修士学位論文

題目

Gradle ビルドスクリプトに対する自動テスト実現のための テストライブラリの提案

指導教員

楠本 真二 教授

報告者

藪下 友

令和 8 年 2 月 2 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

令和 7 年度 修士学位論文

Gradle ビルドスクリプトに対する自動テスト実現のための
テストライブラリの提案

藪下 友

内容梗概

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．

主な用語

Gradle，ビルドスクリプト，build.gradle，テストライブラリ，スタブ，アサーションメソッド

目次

図目次

表目次

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][2][3]．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 [4]．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 [5]．

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 [6][7][8][9]．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 [10][11][12]．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 [13]．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 [14]．

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，リターンコードなど多岐にわたる [15]．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．

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．

```
jar {
  // Main-Classが設定されていない
  // manifest { attributes( "Main-Class": "com.app.Main")}
  from {
    configurations.runtimeClasspath.collect {
      it.isDirectory() ? it : zipTree(it)
    }
  }
}
```

(a) JAR 実行エントリ未設定のビルドスクリプト

```
no main manifest attribute, in build/libs/android-app.jar
```

(b) エントリが存在しないことによる JAR 実行時エラー出力

図 1: ビルドスクリプトに含まれるバグの例

2 準備

2.1 ビルドスクリプトのバグ

ビルドスクリプトのテストにおける検出対象はビルドスクリプトに存在するバグである．図 1 にビルドスクリプトのバグ事例を示す．図 1a は実行可能 JAR を生成するためのビルドスクリプトである．`jar` ブロックでは JAR ファイル生成タスクの性質を定義しており，`from` ブロックでは JAR ファイルに含める各種クラスファイルを指定する．この例では `from` の宣言自体は正しいが，JAR ファイルの実行時エントリを指定する `manifest` が未指定である．そのため，生成される JAR を実行すると 図 1b に示す通りエントリが見つからず実行時エラーを出力する．この事例ではビルドプロセス自体は成功してもビルドスクリプト中の誤りが原因となり，開発者の意図しない成果物が生成される．本研究では，このようなビルドスクリプト中の誤りにより成果物が期待通り振る舞わなくなる要因を，ビルドスクリプトのバグと定義する．本研究ではこのようなビルドスクリプトのバグの検出を目的としてテストを行う．

2.2 Java ソースコード中のビルド失敗要因

Java ソースコード中にビルド失敗要因が存在する場合，ビルドスクリプトに限定した検証作業を実施できない．図 2 に Java ソースコードが原因による `build` タスク失敗例を示す．Gradle における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 2 のように，Java ソースコードにランタイムエラー等のビルド失敗要因が存在した場合に，それに依存する `test` タスクが失敗

し依存関係を通じ build タスクも失敗してしまう．2.1 節で述べたようにビルドスクリプトのバグはビルド成果物に現れる．そのため，図 2 のように test タスクが失敗し一部のビルド成果物が生成されないと，ビルドスクリプトのバグが反映されるはずのビルド成果物が生成されずテストが実施できない．また，ビルド成果物の未生成がビルドスクリプトのバグに起因するか，ビルド失敗に起因するかが判別できない．したがって，自動テスト実施のために Java ソースコードをビルドプロセスから分離しビルドを成功させる必要がある．

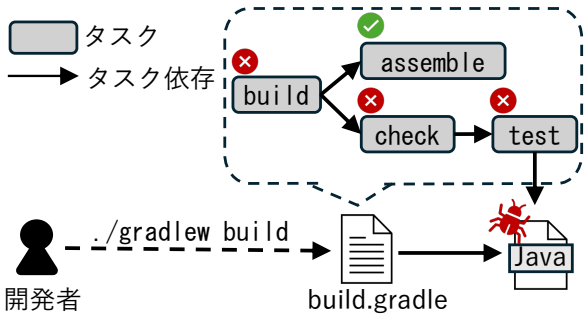

図 2: Java ソースコードの欠陥により build タスクが失敗

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[*1]．このライブラリを用いてビルドスクリプトに対するテストケースを記述でき，Java ソースコードに対する自動テストと同時にビルドスクリプトに対しても自動テストを実施することが可能となる．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．

ここで，本研究で対象とするビルド成果物をそれぞれの成果物の情報源と共に 表 1 へ示す．ビルド成果物は大きくファイルとログに分類できる．各ビルド成果物の情報源について，`build/`は Gradle においてビルド成果物が配置されるデフォルトのディレクトリであり，Java のコンパイル結果であるクラスファイルや JAR などが配置される．JUnit の生成するテストレポートは，HTML 形式と XML 形式で異なるディレクトリ（`build/test-results/`，`build/reports/tests/test/`）へと配置されるため，検証時にはこれらの両ディレクトリが対象となる．また，ログ情報としてはビルド中に実行されたタスクやコンパイルのターゲット Java バージョン，リターンコードがある．これらは標準出力やリターンコードに含まれる．

表 1: ビルド成果物とその情報源

大分類	小分類	情報源
ファイル	クラスファイル	`build/classes/`
	JAR 等パッケージファイル	`build/libs/`
	テストレポート	`build/test-results/`, `build/reports/tests/test/`
	公開ファイル	`~/.m2/`
ログ	実行されたタスクリスト	標準出力
	コンパイル Java バージョン	標準出力
	リターンコード	リターンコード

[*1] `https://central.sonatype.com/artifact/io.github.y-yabust/BScriptTest/`

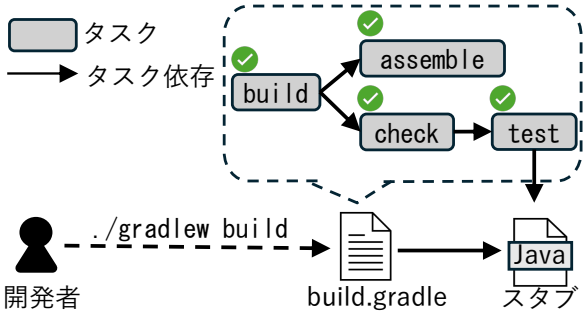

図 3: スタブにより Java ソースコードの欠陥を隠蔽

3.1 Java ソースコードのスタブ

Java ソースコードをビルドプロセスから分離しビルドを成功させるために，Java ソースコードの代替となるスタブを提案する．図 3 に提案スタブを用いた build タスクの検証例を示す．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 3 の通り，ソースコード中の欠陥を隠蔽しビルドを成功させる．また他に Java ソースコードをビルドプロセスから分離する方法として，ビルドスクリプトにおけるエラーハンドリングや Java ソースコードの削除が考えられる．例として `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．また，クラスファイルやテストレポートなどのビルド成果物は Java ソースコードの構文要素に基づき生成されるため，Java ソースコードを削除すると検出対象であるビルド成果物へと影響を及ぼしてしまう．そのためスタブは，Java ソースコードの代替として機能し，Java ソースコードを完全に分離させるのではなく，直接的にソースコード中に存在するビルド失敗要因のみを隠蔽しビルドを成功させる．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 [16].

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 ソースコード中のビルド失敗要因を隠蔽し，ビルドを成功させることにある．しかし，ビルドスクリプトの作用はクラスファイル等の 表 1 に示すテストにおける検証対象である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する必要がある．ここで，スタブの利用によりスタブを利用しない場合のビルド成果物へ影響を及ぼしてしまうことをスタブの副作用と定義する．副作用は大きく 2 つに分類できる．1 つ目は，スタブの利用によりビルドプロセスが失敗しビルド成果物が完全に生成されないことである．また 2 つ目は，ビルドプロセスは成功するが，元のプロジェクトと比べてファイルやログ出力などのビルド成果物へ差異

表 2: 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

を生じることである．1 つ目の副作用に関しては，スタブの目的であるビルドの成功を阻害するため原則許容できない．一方で 2 つ目の副作用に関しては，ビルド成果物の中でも開発者が通常生成を期待しない一時ファイルなどに影響する場合は，テストに影響を及ぼさないため許容可能である．よって，2 つ目の副作用に関してはこれら全てがテストに悪影響を及ぼすわけではないが，正確なテストを行うためには最低限に留める必要がある．そのため，Java Language Specification（JLS）における構文記法（Chapter 19. Syntax[*2]）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 2 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テストケースのみを保持する．

例として，プロダクトコードとテストコードから生成されるスタブのソースコードを，元ソースコードからの差分とともに 図 4 に示す．−（赤色）の行はビルド失敗要因を隠蔽するため元ソースコードから除去された箇所を示し，＋（緑色）の行はビルド成果物維持のために追加された箇所を示す．図 4a はプロダクトコードのスタブ例である．図よりビルド成果物であるクラスファイルに影響する `package`

[*2] https://docs.oracle.com/javase/specs/jls/se24/html/jls-19.html

```
1   package com.order;
2
3 - import lombok.Builder;
4 - import java.util.List;
5
6 - public class OrderProc extends Base {
7 + public class OrderProc {
8 -   private int counter = 0;
9
10 -  public OrderProc(int counter) {...}
11 -  public static void main(...) {...}
12 +  public static void main() {}
13 -  @Builder
14 -  public static class Config {...}
15 +  public static class Config {}
16  }
```

(a) プロダクトコードのスタブ例

```
1   package com.order;
2
3   import org.junit.jupiter.api.Test;
4 - import org.junit.jupiter.params.
      ParameterizedTest;
5 - import org.junit.jupiter.params.
      provider.ValueSource;
6 - import static org.assertj.core.api.
      Assertions.assertThat;
7
8   public class OrderProcTest {
9 -  private OrderProc target = new
      OrderProc(5);
10
11     @Test
12 -   void testMainRuns() {...}
13 +   void testMainRuns() {}
14 -   @ParameterizedTest @ValueSource(
      ints = {1, 2, 3})
15 -   void testWithParameters(int input)
      {...}
16 +   @Test
17 +   void testWithParameters() {}
18  }
```

(b) テストコードのスタブ例

図 4: 元ソースコードからの差分を含めたスタブ例（– 行：削除箇所，＋行：追加箇所）

や，クラス宣言である OrderProc，インナークラスである Config は，不要な extends や@Builder などの修飾子を除去した上で保持されているが（1，6-7，13-15 行目），ビルド成果物に影響しない依存関係や，クラス内部のフィールド，コンストラクタなどは除去されていることがわかる（3-4，8-10 行目）．また，実行可能 JAR のエントリポイントとなる main メソッドは，内部の不要な処理を除去した上で保持される（11-12 行目）．図 4b はテストコードのスタブ例である．テストコードでも同様に，package やクラス宣言は保持され（1，8 行目），不要な依存関係は除去されている（4-6 行目）．一方で，ビルド成果物であるテストレポートに影響するテストケースに関しては，内部の不要な処理を除去した上で保持され（12-13 行目），テストケースとして判定されるために必要な依存関係のみが保持される（3 行目）．また，@ParameterizedTest や@ValueSource などのアノテーションが付与されたテストケースは，参照不整合を起こす可能性があるため@Test に置換されて保持される（14-17 行目）．

他にスタブのメソッドとして一部構文要素を保持する保持メソッドと指定したファイルへエラーを注入するエラー注入メソッドを実装している．各メソッドを 表 3 へ示す．保持メソッド（keep 系メソッド）の利用によりファイルとクラス，メソッド，アノテーション，インポート文単位で元の Java ソース

表 3: スタブが持つ保持メソッドとエラー注入メソッド

メソッド	詳細
keepFile()	指定したファイルを保持
keepClass()	指定したクラスを保持
keepMethod()	指定したメソッドを保持
keepAnnotation()	指定したアノテーションを保持
keepImport()	指定したインポート文を保持
injectCompileError()	指定ファイルへコンパイルエラーを注入
injectRuntimeError()	指定ファイルへ実行時エラーを注入
injectTestFailure()	指定ファイルへアサーションエラーを注入

コードの構文要素を一部保持したスタブを生成できる．これにより，テストを実施するにあたり必要な構文要素を例外的に保持したスタブを生成できる．ただし，一部構文要素を残すことにより参照不整合などが生じコンパイルエラーとなる可能性もあるため注意して使用する必要がある．また，エラー注入メソッド（inject 系メソッド）の利用によりコンパイルエラーと実行時エラー，アサーションエラーを任意のファイルへと注入できる．指定したファイル内のクラスへそれぞれのエラーを生じるメソッドを配置することでエラーを生じさせている．これによりエラーを生じるスタブを生成でき，Java ソースコード内でエラーが発生した場合のビルドスクリプトのエラーハンドリングが正常に動作しているかなどの異常系テストが可能となる．

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[*3]を基盤として実装する [17]．さらに，本ライブラリでは 表 1 のファイルとログから定義されるビルド成果物を参考に，成果物を抽象化した BuildArtifact オブジェクトを提供する．図 5 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[*4]及び Gradle Forums[*5]を対象とし, 検出対象

[*3] https://joel-costigliola.github.io/assertj/

[*4] https://stackoverflow.com/

[*5] https://discuss.gradle.org/

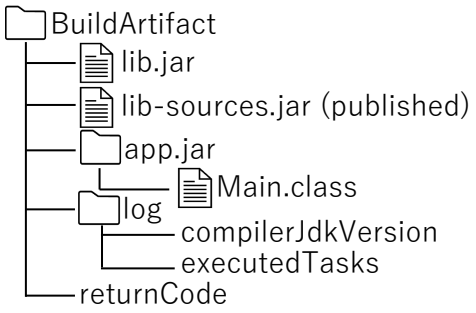

図 5: `BuildArtifact` が保持する成果物例

であるビルドスクリプトに起因するバグ事例の予備調査を行った．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．これらのバグ事例と収集件数を，表 1 に示すビルド成果物に基づき分類した．分類結果を 表 4 へ示す．例えば B2.1 の JAR 内に必要なファイルが含まれない事例としては，`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ある．また B3.2 のタスクが実行されない事例としては，`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ある．ここで，表 4 に示す各バグ事例はアサーションメソッド実装の指針を得るための予備調査結果であり，現時点でビルドスクリプトのバグを完全に網羅するものではない．そのため今後新たなバグ事例の発見に伴い，各バグカテゴリ内で事例の追加及びそれに基づく新たなアサーションメソッドの実装が求められる．

表 4 に示したバグカテゴリの各事例の検出を目標に，アサーションメソッ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 5，表 6，表 7 に示す．各表は 表 4 に示す B1（ 表 5）と，B2（ 表 6），B3（ 表 7）それぞれのバグカテゴリを主に検出するように対応しており，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事例を 表 4 中の ID を用いて示している．図 5 における

表 4: バグカテゴリと確認された事例

バグカテゴリ	事例	件数
[B1] ファイルパスの誤り	[B1.1] 成果物が生成されない	1
	[B1.2] 公開先リポジトリへの配置に失敗	3
[B2] ファイル内容の誤り	[B2.1]JAR 内に必要なファイルが含まれない	6
	[B2.2]JAR 内マニフェストに実行エントリが存在しない	1
[B3] ログの誤り	[B3.1] 期待通りでないクラスファイル Java バージョン	1
	[B3.2] タスクが実行されない	5
	[B3.3] タスクの実行順序が期待と異なる	4
	[B3.4] テスト失敗無視（`ignoreFailures`）されない	2

表 5: 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	B1.1,B2.1
containsExecutedTestReport()	テストレポートの存在	B3.4
containsPublication()	公開物の存在	B1.2
containsBuildOutcome()	ビルド成否	B3.4
containsAnyJars()	JAR の生成有無	B1.1
containsAnyWars()	WAR の生成有無	B1.1

表 6: 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	B2.1
runsSuccessfully()	JAR 実行のリターンコードが 0 か	B2.1,B2.2
isNotEmptyJar()	JAR が META-INF 以外を含むか	B2.1
targetsJavaVersion()	クラスファイルの Java バージョン	B3.1

表 7: 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	B3.2,B3.3
containsExecutedTasksSequence()	タスクの実行順	B3.3
containsTaskOutcome()	タスク成否	B3.4
containsCompilerJdkVersion()	コンパイラの JDK	B3.1

lib.jar などの BuildArtifact 直下の成果物については，表 5 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 5 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 6 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 7 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．ここで 表 5，表 6，表 7 は基本的には B1，B2，B3 へ対応づけられるが，複数のビルド成果物へ影響を及ぼすバグも存在するため，単一のアサーションメソッドがカテゴリを跨いでバグ事例を検証できる場合がある．例えば，B3.4 のテストタスクが失敗しても無視されないバグは，ログ中のタスク実行へ影響するが同時にテストレポート生成へも影響する．そのため，ログの誤りとして B3 へ分類されるが，主に B1 へ対応する 表 5 に属するメソッドでも検証できる．

図 6 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）．GradleRunner は，スタブオブジェクトを用いて Gradle のビルドを実行するために実装したメソッドである．このメソッドでは from() メソッドの引数に生成したスタブオブジェクトを指定することでスタブを用いたビルドを可能としているが，

```
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 6: テストケース例

実プロジェクトのパスを指定することで実プロジェクトを用いたビルドも可能である．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 5 の階層構造に従い，BuildArtifact 直下の成果物 lib.jar については，表 5 の contains() で存在を確認できる（7 行目）．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 6 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 7 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）．

テストケースを記述する際の，これらのアサーションメソッドの選択手順は次の通りである．

1. テスト対象タスク（ロジック）の決定
2. 対象タスクが生成するビルド成果物の特定
3. 各ビルド成果物に対する検証項目の決定
4. 各ビルド成果物と検証項目に対応するアサーションメソッドの選択

まずテスト対象タスクを決定した後は，そのタスクが生成するビルド成果物を特定する．次に，ファイルやログなど検証対象のビルド成果物ごとにファイル生成有無やタスク成否などの検証項目を設定し，それぞれに対応する適切なアサーションメソッドを 表 5，表 6，表 7 から選択する．ビルドスクリプトのバグは出力であるビルド成果物へと影響するため，テストではこの出力を検証することによりビルドスクリプトのバグを検出できる．したがって手順に示す通り，ビルドスクリプトで定義されたタスクの挙動を検証するためには，それぞれのタスクが生成するビルド成果物を検証対象とするアサーションメソッドを選択すれば良い．例として，ビルドスクリプト内で定義された assemble タスクのテストを考える．このタスクが生成するビルド成果物としては複数の JAR であり，それぞれの JAR について生成有無や依存クラスの包含有無，実行成否などの検証項目が考えられる．そのため，表 5 中の contains() を選択し JAR 生成有無を確認し，表 6 中の contains() や runsSuccessfully() などを選択し JAR 内のコンテンツ存在や実行成否の検証を行う．

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．ここでの適用可能とは，本テストライブラリを用いてスタブの副作用を生じることなくテストを実施できるかどうかを意味する．つまり，テストケースにおいて検証の役割を持つアサーション部が実行されるまでの処理でエラーを生じず，検証対象であるビルド成果物がスタブを利用しない場合と比べ欠損なく生成されている状態を意味する．そのため，異なるプロジェクト間でも共通してアサーション部を実行するために必要となる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 6: 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．

- 評価 1：最小限テストの成功率
- 評価 2：スタブなしビルド成果物との差異

表 8: 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

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 8 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可能性のあるスタブの副作用を示している．これら 2 つの評価項目によりスタブの副作用を生じることなくテストを実施できているかという適用可能性を評価する．

実験題材として，サンプルプロジェクトや独立するサブプロジェクト同士から構成されるマルチプロジェクト構造などのテスト対象のビルドスクリプトが一意に定まら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 評価 1：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 9 に示す．これらはすべてスタブの副作用によるものである．特に，

表 9: 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

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．

4.2.3 評価 2：結果と考察

評価 1 における最小限テストが成功した 84 プロジェクトに対して，表 8 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 10 に示す．ここで，Files カテゴリにカウントされた差異が影響し，他カテゴリでカウントされてしまう副次的な差異が存在した．観測された副次的な差異としては，元のプロジェクトで JAR に含めるように指定されていたクラスファイルが，スタブの利用により生成されない（30 件），または別名で生成されてしまうことで（1 件），中身が異なる JAR を生成し JAR

表 10: 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	許容可能
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	継承テスト消失	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	×

Contents カテゴリへと影響を及ぼした（31 件）．加えて，同様の Files の差異によりクラスファイルのバージョン比較が行えず Class File Version カテゴリへと影響を及ぼした（31 件）．これらの副次的な差異は，JAR Contents と Class File Version カテゴリからは除外し，各カテゴリごとの独立した差異のみを集計した．表 10 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．しかし，3.1 節でも記載した通り，これら全ての差異がスタブの副作用として許容できないわけではない．ここで許容可能な差異とは，ビルド成果物の中でも開発者が通常生成を期待しない一時ファイルなどに影響するような，テストへ影響しない差異である．表 10 の許容可能かの列へそれぞれの差異が副作用として許容できるかを示す．これに基づき，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し，どの程度までなら副作用として許容できるかを以下で示す．

Files

許容可能な差異としては以下が挙げられる．

- 匿名クラス未生成
- ローカルクラス未生成
- 異なるファイル名

まず匿名クラスやローカルクラス未生成に関して，これらの内部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．通常，スタブ生成のための処理規則ではクラス内に宣言された内部クラスに関しても再帰的に保持されるが，匿名クラスやローカルクラスのような一時的な内部処理のために宣言される補助的なクラスに関しては，スタブ生成時にクラス宣言として認識できないため保持されずクラスファイルも生成されない．しかし，これらの補助的なクラスのクラスファ

イルは開発者が明示的に参照・操作する対象ではない．これらのクラスファイルは通常 `OuterClass$1` のように連番付きで生成され，ビルド結果の検証や公開の際に個別に認識されない．そのため，スタブの利用によりこれらのクラスファイルが欠落しても，開発者が期待するであろうビルドスクリプトが本来生成すべき主要なクラスファイルには影響しない．したがって，この差異はスタブの副作用として許容可能である．

次に，jgitver プラグイン適用により異なるファイル名の生成に関しては，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許容可能である．一方で，許容できない差異としては以下が挙げられる．

- Builder.class 未生成
- マッピングファイル未生成
- package-info.class 未生成

これら全ての差異は共通して依存関係先のアノテーションの隠蔽に起因している．また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．そのため，これらのファイルの生成をテストで検証する際にファイルが未生成だと，ビルドスクリプト内のタスク定義が正しいにも関わらずテストに失敗し，タスクが正常に動作していないと開発者に誤解を与える可能性がある．したがってこれらの差異は許容できない副作用である．

JAR Contents

許容可能な差異として匿名クラス未生成が挙げられる．これは Files の場合と同様にスタブの副作用として許容可能である．一方で，許容できない差異としては以下が挙げられる．

- Javadoc 関連ファイル未生成
- Mixin 等設定ファイル未生成
- 未使用依存関係除外

まず，Javadoc 関連の HTML ファイルや Mixin 等の設定ファイルの未生成などは，Javadoc コメントや `Mod` アノテーション等の隠蔽に起因している．これらのファイルは，Files での許容できない差異と同様に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．よってこれらのファイルが生成されないと，開発者へタスクが正常に動作していないと誤解を与える可能性があるため，これらの差異は許容できない副作用である．また未使用依存関係除外については，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 から除外されることで差異を生じる．スタブでは可能な限り参照を隠蔽するため，利用する依存関係はテストケースに関するもののみ

であり大部分のクラスファイルが JAR 内から削除されてしまう．よってビルドスクリプトで定義した JAR に含めるクラスパスが誤っていると開発者に誤解を与える可能性があるため，この差異は許容できない副作用である．

Jar Return Code

許容できない差異として成功する実行可能 JAR 生成が挙げられる．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ため，この差異は許容できない副作用である．

Task Execution Order

許容可能な差異として，タスクの実行数減少・早期終了が挙げられる．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．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

許容できない差異として継承テスト消失が挙げられる．スタブにより，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が隠蔽され `test` タスクで実行されず，それらのテストケースに対応するテストレポートが生成されない差異が生じた．テストレポートは，Files での許容できない差異と同様にビルドスクリプトで定義した `test` タスクにより自動生成される．よってテストレポートが生成されないと，開発者へ `test` タスクが正常に動作していないと誤解を与える可能性があるため，これらの差異は許容できない副作用である．

4.2.4 スタブ副作用への対策

評価 1 と評価 2 で現れたビルドプロセスの失敗及びビルド成果物への差異といったスタブの副作用の内容を基に，スタブ作成方針を示す．

ビルドプロセスの失敗

評価 1 にて観測されたスタブの副作用は 表 9 に示す通りであり，各失敗原因ごとに対策を考える．まず他言語ファイルや構成ファイル，タスクからの参照先がスタブで存在しないような依存欠如によるビルド失敗に関しては，スタブの保持メソッドの利用が考えられる．参照される構文要素を保持するスタブを生成することで，ソースコード中のビルド失敗要因を隠蔽しながら参照によるビルド失敗も防ぐことができる．ただし，保持メソッドを用いたとしても依存が多く失敗してしまう場合に関しては，実プロジェクトを用いたビルド実行が考えられる．3.2 節でも示すようにビルド実行を行う `GradleRunner` で元のプロジェクトを指定することで，スタブを用いずにビルドを実行し副作用を防ぐことができる．ただしスタブを利用しないため，Java ソースコード内に実行時エラー等のビルド失敗要因が含まれる場合，それらは隠蔽されずビルドが失敗する可能性がある．そのため，スタブ対象外のファイルからの参照を事前に把握しており，かつ参照先の構文要素の依存が少ない場合には，保持メソッドを用いて参照先の構文要素を保持したスタブの作成を推奨する．また，参照を事前に把握できない場合には一度スタブを用いて実行し，ビルド失敗時に参照先を把握し，保持メソッドで対応可能な構文要素であれば参照先を保持したスタブの作成を推奨する．また，保持メソッドで対応できない場合にはスタブを用いず元のプロジェクトを用いたビルドの実行を推奨する．

次に静的解析違反や動作期待未達によるビルド失敗に関しては，`GradleRunner` で元のプロジェクトを用いたビルドが考えられる．4.2.2 節にも記載する通り，フォーマット規則は全てのプロジェクトで固有に設定可能であるため，スタブが固有の規則に対応できず違反する可能性が高い．またテストカバレッジ等の動作期待を設けている場合，これを達成するためには該当するメソッド等の全ロジックを保持する必要があるが，テストケース等のロジックは基本的には別クラスのメソッドなどに依存するため保持メソッドによる対応は困難となる．そのためこれらのタスクを実行するビルドスクリプトでは，元のプロジェクトを用いたビルドの実行を推奨する．

ビルド成果物への差異

評価 2 で観測されたスタブの副作用のうち，アサーションメソッドの検証に影響しない許容可能な差異については，テスト結果に影響を与えないため特段の対応は不要である．そのためスタブ作成方針を検討すべき対象は，表 10 に示す許容可能列で許容できないとされた，検証に影響を及ぼす差異に限られる．これらの差異を防ぐための対策を以下に示す．

まず Builder.class やマッピングファイル未生成，設定ファイル未生成，Javadoc 関連ファイル未生成，継承テスト消失といったアノテーションや Javadoc コメント，テストケースなどの特定の構文要素がスタブで隠蔽されることでビルド成果物に差異を生じる場合は，スタブの保持メソッドの利用が考

えられる．保持メソッドによりこれらの構文要素を保持するスタブを生成することで，正常にビルド成果物を生成できる．ただし，これらの構文要素を保持することにより参照不整合が生じてしまう場合には，ビルドプロセスが失敗する副作用への対策と同様に `GradleRunner` で元のプロジェクトを用いてビルドを実行することでビルド成果物を生成できる．そのため，検証したいファイルの生成に必要な構文要素がスタブにより隠蔽されることを把握しており，かつ保持することで参照不整合を生じない場合には，保持メソッドを用いてこれらの構文要素を保持したスタブの作成を推奨する．また把握できない場合には，一度通常のスタブを用いて未生成のファイルを検証するテストを実施する．そのテストが失敗した際には，Java ソースコード内に該当する構文要素が存在しないことを確認し，保持メソッドで対応可能な構文要素であればそれらを保持したスタブの作成を推奨する．また，保持したい構文要素の依存先が多いなど保持メソッドでは対応できない場合には，スタブを用いず元のプロジェクトを用いたビルドの実行を推奨する．

次に，JAR 内の未使用依存関係の除外や常に正常終了する JAR が生成されてしまう差異については，`GradleRunner` で元のプロジェクトを用いたビルドが考えられる．4.2.3 節にも記載する通り，これらは依存の多い `main` メソッドやテストケースなどのロジックを保持する必要があり，保持メソッドでの対応は困難となる．そのため，JAR の実行検証などの依存の多いロジックを保持したままビルドを実行する必要があるテストケースの場合は，元のプロジェクトを用いたビルドの実行を推奨する．

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．巨大なシステムを開発する際に，システムをより細かな部品に分解して開発を進めるという考え方は広く受け入れられており [18]，この部品化の考えは Java ソースコードとビルドスクリプトにも適用できる．これによりシステムの本質的なロジックとは独立に，システムのビルド方法も開発を進めることができる．このときに，ソースコードとは切り離されたビルドスクリプトのテストを用意しておくことで，ソースコードの影響を可能な限り軽減し，ビルドスクリプトそのものの正しさを自動検証できる．そのため本実験では，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を評価する．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 11 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 11: 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 4 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（B2.1）
- プロジェクト 2：ビルド続行阻止（B3.4）

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 7 に示す．図 7a 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 7b 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 8 に示す．図 8a はテストケースであり，スタブの作成とビルド実行は 図 6 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できる（3-4 行目）．図 8a のテストケースによるバグ検出時の出力を 図 8b に示す．図 8b 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．

```
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 7: 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 8: JAR に関するテストケースと検出時の出力

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 9 に示す．図 9a 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）．

```
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 9: バグを含むビルドスクリプトとビルド実行時出力

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 9b 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．

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 10 に示す．図 10a はテストケースであり，ビルド実行は 図 6 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）．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 10a のテストケースによるバグ検出の出力を 図 10b に示す．図 10b 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．

```
@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 10: ビルド続行に関するテストケースと検出時の出力

6 関連研究

ビルド失敗に関する原因を調査する研究が数多く行われている [19][20][21][22]．Lou らは [23]，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 [24]，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．

加えて，ビルド工程に存在する欠陥を検出する研究が数多く行われている [25][26][27][28]．Sotiropoulos らは [29]，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 [30]，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 [14]．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．

7　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，バグ事例の調査及びアサーションメソッドの実装が挙げられる．現時点でのアサーションメソッドは，表 4 に示すバグ事例の検出を目的として実装している．しかし，これらが現時点でビルドスクリプトのバグの一部を示すことから，より多くのバグへ対応するにはバグ事例の追加調査及びそれに基づく新たなアサーションメソッドの実装が求められる．

また今後の拡張として，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可能な汎用性を持つ．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

本研究を遂行するにあたり，多数の方にご協力いただき誠にありがとうございました．この場を借りて深く感謝の意を表します．

楠本真二教授には，研究の方向性について常に広い視点から忌憚のないご意見をいただき，自らの思索では及び得なかった多くの新たな視点を示唆していただきました．また，研究室での日々の生活においても，折に触れて温かいお心遣いや対話を通じて，研究活動の活力をいただきました．深く感謝申し上げます．

柗本真佑准教授には，学部生の頃から現在に至るまで多大なるご指導を賜りました．当初は研究の進め方や執筆において未熟な点が多々あり，度重なる再考や修正を余儀なくされました．そんな私に対して，何度もご相談に乗っていただき，時には休日を返上してまで根気強く推敲にお付き合いいただきました．また，先生の研究に対する真摯な姿勢と，学生一人ひとりを深く気遣われるお姿に触れ，研究者としてのみならず一人の人間としても多くを学ばせていただきました．本研究の完遂は偏に先生のご助力の賜物であります．心より御礼申し上げます．

楠本研究室事務補佐員の橋本美砂子様には，3 年間という長きにわたり，公私ともに多大なるご支援をいただきました．特に私が研究室の役割を担っていた際には，親身になって相談に乗っていただき，細やかなお心遣いに幾度となく助けられました．また，共通の趣味や日常のたわいもない会話を通じて，研究の合間に安らぎの場を提供していただきました．ここに深く感謝の意を表します．

ご卒業されました先輩方には，私が未熟であった頃から長きにわたる多大なるご指導を賜りました．常に高い志を持って研究に励まれる先輩方の姿は，私にとって何よりのお手本であり，先輩方が築き上げられた道標を辿ることで，迷うことなく研究を進められました．また，いつ研究室を訪れても誰かが真摯に研究に取り組んでいる環境は，私にとって大きな刺激となり日々の研究の糧となりました．研究面のみならず，日々のたわいもない雑談や趣味を通じた交流は，研究の合間の何よりの安らぎであり，心の支えとなりました．これまでの温かいご支援に，心より深く感謝申し上げます．

研究室の同期の方々には，その能力と主体的な行動力に多くの刺激を受け，多くのことを学ばせていただきました．また，私にはない独自の視点から本研究へ多くの貴重な助言をいただき，そのおかげで論文を改善し続けることができました．また，研究の合間での語らいや親睦を深めた時間は，私の研究生活における大きな活力となりました．心より感謝いたします．

また，研究室の学生の方々には，日々の研鑽に励む誠実な姿を通じて，私自身も研究への意欲を新たにすることができました．自ら学び道を切り拓いていく皆様の姿勢を拝見し，大いに鼓舞されました．研究室での温かい交流が私の大きな支えとなりましたことに感謝いたします．

最後に，今日まで私を支え，温かく見守り続けてくれた家族に深く感謝いたします．大学院での二年

間という長い期間において，困難に直面した際の大きな精神的支えとなりました．

末筆ながら，本研究を支えてくださったすべての方に心より感謝いたします．

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] Fan, G., Wang, C., Wu, R., Xiao, X., Shi, Q. and Zhang, C.: Escaping dependency hell: finding build dependency errors with the unified dependency graph, in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 463–474 (2020).

[3] Hilton, M., Tunnell, T., Huang, K., Marinov, D. and Dig, D.: Usage, costs, and benefits of continuous integration in open-source projects, in *In Proceedings of International Conference on Automated Software Engineering*, pp. 426–437 (2016).

[4] Liu, P., Li, L., Liu, K., McIntosh, S. and Grundy, J.: Understanding the quality and evolution of Android app build systems, *Journal of Software: Evolution and Process*, Vol. 36, No. 5, p. e2602 (2024).

[5] 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).

[6] 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 *In Proceedings of International Symposium on Empirical Software Engineering and Measurement*, pp. 38–47 (2017).

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

[8] Kerzazi, N., Khomh, F. and Adams, B.: Why do automated builds break? an empirical study, in *In Proceedings of International Conference on Software Maintenance and Evolution*, pp. 41–50 (2014).

[9] Kang, M., Kim, T., Kim, S. and Ryu, D.: Gradle-Autofix: An Automatic Resolution Generator for Gradle Build Error, *International Journal of Software Engineering and Knowledge Engineering*, Vol. 32, No. 04, pp. 583–603 (2022).

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

[11] Lyu, J., Li, S., Liu, B., Zhang, H., Rong, G., Zhong, C. and Liu, X.: Detecting Build Depen-

dency Errors by Dynamic Analysis of Build Execution Against Declaration, *Transactions on Software Engineering*, Vol. 51, No. 6, pp. 1745–1761 (2025).

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

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

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

[15] Xiong, J., Shi, Y., Chen, B., Cogo, F. R. and Jiang, Z. M.: Towards build verifiability for java-based systems, in *In Proceedings of International Conference on Software Engineering: Software Engineering in Practice*, pp. 297–306 (2022).

[16] Wu, R., Chen, M., Wang, C., Fan, G., Qiu, J. and Zhang, C.: Accelerating build dependency error detection via virtual build, in *In Proceedings of International Conference on Automated Software Engineering*, pp. 1–12 (2022).

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

[18] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L.: Microservices: yesterday, today, and tomorrow, *Journal of Present and ulterior software engineering*, pp. 195–216 (2017).

[19] Tamanna, M., Chandrani, Y., Burrows, M., Wroblewski, B., Williams, L. and Wermke, D.: Your Build Scripts Stink: The State of Code Smells in Build Scripts, in *In Proceedings of International Conference on Automated Software Engineering* (2025).

[20] Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E. and Bowdidge, R.: Programmers' build errors: a case study (at google), in *In Proceedings of International Conference on Software Engineering*, pp. 724–734 (2014).

[21] Beller, M., Gousios, G. and Zaidman, A.: Oops, my tests broke the build: An explorative analysis of travis ci with github, in *In Proceedings of International Conference on Mining Software Repositories*, pp. 356–367 (2017).

[22] Bezemer, C.-P., McIntosh, S., Adams, B., German, D. M. and Hassan, A. E.: An empirical study of unspecified dependencies in make-based build systems, *Journal of Empirical Software Engineering*, Vol. 22, No. 6, pp. 3117–3148 (2017).

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

[24] 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 *In Proceedings of International Conference on Mining Software Repositories*, pp. 345–355 (2017).

[25] Yasi, T. and Qin, J.: Automatic building of java projects on GitHub: A study on reproducibility (2022).

[26] Lee, J., Li, M. and Hsu, K.-H.: Applying Transformer Models for Automatic Build Errors Classification of Java-Based Open Source Projects, in *In Proceedings of International Conference on Software Engineering: Companion Proceedings*, pp. 282–283 (2024).

[27] Licker, N. and Rice, A.: Detecting incorrect build rules, in *In Proceedings of International Conference on Software Engineering*, pp. 1234–1244 (2019).

[28] Soto-Valero, C., Harrand, N., Monperrus, M. and Baudry, B.: A comprehensive study of bloated dependencies in the maven ecosystem, *Journal of Empirical Software Engineering*, Vol. 26, No. 3, p. 45 (2021).

[29] 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).

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