Unityテストツール ドキュメンテーション

目次

Unityテストツール

インテグレーションテストフレームワーク

インテグレーション（統合／結合）テストは、シーン中でアセットを直接検証するプロセスを自動化します。このフレームワークは、エディター上から直接既存のコンテンツに対して使用することを想定して設計されており、各アセットの挙動と、アセット間の実装の両方を検証するテストを構築できます。

インテグレーションテストフレームワークの使い方

アサーションコンポーネント

アサーションコンポーネント（アサーションコンポーネント）はゲームオブジェクトの不変条件を設定する際に使用します。このコンポーネントのセットアップはエディター UI で完結するため、コードを書く必要が一切ありません。拡張やカスタマイズも容易で、ニーズに合わせて構成できます。

アサーションコンポーネントの使い方

ユニットテストランナー

NUnit Framework がエディターに統合されているため、Unity 上からユニットテストを実行できます。つまりゲームオブジェクトをインスタンス化して、それらに対して処理を行なえます。これは Unity 外では困難な処理です。またテストを実行して結果を報告するテスト実行機能も統合してあります。

ユニットテストランナーの使い方

NSubstituteライブラリ

NSubstitute は Unity Test Framework に同梱されているライブラリです。詳細についてはライブラリのドキュメントを参照してください：http://nsubstitute.github.io/help.html

サンプル

フレームワークにはサンプルも同梱されています。

Q&A

- 互換性のあるUnityバージョンは何ですか?

  このフレームワークはUnity4.xで動作します。

- C#のみでユニットテストを書くことは可能ですか?

  私達はC#で書くことに注力していますが、UnityScriptやBooでも問題なく書けるようにしています。

- いくつかのサブフォルダにこのツールを移動させることは可能ですか?

  はい、可能です。しかし「Icons.cs」ファイル内のリソースパスを更新する必要があります。

- なぜこのツールを「Standard Assets」フォルダに移動させた時、動作しなくなるのですか?

  現在のフォルダ構成では「Standard Assets」フォルダで動作するように設計されていません。ですが、少しの修正を行うだけで移動することが可能です。Editorのコードを「Standard Assets/Editor」へ移動し残りを「Standard Assets」フォルダ配下に移動してください。

- 既知の問題と制限はなにかありますか?
  - インテグレーションテストフレームワーク は インテグレーションテストランナー ウィンドウが表示されている間のみ動作します。
  - フレームワーク同梱の NSubstitute はスレッドセーフではありません。

フレームワーク同梱のサンプル

サンプルシーン

サンプルは Examples フォルダに格納されています。

- IntegrationTestsExample.unity シーン

  このシーンは Game Tests Runner の機能を紹介するものです。使用するには Game Tests Runner を開く必要があります（メニュー内から[Tests] > [Game Test Runner]、または CTRL + ALT + SHIFT + T）。サンプルは合計 6 個用意されています。またシーンには、全テストで共用されるオブジェクト（プレハブ）が 2 つ存在します（CubeTriggerFailure、CubeTriggerSuccess）。これらのプレハブはシンプルな構造をしており、それぞれコライダと Testing.Fail() および Testing.Succeed() を呼び出すスクリプトを持ちます。これらテストはフレームワークの技術的な側面に焦点を当てたものであり、大掛かりなテストではありません。 各テストの目的は次のとおりです:

  - Test1 - Success（成功）

    球体が、テストの成功をトリガーするキューブに向けて落ちていきます。テストの結果は成功となります。

  - Test2 - Timeout（タイムアウト）

    テストに設定されているタイムアウト値が小さすぎる（0.1 秒）場合は、球体がキューブに落ちる前にタイムアウトします。

  - Test3 - FailurePlayerReceivesDamageWhenSpiderExplodes（失敗）

    球体が、テストの失敗をトリガーするキューブに向けて落ちていきます。テストの結果は失敗となります。

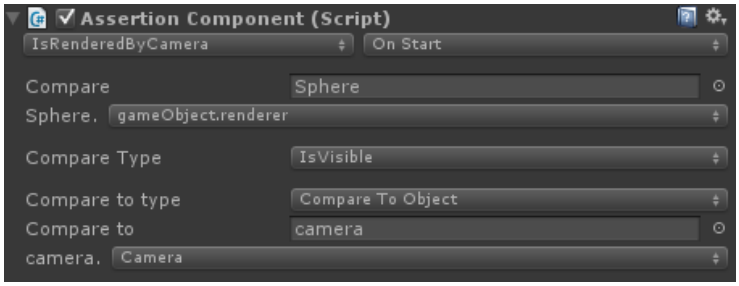
  - Test4 - Ignored（無視）

    [ignore]（無視する）チェックボックスをオンにした状態でテストします。全テストを実行しても、このテストは無視されます。

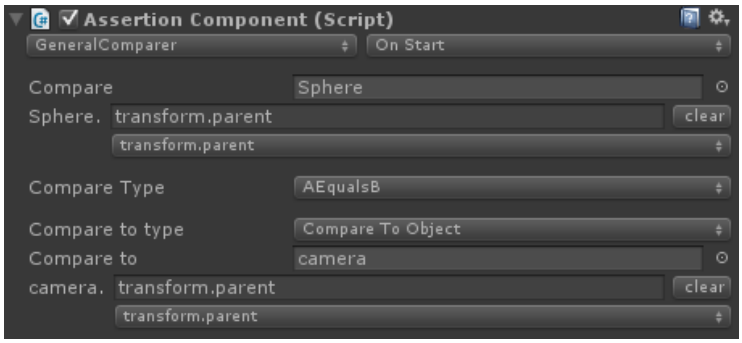
  - Test with Assertions（アサートあり）

    [Succeed after all assertions are executed]（全アサートが実行されたら成功）チェックボックスをオンにした状態でテストします。このテストでは球体に 2 つのアサートがセットされています。

    - 最初のアサートは OnStart コールバック時に設定されており、sphere（球体）のレンダラがカメラによってレンダリングされているかどうかをチェックします。

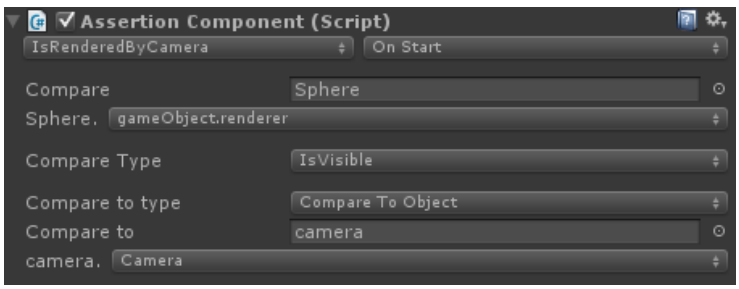
      

    - 2番目のアサートは sphere（球体）と camera（カメラ）の親オブジェクトが同一かどうかをチェックします。

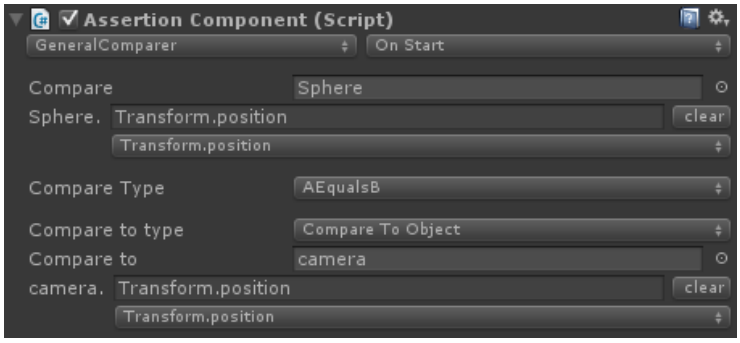
      

  - Test with Assertion Fails（アサートあり、失敗）

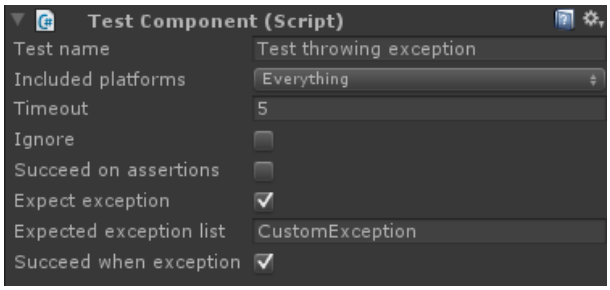
    - 最初のアサートは OnStart コールバック時に設定されており、レンダラが camera（カメラ）によってレンダリングされているかどうかをチェックします。

    - 2番目のアサートは sphere（球体）の transform.position が camera（カメラ）の transform.position と同一であるかどうかをチェックします。この条件は真ではないため、テストは失敗します。



- Test throwing exception（例外をスローするテスト）
  - 例外がスローされた時に成功とみなされるテストです。ここでの例外は CustomException またはその派生タイプである必要があります。



- AssertionExampleScene.unity シーン

  アサーションコンポーネントを使用したデバッグ作業の簡単なサンプルです。このシーンでは、球体が平面に向けて落下していき、その後平面から転がり落ちます。アサートは球体に 2 つ設定されています。 最初のアサートは、球体が OnUpdate の呼び出し時に必ずカメラでレンダリングされることを確認するものです。2 番目のアサートは球体の位置ベクトルの Y 値が常に平面の位置ベクトルの Y 値よりも大きいことを確認します。これは、球体が平面よりも下に行かない、ということです。

- AngryBotsTests/TestScene.unity シーン

  このサンプルは Unity のサンプルプロジェクトである Angry Bots のアセットを使用しています。使用されている Angry Bots のアセットは 2 個のプレハブ、PlayerPrefab、EnemySpider です。最初のプレハブは標準的なプレイヤーコントローラー、もう一方のプレハブは敵スパイダーで、こちらはプレイヤーが近づくと覚醒するようになっています。このテストシーンの目的は、次の 3 点をテストすることです:

  - プレイヤーとの距離が一定以下になるとスパイダーが覚醒してプレイヤーに近づいていく
  - プレイヤーとの距離が離れている場合、スパイダーは覚醒しない
  - スパイダーは爆発時に、プレイヤーにダメージを与える

  このシーンでは手順を自動化するために Game Tests を使用しています。このシーンに対するテストは以下のとおりです:

  - Test_PlayerReceivesDamageWhenSpiderExplodes

    スパイダーがプレイヤーを攻撃して爆発します。プレイヤーがダメージを受け、healthが特定の値より低いかを検証します。

  - Test_SpiderSleepsWhenPlayerNotInRange

    プレイヤーをスパイダーの視認範囲外に位置させ、スパイダーが覚醒して攻撃してこないことを確認します。このテストでは、スパイダーとプレイヤーの間に CubeCollisionFailure を設定し、スパイダーがそれに接触するとテスト失敗と判定する方法で検証を行ないます。また一定時間経過でトリガーされるアサートを含むゲームオブジェクトもあり、こちらはスパイダーの攻撃行動コントローラーが無効であること（スパイダーが休眠モードであること）をチェックします。このアサートがチェックされるとテストは成功となります。

  - Test_SpiderWakesWhenPlayerInRange

    プレイヤーがスパイダーの視認範囲内に位置されると、スパイダーは覚醒してプレイヤーに近づいてきます。スパイダーとプレイヤーの間には、接触時に Testing.Succeed() を呼び出す CubeCollisionSuccess が存在します。スパイダーがプレイヤーに近づいていくと途中でトリガーに接触し、テスト成功となります。

NUnitサンプル

SampleTests.cs には NUnit の基本的な使い方を示すサンプルが含まれています。

- public void ExceptionTest() - 例外がスローされるため失敗します。
- public void IgnoredTest() - ignore（無視）属性の用途を示します。
- public void SlowTest() - 実行に 1 秒かかるテストで、[Notify when test is slow]（テストが遅い時に通知）オプションを設定できます。public void FailingTest() - アサートの使い方、および Assert.Fail() を呼び出して失敗するケースを実証します。
- public void PassingTest() - アサートの使い方、および Assert.Pass() を呼び出して成功するケースを実証します。
- ParameterizedTest、RandomTest、RangeTest が nUnit の機能を示します。

NSubstituteDemo.cs に含まれるクラスは、シンプルな条件下で **NSubstitute** の使い方を実演します。

IGameEvent：抽象ゲームイベントを表すインターフェース。

IGameEventListener：抽象ゲームコンシューマーを表すインターフェース。

GameEventSink：システムからイベントを取得して登録リスナーに渡すオブジェクト。

RegisteredEventListenersGetEventsテスト は、ReceiveEvent メソッドが IGameEventListener を表す登録リスナーで呼び出されたことをチェックします。

IGameEventListener そのものではなく代替物が使用されます。

```
public void RegisteredEventListenersGetEvents()
{
    GameEventSink sink = new GameEventSink();

    //a proxy for IGameEventListener is created.
    IGameEventListener listener = Substitute.For<IGameEventListener>();

    sink.RegisterListener(listener);
    sink.ReceiveEvent(Substitute.For<IGameEvent>());

    //In this line a check that the method was called (with any arguments).
    listener.Received().ReceiveEvent(Arg.Any<IGameEvent>());
}
```

NSubstitute の機能については NSubstitute のドキュメントを参照してください。

UnityScriptとBooでユニットテスト(id:example-2-1)

UnityScriptの例

```
class UnityScriptUnitTests
{
   @NUnit.Framework.Test()
   function UnityScriptTest () {
      NUnit.Framework.Assert.Pass();
   }
}
```

Booの例

```
namespace UnityTest
import NUnit.Framework
class BooUnitTests:
   [Test]
   def BooTest ():
      Assert.Pass();
```

インテグレーションテストフレームワークの使い方

インテグレーションテストランナーの開き方

インテグレーションテストランナーはメニューバーから [Unity Test Tools] > [Integration Tests] >[Integration Tests Runner]または Shift + Ctrl + Alt + T で開きます。

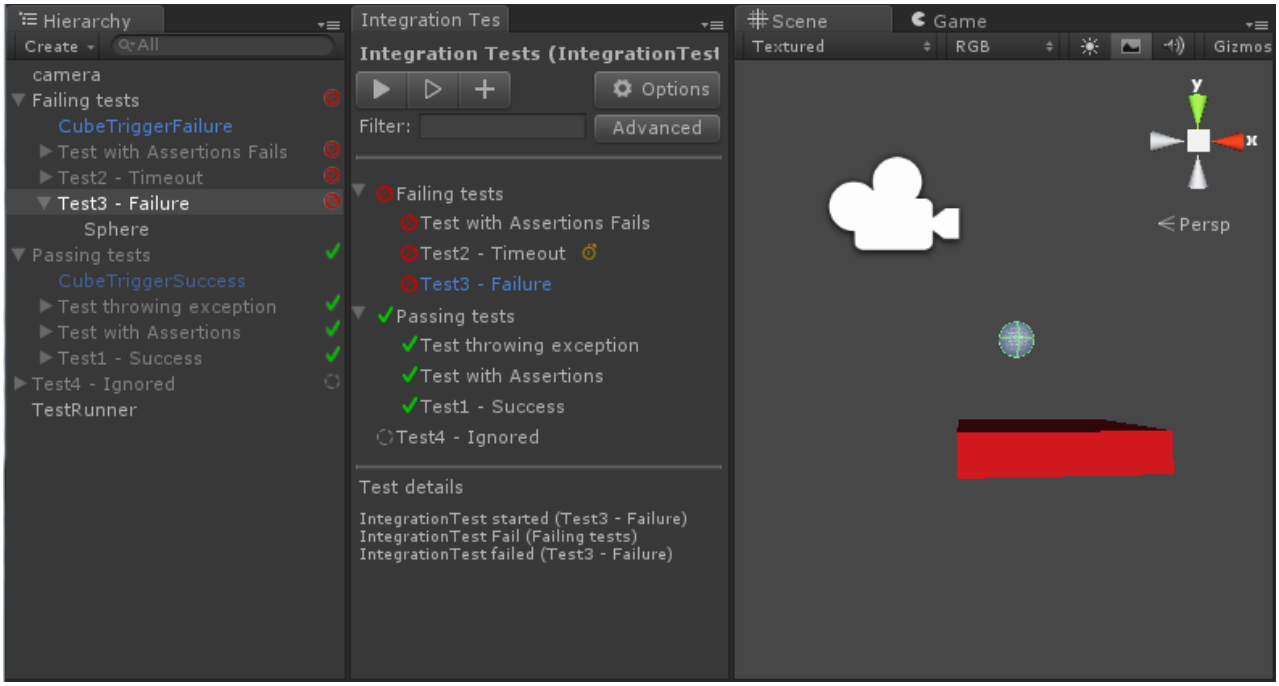
ショートカットにはこの他にも、Ctrl + Alt + T：全テストを実行、Ctrl + T：選択したテスト（複数可）を実行、などがあります。

インテグレーションテストランナーの構造

インテグレーションテストランナーは、独立したシーンでの利用を想定して設計されています。実施するすべてのテストを集めた、テストスイートのようなものを想像して頂ければ分かりやすいでしょう。テストは 1 つのシーンに複数個配置できます。成果物となるシーンにはテストを配置せず、それらのシーンから別途テスト用シーンを作成してください。

テストオブジェクトとは、シーン内にあり、TestComponent がアタッチされたゲームオブジェクトのことです。テストオブジェクトの階層下にあるオブジェクトは、すべて当該テストに属するものとして扱われます。他方、テストオブジェクトの階層下にないオブジェクトは、シーン中の全テストの共通オブジェクトとなります（通常は床や壁などの環境要素）。なお、テストオブジェクトを手動で作成する必要はありません。すべての手順は Test Runner で行われます。テストは階層構造を持つことが出来ます。テストオブジェクトに他のテストオブジェクトを子要素とすることが可能です。配下にあるテストオブジェクトはグループとなり個別のテストとして実行はされません。

テストは同時に 1 つしか有効化されません。テストを 1 つ選択すると他のテストは無効化されるため、一度に作業できるテストは 1 つだけです。



このシーンはフレームワークに同梱されているサンプルシーンです。階層ビューを見ると、このシーンには 7 個のテストオブジェクトと2つのグループがあることを確認できます。1つ目のグループは全て失敗するテストです。2つ目が全て成功するテストで構成されています。残りのグループでないテストは無効で実行されないテストです。各オブジェクトには、テストの前回実行結果を示すアイコンが表示されています。サンプルでは「Test3 - Failure」が選択されているため、他のテストは無効化されています。このテストには Sphere（球体）ゲームオブジェクトのみが存在しています。しかしシーンを見てみると、他にも赤と緑の 2 個のキューブがあることに気づきます。これらのキューブはテストオブジェクトの下に配置されていないため、シーン内の全テストで共有されます。これらキューブは、階層ウィンドウ上では CubeTriggerSuccess および CubeTriggerFailure として表示されます。また、この画面では TestRunner オブジェクトも確認できます。このオブジェクトは実行開始後にテストを進行させる役割を果たします。なお、このオブジェクトはシーンに最初のテストを追加すると自動的に追加されます。

テストを実行すると、Test Runner が以下の手順を実行します:

1. プレイモードを有効化する
2. 最初のテストを有効化（アクティブ化）する
3. テスト完了まで待機（またはタイムアウトするまで待機）する
4. 現在アクティブなテストを無効化する
5. キューに他のテストがある場合は、次のテストを有効化して手順 3 に戻る
6. 結果をレポートし、テストランを終了する

テストフローの制御方法（テストの開始／終了方法）

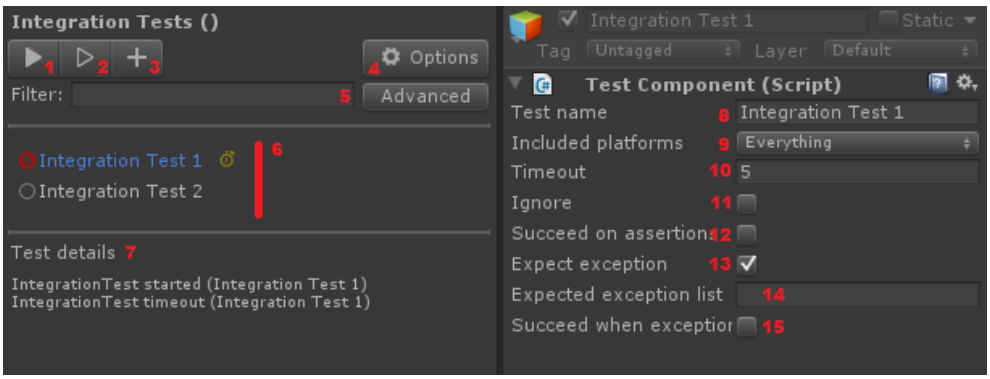
テストはテストオブジェクトが有効化されると開始されます。なお、テストの終了には様々なタイプがあります:

- 関数 Testing.Pass() が呼び出される場合。テストは正常終了します。
- 関数 Testing.Fail() が呼び出される場合。テストは失敗します。
- 実行がタイムアウトする場合。指定時間内に上記のいずれの関数も呼び出されない場合に発生します（タイムアウトの値はテストごとに指定可能）。
- ハンドルされない例外がスローされる場合。
- 予期された例外がスローされた場合（[Expect exception]（例外を予期）がオンになっている場合のみ）
- テストの下に配置された全オブジェクトの Assertion Component は、最低 1 回はチェックされます（[Succeed after all assertions are executed]（全アサートが実行されたら成功）オプションの設定オンが必須）。

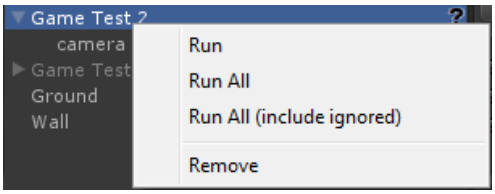
テストの下に配置された全オブジェクトの Assertion Component は、最低 1 回はチェックされます（[Succeed after all assertions are executed]（全アサートが実行されたら成功）オプションの設定オンが必須）。

インテグレーションテストランナー

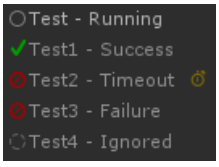
インテグレーションテストランナー ウィンドウの機能:



1. シーン内の全テストを実行（無視されるテストは除く）
2. 選択中のテストを実行（複数可能）
3. 新規テストを追加： シーン内にテストオブジェクトを新規作成します。
4. オプション： インテグレーションテストのオプション
    a. 選択中のテストの下にゲームオブジェクトを追加： オンにすると、シーンにオブジェクトを新規追加する際に自動的にテストオブジェクトの下に配置されます（ルート階層に配置されない）。
    b. 実行時に UI をブロック： オンにすると、テスト実行時にダイアログが表示されます。
5. テストフィルタ： 指定文字列を含まないテストを除外して表示します。
    a. 成功を表示： 成功したテストを表示します。
    b. 失敗を表示： 失敗したテストを表示します。
    c. 無視を表示： 無視されたテストを表示します。
    d. 未実行を表示： 実行されなかったテストを表示します。
6. テストのリスト： シーン内にあるテストの一覧を表示します。
7. テストのログと例外メッセージ
8. テスト名： テストの名前です。
9. 含まれるプラットフォーム： テストに含まれるべきプラットフォームを示します。
10. タイムアウト：タイムアウトするまでの秒数を表示します。
11. 無視するテスト：全テストの実行時に無視するテストです。
12. 全アサートが実行されたら成功： ゲームオブジェクトに含まれる全アサートが 1 回以上チェックされて初めてテスト終了とする場合は、このオプションをオンにします。
13. 例外を予期： オンにすると、例外がスローされても対象テストが失敗しなくなります。
14. 予期される例外リスト： スローされても失敗と見なさない例外のリストです。複数個を指定する際は、間をカンマ「,」で区切ります。リストにある例外から派生した例外タイプも、同じく予期されるものとして扱われます。このリストが空の場合、すべての例外タイプが予期されるものとして扱われます。
15. 例外がスローされた時に成功： このテストは一部の例外を除いて、例外がスローされた時に成功します。
    1. 選択中のテスト
    2. 結果アイコン
    3. Test Runner オブジェクト
    4. 共通オブジェクト： テストノードの下に配置されていないオブジェクトは、全テストでアクティブになります。



テストのコンテキストメニュー（テストを右クリック）



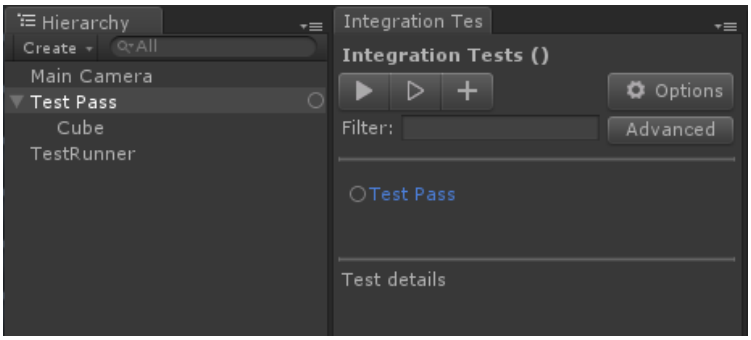
アイコン

シンプルなテストを作成する

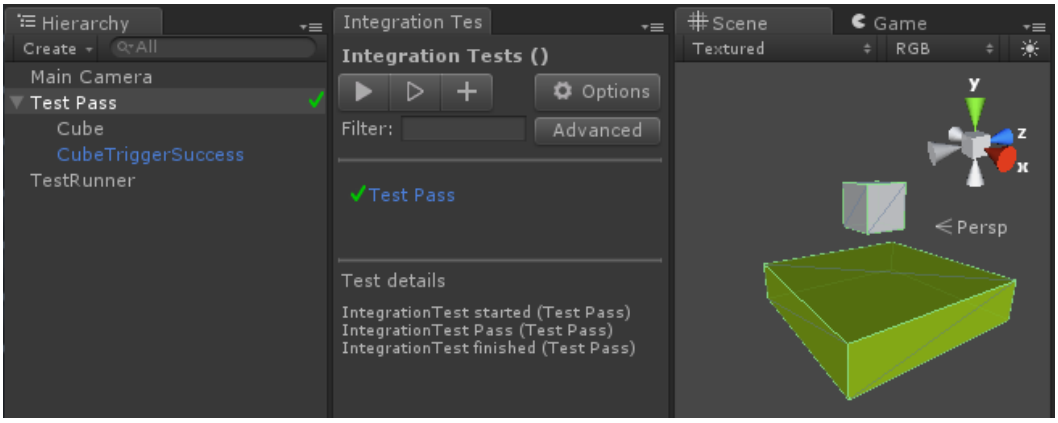
次に 2 個のシンプルなテストを記述する手順を解説します:

1. テストを含むシーンを新規作成します。
2. インテグレーションテストランナーウィンドウを開きます（メニューバーから [Unity Test Tools] > [Integration Tests Runner] 、または Shift + Ctrl + Alt + T)。
3. プラスボタンをクリックしてテストを新規作成し、名前を「Test Pass」に変更します。なお、TestRunner オブジェクトは自動的に追加されます。
4. 作成したテストを選択します。
5. シーンに Cube（キューブ）を追加します。[Add new GameObjects under selected test]（選択中のテストの下にゲームオブジェクトを追加）オプションがオンの場合、キューブが自動的にテストノードの下に配置されます。このオプションがオフの場合は、手動でテストノードの下に移動してください。

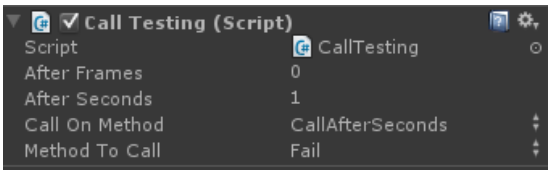
   ここまでの手順が完了すると、階層は次のような状態になります。



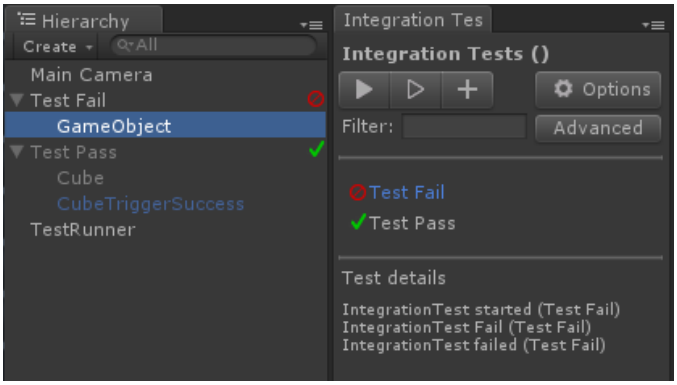
6. キューブに Rigidbody コンポーネントを追加します。ここでテストを実行すると、キューブが落下することを確認できます。実行するには、テストを右クリックして [Run]（実行）をクリックします（テストを選択した状態で CTRL + T を押しても同様に実行できます）。テストを実行すると、Testing.Pass() が呼び出されないため 5 秒後（デフォルト設定）にタイムアウトします。
7. 最初のキューブ（キューブ 1）の下にもう 1 つキューブを追加し（キューブ 2）、衝突時に Testing.Pass() を呼び出すよう設定します。これにはフレームワーク同梱のテストアセットを使用できます。Assets\UnityTestFramework\IntegrationTestsFramework 以下にある CubeTriggerSuccess プレハブを見つけ、シーン内でキューブ 1 の真下に配置します。これでキューブ 1 が落下すると、配置したプレハブと衝突するようになります。配置したプレハブをよく見てみると、スクリプトがアタッチされている事が分かります。このスクリプトが OnCollisionEnter 関数内で Testing.Pass() を呼び出す役割を果たしています。
8. テストを再度実行します。この時点で、テストは成功するようになっています。



9. 次は別のテストを追加し、続いて空っぽのゲームオブジェクトを作成します。Testing Assets にある CallTesting スクリプトをアタッチして、1 秒後に失敗するようにします。



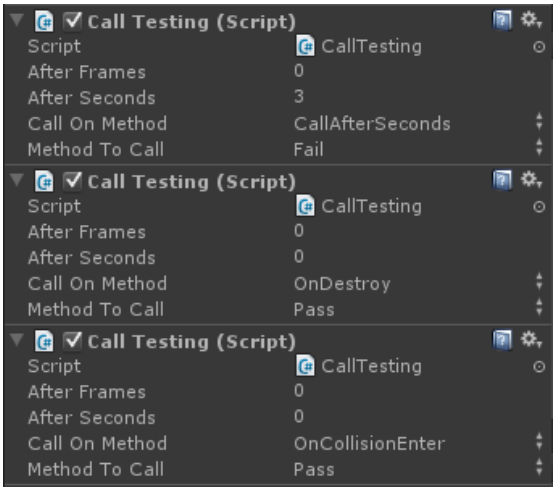
10. 今度は両方のテストを実行します。次のような結果が返されます。



アセットをテストする

フレームワークには次に示すテスト用アセットが含まれています:

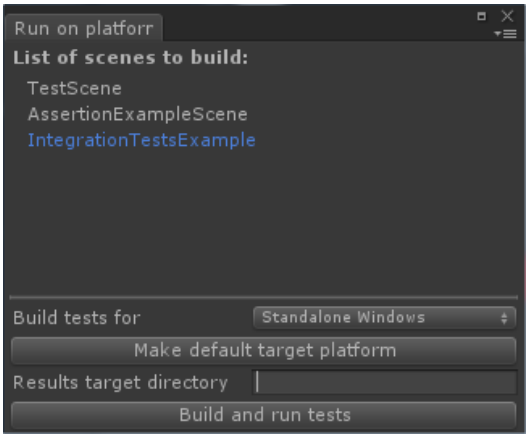
- CallTesting.cs スクリプト： 選択したメソッドから、または一定時間／フレーム経過後に Testing.Pass() または Testing.Fail() を自動的に呼び出すことができます。



- この構成は、オブジェクトがコリジョンに入った時（OnCollisionEnter）または破壊された時（OnDestroy）に成功します。それ以外の場合には、3秒後にテストが失敗します。
- CubeTriggerSuccess、CubeTriggerFailure：OnTriggerEvent 時に成功 / 失敗するキューブのプレハブです。

プラットフォーム上でテストを実行する

Platform Runner ([Unity Test Tools] > [Integration Tests] > [Platform Runner] > [Run on platform], Ctrl+Shift+R)を使用して目的のプラットフォームで「Build & Run」時にテストシーンを実行します。



1. List of scenes to build - ビルドするシーンの複数選択可能なリストです。テストの行う必要のあるシーンの順を選択し実行します。
2. Build tests for - ビルド対象のプラットフォームのリスト。このリストはBuildTargetと一致しています。
3. Make default target platform - 現在選択中のデフォルトのプラットフォームでビルドします。
4. Result target directory - テスト結果を保存するディレクトリ。このパスは必ず有効で存在することが条件です。パスはネットワーク上の場所も指定することが可能です。例: \\network-drive\results
5. Build and run tests - ビルドを行いテストを実行します。

さらに、[Unity Test Tools] > [Integration Tests] > [Platform Runner] > [Run current scene] (Ctrl+Alt+Shift+R) で現在開いているシーンをデフォルトのプラットフォームでテストを実行することが出来ます。

結果をレポートする

エディター内でテストを実行した場合、結果はTest Runnerに表示されます。バッチモードやプレイヤー上では、結果を示す XML ファイルが作成されます。このファイルはパラメータを指定していない限り、プロジェクトのルートフォルダに格納されます。この機能は現在、エディター、エディターのバッチモード、スタンドアロンでの実行に対応しています。結果はnUnitフォーマットで取得できます。詳しくはここをご覧ください: http://www.nunit.org/docs/2.6.2/files/Results.xsd. ファイルシステムをサポートしていないプラットフォームは出力された結果を解析することが可能です。

CUIからの実行（バッチモード）

インテグレーションテストはコマンドラインからも実行可能です。コマンドラインから実行するには、Unity をバッチモードで実行し、起動時に UnityTest.UnitTestView.RunAllTestsBatch メソッドを実行します。

パラメーター名	説明
testscenes	実行するシーンのカンマ区切りのリスト
targetPlatform	実行するプラットフォーム。値はBuildTargetに合わせるようにしてください。もし値が存在しない場合はEditor上で実行されます。
resultsFileDirectory	結果を保存するフォルダのパス。指定しない場合はプロジェクトのルートフォルダに保存します。

例:

```
>Unity.exe -batchmode -projectPath PATH_TO_YOUR_PROJECT -executeMethod UnityTest.Batch.RunIntegrationTests -
testscenes=TestScene1,TestScene2 -targetPlatform=StandaloneWindows -resultsFileDirectory=C:\temp\
```

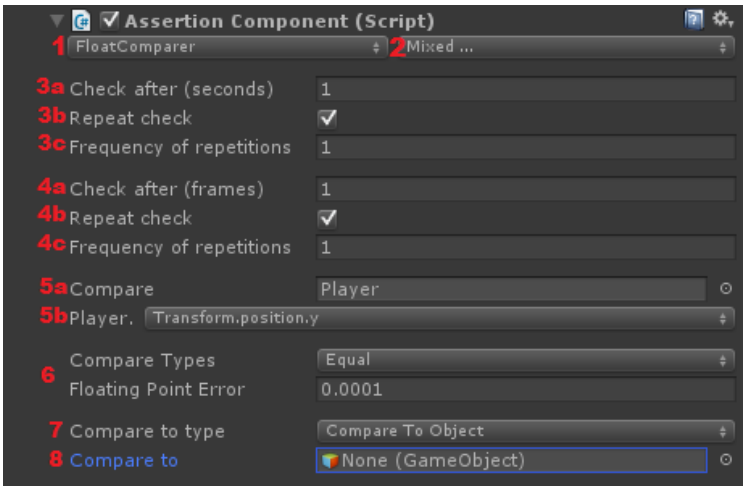
この例では、TestScene1とTestScene2をWindowsスタンドアロンプレイヤーでテストを実行し、結果を `C:\temp\` へ保存します。

アサーションコンポーネントの使い方

アサーションコンポーネントの概要

アサーションコンポーネント は、ゲームオブジェクトが指定した状態になった時にアサートする機能を持ちます。これはコードを書く必要のないビジュアルツールなのです。設計段階から拡張性と順応性を考慮しているため、プロジェクトのコンテンツやニーズに合わせて利用できます。

本コンポーネントの仕組みは極めてシンプルです。まず不変条件として「常に真であるべき条件」を設定し、さらにその条件をチェックするタイミングを指定します（サンプルでは Update メソッド毎）。プロジェクト実行中に設定したアサートが失敗すれば例外が出力され、これによって開発中のアプリケーションが望ましくない状態にあることを把握できるため、問題点を調査できます。通常は [Console] ウィンドウで [Error pause]（エラー時に一時停止）オプションを有効にしておき、エラー発生時に一時停止するようにしておきます。



1. Comparer（比較オプション）の選択： 2 つの値の比較方法を指定します。この設定がアサートの結果を決めます。
2. チェック頻度： アサートをチェックするタイミングを指定します。複数選択可能なオプションです。
3. [After period of time]（指定時間後）の頻度（2. を参照）。オプションが有効でない場合は表示されない。
    a. 初回チェックを実行するまでの秒数。
    b. チェックを繰り返し行なうかどうか。
    c. チェックを繰り返す頻度。
4. [Update]（更新）の頻度（2. を参照）。オプションが有効でない場合は表示されない。
    a. 最初のチェックを実行するまでのフレーム数。
    b. チェックを繰り返し行なうかどうか。
    c. チェックを繰り返す頻度。
5. 比較に使用される最初のゲームオブジェクト。デフォルトでは、コンポーネントがアタッチされているゲームオブジェクトになる。
    a. ゲームオブジェクト参照フィールド

    b. パスから変数をチェック
6. 選択した Comparer のカスタムフィールド（FloatComparer）。この例では比較の種類と精度を定義。
7. （5 で指定された）ゲームオブジェクトの比較対象。他のゲームオブジェクト、静的な数値や null 値との比較も可能。
8. 比較対象とする他のオブジェクト。

アサーションコンポーネントのセットアップ

アサーションコンポーネントのセットアップは簡単に行なえます。シンプルなアサートであれば数ステップで設定可能です。

1. アサートのチェックに使用する Comparer（1）を指定します。Comparer は通常、許容する型を定義し、比較するプロパティの選択時に便利なフィルタとして機能します。
2. アサートをチェックするタイミング（2）を選択します。MonoBehaviour のコールバックメソッドはほぼすべて選択可能（OnStart、OnUpdate など）です。また、チェック実施までの時間を設定することもできます（[After period of time（指定時間後）]）。OnUpdate および AfterPeriodOfTime ではチェック頻度を定義する追加パラメータを指定できます（3、4）。
3. 比較対象とするプロパティ値のパス（5b）を選択します。ここには Comparer で選択した型の値だけがフィルタされて表示されます。たとえば Float Comparer は浮動小数点型の値にのみ対応するため、float 型のプロパティおよびフィールドだけが表示されます。
4. Comparer では、フィールドを明瞭化して挙動をカスタマイズできます。たとえば Float Comparer では比較演算の種類（Equal、Greater、Less）と、浮動小数点演算の精度（6）を選択できます。
5. 次に、値を比較する対象を選択します。デフォルトでは別のゲームオブジェクトのプロパティと比較できます。また、（Comparer が対応している場合は）静的な値、そして null と比較することもできます。
6. これまでの設定内容に応じて比較用の値を選択します。

アサーションコンポーネントの機能

- Comparer（比較オプション）

  Comparer（比較オプション）はアサートの動作を定義します。Comparer は ObjectComparerBase の派生クラスである必要があり、Compare メソッドを実装する必要があります。

  Comparer 実装サンプル:

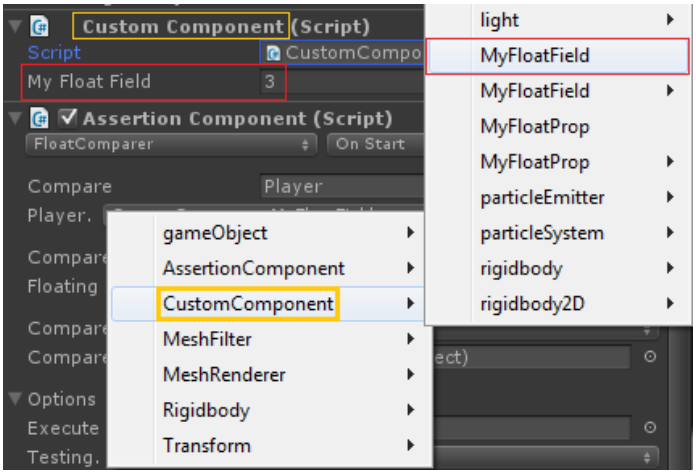
```
public class FloatComparer : ObjectComparerBase<float> 1
{
    public enum CompareTypes
    {
        Equal,
        NotEqual,
        Greater,
        Less
    }

    public CompareTypes compareTypes;
    public double floatingPointError = 0.0001f;                    2

    protected override bool Compare (float a, float b)
    {
        switch (compareTypes)
        {
            case CompareTypes.Equal:
                return Math.Abs (a - b) < floatingPointError;
            case CompareTypes.NotEqual:
                return Math.Abs (a - b) > floatingPointError;
            case CompareTypes.Greater:
                return a > b;
            case CompareTypes.Less:
                return a < b;
        }
        throw new Exception();                                     3
    }
    public override int GetDepthOfSearch() 4
    {
        return 3;
    }
}
```

  1. Comparer は ObjectComparerBase クラスを継承する必要があります。
  2. Public かつシリアライズ化可能なフィールドは Comparer に対して明瞭化されます。これらフィールドを使うと Comparer をカスタマイズできます。
  3. Compare メソッドはアサート実行時に呼び出されます。また抽象メソッドで、Comparer で定義された型の値 2 個を引数として取ります。型が定義されていない（Comparer がジェネリックでない ObjectComparerBase を継承）場合、引数は System.Object から得られます。
  4. またメソッドをオーバーライドすることで、さらなるカスタマイズが可能です。GetDepthOfSearch メソッドは、プロパティ検索アルゴリズムのデフォルト深度をオーバーライドします。

- ゲームオブジェクトのプロパティへのパスを選択する

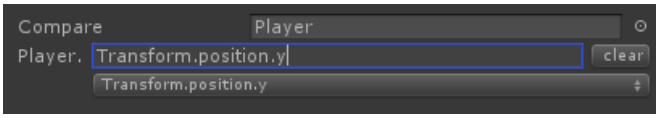
  プロパティへのパス選択時には、指定 Comparer が対応する型のフィールドがリスト表示されます。リストには対象ゲームオブジェクト自体のプロ

パティと、アタッチされている全コンポーネントのプロパティ（カスタムスクリプト含む）が表示されます。



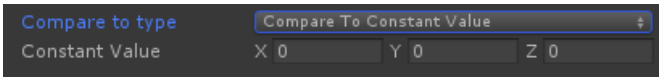
- 手動でのパス選択

  Comparer に対応する型が指定されていない場合には、選択可能な値が存在しないため何も表示されません。そのような場合にはプロパティへのパスを手動で指定する必要があります。エディタでは指定パスが正しくないと思われる場合にヒントを表示します。ヒントリストからは提案された値を選択できます。ヒント：パスの入力中に矢印キーを押すとパスのヒントがポップアップ表示されます。



- 定数値と比較する

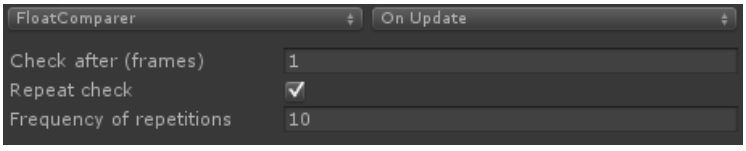
  最初の値は、コンポーネントで指定した静的な値と比較できます。比較する場合は [Compare to type]（比較する型）フィールドで [Compare To Constant Value]（定数値と比較）を選択します。Comparer が対応する型が Unity がデフォルトでシリアライズ可能な場合は、値を入力するためのエリアが表示されます。



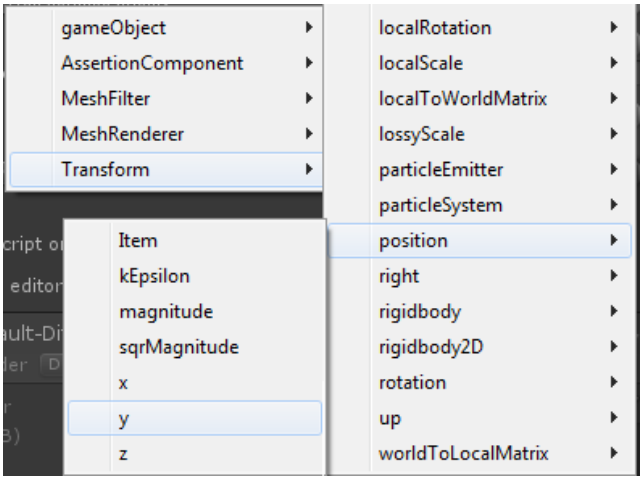
始め方

以下では、シンプルな Assertion Component をセットアップするための手順を紹介していきます。

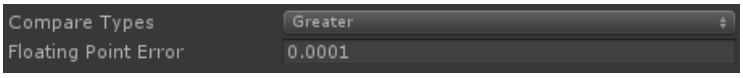
1. シーン中にオブジェクトを新規作成します（ここでは Sphere（球体）を使用）。次に、そのオブジェクトに Rigidbody コンポーネントを追加します。追加し終えたらオブジェクトを選択し、インスペクタに移動します。
2. Assertion Component を追加します。
3. Float Comparer を選択します。
4. アサートを検証するタイミングとして OnUpdate を選択します。これは複数選択コントロールなのでデフォルト設定のOnStartのチェックを外すことを覚えておいてください。
5. アサートを毎フレーム検証する必要がない場合は [Frequency of repetitions]（繰り返し頻度）を 10 に設定します。



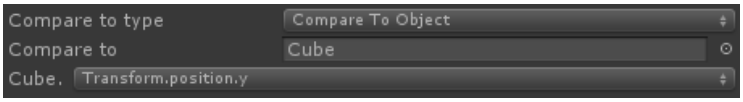
6. Sphere.Transform.position.y の値を選択します。



7. ここでは Sphere.Transform.position.y の値が常に比較対象の値よりも大きいことを想定しているため、Compare の種類にGreater（より大きい）を選択します。



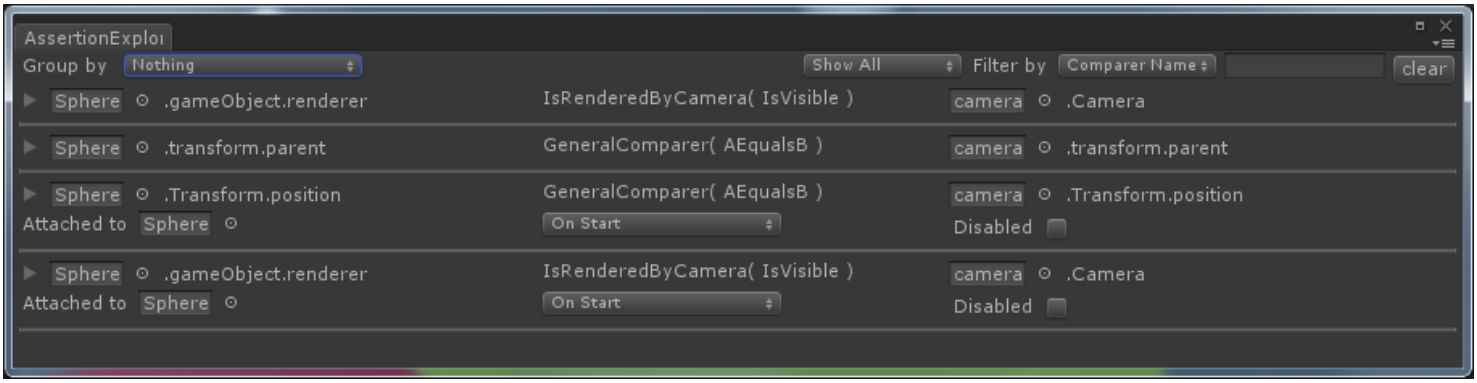
8. さらにゲームオブジェクトを追加し（ここでは Cube（キューブ）を使用）、先に作成したオブジェクトよりも下に配置します。
9. キューブのゲームオブジェクトを Assertion Component の [Compare to]（次と比較）フィールドにドラッグします。
10. パス Cube.Transform.position.y を選択します。



11. シーンを実行します。
12. 球体が落ちはじめ、キューブよりも下に落ちた瞬間にエディタが一時停止します。この一時停止は、アサートのチェックが失敗した（条件 Sphere.Transform.position.y > Cube.Transform.position.y が真でなくなった）ことで発生します。もしエディタが一時停止しない場合、コンソールで「Error Pause」をチェックしているか確認して下さい。

アサーションエクスプローラー

アサーションエクスプローラー は Unit Test Tools メニューからアクセスできます。ここには現在のシーンにあるオブジェクトのアサートがすべて表示されます。フィールドは基本的に読み取り専用で、コンポーネントの有効化／無効化のみがエクスプローラから設定可能です。またリストはグループ化や基本的なフィルタ機能を備えています。



コードを取り除く

アサーションは開発ビルドでなければゲームオブジェクトから取り除かれます。完全にコードの依存関係を削除したい場合はカスタムビルドを構築する必要があります。以下のコードはコードを完全に取り除く例です。

```
using UnityEditor;
using UnityEngine;

public class StripCodeAndBuild
{
    private static string assetsPath = "Assets/UnityTestTools";
    private static string buildTempPath = "Assets/Editor/UnityTestTools";

    [MenuItem ("Unity Test Tools/StripCodeAndBuild")]
    public static void StripCodeAndBuildStandalone ()
    {
        AssetDatabase.CreateFolder ("Assets", "Editor");
        var result = AssetDatabase.MoveAsset (assetsPath, buildTempPath);

        if (string.IsNullOrEmpty (result))
        {
            BuildPipeline.BuildPlayer (new[] {EditorApplication.currentScene}, @"C:\temp\release.exe",
BuildTarget.StandaloneWindows, BuildOptions.None);
            result = AssetDatabase.MoveAsset (buildTempPath, assetsPath);
            if (string.IsNullOrEmpty (result))
                AssetDatabase.Refresh ();
            else
                Debug.LogWarning (result);
        }
        else
        {
            Debug.LogWarning (result);
        }
    }
}
```

備考

- 既知の問題： クラス名が小文字で始まる MonoBehaviour がアタッチされている場合、このコンポーネントは動作しません。

ユニットテストランナーの使い方

ユニットテストランナーの開き方

Unit Test Runner はメニューバーから [Unity Test Tools] > [Unit Test Runner] 、または Shift + Ctrl + Alt + U で開きます。

NUnit を使う

Unity Test Framework パッケージのインポートが完了すると、プロジェクトに NUnit ライブラリ（バージョン 2.6.2）が含まれるようになります。

初めて NUnit をお使いになる場合は、NUnit の Quick Start guide（クイックスタートガイド） をご覧ください。この記事では C# で書かれた銀行系アプリケーション開発プロセスにおける NUnit の使い方を紹介しています。

ユニットテストを開始するには、まず [Test] > [Unit Test Runner] の順に選択して Test Runner ウィンドウを開きます。これで、Unity Test Framework 同梱のテストサンプルが開きます。

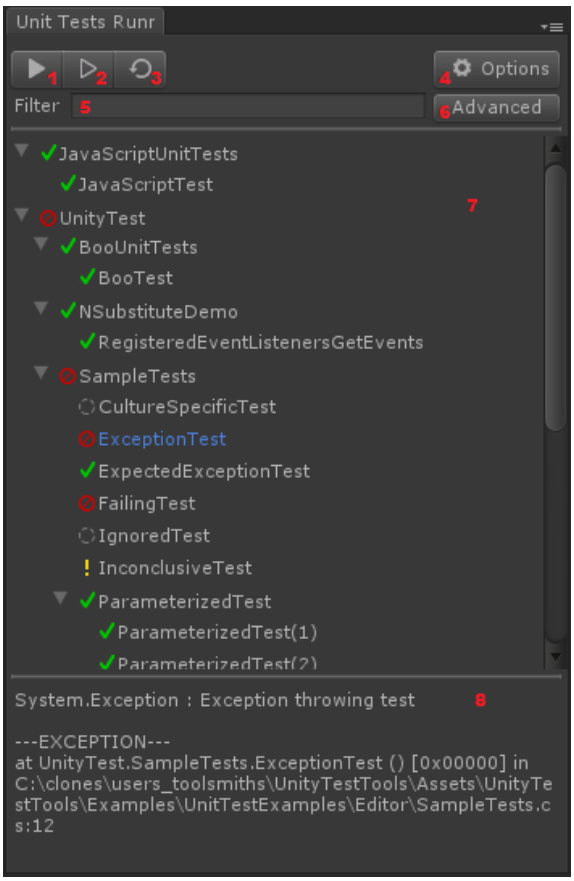
ユニットテストランナーの仕組み

ユニットテストランナーはプロジェクトに含まれる NUnit ライブラリ（nunit.core.dll、nunit.core.interfaces.dll、nunit.framework.dll）を使用しており、Assembly-CSharp.dll および Assembly-Editor-CSharp.dll 中に存在するテストを探します。

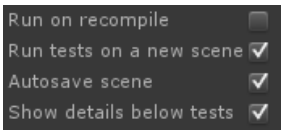
テストの実行前には Runner により新しいシーンが開かれるため（オプションをオフにしている場合を除く）、場合により作業中のシーンを保存するようダイアログが表示されます。実行後は自動的に、直前に作業していたシーンがロードされます。実行の合間にクリーンアップは行なわれないため、必要な場合はテストスイート側で実施する必要があります。シーン上の GameObjects を制御するには UnityUnitTest クラスが役立ちます。このクラスには GameObject を作成したり、自動クリーンアップを実行したりするメソッドが用意されています。

なおユニットテストファイルは Editor フォルダの下に配置し、ビルドに含まれないようにしておくことを推奨します。

ユニットテストランナーウィンドウ

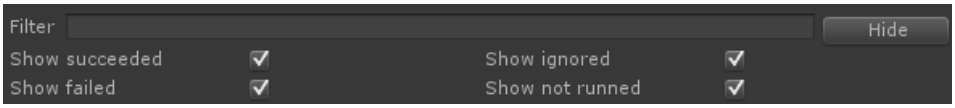


1. 全てのテストを実行
2. 選択中のテストを実行
3. 失敗したテストを実行
4. オプション - オプションパネルを表示します



   a. Run on recompilation（再コンパイルして実行） - テストごとにコンパイルして実行を行います。（コンパイルが失敗した時を除く）
   b. Run tests on a new scene（新規シーンで実行） - ランナーが新規シーンを開きテストを実行し、テストが終了したら現在のシーンに戻ります。
   c. Autosave scene（シーンを自動保存） - 自動的に実行前にシーンを保存します。（「Run tests on a new scene」にチェックが付いている場合）
   d. Show details below tests（テストの詳細を表示） - テストのリストの下に詳細タブを表示します。

5. フィルタ： 指定した文字列を含むメソッドとクラスだけをフィルタして表示します。
6. 詳細ボタン： フィルタの詳細設定を表示します



7. Tests 階層ウィンドウ： 各テストとその実行結果を示します
8. 例外または失敗したテストのスタックトレース表示領域

結果をレポートする

テストの結果は、実行完了時に毎回 Test Runmer Windowに表示されます。バッチモードの時はnUnit スタイルの XML ファイルとして生成されます。このファイルはパラメータを指定していない限り、プロジェクトのルートフォルダに格納されます。ファイルのスキーマは個々で見ることが出来ます: http://www.nunit.org/docs/2.6.2/files/Results.xsd

CUIからの実行（バッチモード）

テストはコマンドラインからも実行可能です。コマンドラインから実行するには、Unity をバッチモードで実行し、起動時に UnityTest.Batch.RunUnitTests メソッドを実行します。

パラメーター名	説明
resultFilePath	結果を保存するファイルパス。パスがフォルダの場合、デフォルトのファイル名が使用されます。指定しない場合は、プロジェクトのルートフォルダに保存されます。

例:

```
>Unity.exe -projectPath PATH_TO_YOUR_PROJECT -batchmode -quit -executeMethod UnityTest.Batch.RunUnitTests -
resultFilePath=C:\temp\results.xml
```

ファイルからテストをロードする（ユニットテストファイルのコンテキストメニュー）

実行したいテストがすべて単一ファイルに収まっている場合は、当該ファイルを右クリックして [Unity Test Tools] > [Load tests from this file]（このファイルからテストをロード）を選択すれば実行できます。

備考

- 本ツールでは nUnit の全機能がサポートされているわけではありません。サポートされていない機能は以下のとおりです：CategoryAttribute、RandomAttribute、ExplicitAttribute

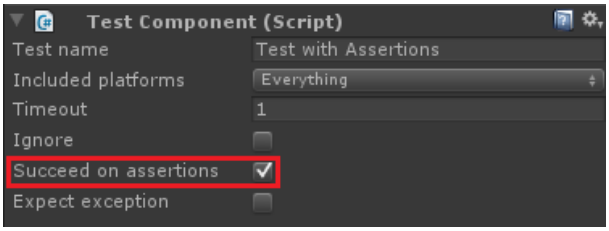
インテグレーションテストでアサーションコンポーネントを使用する

アサーションコンポーネントは、インテグレーションテストにおいて予期した通りの挙動が生じているかどうかを検証する目的に利用できます。Integration Test テストを選択すると、インスペクタで [Succeed on assertions]（アサート成功）オプションを選べるようになります。

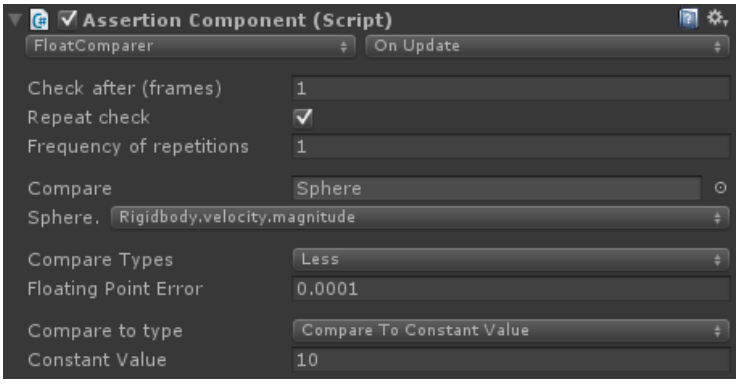
このオプションを有効にしておくと、テストのために Testing.Succeed() や Testing.Fail() メソッドを明示的に呼び出す必要がなくなります。これで、テストの下にある各オブジェクトへアタッチされている全アサーションコンポーネントがチェックされ、全アサートが 1 回以上チェックされると次に進むようになります。

実際の手順

1. シーンを新規作成します
2. Integration Tests Runner ウィンドウを開きます（メニューバーから [Unity Test Tools] > [Integration Tests Runner] 、または Shift + Ctrl + Alt + T）
3. テストを新規作成します
4. インスペクタで [Succeed on assertions]（アサート成功）オプションをオンにします



5. 球体（Sphere）に Rigidbody コンポーネントを追加します
6. Assertion Component をアタッチします
7. Assertion Component を次のように構成します:
    - 使用する比較クラス： **FloatComparer**
    - チェック: **On Update**
    - 値: **Sphere.Rigidbody.velocity.magnitude**
    - Compare Types（比較タイプ）： **Less**
    - Compare to type（比較する型）： **Compare to constant value（定数値と比較）**
    - Constant value（定数値）： **10**



この構成は、「球体（Sphere）の速度（velocity）が常時 10 未満であるよう Update 呼び出し時に毎回チェックする」とも読み替えられます。

8. Integration Tests Runner ウィンドウからテストを開始します（テストを選択した状態で CTRL + T を押しても開始できます）。球体が落ち始め、加速していきます。速度が 10 に達するとアサートが失敗し、テストは失敗として終了します。