクを設定する．テストライブラリの適用にはこの最小限テストが成功し，ビルド成

表 6: 最小限テストで確認すべきビルド成果物の差異

差異カテゴリ	詳細
Files	build/libs,classes 下のファイル存在
JAR Contents	JAR 内コンテンツの存在
JAR Return Code	JAR 実行の成否
Task Execution Order	実行したタスクの順番
Class File Version	クラスファイルの major version
Java Compiler Version	javac, -source/target のバージョン
Test Reports	テストレポートの存在
Published Artifacts	Maven ローカルへの公開物の存在

果物がスタブを用いない場合の成果物に比べて差異がないことが求められる．これは，ビルド成果物に差異が存在するとアサーションメソッドによる正確な検証が困難となるためである．そのため，評価は以下の 2 つの観点から行う．

- 最小限テストの成功率
- スタブなしビルド成果物との差異

最小限テストの成功率に関しては，定義した最小限テストを複数のプロジェクトに対して実行し，その成否を確認する．スタブなしビルド成果物との差異については，最小限テストに成功したプロジェクトのビルド成果物を，スタブを用いずにビルドを実行した場合の成果物と比較する．比較対象となる差異カテゴリを表 6 に示す．各項目はビルド成果物に基づき，差異が存在した場合にアサーションメソッドによる検証に影響が出るものを示している．

実験題材として，サンプルプロジェクトや提案手法が適用できないプロジェクト構造を避けるため以下の選定基準を設けた．この基準に従う 103 プロジェクトを実験対象として選定した．

- GitHub 上で公開
- 主要言語が Java
- star 数が 10 以上
- OSS のライセンスが存在
- build.gradle または build.gradle.kts が存在
- JDK8,11,17,21 のいずれかで `./gradlew build` が成功
- テスト対象のビルドスクリプトが一意に定まる構造

4.2.2 結果と考察

最小限テストの成功率は，103 プロジェクト中 84 プロジェクトの 81%であった．テストに失敗した 19 プロジェクトの失敗原因を表 7 に示す．特に，Groovy など他言語ファイルからの参照先の喪失や，タスク実行に伴い必要となる Java ソースコードへの依存の欠如が存在した．また，静的コード解析ルールの違反などフォーマット規約が原因によるテスト失敗も存在した．

結果より，基本的には 8 割以上と多くのプロジェクトへ適用可能であった．しかし，現時点のスタブ作成方針では適用できないプロジェクトも一部存在した．タスクや他ファイルからの参照に関しては，現時点では Java ソースコードのみをスタブの対象にしているため隠蔽化できない．これらの Java ソースコード外からの参照はプロジェクトごとで定められ一般化できないため，全てに対応するのは困難である．また，テストカバレッジなどの実行時パフォーマンスに関しては，メソッド内のロジックを隠蔽化するためスタブでは再現できない．実際に，失敗原因となった高いテストカバレッジを維持するにはテストケース内のロジックを隠蔽すべきでないが，ほとんどのテストケースは他クラスのメソッドに依存するため，可能な限り参照を隠蔽するスタブ方針と矛盾してしまう．ただし，失敗プロジェクト数自体が少ないことから，現行のスタブ作成方針を変更する必要性は低い．そして，フォーマット規約によるビルド失敗に関しては，フォーマット規則が各プロジェクトで固有に設定可能である以上，全てに対応することは不可能である．

表 7: 最小限テスト失敗原因の分類

大分類	小分類	件数
依存欠如	他言語ファイルの参照	8
	構成ファイルからの参照	1
	タスクが生成する Java コードからの参照	1
	タスクからの参照	2
静的解析違反	-	5
動作期待未達	テストカバレッジ閾値未達	2

次に，最小限テストが成功した 84 プロジェクトに対して，表 6 に示したカテゴリごとに，スタブを用いないビルド成果物との差異を調査した．84 プロジェクトの内，39 プロジェクトでは各カテゴリで差異が存在せず，残りの 45 プロジェクトではいずれかのカテゴリで差異が存在した．各カテゴリごとの差異の原因と件数を表 8 に示す．ここで，Files で特定のクラスファイルが生成されなかった場合に，JAR Contents で同クラスファイルが JAR に含まれないような，他カテゴリが原因となる副次的な差異は除外した．このように，各カテゴリごとの独立した差異のみを集計した．表 8 より，Files，JAR Contents，JAR Return Code，Task Excecution Order，Test Reports の 5 つのカテゴリで差異が確認された．これらの差異がアサーションメソッド実行時にどのような影響を及ぼすのか，またどの程度までなら副作用として許容できるかを考察する．

Files

生成されるファイル群では，インナークラスが生成されない差異が多く確認され，特に匿名クラスによるものが大部分を占める．しかし，これらのクラスファイルは開発者の明示的な操作対象ではないため，この差異はスタブの副作用として許容可能である．また，jgitver プラグイン適用によるファイル名変更に関しては，スタブ作成過程で未コミット状態のファイルが生じるため避けられないが，このプラグインを利用するプロジェクト自体が少数であるため

表 8: 差異カテゴリごとに確認されたビルド成果物の差異の原因分類

カテゴリ	原因	詳細	件数
Files	匿名クラス未生成	匿名クラスが隠蔽され生成されない	34
	Builder.class 未生成	Lombok の`@Builder` が隠蔽され生成されない	4
	ローカルクラス未生成	メソッド隠蔽化によりローカルクラスも隠蔽され生成されない	1
	マッピングファイル未生成	Mixin の`@Mixi` が隠蔽され生成されない	1
	異なるファイル名	jgitver により未コミット状態のファイル名に dirty 付与される	1
	package-info.class 未生成	RUNTIME アノテーションが隠蔽されコンパイル対象外となる	1
JAR Contents	Javadoc 関連ファイル未生成	Javadoc 文が隠蔽され HTML 成果物が生成されない	5
	Mixin 等設定ファイル未生成	Mixin の`@Mod` 等が隠蔽され生成されない	4
	匿名クラス未生成（JAR 内）	匿名クラスが隠蔽され生成されない	2
	未使用依存関係除外	`minimize()` 使用により未使用外部ライブラリが除外される	1
JAR Return Code	成功する実行可能 JAR 生成	Java ソースコードの参照やロジック隠蔽されるため必ず正常終了する	9
Task Execution Order	実行数減少・早期完了	不要な処理が隠蔽され実行されず順序が変化する	4
Test Reports	テストアノテーション消失	`@ParameterizedTest` 等が隠蔽される	4
	継承テスト消失	テストクラス継承でオーバーライドされないテストケースが隠蔽される	1

許容可能である．一方で，Builder.class 未生成やマッピングファイル未生成といった差異に関しては，依存関係先のアノテーションの隠蔽に起因している．これらのファイルは，ビルドスクリプトで定義したタスクにより自動生成されるため，ビルドスクリプトの検証においては望ましくない副作用である．そのため，ファイル生成に関連するアノテーションを保持したスタブ作成が求められる．

JAR Contents

Files の場合と同様に，匿名クラス未生成についてはスタブの副作用として許容可能である．一方で，Javadoc 関連の HTML ファイルや設定ファイルの未生成などは，ビルドスクリプトの記述と関連するため望ましくない副作用である．これらのファイル未生成に対し，Files カテゴリと同様に関連するアノテーションを保持したスタブが求められる．一方，ビルドスクリプト内で使用される `minimize()` により，スタブ中で使用されていない依存関係が JAR から除外される差異については，可能な限り参照を隠蔽するスタブの方針と本質的に相容れない．そのため，現行のスタブ作成方針ではこの差異を回避することは困難である．

Jar Return Code

スタブにより，JAR に含まれる `main` メソッド内のロジックが隠蔽されるため，生成された実行可能 JAR が必ず正常終了する．これにより，JAR の依存関係などのビルドスクリプトの記述にバグが存在した場合に，本来異常終了する JAR が正常終了してしまいバグを検出できなくなる．そのため，望ましくない副作用である．ただし，現状のスタブ作成方針では参照を隠蔽する上でメソッド内のロジックを完全に維持できないため回避困難である．

Task Execution Order

スタブにより，一部のアノテーションやクラス定義が隠蔽されるため，タスクの実行数が減少したり，特定のタスクが早期終了する事例が確認された．これにより，ログ上でタスク実行順序に変化が生じる例も見られた．しかし，Gradle ではタスクの実行順は原則として `dependsOn` などの依存関係によって制御され，その他の依存関係を持たないタスク同士の実行順序は保証されない．そのため，これらの実行順序の変化は開発者の意図に反するものではなく，許容可能な副作用である．

Test Reports

一部のテストケースが実行されない差異が確認された．具体的には，`@ParameterizedTest` などの参照を生むアノテーションを持つテストケースや，他テストクラス継承時にオーバライドされないテストケースは，スタブにより隠蔽され `test` タスクで実行されない．これにより，ビルドスクリプトで定義するテストレポートが生成されないため望ましくない副作用である．そのため，`@ParameterizedTest` が付くテストケースは付随する参照を隠蔽した上で保持し，継承元にテストケースが存在するテストクラスは，継承元と同一のテストケースを追記することで対策すべきである．

4.3 実験 2：エラー隠蔽効果の検証

4.3.1 実験設計

本実験では，スタブを用いることで Java ソースコードに存在するビルド失敗要因を隠蔽できているか検証する．これは Java ソースコード側に欠陥が含まれているという状況下において，スタブによりその欠陥を隠蔽しビルドスクリプトに記述されたロジックそのものを適切に検証できるかという実験に該当する．検証では，実験 1 で最小限テストが成功したプロジェクトの一部を対象にビルド失敗要因を注入し，実験 1 同様に最小限テストを実行する．注入するビルド失敗要因は，ランタイムエラーとアサーションエラーである．確実にこれらのエラーを発生させるために，プロジェクト内のテストクラスを 1 つ選択し各エラーに対

応するテストケースを配置する．これにより，test タスクで確実に注入したエラーが発生する．スタブにより Java ソースコードのビルド失敗要因が隠蔽できビルド成果物への影響がないか評価するため，最小限テストの成功率と，実験 1 で評価した「スタブなしビルド成果物との差異」と比べて新たに生じた差異の有無を確かめる．

実験対象として，実験 1 で最小限テストに成功したプロジェクトの中から，以下条件を満たすものを star 数の多い順で 3 プロジェクト選択する．

- build ディレクトリ下に classes 及び libs が存在し，ファイルが 1 つ以上生成されている
- test タスクが実行されている

1 つ目の条件は，実験 1 から新たに成果物に生じた差異を確かめる上で，ある程度の成果物が存在していることが望ましいためである．2 つ目の条件は，注入したビルド失敗要因が確実に実行されるためである．

4.3.2 結果と考察

結果として，いずれのプロジェクトでも最小限テストが成功した．さらに，実験 1 で確認された「スタブなしビルド成果物との差異」と比べて新たな差異も生じなかった．一方で，ビルド失敗要因注入後のプロジェクトに対してスタブを用いずに./gradlew build を実行すると，各プロジェクトは注入した 2 つのテストケースが原因で test タスクでビルドが失敗した．これはすなわち，スタブによって欠陥の隠蔽化に成功しているといえる．

4.4 実験 3：ビルド実行時間短縮効果の検証

4.4.1 実験設計

本実験では，スタブを用いることでビルド実行時間を短縮できているか検証する．本来のスタブの実装目的は，Java ソースコード側に存在し得る欠陥を隠蔽することにあるが，3.1 節で示す通りスタブでは元の Java ソースコードから検証対象であるビルド成果物に影響しない構文要素が除外される．そのため，スタブを利用する副次的な効果としてビルド成果物生成に不要な処理が実行されず，ビルド実行時間の短縮が期待できる．検証では，実験 1 で最小限テストが成功した 84 プロジェクトを対象に最小限テスト内のスタブ生成時間及びビルド時間を計測する．また比較対象として同プロジェクトに対し，スタブを用いない build タスクの実行時間を計測する．スタブの利用有無に限らずビルドは同一環境で計 5 回実行されその平均値を計測時間とする．また，各ビルド環境を統一するため増分ビルドキャッシュは用いずにビルドを実行する．これらのスタブを用いたビルドと用いないビルドでの実行時間の比較により評価を行う．

4.4.2 結果と考察

結果として，スタブの利用により全てのプロジェクトでビルド時間の短縮が確認できた．対象プロジェクトにおける計測結果の平均値を表 9 に示す．スタブを利用しないビルド時間が 18.0 秒であるのに対し，スタブを用いたビルド時間は 9.02 秒であり全体で 2.65 倍高速化できている．また，提案手法のオーバーヘッドとなるスタブ生成時間は 0.40 秒とビルド時間に比べて十分小さく，スタブ生成時間を含めたビルド時間に関しても 9.42 秒とスタブを用いないビルドと比べ全体で 1.91 倍高速化できている．そのため，スタブの利用によりビルド時間は短縮できビルドスクリプトのテストにおける時間的コストを削減可能だといえる．

表 9: ビルド時間とスタブ生成時間に関する平均値

	nostub(sec)	stub(sec)			speedup(×)	
	Build	Gen	Build	Total	Build	Total
average	18.0	0.40	9.02	9.42	2.65	1.91

5. ケーススタディ

提案手法の有効性を確認するため，提案手法を用いて実在するバグを検出できるか検証する．検証対象として 3.2 節で収集したバグ事例に基づき，以下に示すバグを再現する 2 つのプロジェクトを作成した．各プロジェクトが対応する表 2 のカテゴリの ID を併記している．

- プロジェクト 1：期待通りでない JAR（B1）
- プロジェクト 2：ビルド続行阻止（B7）

これらのプロジェクトに対して，バグを含むビルドスクリプトと，バグによる出力，バグを検出するためのテストケース，バグ検出時の出力をそれぞれ示す．

5.1 期待通りでない JAR

ここでは，開発者の期待通りでない JAR を生成するプロジェクトについて提案手法を適用する．前提として，開発者はビルドにより正常終了する実行可能 JAR の生成を期待している．まずは，JAR 生成に関してバグを含むビルドスクリプトとバグによる出力を図 4 に示す．図 4(a) のビルドスクリプトでは，JAR に含める依存関係に compileClasspath を指定している（8 行目）．そのため，コンパイル時の依存関係のみが JAR へ含まれ実行時に必要な推移的依存関係は含まれない．しかし，このプロジェクトは utils プロジェクトへ依存しているため（3 行目），生成された JAR を実行すると図 4(b) の通り，utils プロジェクトが依存する com/google/common/base/Strings が JAR へ含まれず実行時エラーを出力する．

このバグを検出するためのテストケースと検出時の出力を図 5 に示す．図 5(a) はテストケースであり，スタブの作成とビルド実行は図 3 に示すものと同様であるため省略する．例えば推移的依存関係を含んでいるかは，検出対象の app.jar を抽出し，contains() で推移的依存と思われるライブラリのクラスファイルを 1 つ指定することで検証

```
1 ...
2 dependencies {
3   implementation project(":utils")
4 }
5 jar {
6   from {
7     // BUG: compileClasspath では推移的依存が含まれない
8     configurations.compileClasspath.filter{it.exists()}
9          .collect{it.isDirectory() ? it : zipTree(it)}
10  }
11 }
```

(a) 期待通りでない JAR を生成するビルドスクリプト

```
Exception in thread "main" java.lang.NoClassDefFoundError:
    com/google/common/base/Strings at ...
```

(b) 依存関係の欠如による JAR 実行時エラー出力

図 4: バグを含むビルドスクリプトと JAR 実行時出力

```
1 @Test void testJar() {
2   ... // スタブ生成とビルド実行は省略
3   assertThat(artifact).extractingFile("app.jar")
4     .contains("com/google/common/base/Strings.class");
5 }
```

(a) 依存関係の欠如を検出するテストケース例

```
BscriptTest > testJar() FAILED
  java.lang.AssertionError:
    Expected file <com/google/common/base/Strings.class>
    to be in JAR <app.jar>, but it was not found.
```

(b) バグ（依存関係の欠如）検出時の出力

図 5: JAR に関するテストケースと検出時の出力

できる（3-4 行目）．図 5(a) のテストケースによるバグ検出時の出力を図 5(b) に示す．図 5(b) は，`contains()` 実行時のアサーションエラーである．出力から，指定した推移的依存関係が JAR に含まれていないことがわかり，バグを検出できる．

5.2 ビルド続行阻止

ここでは，ビルドの続行を阻止するプロジェクトについて提案手法を適用する．前提として，開発者は `test` タスクでテストが失敗した場合でも，その後に続くレポート出力やソースコードのフォーマットチェックのため，ビルドを続行したいと考えている．まずは，ビルド続行においてバグを含むビルドスクリプトとバグによる出力を図 6 に示す．図 6(a) のビルドスクリプトでは，`ignoreFailures` を `true` としてテストが失敗しても，`test` タスクを成功とみなしビルドを続行するよう設定されている（4 行目）．しかし，テスト失敗時の後処理としてログ出力をしており（5-12 行目），その際に `result.failures` リストの存在しない要素を参照している（9 行目）．これにより，`test` タスクでテストが失敗した際にビルドスクリプトで参照エラーとなり，図 6(b) の通り意図せず実行時エラーが出力されビルドが中断してしまう．また，全てのテストケースが成功した場合にはこのバグは顕在化せず，バグが見逃される可能性もある．

```
1 ...
2 test {
3   useJUnitPlatform()
4   ignoreFailures = true // テストに失敗してもビルド継続
5   afterSuite { desc, result -> // テスト失敗時にログ出力
6     if (!desc.parent) {
7       if (result.failedTestCount > 0) {
8         // BUG: 参照によるランタイムエラー
9         println "${result.failures[6].message}"
10      }
11    }
12  }
13 }
```

(a) ビルドが続行できないビルドスクリプト

```
4 tests completed, 3 failed
> Task :test
  FAILURE: Build failed with an exception.
```

(b) 実行時エラーによる `build` タスク中断時の出力

図 6: バグを含むビルドスクリプトとビルド実行時出力

このバグを検出するためのテストケースと検出時の出力を図 7 に示す．図 7(a) はテストケースであり，ビルド実行は図 3 に示すものと同様であるため省略する．ここで検証すべきは，テスト失敗時にビルドが続行できているかである．しかし，通常のスタブではテストケース内のロジックは全て隠蔽されテストは失敗しない．そのため，指定したファイルでテスト失敗を誘発するスタブのエラー注入メソッド `injectTestFailure()` を実行している（3 行目）．他のメソッドとしては，ランタイムエラーを注入するメソッドが実装されている．これにより，テストが必ず失敗するスタブが作成され理想の検証環境を構築できる．テスト失敗時にビルドが続行されているかどうかは，`test` タスクの実行成否を検証すればよいため，`test` タスクに対して `containsTaskOutcome()` を実行することで検証している（7 行目）．図 7(a) のテストケースによるバグ検出の出力を図 7(b) に示す．図 7(b) は，`containsTaskOutcome()` 実行時のアサーションエラーである．出力から，`test` タスクが失敗しておりテストが失敗する場合にビルドを続行できていないことがわかり，バグを検出できる．

6. おわりに

本研究では，ビルドスクリプトに対するテストを目的として，Gradle のビルドスクリプトの検証に特化したテストライブラリを提案した．テストライブラリはスタブとアサーションメソッドから構成され，有効性の評価のために評価実験とケーススタディを実施した．その結果，対象プ

```java
@Test void testBuildExecutableWithError() {
  StubProject project = StubGenerator.from("android-app")
                .injectTestFailure("TestUtils.java")
                .generate();
  ...  // ビルド実行は省略
  assertThat(artifact).extractingLog()
        .containsTaskOutcome("test", "SUCCESS");
}
```

(a) test タスク失敗を検出するテストケース例

```
BscriptTest > testBuildExecutableWithError() FAILED
  java.lang.AssertionError:
    Expected task <test> to have outcome <SUCCESS>,
    but actual outcome was <FAILED> at ...
```

(b) バグ（ビルド中断）検出時の出力

図 7: ビルド続行に関するテストケースと検出時の出力

ロジェクトの 8 割以上へ提案手法が適用可能であり，実在するバグの検出も可能であることを確認した．また，スタブの副次的な効果としてビルド実行時間を短縮することも確認した．

今後の課題として，スタブの改善が考えられる．現時点のスタブでは一部のアノテーションやテストケースを保持できず，適用可能ではあるがスタブなしの場合と比べて生成されるファイルやテストレポートに差異を生じた．そのため，スタブで保持すべきアノテーションのリスト化や，テストケース継承処理の実装が求められる．また，別の課題として実装したアサーションメソッドの評価が挙げられる．本稿では実装したアサーションメソッドがどれだけのバグを検出できるかといった網羅性を検証できておらず，このためにはビルドスクリプトに存在し得るバグ事例を網羅的に収集し，その全体像を把握する必要がある．しかし，本研究で収集されたバグ事例は 23 件と少なく，再現性のある調査方法の確立やバグ事例の体系化も不十分である．したがって，Gradle Forums の投稿などを対象に投稿期間等の条件を限定したデータセットを定め，全てのバグ事例の収集及び体系化のための追加調査を行う必要がある．これにより，実験を行う上でのバグカテゴリの全体像を定義でき，アサーションメソッドの評価が可能となる．

加えて，提案手法の拡張として他ビルドツールへの適用が考えられる．提案手法ではビルドスクリプトに対する静的解析などを必要とせず，ビルドによって生成されたビルド成果物を検証することでビルドスクリプトのバグを検出している．そのため，この手法は特有のビルドツールへと依存せず Java ソースコードを対象とした他のビルドツールへと適用できる可能性がある．他の主要な Java 用のビルドツールとしては Maven などが存在するが，Maven においてもビルド実行により Gradle 同様に決められたフォルダへビルド成果物を配置する．そのため，アサーションメソッドによるビルド成果物への検証を Maven などのビルド成果物に対しても拡張できれば，他ビルドツールへの適用も可能だと考える．

謝辞 本研究の一部は，JSPS 科研費（JP25K15056, JP25K03102, JP24H00692）による助成を受けた．

参考文献

[1] Misu, M. R. H., Achar, R. and Lopes, C. V.: SourcererJBF: A java build framework for large-scale compilation, *Transactions on Software Engineering and Methodology*, Vol. 33, No. 3, pp. 1–35 (2024).

[2] Spall, S., Mitchell, N. and Tobin-Hochstadt, S.: Build scripts with perfect dependencies, *In Proceedings of the ACM on Programming Languages*, Vol. 4, No. OOPSLA, pp. 1–28 (2020).

[3] Hassan, F., Mostafa, S., Lam, E. S. and Wang, X.: Automatic building of java projects in software repositories: A study on feasibility and challenges, *In Proceedings of International Symposium on Empirical Software Engineering and Measurement*, pp. 38–47 (2017).

[4] Zhang, C., Chen, B., Hu, J., Peng, X. and Zhao, W.: BuildSonic: Detecting and repairing performance-related configuration smells for continuous integration builds, *In Proceedings of International Conference on Automated Software Engineering*, pp. 1–13 (2022).

[5] Lyu, J., Li, S., Zhang, H., Zhang, Y., Rong, G. and Rigger, M.: Detecting build dependency errors in incremental builds, *In Proceedings of International Symposium on Software Testing and Analysis*, pp. 1–12 (2024).

[6] Nejati, M., Alfadel, M. and McIntosh, S.: Understanding the implications of changes to build systems, *In Proceedings of International Conference on Automated Software Engineering*, pp. 1421–1433 (2024).

[7] Nejati, M., Alfadel, M. and McIntosh, S.: Code review of build system specifications: Prevalence, purposes, patterns, and perceptions, *In Proceedings of International Conference on Software Engineering*, pp. 1213–1224 (2023).

[8] Spadini, D., Palomba, F., Baum, T., Hanenberg, S., Bruntink, M. and Bacchelli, A.: Test-driven code review: an empirical study, *In Proceedings of international conference on software engineering*, pp. 1061–1072 (2019).

[9] Lou, Y., Chen, Z., Cao, Y., Hao, D. and Zhang, L.: Understanding build issue resolution in practice: symptoms and fix patterns, *In Proceedings of Joint Meeting on Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 617–628 (2020).

[10] Rausch, T., Hummer, W., Leitner, P. and Schulte, S.: An empirical analysis of build failures in the continuous integration workflows of java-based open-source software, *In Proceedings of International Conference on Mining Software Repositories*, pp. 345–355 (2017).

[11] Sotiropoulos, T., Chaliasos, S., Mitropoulos, D. and Spinellis, D.: A model for detecting faults in build specifications, *In Proceedings of the ACM on Programming Languages*, Vol. 4, No. OOPSLA, pp. 1–30 (2020).

[12] Hassan, F. and Wang, X.: HireBuild: an automatic approach to history-driven repair of build scripts, *In Proceedings of the International Conference on Software Engineering*, pp. 1078–1089 (2018).

[13] Wu, R., Chen, M., Wang, C., Fan, G., Qiu, J. and Zhang, C.: Accelerating build dependency error detection via virtual build, *In Proceedings of International*

Conference on Automated Software Engineering, pp. 1–12 (2022).

[14] Leotta, M., Cerioli, M., Olianas, D. and Ricca, F.: Fluent vs basic assertions in Java: an empirical study, *In Proceedings of International conference on the quality of information and communications technology*, pp. 184–192 (2018).