



Woodland  
Academy

## 6.7 開発テスト

- テストの基本
- Spring でのテスト



*Shape Your Future*

# 目次

- 1 テストの基本
- 2 Spring でのテスト

# 開発テスト

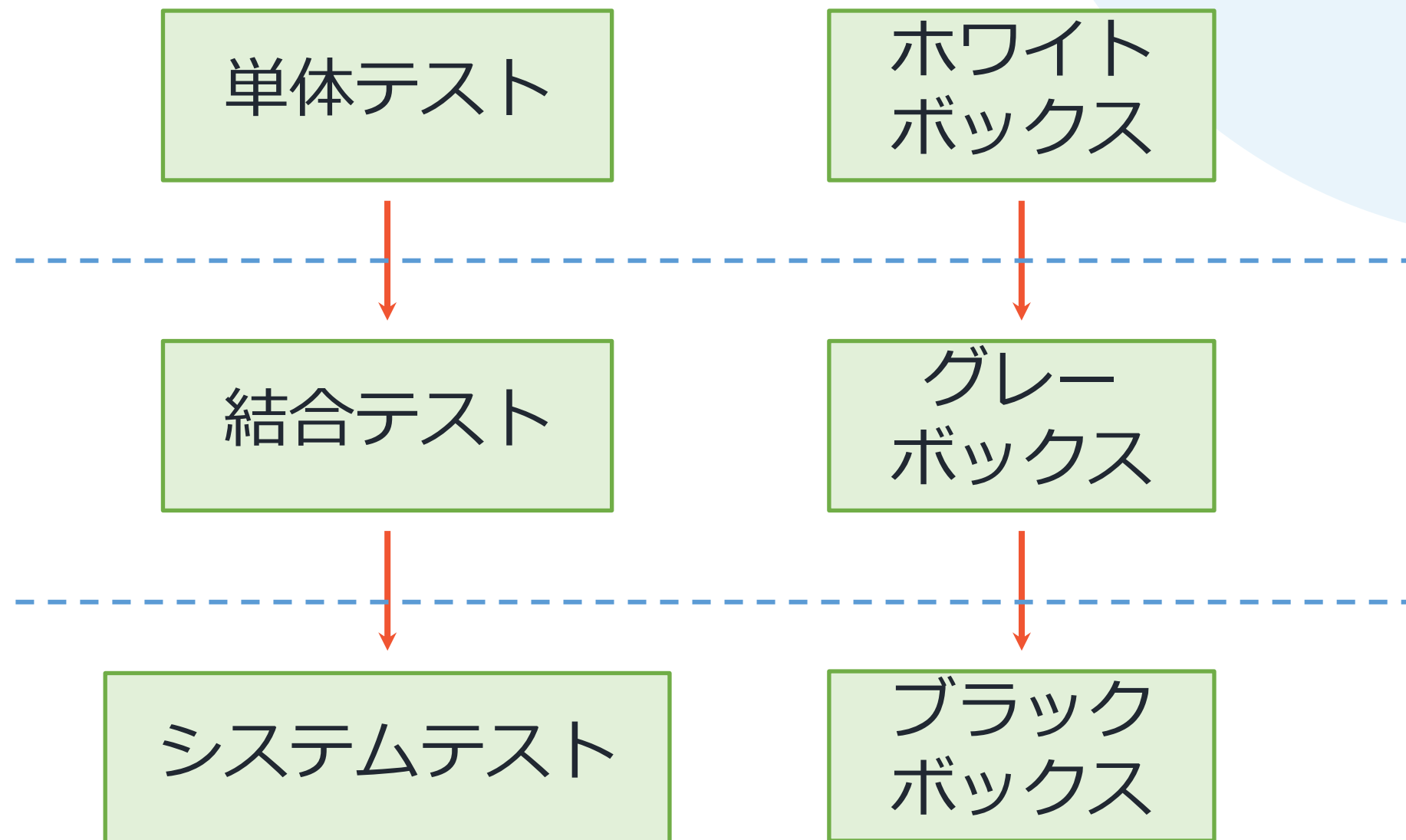
- ソフトウェア開発では、コーディングした後、そのコードが正しいかどうかをテストするために、あらゆる状況を想定して実行する必要があります。しかし、手作業でのテストは時間と手間がかかり、特殊なケースを見逃してしまう可能性があります。
- **開発テスト**<sup>[Development Testing]</sup>とは、既に書かれたコードの正しさをテストするために、テスト専用のコードを追加で書くことによって、**テストの流れを自動化**したものです。

# テストケース

- 開発テストを正しく行うためには、あらゆる考えられる実行状況、即ち**テストケース**[Test Case]を模擬し、機能要件に従って各ケースで期待される結果を出すためのコードを書く必要があります。
- テストコードを実行した結果、どのような場合にコードが正しく動作し、機能要件を満たしているのか、どのような場合にコードが期待通りに動作できないのかを知ることができます。後者の場合、既存のコードを修正し、すべてのテストが合格するまでテスト → 修正の繰り返しをすることが必要です。

# テストの 3 つの段階

- 開発テストは大きく分けて 3 つのフェーズに分けられ、それぞれ 3 種類のテストに対応しています：





# 単体テスト

- **単体テスト**[Unit Test, UT]とは、最小限の単位あたりのコードを、**独立的にテスト**します。
- 従来のプロセス指向開発では、最小単位はプログラムや、関数であったのに対し、現代のオブジェクト指向開発では、最小単位は一般にクラスの**メソッド**を指します。
- 単体テストの目的は、ソフトウェアを構成する個々の部分が、独立して正しく動作することを証明することです。したがって、単体テストは一般的に**最初に行う**テストです。
- Java (Spring) は非常に便利な単体テストのフレームワークを提供しています。ほとんどの場合、これらのフレームワークを使用して単体テストのコードを作成します。

# 単体テストの重要性


- 信頼性が高い単体テストは、製品開発の初期段階のどこかの段階で多くのバグを発見し、修正にかかる費用を少なくすることができます。システム開発の後期になると、バグの発見と修正が難しくなり、多くの時間と開発コストを消費することになります。
- したがって、製品のコードに変更を加える場合は、常に完全な**回帰テスト** [Regression Testing] を実施する必要があります。ライフサイクルの早い段階で製品のコードをテストすることは、開発効率と製品品質を保証する最良の方法です。
- 健全な単体テストは、システム統合のプロセスを大幅に簡素化することができます。開発者は、バグだらけの個々のユニットにこだわることなく、ユニット間の相互作用や全体の機能実装に集中することができます。

# 優れた単体テスト

- 優れた単体テストは、**素早く、反復可能で、自動化される**べきです。
  - ソフトウェアの単体テストをできるだけ早く実施することは、その後の開発テストを円滑に進めるためのカギになる。
  - コードの拡張や修正に対応できるよう、単体テストのコードを繰り返し行えることを確保すること。
  - 高速化と重複作業の削減のため、単体テスト活動には可能な限り自動テスト手段を使用すること。
- テストの生産性を最大化する鍵は、適切なテスト戦略を選択することです：単体テストの概念を完全に理解するだけでなく、テストプロセスを適切に管理し、テストプロセスをサポートする優れたツールを使用することも必要です。

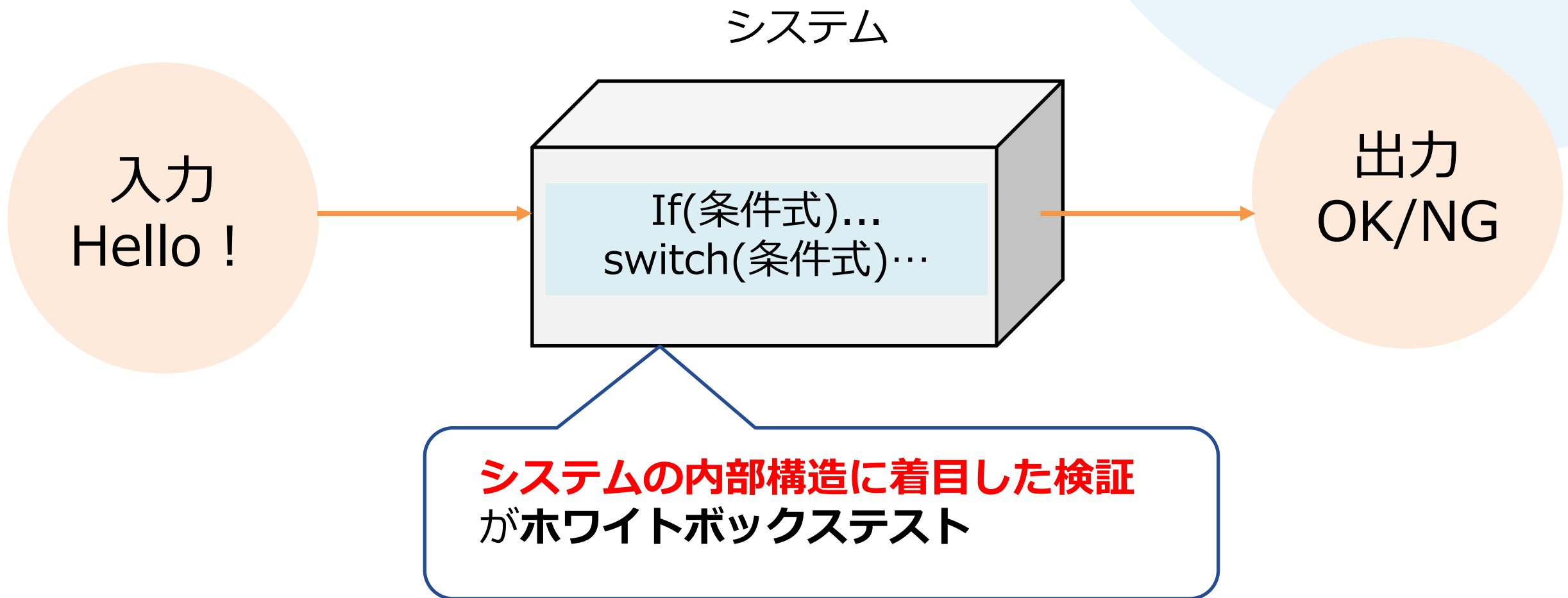


# 単体テストとカプセル化

- すでに知っている（ § 2.3.2）ように、**カプセル化**は、オブジェクト指向において最も重要な概念の一つです。特定のメソッドや属性は、外部クラスからアクセスされないようにする必要があります。
- しかし、単体テストでは、これらのプライベートメソッドの**具体的な実装方法**を理解する必要があります。これらのコードは、テストの失敗の原因となる可能性があるからです。
- そのため、単体テストではよく**ホワイトボックス**<sup>[White Box]</sup>**テスト**モデルを採用しています。単体テストのコードは、実際にテストされるユニットを**開発したプログラマ**が書くことになります。これらのプログラマは、これらのユニットの基本的な実装を理解し、テストの結果に基づいて、プライベートメソッドのコードを修正することができます。

# ホワイトボックステスト

- ホワイトボックステストは、システムの内部構造（プログラムの内容）が意図した通りに動作するかを確認するテスト技法です。



# カバレッジ

- ホワイトボックステストにおける**カバレッジ**<sup>[Coverage]</sup>とは、プログラムの全コードのうち、テスト中に実行されるコードの割合のことです。
- 一般に、単体テストのカバレッジは可能な限り 100% にしたいものです。もし、カバレッジが 100% でなければ、開発中に考慮した全ての状況をカバーしていないことを意味し、いくつかの新しいテストケースを書く、あるいは冗長で無意味なコードを削除する必要があります。
- カバレッジはあくまでテスト完了の参考基準の一つであり、100% になっても、**必ずしもコードに問題がないことを証明するわけではない**ことに注意しつつこと。例えば、コードもテストケースも考慮されていない特殊なケースが実際にあるかもしれません。

# カバレッジの基準

- 100% のカバレッジ、即ち「**網羅**」という概念の基準を大きく 3 つに分類できます：
  - **命令網羅**[Statement Coverage]：コードの**すべての行**が実行されました。
  - **分岐網羅**[Branch Coverage]：すべての行が実行され、かつ、各分岐文は条件が**真の時**も、**偽の時**もそれぞれ少なくとも一回実行されました。
  - **条件網羅**[Condition Coverage]：分岐網羅の条件に基づき、判定条件が複数の条件の組み合わせである場合、各**条件の真偽のすべての組み合わせ**が発生しました。

- 例えばこのコードに対し：

```
1 if (x > 3 && y < 4) {  
2     System.out.println(x - y);  
3 }  
4 System.out.println(x + y);
```

- 命令網羅を実現するためには、 $x = 4$ 、 $y = 3$  の場合のみテストする必要があります。
- 分岐網羅を実現するためには、少なくとも  $x = 3$ 、 $y = 3$  の場合をもう一度テストして、条件が成立しないときをテストする必要があります。
- 条件網羅を実現するために、 $x = 4$ 、 $y = 5$  のような他の条件の組み合わせもテストする必要があります。



# 結合テスト

- **結合テスト** [Integration Testing, IT] は、**2 つ以上の依存関係があるモジュールを全体**としてテストするテストです。
- 普通、単体テストの後に結合テストが行われます。単体テストに合格したユニットは、それ自体では正しく機能することが保証されていますが、他のユニットと相互作用したときに問題を引き起こす可能性があります。結合テストの目的は、異なるユニットが相互作用し、協力する場合でも、意図した機能が正しく実装されていることを確認することです。
- 多くのプロジェクトでは、**複数の開発者**が異なるモジュールやユニットのコードを記述することになります。そのため、異なる人が書いたコードで構成されるシステムが正しく動作することを、結合テストによって検証することが不可欠です。

# 結合テストの要点

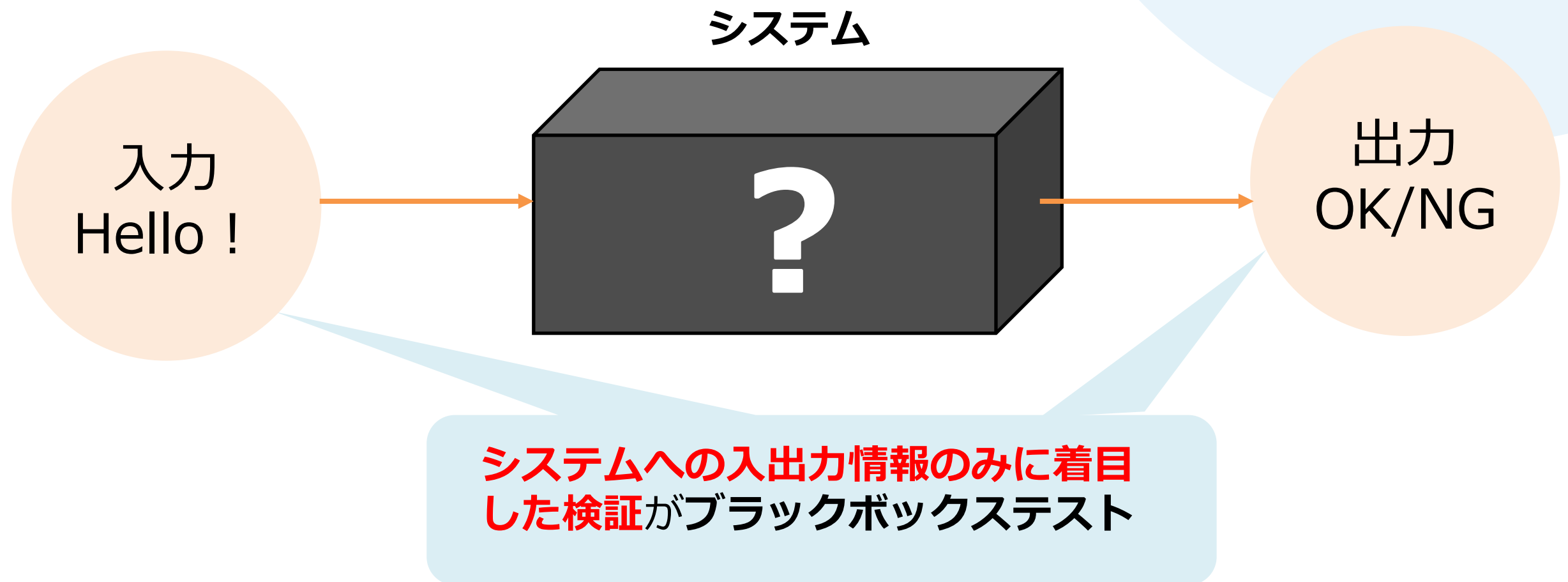
- 結合テストは、以下の 2 種類（5 つ）の課題を優先的に取り組む必要があります：
  - モジュール間のインタフェース（インタフェースのカバレッジ）：
    1. 各モジュールを接続する際、モジュールインタフェースのメソッドを使うことによって情報が失われることはありませんか。
    2. 全体のデータ構造に問題はあって、データが予期せずに変更される恐れはありませんか。
  - 結合後の機能（パラメータの受け渡し）：
    3. 各機能の部分を組み合わせて、必要な全体の機能を実現できますか。
    4. あるモジュールの機能が、他のモジュールの機能に悪影響を与えませんか。
    5. 個々のモジュールにエラーが蓄積され、許容できないレベルまで拡大されていませんか。

# 結合テストを行う方式

- 結合テストの基本的な目的は、機能が正しく動作することを検証することです。そのため、**ブラックボックス**<sup>[Black Box]</sup>や、**グレーボックス**<sup>[Gray Box]</sup>テストがよく採用され、テストする方は、内部のコード実装を理解せずに、要求された機能だけに基づいてテストケースを選択します。
- また、ソフトウェアシステムには非常に多くのモジュールや階層があるため、最初から全てのモジュールを結合してテストしてしまうと、問題が発生したときに、不具合のある部分を特定することが難しくなってしまいます。そのため、実際には、まず少数のモジュールを結合してテストを行い、テストに合格してから**新たなモジュールを結合**して、すべてのモジュールが結合テストに合格するまでこれを**繰り返**します。モジュールを追加する順番によって、**ボトムアップ**<sup>[Bottom-up]</sup>型、**トップダウン**<sup>[Top-down]</sup>型などの方式があります。

# ブラックボックステスト

- ブラックボックステストは、**システムの内部構造（プログラムの内容）は意識せず**、仕様を満たしているかを確認するテスト



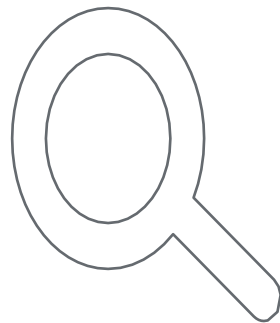
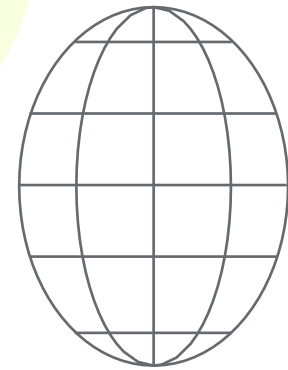
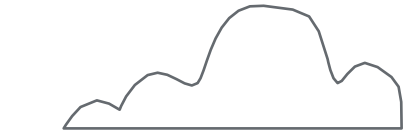
# システムテスト

- **システムテスト** [System Testing, ST] は、実稼働環境における**製品コード全体**の完全なるテストのことです。
- システムテストでは、統合テストに合格ソフトウェアは、コンピュータシステムの一部としてシステムの他の部分と統合され、**実際の動作環境**において一連のテストが実施されることとなります。ソフトウェアの潜在的な問題を特定し、システムが適切に機能することを保証します。
- 結合テストとシステムテストの比較：
  - テストの内容：結合テストは各モジュール間の結合をテストし、システムテストはシステム全体の機能と性能をテストします。
  - テストの視点：結合テストは技術的な観点に、システムテストはビジネス的な観点に注目します。





# Q&A




# 目次

1 テストの基本

2 Spring でのテスト

# JUnit

- **JUnit** は、Java で最も有名な単体テストのフレームワークです。JUnit を使うことで、単体テストのコードを統一的かつ効率的に開発することができ、テストの結果も明確で読みやすいものにしてくれます。主に**単体テスト**に使用されますが、**結合テスト**の記述もある程度サポートします。
- Spring Bootは、最新版の JUnit 5 をサポートし、使用を推奨しています。
- JUnit 5 の完全な API と詳細なチュートリアルは、公式ドキュメントに記載されています：  
 <https://junit.org/junit5/docs/current/user-guide/>

# JUnit の利用

- Spring Boot には JUnit のコンポーネントが統合されています（spring-boot-starter-test）。依存関係を追加せずに、直接テストコードを書き始めることができます。
- Spring 以外のプロジェクトで JUnit を使用したい場合は、対応する Maven の依存関係を追加するだけです：

```
1 <dependency>
2     <groupId>org.junit.jupiter</groupId>
3     <artifactId>junit-jupiter-engine</artifactId>
4     <version>5.4.0</version>
5     <scope>test</scope>
6 </dependency>
```

# テストクラスの宣言

- 通常、テストコードを開発コードとは別のテストディレクトリに配置します。単体テストでは、テストされる各クラスに対応するテストクラスを作成します。それらの**パッケージ名とクラス名**は、テストされるクラスと一致させます：

```

▼ 📁 src/main/java
  ▼ 📁 net.lighthouseplan.spring.junit
    ▼ 📁 controllers
      > 📄 LoginController.java
      > 📄 ReigisterController.java
    > 📁 models
    > 📁 repositories
  ▼ 📁 services
    > 📄 AccountService.java
    > 📄 DemoMethods.java
  
```

```

▼ 📁 src/test/java
  ▼ 📁 net.lighthouseplan.spring.junit
    ▼ 📁 controllers
      > 📄 LoginControllerTest.java
      > 📄 RegisterControllerTest.java
    > 📁 integration
  ▼ 📁 services
    > 📄 AccountServiceTest.java
    > 📄 DemoMethodsTest.java
  
```

Try  SpringTest



# テストメソッドの宣言

- 単位テストのテスト対象になるユニットは、オブジェクトやクラスのメソッドです。通常、テスト対象のクラスの各メソッドに対応するテストメソッドを宣言します。テストメソッドは **public** に設定すべきです。**パラメータと戻り値を持つべきではありません。**
- JUnit では、**@Test** アノテーションを使って、メソッドがテストメソッドであることを宣言します：

```

1 @Test
2 public void testIsOdd() {
3     // Test code
4 }

```

# 幅広い状況に対応するテストメソッド

- 同じテストメソッドであっても、入力値の違いによって結果が異なる場合があります。例えば、サービスクラスでユーザ名やパスワードを検証するメソッドは、成功することもあるれば、ユーザ名やパスワードが正しくないために失敗することもあります。これらのケースをテストするためには**別々のテストメソッド**を定義する必要があります：

```
// 正しいユーザーネームとパスワードを入力した場合
public void testValidateAccount_CorrectInfo_ReturnTrue() { }

// 間違ったユーザーネームを入力した場合
public void testValidateAccount_WrongUsername_ReturnFalse() { }

// 間違ったパスワードを入力した場合
public void testValidateAccount_WrongPassword_ReturnFalse() { }
```

# アサーション

- **アサーション**<sup>[Assertion]</sup>は、特定の変数やオブジェクトが満たすべき条件の宣言を指します。テストメソッドの中で、  
「テストメソッドを実行した後、これらの変数はこれらの条件を満たすべきである」という複数のアサーションを宣言することができるのです。
- テストコードを実行した際に、これらのアサーションがすべて成立していれば、そのメソッドはテストに合格しています。一つでもアサーションが成立しない場合、テストは直ちに終了して例外を投げるので、実際の実行結果や期待した結果との違いがわかり、コードの修正に役立ちます。

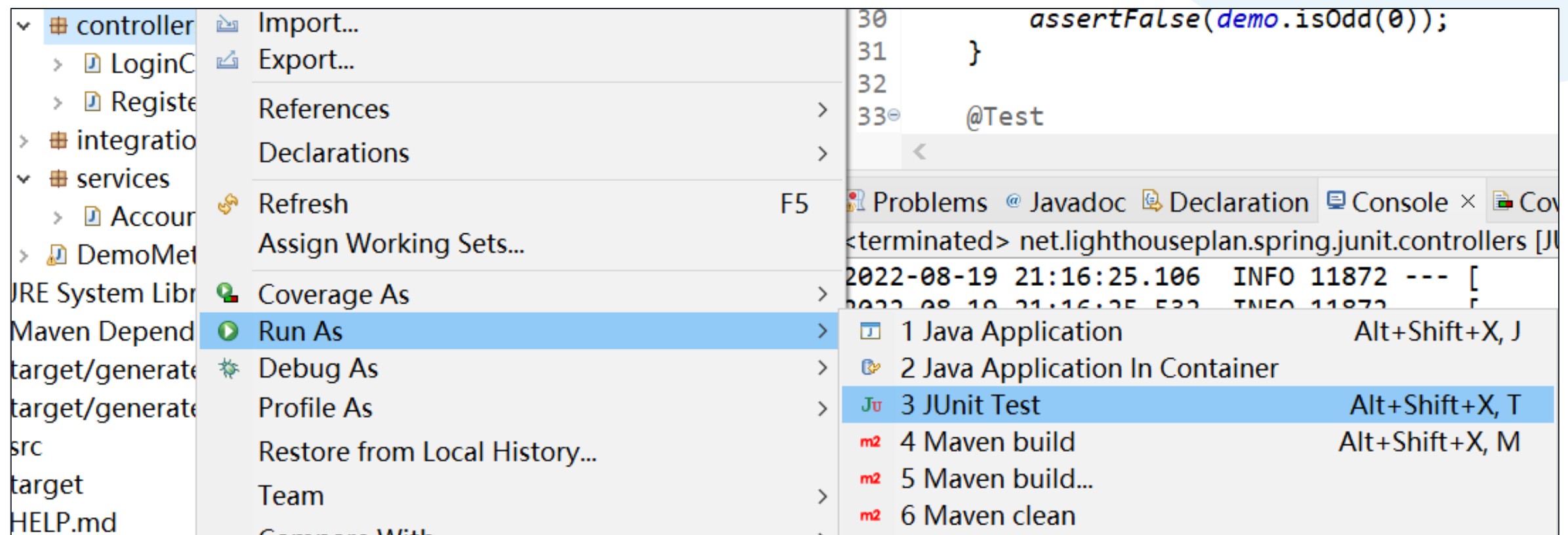
# アサーションメソッド

- JUnit では、**assertXxx()** 系のメソッドを用いて**アサーションを宣言**します。よく使われるメソッドは：

メソッド	説明
assertNull() assertNotNull()	オブジェクトが null・null じゃないことを宣言
assertEquals() assertNotEquals()	変数値が等しい（近い）・等しくない（近くない）ことを宣言
assertSame() assertNotSame()	オブジェクト（の参照）が同じ・異なることを宣言
assertTrue() assertFalse()	ブール式が true・false であることを宣言
assertThat()	様々な条件が成立することを宣言

# テストの実行

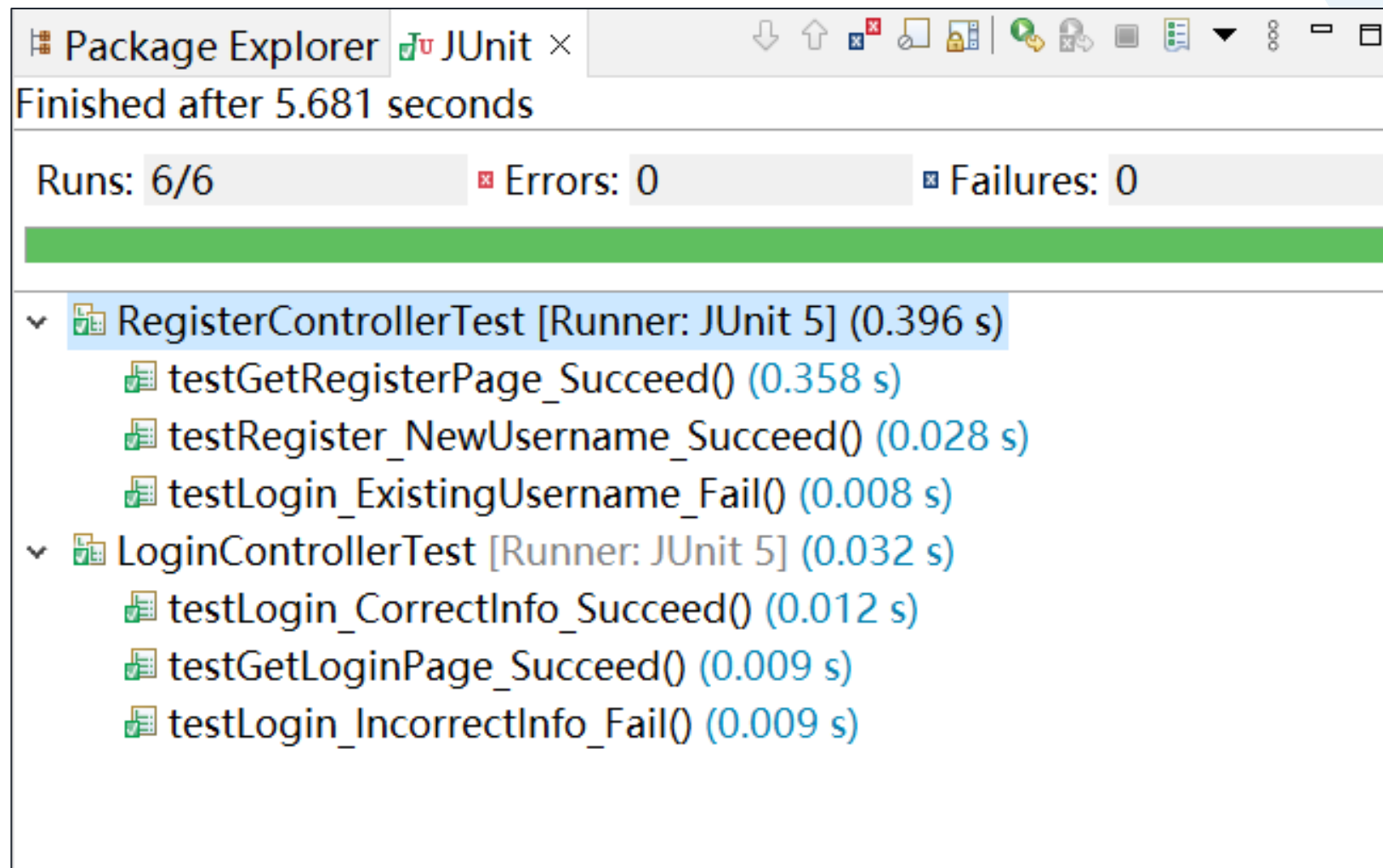
- テストメソッドを書いたら、単一のメソッドか、クラス内のすべてのメソッドか、パッケージ内のすべてのメソッドかのどちらかを実行するように選択できます。メソッド、クラス、パッケージのどちらかを右クリックして、Run As → JUnit Test を選択するだけです：





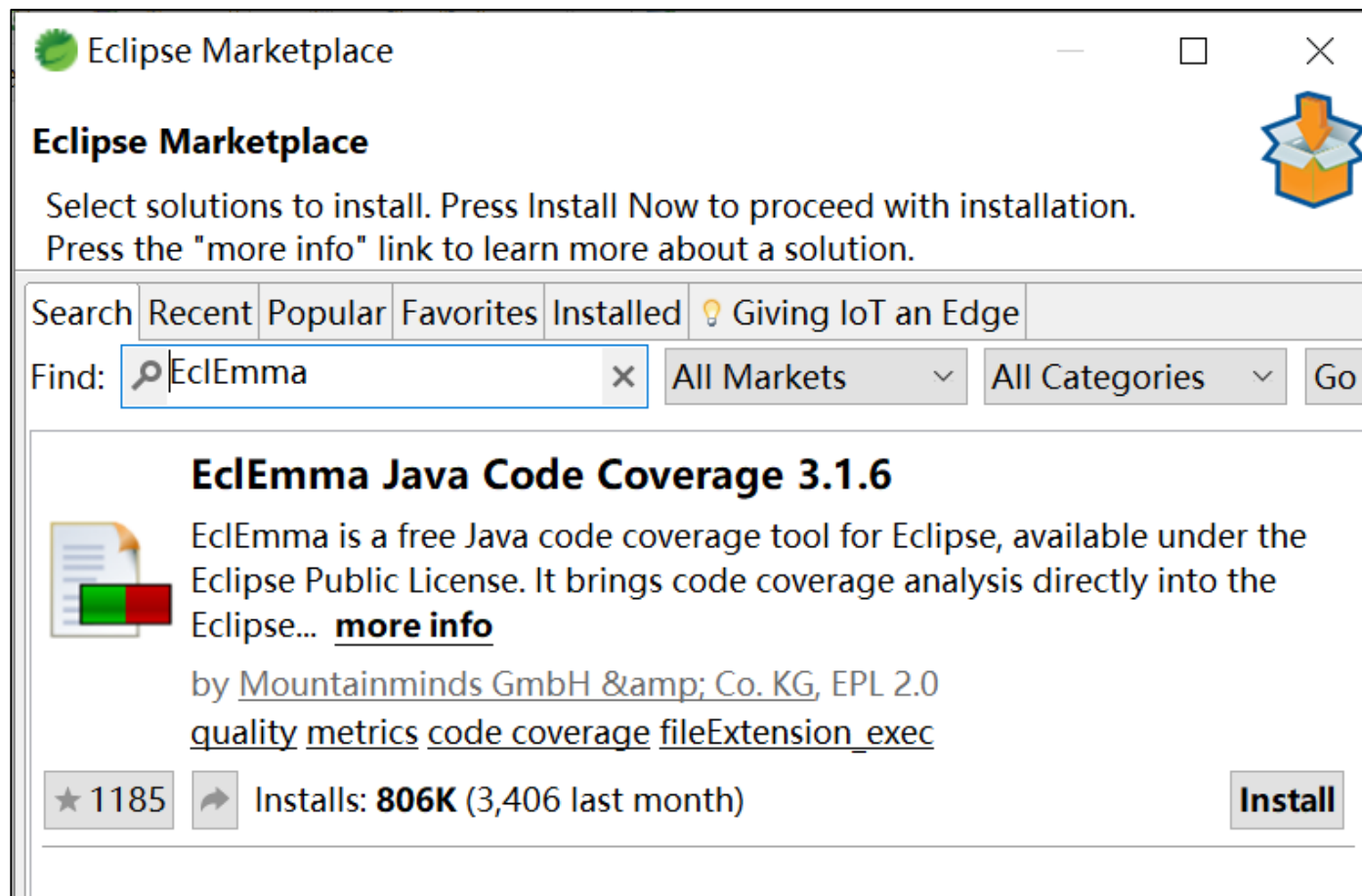
# テストの結果

- パッケージブラウザの右側にある JUnit タブをクリックすると、各テストメソッドの結果が表示されます：



# EclEmma のインストール

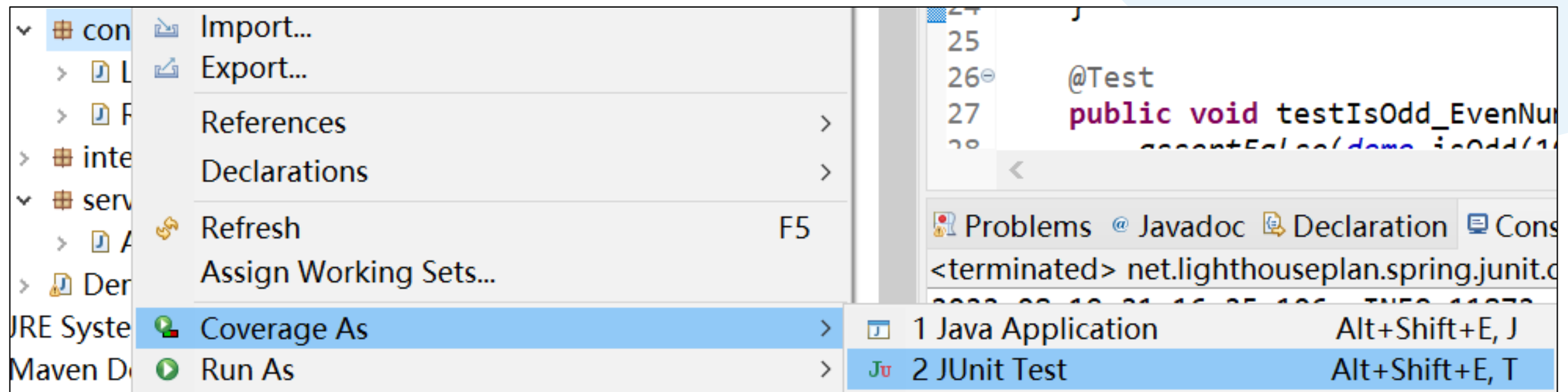
- テストの**カバレッジ**を確認するには、追加のプラグインのインストールが必要です。Help → Eclipse Marketplace をクリックし、「**EclEmma**」を検索してインストールします：



- インストール完了後、STS を再起動する必要があります。

# カバレッジを表示

- カバレッジを表示するテストは、Run As の代わりに Coverage As を選択することを除けば、普通のテストと同様です：



# カバレッジの結果

- テストが完了したら、コンソールパネルの Coverage タブをクリックして、各パッケージ、クラス、メソッドのコードカバレッジをチェックすることができます：

Problems @ Javadoc Declaration Console Coverage ×	
net.lighthouseplan.spring.junit.controllers (2022年8月19日 下午9:25)	
Element	Cover...
▼ SpringTest	31.7 %
▼ src/test/java	41.6 %
> net.lighthouseplan.spring.junit.integration	0.0 %
> net.lighthouseplan.spring.junit	0.0 %
> net.lighthouseplan.spring.junit.services	0.0 %
▼ net.lighthouseplan.spring.junit.controllers	100.0 %
> LoginControllerTest.java	100.0 %
> RegisterControllerTest.java	100.0 %

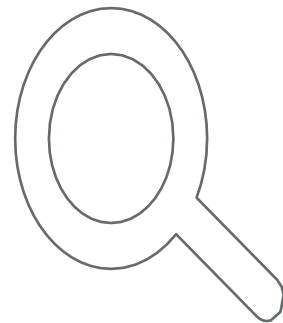
# JUnit の特別なアノテーション

- 偶に、テストクラスの個々のメソッドの中に、**重複な操作**を行うことがあります。例えば、各テストの前にクラスのオブジェクトを作成する必要があります。また、終了後にオブジェクトの `close()` メソッドを呼び出す必要があります。
- JUnit ではメソッドのアノテーションを多数提供：

アノテーション	説明
@BeforeEach	メソッドは、 <b>各</b> テストが実行される <b>前</b> に実行
@AfterEach	メソッドは、 <b>各</b> テストが実行される <b>後</b> に実行
@BeforeAll	メソッドは、 <b>すべての</b> テストが実行される <b>前</b> に実行
@AfterAll	メソッドは、 <b>すべての</b> テストが実行された <b>後</b> に実行



Q&A





# Spring のコントローラをテスト

- MVC モデルのコントローラの各メソッドは、通常、HTTP リクエストを受け付けたときだけ呼び出されます。そのため、**HTTP リクエストの送信処理を模擬**する方法が必要です。
- コントローラをテストするには、まずテストクラスにアノテーションを追加し、Spring が MVC テストの環境を自動設定してくれるようにする必要があります：

```
1 @SpringBootTest
2 @AutoConfigureMockMvc
3 public class RegisterControllerTest {
4     // Test code
5 }
```

# MockMvc オブジェクト

- 次に、**MockMvc** オブジェクトを作ります。このオブジェクトは、私たちが送る HTTP リクエストを受け入れ、私たちが検証しやすい形で応答するサーバを模倣します：

```

1 @SpringBootTest
2 @AutoConfigureMockMvc
3 public class RegisterControllerTest {
4     @Autowired
5     private MockMvc mockMvc;
6 }

```

- 前と同じように、**@Autowired** アノテーションを使用すると、mockMvc オブジェクトを明示的にインスタンス化することなく直接使用できるようになります。

# HTTP リクエストの模擬

- クライアントから送信される HTTP リクエストを模擬するには、**MockMvcRequestBuilders** が提供するメソッドを使用して、リクエストを手動で生成する必要があります：

```
1 RequestBuilder request = MockMvcRequestBuilders
2     .post("/register")
3     .param("username", "Bob")
4     .param("password", "Bob54321");
```

- 2行目の **post()** は、HTTP リクエストが POST メソッドを使うことを示し、GET メソッドの場合は **get()** と記述することができます。
- **param()** メソッドは、リクエストに含まれる各パラメータを設定します。

# リクエスト処理の模擬

- 次に、MockMvc オブジェクトの **perform()** メソッドを使用して、リクエストを処理し、いくつかのアサーションを宣言します：

```
1 mockMvc.perform(request)
2           .andExpect(view().name("login.html"))
3           .andExpect(model().attributeDoesNotExist("error"));
```

# MVC のアサーションメソッド

- アサーションメソッドには様々なものがありますが、一般的には以下のようなものが使われています：
  - `view().name("xxx.html")` : ビューの名前を宣言
  - `model().attribute("error", true)` : ページに渡されたパラメータが特定の値であることを宣言
  - `model().attributeDoesNotExist("error")` : パラメータが存在しないことを宣言
  - `status().is(200)` : HTTP ステータスコードを宣言
  - `redirectedUrl("http://localhost:8080/login")` : リダイレクト URL を宣言
  - `content().string()` など : レスポンス内容を宣言
- これらのメソッドを使えば、もう `assertXxx()` メソッドを呼び出す必要はありません。



# Spring Security の対応

- Spring Security を使用するプロジェクトでは、リクエストを作成する際に、ログインしているユーザーに関連する情報を設定する必要があります：

```

1 UserDetails alice = User.withDefaultPasswordEncoder( )
2     .username( "Alice" )
3     .password( "123456" )
4     .roles( "USER" )
5     .build( );
6
7 RequestBuilder request = MockMvcRequestBuilders
8     .get( "/hello" )
9     .with( csrf( ) )
10    .with( user( alice ) );

```



# 外部依存の解除

- コントローラのテスト中に、コントローラがサービスのメソッドを呼び出し、それが他のクラスのメソッドを呼び出すようなことがあります。単体テストの原則は 1 つのクラスだけをテストするから、実際に行っているのは**結合テスト**になってしまいます。
- 単体テストを行うためには、サービス層のメソッドの振る舞いを模擬する**スタブ**<sup>[Stub]</sup>オブジェクトを使用することができます。

- スタブオブジェクトは、サービスクラスのメソッドが特定のパラメータを受け取ったときに、実際のサービスクラスのコードを呼び出すことなく、いくつかの**固定の戻り値を直接返す**ように定義することができます。
- スタブオブジェクトを定義するには、まずテストクラスで置き換えたいクラスのオブジェクトを宣言し、その宣言の前に **@MockBean** アノテーションを付けます：

```
@MockBean  
private AccountService accountService;
```

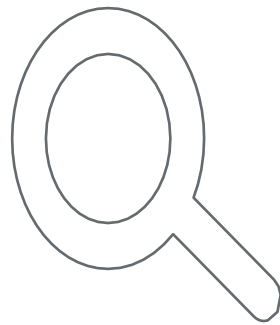
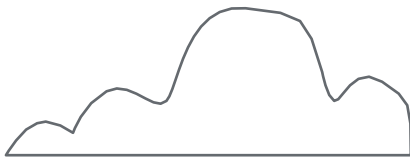
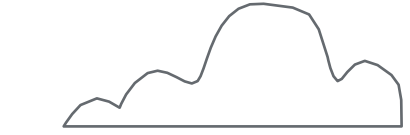
- **when()** メソッドを使用して、個々のメソッドが呼び出された際のスタブオブジェクトの動作を定義します：

```
1 @BeforeEach
2 public void prepareData() {
3     when(accountService.createAccount(any(), any())).thenReturn(true);
4     when(accountService.createAccount(eq("Alice"), any())).thenReturn(false);
5     when(accountService.validateAccount(any(), any())).thenReturn(false);
6     when(accountService.validateAccount("Alice", "ABC12345")).thenReturn(true);
7 }
```

- このコードは、サービスクラスでアカウントを作成したければ、ユーザ名が Alice でない限りに、コントローラに true の結果がもらうことを表現しています。そして、ユーザ名とパスワードがそれぞれ Alice と ABC12345 でない限り、コントローラはいずれかのアカウントを検証しようとしたときに、false が返されます。
- スタブオブジェクトを定義した後に、直接にテストを行うと、元々書いたサービスクラスのコードは呼び出されず、定義された結果が直接返されることがわかります。



# Q&A



# テストメソッドのネーミング

- 単体テストメソッドのネーミングには、様々なものがあります。ここでは、読みやすいネーミング法を一つ紹介します。
- メソッド名は**大文字**か、「**test + 大文字**」で始まります。メソッド名の各部分は、**大文字のキャメルケース**で、アンダースコア「**\_**」で区切ります。
- メソッド名の構成は、**[テストされるメソッド名]\_[テスト状況]\_[期待される結果]**。例えば：

```
testGetHelloPage_DoesNotLogin_Redirected( )
```



# まとめ

Sum Up



## 1.開発テストの概念：

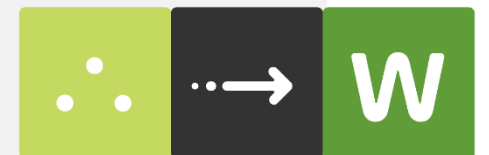
- ① 単体、結合、システムテスト
- ② ホワイトボックス、グレーボックス、ブラックボックステスト
- ③ カバレッジ

## 2.Java・Spring でのテスト方法：

- ① JUnit による単体テスト：assertXxx() メソッド。
- ② MVC のテスト方法：
  - a.@MockMvc、
  - b.@MockBean。

# Thank you!

From Seeds to Woodland — Shape Your Future.



*Shape Your Future*