Unity®

UNITY 開発者向け → E ブック

C# スタイル ガイドの作成

スケーラブルでクリーンなコードの記述

目次

“クリーンコード（CLEAN CODE）には、必ずその作成者の配慮が伺えます。」

《フェザーズ・マイケル・C .》(《平澤章、越智典子、稲葉信之、田村友彦、小堀真義》訳)『《レガシーコード改善ガイド（"Working Effectively With Legacy Code"）》』

はじめに

クリエイティブな仕事には苦労させられます。

一瞬のひらめきからコードを思いつき、そこから生み出したプロトタイプがちゃんと動いた！まず第一関門の突破、おめでとうということになります。しかし、単にコードを動作させるだけでは不十分です。ゲーム開発というものには、数多の課題があるものです。

ロジックが機能したら、そこからリファクタリングとクリーンアップのプロセスが始まります。

このガイドは、コードスタイルガイドの作成方法に関する業界の専門家からのアドバイスをまとめたものです。チームの各メンバーごとに、指針となるべくした手順書をしっかり作成さえしておけば、そのコードベース（codebase）が対象のプロジェクトを商業規模の製品レベルまでに成長させる、ということが保証されたようなものです。

以下に示すヒントやコツは、たとえ準備に余計な労力がかかったとしても、長期的には開発プロセスにとって助けとなります。また、よりクリーンで柔軟に対応できるようなコードベースを作成することで、チームを拡張する際にも効率的に新しい開発者が参入してもらうように手配することができます。

プロジェクトに参加する、すべての人々、そして自身の生活も、さらに快適にするために、コードをクリーンに保つように心がけましょう。

協力者

このガイドは、ビジュアルエフェクトアーティストとして映画業界やテレビ業界にて 3D と VFX の制作に 15 年以上にわたって取り組んできた経験を持ち、現在は独立系ゲーム開発者兼教育者である、Wilmer Lin 氏によって制作されました。制作には、シニアテクニカルコンテンツマーケティングマネージャー Thomas Krogh-Jacobsen のほか、Unity のシニアエンジニア Peter Andreasen、Scott Bilas、Robert LaCruise も大きく貢献しています。

© 2022 Unity Technologies

そもそも、クリーンコードとは何か？

ほとんどのゲーム開発者は、クリーンコードが読みやすくて管理しやすいコードであるということ認識しているでしょう。

クリーンコードとは、洗練されており、効率的で、読みやすいコードのことです。

これにはしっかりとした理由があります。まず、元々の作成者にとっては明白なことであっても、他の開発者にははっきり理解できないという場合があります。同様に、3 か月後に数点ロジックを実装する際に、特定のコードスニペットが何のためのものだったのかもう忘れてしまっているという可能性もあります。

クリーンコードは、開発のスケーラビリティを高めるとともに、以下に示す一連の生産標準に準拠することを目的としたものなのです。

— 一貫した命名規則に従う

— 読みやすさを前提にしてコードをフォーマットする

— クラスとメソッドを整理して、内容を小さくまとめ、読みやすくする

— 自明ではないコードには、どんなものであれコメントをつける

作成するのがモバイル向けのパズルであっても、コンソール向けの大規模な MMORPG であっても、コードベースをクリーンに保つことは、ソフトウェアメンテナンスの総コストを削減することにつながります。そうすれば、既存のソフトウェアに新機能を実装したりパッチをあてるのも、より簡単にできるようになるのです。

将来のチームメイトにとっても、また将来のあなた自身にとっても、これはありがたく感じることになるはずです。

© 2022 Unity Technologies

1 チームとしての開発

“コンピューターが理解できるコードなら、誰でも書けます。人間が理解できるコードを書けてこそ、優れたプログラマーだと言えるのです。」

– Martin Fowler（『*Refactoring*』著者）

すべてが 1 人で完結している開発者は存在しません。ゲームアプリケーションの技術的ニーズが高まっていけば、サポートが必要になります。そして必然的に、多彩なスキルを持ち合わせるチームメンバーたちを追加することになります。常に増員されるチームのためにクリーンコードでコーディング標準（Coding standards）を導入すれば、全員が同じ認識で作業に取り掛かれるようになります。そうすると、より一貫した形のガイドラインに沿って、皆が同じプロジェクト上で作業できるようになります。

スタイルガイドの作成方法について見ていく前に、まず、Unity 開発の拡張に役立つ一般的なルールをいくつか確認しましょう。

KISS（keep it simple, stupid）

コンピュータの扱いやプログラミングは困難を極めるものではありますが、否定しがたい事実として、エンジニアや開発者が物事を複雑にしすぎてしまうことは少なくないのです。目の前の問題に対する最も簡単な解決策を見つけるには、「シンプルに、馬鹿なままに」を意味する、KISS の原則を指針として取り入れるのが効果的です。

実績のあるシンプルな手法で課題を解決できるのなら、すべてを一から作り出す必要はありません。ただ使ってみたいというだけの理由で、思いつきで新しいテクノロジーを導入することにメリットがあるでしょうか？ Unity のスクリプティング API には、すでに多数のソリューションが内蔵されています。たとえば、既存の六角形タイルマップがあなたの戦略ゲームに適しているのであれば、マップの作成作業をスキップすることもできるのです。一番良いコードを書くということは、つまりコードを一切書かないということなのです。

KISS の原則

有名な KISS の原則は、設計のシンプルさを重視する考え方で、以下の引用からもわかるように、さまざまな時代にわたって広く支持されてきた考え方です。

「シンプルさは究極の洗練性である。」

– Leonardo da Vinci

「シンプルなタスクをシンプルにすべし！」

– Bjarne Stroustrup

「シンプルさは信頼性の前提条件である。」

– Edsger W. Dijkstra

「ものごとはできるかぎりシンプルにすべきだ。しかし、シンプルすぎてもいけない。」

– Albert Einstein

プログラミングの場合、これはコードをできるだけ簡素化することを意味します。
不必要に複雑さを増させることは避けましょう。

YAGNI 原則

関連する考え方で、YAGNI 原則（「you aren’ t gonna need it」）というものもあります。これは、実際に必要になるまで、新しい機能は追加しないほうがよいとする考え方です。現時点で導入する必要がない機能について心配することはせず、現在必要としている、最もシンプルな機能のみを構築するようにしましょう。

© 2022 Unity Technologies

問題をコードで解決しない

ソフトウェア開発の最初のステップは、自分が何を解決しようとしているのかを理解することです。このことは当たり前のように思えるかもしれませんが、開発者が実際の問題を理解しないままコードを実装しようとして作業に行き詰まるという事態は、ありふれるほどよく起こります。そうでなくとも問題の原因を完全に把握しないままに、コードが機能するまでコードを修正し続けることになります。

たとえば、メソッドの先頭に if-null ステートメントを手っ取り早く置くことで Null Reference Exception を修正できたとしても、それが本当の原因だったのか、それとも、より深い部分にある別のメソッドへの呼び出しが原因だったのか、判断できるでしょうか？

問題を解決するためのコードを単に追加するのではなく、根本的な原因を調査するようにしましょう。その場しのぎの解決策を適用するのではなく、問題が起こっている原因を自分自身で追求するのです。

日々、段階的に改善していく

クリーンコードを作成することは、流動的で継続的なプロセスです。この考え方をチーム全体に根付かせましょう。コードのクリーンアップを、開発者の日常生活の一部として捉えるのです。通常、壊れたコードを意図的に書く人はいません。しかし時間の経過とともに、そのようなコードが生まれてくるのです。コードベースに対しては、定期的なメンテナンスが欠かせません。そのための時間と予算を確保し、確実に実施するようにしましょう。

良いものを目指しながらも、完璧は求めない

一方で、完璧を求めることも禁物です。コードが生産基準を満たしたら、それをコミットして次に進むようにしましょう。

最終的には、コードによって何らかに機能を果たす必要があるのですから、新しい機能の実装と、コードクリーンアップとのバランスを図るようにしましょう。むやみにリファクタリングすることは避けてください。リファクタリングは、それが自分や他の誰かの利益になると判断できるときに行いましょう。

計画は必要だが、適応力も重要

Andy Hunt 氏と Dave Thomas 氏は、著書の『*《村上雅章》訳）『《新装版 達人プログラマー 職人から名匠への道（"The Pragmatic Programmer"》』*の中で、「プログラミングは建設というよりガーデニングに近い」と記しています。ソフトウェアエンジニアリングは有機的なプロセスなのです。すべてが計画通りにいくわけではないということも、あらかじめ覚悟しておきましょう。

たとえ緻密な図面を描けるとしても、紙の上で庭を設計することが良い結果を保証してくれるわけではありません。植えた植物が、予想とは違う花の咲かせ方をするかもしれません。その庭を思い通りに完成させるには、コードの一部を剪定したり、移植したり、入れ替えたりする必要があるのです。

ソフトウェアの設計は、建築家が設計図を描くようにはいきません。なぜなら、設計図よりも内容が変わりやすく、機械的ではないからです。開発者は、コードベースの成長に応じて対応をとっていく必要があります。

© 2022 Unity Technologies

一貫性を保つ

問題への対処方法が決まったら、それに似た他の問題にも同じ方法でアプローチしましょう。これは難しいことではありませんが、たゆまぬ努力が必要です。この原則は、命名規則（クラス、メソッド、大文字 / 小文字など）はもちろん、プロジェクトフォルダーやリソースの整理方法に至るまで、あらゆることに適用するようにしてください。

何よりも、スタイルガイドについてチームの同意を得たうえで、それに従ってもらうことが重要です。

チーム全体で取り組む

コードをクリーンかつシンプルに保つことはすべての人に最大の利益をもたらしますが、「クリーンかつシンプルである」ことは、「簡単である」ことと同じではありません。クリーンでシンプルなコードにするには努力が必要です。これは初心者にとっても、経験豊富な開発者にとっても大変な取り組みです。

チェックを怠れば、プロジェクトが混乱に陥ることになります。プロジェクトのさまざまな部分を色々な人々が担当するプロジェクトにおいて、それは当然の結果と言えます。コードが散らかった状態にならないようにするためには、全員が責任を負い、各チームメンバーがスタイルガイドを読んで、それに従う必要があります。クリーンアップはグループでの取り組みなのです。

自分とチームのためのスタイルガイド

このガイドでは、Unity での開発中に皆さんが遭遇するであろう、最も一般的なコーディング規則に焦点を当てています。これらは、Microsoft Framework Design Guidelines のサブセットです。同ガイドラインには、このガイドで示すもの以外にも多数の規則が含まれています。

これらのガイドラインは推奨事項であり、絶対に守るべき厳格なルールではありません。チームの好みに合わせてカスタマイズしてください。すべてのメンバーに適合するスタイルを選び、それを確実に適用するようにしましょう。

何より大事なことが一貫性なのです。以下の推奨事項に従えば、将来スタイルガイドを変更する必要が生じた場合にも、いくつかの検索操作と置換操作を行うことで、コードベースを迅速に移行することができます。

ご使用のスタイルガイドがこのドキュメントや Microsoft Framework Design Guidelines と食い違う場合は、スタイルガイドを優先してください。そうすることで、プロジェクト全体を通じて一貫したスタイルを維持できます。

© 2022 Unity Technologies

2 スタイルガイドの作成

“コンピューター科学において難しいことは 2 つしかありません。それは、キャッシュの無効化と名前の付け方です。」

– Phil Karlton 氏（ソフトウェアエンジニア）

アプリケーションは、それぞれ違った考え方をする可能性がある一個人の集まりによって作られます。スタイルガイドは、そのような違いによって生じるばらつきを抑え、まとまりのある最終製品を完成させるのに役立ちます。これを徹底すれば、Unity プロジェクトに参加したメンバーがどれだけ多くても、まるで 1 人の作成者によって開発されたかような製品が作れるでしょう。

Microsoft と Google のどちらからも、包括的なサンプルガイドが提供されています。

— Microsoft C# のコーディング規則

— Google の C# スタイルガイド

これらは、Unity 開発を管理するための優れた開始点となります。各ガイドでは、名前の付け方、フォーマット、およびコメント記述のためのソリューションが紹介されています。単独の開発者の場合、はじめはこれが制約のように感じるかもしれませんが、チームで作業するようになると、スタイルガイドに従うことが不可欠になります。

スタイルガイドは、後で配当を受け取るための初期投資だと考えましょう。1 つの標準セットを守るようにすれば、誰かを別のプロジェクトへと移動させる場合にも、再学習に費やされる時間を短縮できます。

スタイルガイドを利用すれば、コーディングの規則やフォーマットから曖昧さを排除することができます。一貫したスタイルを定めることで、明確な指示に従えるようになるのです。

Unity では、C# のサンプルスタイルシートを作成しました。このシートを参考にして、独自のガイドを作成することもできます。内容を自由にコピーし、必要に応じて調整を加えてください。

それでは始めていきましょう。

命名規則

名づけを行うということには、内面的心理学が関わっています。名前とは、ある物がどのような意味を持ってこの世に存在していのるかを示すものなのです。それは何なのか？それは誰なのか？それは何の役に立つのか？などといったことです。

変数、クラス、メソッドの名前は単なるラベルではありません。これらには意味があり、影響力があります。命名方法が効果的かどうかによって、プログラムを読んだ人が、プログラマーの伝えようとしている考えをどの程度理解できるかが変わってきます。

以下に示すのは、命名方法に関するいくつかのガイドラインです。

識別子名

識別子は、型（クラス、インターフェース、構造体、デリゲート、または enum）、メンバー、変数、または名前空間に割り当てる任意の名前です。識別子は、文字またはアンダースコア（_）で始まる必要があります。

C# では許可されていますが、識別子に特殊文字（バックスラッシュ、記号、Unicode 文字）を含めることは避けてください。これらは、特定の Unity コマンドラインツールの妨害要因となる可能性があります。多くのプラットフォームとの互換性を確保するために、特殊な文字の使用は避けてください。

ケーシングの用語

C# では、スペース文字は識別子を区切るために使用されるので、変数の名前にスペースを使用することはできません。ケーシングスキームは、ソースコードでの複合名やフレーズの使用に関する問題を軽減するのに役立ちます。命名規則とケーシング規則には、いくつか有名なものがあります。

キャメルケース

キャメルケース（キャメルキャップとも呼ばれます）は、単語を大文字で区切って、スペースや句読点を使わずにフレーズを記述する方式です。最初の文字は小文字にします。ローカル変数とメソッドパラメーターはキャメルケースです。

以下に例を示します。

```
examplePlayerController
maxHealthPoints
endOfFile
```

パスカルケース

パスカルケースはキャメルケースの一種で、頭文字を大文字にするものです。Unity開発では、クラス名とメソッド名にこれが使用されます。パブリックフィールドも、パスカルケースになる場合があります。以下に例を示します。

```
ExamplePlayerController
MaxHealthPoints
EndOfFile
```

スネークケース

スネークケースでは、単語間のスペースがアンダースコア文字に置き換えられます。以下に例を示します。

```
example_player_controller
max_health_points
end_of_file
```

ケバブケース

これは、単語間のスペースをダッシュに置き換えるというものです。単語がダッシュ文字によって「串刺し」にされたように見えることが、名前の由来です。以下に例を示します。

```
example-player-controller
Max-health-points
end-of-file
naming-conventions-methodology
```

© 2022 Unity Technologies

ケバブケースの問題点は、多くのプログラミング言語でダッシュがマイナス記号として使用されるということです。一部の言語では、ダッシュで区切られた数字がカレンダー日付として解釈されます。

ハンガリアン記法

変数や関数では、名前でその意図や型が示されることがよくあります。以下に例を示します。

```
int iCounter
string strPlayerName
```

ハンガリアン記法は古い慣習であり、Unity 開発では一般的ではありません。

フィールドと変数

変数とフィールドについては、以下の規則を検討してください。

- **変数名には名詞を使用する**：変数名は、物または状態を表すので、わかりやすく、明確である必要があります。そのため、変数が bool 型の場合を除き、名詞を使って名前を付けるようにしてください（以下を参照）。
- **Boolean には動詞でプレフィックスを付ける**：これらの変数は、true または false の値を示します。多くの場合、これらは質問に対する回答になります（「プレイヤーは走行中ですか？」、「ゲームは終了していますか？」など）。これらの意味をより明確にするために、動詞でプレフィックスを付けるようにしましょう。多くの場合、これは説明や条件と対になります（例：`isDead`、`isWalking`、`hasDamageMultiplier` など）。
- **意味のわかる名前にする。（数式でない限り）省略はしない**：変数名では、その意図を示すようにします。発音しやすく、検索しやすい名前を選びましょう。
- ループや数式の場合は 1 文字の変数でも構いませんが、それ以外の場合は省略を使用しないようにしましょう。一部の母音を省略して時間を節約することよりも、意味の明瞭さのほうが重要です。
- 即時のプロトタイピングを行うときには短い「ジャンク」名を使用してもかまいません。その場合は、後で意味のわかる名前にリファクタリングするようにしましょう。

© 2022 Unity Technologies

避けるべき名前の例	代わりに使用するべき名前	注記
`int d`	`int elapsedTimeInDays`	カウンターや式の場合を除き、1 文字の略語を使用することは避けてください。
`int hp`、`string tName`、`int mvmtSpeed`	`int healthPoints`、`string teamName`、`int movementSpeed`	変数名では意図を表すようにします。検索しやすく、発音しやすい名前にしましょう。
`int getMovememementSpeed`	`int movementSpeed`	名詞を使用します。Boolean（下記）の場合を除き、動詞はメソッドにのみ使用するようにしてください。
`bool dead`	`bool isDead` `bool isPlayerDead`	Boolean では、true か false で答えられる事柄を表します。

— **パブリックフィールドにはパスカルケースを使用する。private 変数にはキャメルケースを使用する：**パブリックフィールドの代わりに、パブリックの getter を使ったプロパティを使用してください（下記の「フォーマット」を参照）。

— **プレフィックスや特殊なエンコーディングを使いすぎないようにする：**プライベートメンバー変数のプレフィックスにアンダースコア（_）を付ければ、ローカル変数と区別できるようになります。

または、this キーワードを使用してコンテキスト内のメンバー変数とローカル変数を区別すれば、プレフィックスを省略することもできます。通常、パブリックフィールドとプロパティにはプレフィックスはありません。

一部のスタイルガイドでは、名前を見ただけで変数の種類わかるように、プライベートメンバー変数（m_）、定数（k_）、または静的変数（s_）のプレフィックスが使用されます。

多くの開発者はこれらを使用せず、エディターの機能に頼るというアプローチをとっています。ただし、すべての IDE で強調表示やカラーコーディングがサポートされているわけではありませんし、ツールによってはリッチコンテキストを一切表示できない場合もあります。プレフィックスをチーム全体でどのように適用するか（あるいは適用するかどうか）を決定する際には、この点を考慮してください。

— **アクセスレベルモディファイアを一貫して指定（または省略）する：**アクセスモディファイアを省略にした場合、コンパイラーはアクセスレベルを `private` と見なします。これ自体は問題ではありませんが、デフォルトのアクセスモディファイアを省略する方法については、一貫性を持たせるようにしてください。なお、これを後からサブクラスで使用する場合は、protected を使用する必要があります。

© 2022 Unity Technologies

サンプルコードスニペット

このガイド内のコードスニペットは内容が省略されており、機能しません。これらは、スタイルとフォーマットを示す目的でこちらに掲載しているものです。

なお、Microsoft の「フレームワークデザインのガイドライン」の修正版に基づいた、Unity 開発者向けのこちらのサンプル C# スタイルシートを参照することもできます。ただしこれは、チームのスタイルガイドを設定する方法の一例にすぎません。

以下のコード例に含まれている、特定のスタイルルールに注目してください。

- デフォルトの private アクセスモディファイアは省略しない。
- パブリックメンバー変数にはパスカルケースを使用する。
- プライベートメンバー変数はキャメルケースにし、プレフィックスとしてアンダースコア（_）を使用する。
- ローカル変数とパラメーターにはキャメルケースを使用し、プレフィックスは付けない。
- パブリックメンバー変数とプライベートメンバー変数はグループ化する。

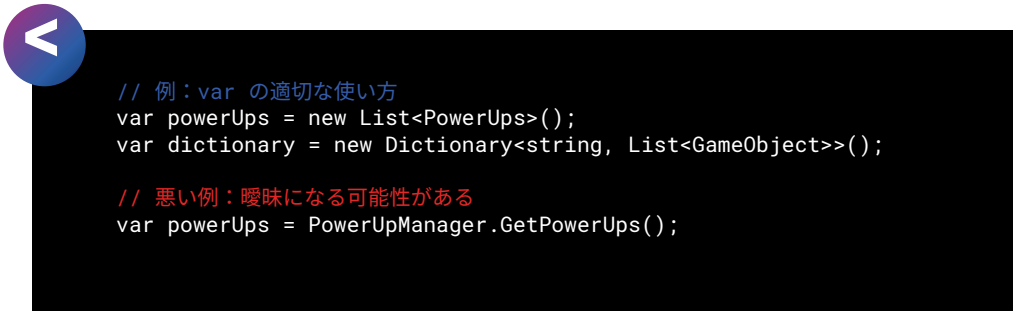
サンプルスタイルガイド内の各ルールを確認し、チームの意向に応じてカスタマイズしてください。優先すべきは、個々のルールの仕様そのものではなく、そのルールに従うことに全員が同意できるかどうかということです。同意が得られない場合は、チームの独自のガイドに従ってスタイル上の不合意を解決してください。

```
// 例：public 変数と private 変数

public float DamageMultiplier = 1.5f;
public float MaxHealth;
public bool IsInvincible;

private bool _isDead;
private float _currentHealth;

// パラメーター
public void InflictDamage(float damage, bool isSpecialDamage)
{
    // ローカル変数
    int totalDamage = damage;

    // ローカル変数とパブリックメンバー変数
    if (isSpecialDamage)
    {
        totalDamage *= DamageMultiplier;
    }

    // ローカル変数とプライベートメンバー変数
    if (totalDamage > _currentHealth)
    {
        /// ...
    }
}
```

© 2022 Unity Technologies

- **変数の宣言は 1 行に 1 つとする：**コンパクトさは低下しますが、読みやすさが向上します。
- **冗長な名前を避ける：**クラス名が Player である場合、PlayerScore や PlayerTarget というメンバー変数を作成する必要はありません。名前を縮めて、Score や Target などとしましょう。
- **冗談や駄じゃれを使わない：**これらは最初のうちは笑いを誘うかもしれませんが、何十回も見ていくうちに、infiniteMonkeys や dudeWheresMyChar などといった変数に面白みは感じられなくなります。
- **暗黙的に型指定されるローカル変数には var キーワードを使用する（それによって可読性が高まり、かつ型が明白な場合）：**どのような場合に var を使用するのかを、スタイルガイドで指定してください。たとえば、多くの開発者は、変数の型が曖昧になったり、ループの外側にプリミティブ型がある場合、var を使いません。

  一般的に var は、それによってコードが読みやすくなる場合（たとえば、型名が長い場合）で、かつ型が曖昧ではない場合に使用されます。



```
// 例：var の適切な使い方
var powerUps = new List<PowerUps>();
var dictionary = new Dictionary<string, List<GameObject>>();

// 悪い例：曖昧になる可能性がある
var powerUps = PowerUpManager.GetPowerUps();
```

enum

enum は、一連の名前付き定数によって定義される特別な値型です。デフォルトでは、定数は整数で、0 からカウントアップされます。

enum の名前と値には、パスカルケースを使用してください。パブリックの enum は、クラスの外部に配置してグローバルにすることができます。enum 名には単数形の名詞を使用します。

> **注：**System.FlagsAttribute 属性でマークされたビット単位の enum は、この規則の例外です。これらは複数の型を表すので、通常は複数形にします。

```
// 例：enum には単数形の名詞を使用します
public enum WeaponType
{
    Knife,
    Gun,
    RocketLauncher,
    BFG
}

public enum FireMode
{
    None = 0,
    Single = 5,
    Burst = 7,
    Auto = 8,
}

// 例：ただし、ビット単位の enum は複数形にします

[Flags]
public enum AttackModes
{
  // 10 進値                          // バイナリ値
   None = 0,                          // 000000
   Melee = 1,                         // 000001
   Ranged = 2,                        // 000010
   Special = 4,                       // 000100

   MeleeAndSpecial = Melee | Special  // 000101
}
```

クラスとインターフェース

クラスやインターフェースに名前を付ける際には、以下の標準ルールに従ってください。

- **クラス名にはパスカルケースを使用する。**
- **ファイル内に Monobehaviour がある場合は、ソースファイル名がそれに一致する必要がある：**ファイル内に他の内部クラスが存在する場合でも、Monobehaviour は 1 つのファイルにつき 1 つだけ存在するようにしてください。
- **インターフェース名には大文字の I でプレフィックスを付ける：**その後に、機能を説明する形容詞を続けます。

© 2022 Unity Technologies

```
// 例：クラスのフォーマット
public class ExampleClass : MonoBehaviour
{

    public int PublicField;
    public static int MyStaticField;

    private int _packagePrivate;
    private int _myPrivate;

    private static int _myPrivate;

    protected int _myProtected;

    public void DoSomething()
    {

    }
}

// 例：インターフェース
public interface IKillable
{
    void Kill();
}

public interface IDamageable<T>
{
    void Damage(T damageTaken);
}
```

メソッド

C# では、実行された命令はすべて、メソッドのコンテキストで実行されます。

> **注：**Unity 開発では、「関数」と「メソッド」という言葉が区別なく使われることも少なくありません。ただし、C# では関数はクラスに組み込まないと記述できないので、「メソッド」という用語が広く使われています。

メソッドではアクションが実行されるので、以下の規則を適用して適切な名前を付けてください。

- **動詞で始まる名前にする：**必要な場合はコンテキストを追加します（たとえば、`GetDirection`、`FindTarget` など）。
- **パラメーターにはキャメルケースを使用する：**ローカル変数など、メソッドに渡されるパラメーターをフォーマットしてください。
- **ブール値を返すメソッドは、質問の形にする必要がある：**Boolean 変数自体と同様、true/false の状態を返すメソッドには、動詞のプレフィックスを付けます。そうすることで、名前を質問の形にします（たとえば、`IsGameOver`、`HasStartedTurn` など）。

© 2022 Unity Technologies

```
// 例：メソッド名は動詞で始めます
public void SetInitialPosition(float x, float y, float z)
{
    transform.position = new Vector3(x, y, z);
}

// 例：ブール値を返すメソッドは、質問の形にします
public bool IsNewPosition(Vector3 currentPosition)
{
    return (transform.position == newPosition);
}
```

イベントとイベントハンドラー

C# のイベントはオブザーバーパターンを実装します。このソフトウェア設計パターンでは、サブジェクト（パブリッシャー）と呼ばれる 1 つのオブジェクトと、オブザーバー（サブスクライバー）と呼ばれる一連の依存オブジェクトとの間の関係を定義し、前者が後者に対して通知を行えるようにします。サブジェクトは、関連するオブジェクトを密結合することなく、オブザーバーに状態変化をブロードキャストできます。

サブジェクトとオブザーバー内のイベントやその関連メソッドについては、いくつかの命名方式があります。以下の方法を試してください。

- **英語の動詞句でイベント名は名づけをする：**状態の変化を正確に伝えられる名前にしてください。出来事の「前」や「後」のイベントであることを示すには、現在分詞または過去分詞を使用します。たとえば、ドアを開く前のイベントなら「OpeningDoor」とし、開いた後のイベントなら「DoorOpened」とします。
- **イベントには、System.Action デリゲートを使用する：**ほとんどの場合は、Action<T> デリゲートでゲームプレイに必要なイベントを処理できます。戻り値の型を void にすると、型の異なる 0 から 16 までの任意の入力パラメーターを渡すことができます。事前定義済みのデリゲートを使用すると、コードを節約できます。

> **注：**EventHandler デリゲートや EventHandler<TEventArgs> デリゲートを使用することもできます。イベントの実装方法については、チーム全員の合意を得るようにしてください。

```
// 例：イベント

// System.Action デリゲートの使い方

public event Action OpeningDoor;    // 前のイベント
public event Action DoorOpened;     // 後のイベント

public event Action<int> PointsScored;
public event Action<CustomEventArgs> ThingHappened;
```

© 2022 Unity Technologies

— **イベントを起こすメソッド（サブジェクト内）には、「On」というプレフィックスを付ける**：イベントを呼び出すサブジェクトは、通常、「On」というプレフィックスが付いたメソッドから呼び出しを実行します（たとえば、「OnOpeningDoor」や「OnDoorOpened」など）。

```
// サブスクライバーがある場合は、イベントを発生させます
public void OnDoorOpened()
{
    DoorOpened?.Invoke();
}

public void OnPointsScored(int points)
{
    PointsScored?.Invoke(points);
}
```

— **イベント処理メソッド（オブザーバー内）には、サブジェクトの名前とアンダースコア（_）を使ってプレフィックスを付ける**：サブジェクトの名前が「GameEvents」の場合、オブザーバーのメソッドには「GameEvents_OpeningDoor」や「GameEvents_DoorOpened」といった名前を付けることができます。

なお、これは「イベント処理メソッド」と呼ばれます。EventHandler デリゲートと混同しないように注意してください。

チームのための一貫した命名方式を決定し、それらのルールをスタイルガイドに明記しましょう。

— **必要に応じて、カスタムの EventArg を作成する**：イベントにカスタムデータを渡す必要がある場合は、System.EventArgs またはカスタム構造体から継承した、新しい型の EventArg を作成します。

```
// 必要な場合は EventArgs を定義します

// 例：ID と色を渡すために使用される、読み取り専用のカスタム構造体
public struct CustomEventArgs
{
    public int ObjectID { get; }
    public Color Color { get; }

    public CustomEventArgs(int objectId, Color color)
    {
        this.ObjectID = objectId;
        this.Color = color;
    }
}
```

© 2022 Unity Technologies

名前空間

名前空間は、クラス、インターフェース、enum などが、他の名前空間やグローバル名前空間にある既存のものと競合しないようにするために使用します。名前空間は、アセットストアのサードパーティアセットとの競合を防ぐ目的でも使用できます。

名前空間を適用する際には、以下のことを守ってください。

- パスカルケースを使用し、特殊記号やアンダースコアは使用しないでください。
- 名前空間のプレフィックスを繰り返し入力しないように、ファイルの先頭に using ディレクティブを追加してください。
- サブ名前空間も作成してください。名前のレベルを区切るには、ドット（.）演算子を使用します。これによって、スクリプトを階層型のカテゴリに編成することができます。たとえば、MyApplication.GameFlow、MyApplication.AI、MyApplication.UI などを作成して、ゲームの複数の論理コンポーネントを保持することができます。

```
namespace Enemy
{
    public class Controller1 : MonoBehaviour
    {
        ...
    }

    public class Controller2 : MonoBehaviour
    {
        ...
    }
}
```

コードでは、これらのクラスはそれぞれ Enemy.Controller1 および Enemy.Controller2 と呼ばれます。プレフィックスの入力を省略するために、using 行を追加してください。

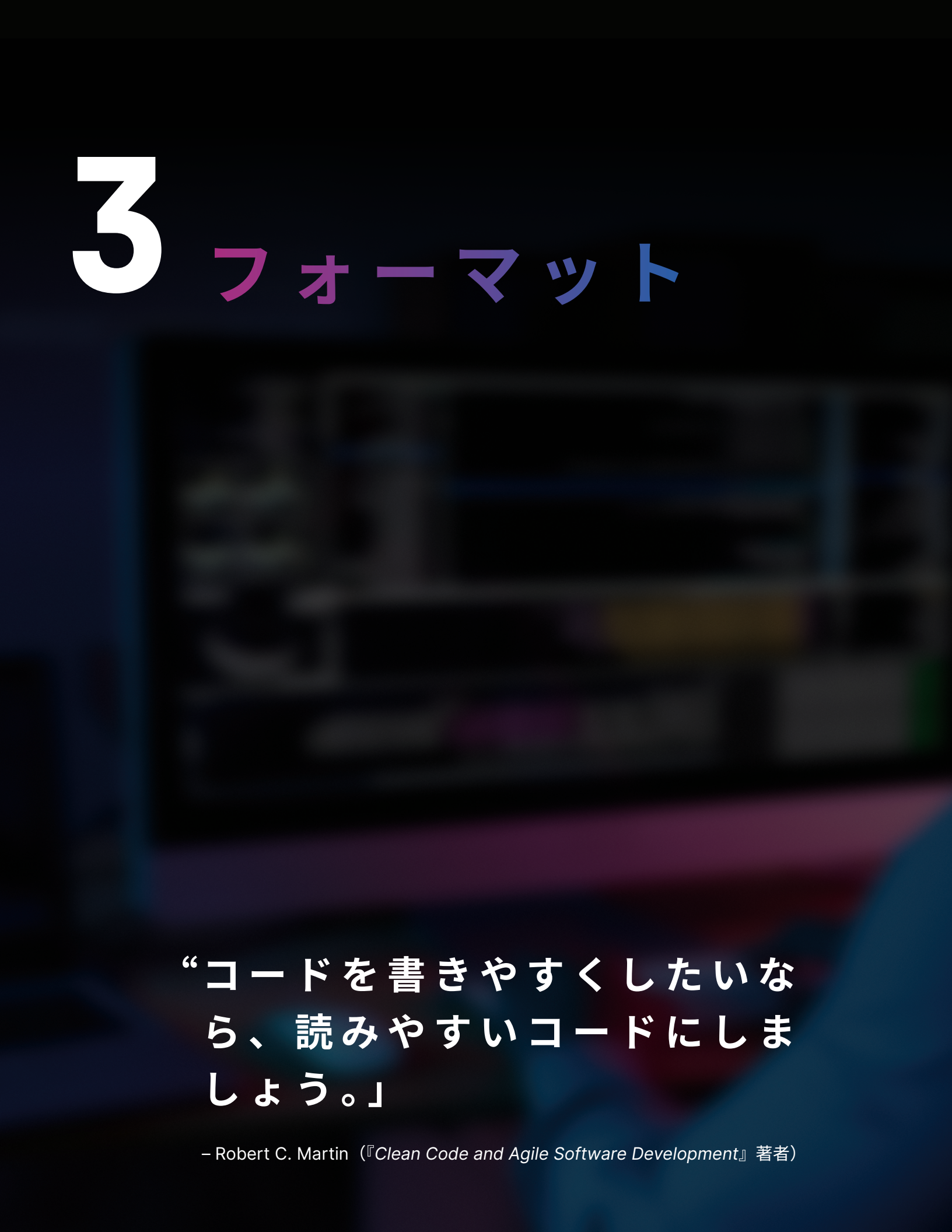
コンパイラーは、Controller1 および　Controller2 というクラス名を見つけると、それらが Enemy.Controller1 と Enemy.Controller2 を意味していると解釈します。

```
using Enemy;
```

スクリプトで、複数の異なる名前空間にある同じ名前のクラスを参照する必要がある場合は、プレフィックスを使ってクラスを区別します。たとえば、Player 名前空間に Controller1 クラスと Controller2 クラスがある場合、Player.Controller1 と Player.Controller2 のように記述することで、競合を回避できます。そうしないと、コンパイラーでエラーが報告されます。

© 2022 Unity Technologies

3 フォーマット



“コードを書きやすくしたいなら、読みやすいコードにしましょう。」

– Robert C. Martin（『*Clean Code and Agile Software Development*』著者）

名前の付け方に加えて、フォーマットによっても曖昧さを減らし、コードの明瞭さを高めることができます。標準化されたスタイルガイドに従えば、コードレビュー時に表記関連のチェックをする負担が減り、機能面をより重点的にチェックできるようになります。

スタイルガイドの作成時には、コードのフォーマット方法をチームでパーソナライズしましょう。Unity 開発スタイルガイドを定める際、コードフォーマットに関する以下の各提案を考慮し、チームのニーズに合わせて、これらのサンプルルールを省略、拡張、または変更してください。

いかなるケースにおいても、各フォーマットルールの実装方法をチームで検討し、全員に統一的に適用するようにしましょう。また、チームのスタイルを後から振り返って確認し、不一致があれば解決するようにしましょう。フォーマット関連の検討事項が減っていくにつれて、他の作業にも集中して取り組めるようになります。

それでは、フォーマットのガイドラインを見ていきましょう。

プロパティ

プロパティは、クラス値の読み取り、書き込み、または計算のための柔軟なメカニズムを提供します。プロパティは、パブリックメンバー変数のように動作しますが、実際にはアクセサーと呼ばれる特殊なメソッドです。各プロパティには、バッキングフィールドと呼ばれるプライベートフィールドにアクセスするための get メソッドと set メソッドがあります。

この方法によって、プロパティはデータをカプセル化し、ユーザーや外部オブジェクトによる不要な変更からデータをかくまいます。getter と setter にはそれぞれ独自のアクセスモディファイアがあります。これによってプロパティを、読み取り / 書き込み、読み取り専用、または書き込み専用にすることができます。

また、アクセサーを使用してデータを検証したり変換したりすることもできます（たとえば、データが希望の形式に適合しているかどうかを確認したり、値を特定の単位に変更したりすることもできます）。

プロパティの構文にはばらつきが生じる可能性があるため、スタイルガイドでプロパティのフォーマット方法を定義する必要があります。次の用法を参考にして、コード内でのプロパティの一貫性を確保してください。

— **単一行の読み取り専用プロパティには、式形式のプロパティを使用する（⇒）：**
これによって、private のバッキングフィールドが返されます。

```
// 例：式形式のプロパティ
public class PlayerHealth
{
    // private のバッキングフィールド
    private int maxHealth;

    // 読み取り専用。バッキングフィールドを返します
    public int MaxHealth => maxHealth;

    // 下記と同等：
    // public int MaxHealth { get; private set; }
}
```

— **他のすべてのプロパティでは、旧来の { get; set; } 構文を使用する：**バッキングフィールドを指定せずにパブリックプロパティを公開したい場合は、自動実装プロパティを使用します。

set および get アクセサーには、式形式の構文を適用します。

書き込みアクセスを許可したくない場合は、必ず setter を private にしてください。複数行のコードブロックでは、開き波括弧の位置に合わせて閉じ波括弧を配置しましょう。

```
// 例：式形式のプロパティ
public class PlayerHealth
{
    // バッキングフィールド
    private int _maxHealth;

    // getter と setter を明示的に実装する
    public int MaxHealth
    {
        get => _maxHealth;
        set => _maxHealth = value;
    }

    // 書き込み専用（バッキングフィールドを使用しない）
    public int Health { private get; set; }

    // 書き込み専用。明示的な setter なし
    public SetMaxHealth(int newMaxValue) => _maxHealth = newMaxValue;

}
```

シリアル化

スクリプトのシリアル化は、データ構造やオブジェクトの状態を、Unity が後で保存および再構築できる形式へと変換するための自動プロセスです。パフォーマンス上の理由から、Unity では他のプログラミング環境とは異なる方法でシリアル化が処理されます。

シリアル化されたフィールドはインスペクターに表示されます。ただし、静的、定数、または読み取り専用フィールドをシリアル化することはできません。これらは、パブリックであるか、[SerializeField] 属性でタグ付けされている必要があります。Unity では、特定のフィールドタイプだけがシリアル化されます。シリアル化ルールの完全なセットについては、ドキュメントページを参照してください。

シリアル化されたフィールドを使用する際には、以下の基本的なガイドラインに従ってください。

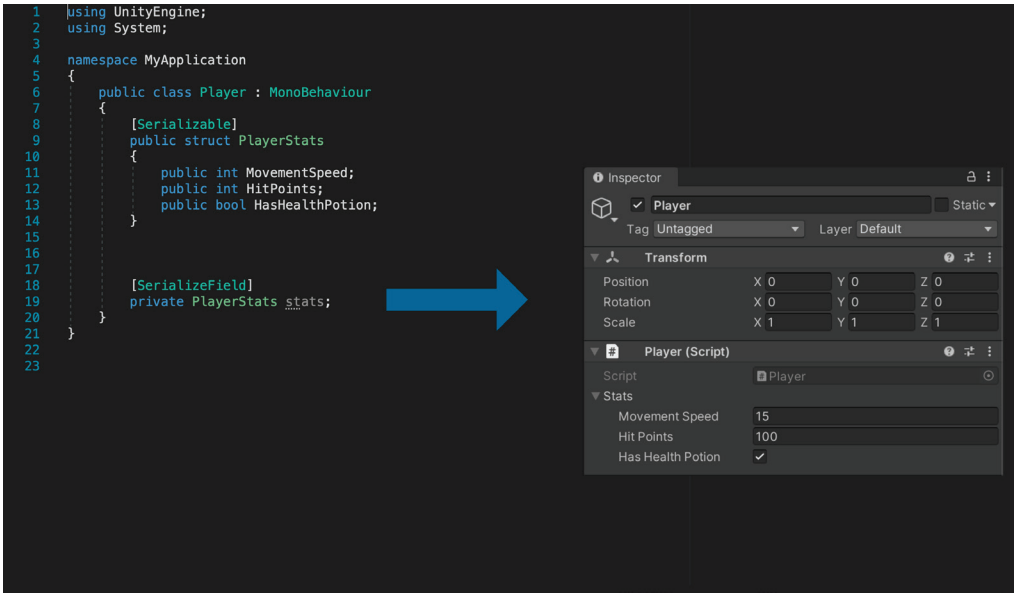
— **[SerializeField] 属性を使用する：**SerializeField 属性は、private 変数または protected 変数と連携させて、インスペクターに表示することができます。これにより、変数を public としてマークするよりもデータが効果的にカプセル化され、外部オブジェクトによる値のオーバーライドが防止されます。

— **最小値と最大値の設定には、Range 属性を使用する**：[Range(min, max)] 属性は、ユーザーが数値フィールドに割り当てることのできる値を制限したい場合に便利です。また、インスペクターでフィールドをスライダーとして表すこともできるので便利です。

— **シリアル化可能なクラスや構造体にデータをグループ化して、インスペクターをクリーンアップする**：パブリックのクラスまたは構造体を定義し、[Serializable] 属性でマークしましょう。インスペクターで公開したい型ごとに、public 変数を定義してください。

```
// 例：PlayerStats 用のシリアライズ可能クラス

using System;
using UnityEngine;

public class Player : MonoBehaviour
{
    [Serializable]
    public struct PlayerStats
    {
        public int MovementSpeed;
        public int HitPoints;
        public bool HasHealthPotion;
    }

// 例：インスペクターにプライベートフィールドが表示されます

    [SerializeField]
    private PlayerStats _stats;
}
```

このシリアル化可能クラスを別のクラスから参照すると、結果の変数が、インスペクター内の折りたたみ可能なユニット内に表示されます。



シリアル化可能なクラスや構造体は、Inspector を整理するのに役立ちます。

© 2022 Unity Technologies

波括弧またはインデントのスタイル

C# では、インデントのスタイルとして一般的なものが 2 つあります。

- オールマンスタイルでは、開き波括弧を新規行に配置します。これは BSD スタイルとも呼ばれます（BSD Unix が由来）。
- The K&R スタイル（または「one true brace style（1 つの真の波括弧スタイル）」）では、開き波括弧を前のヘッダーと同じ行に配置します。

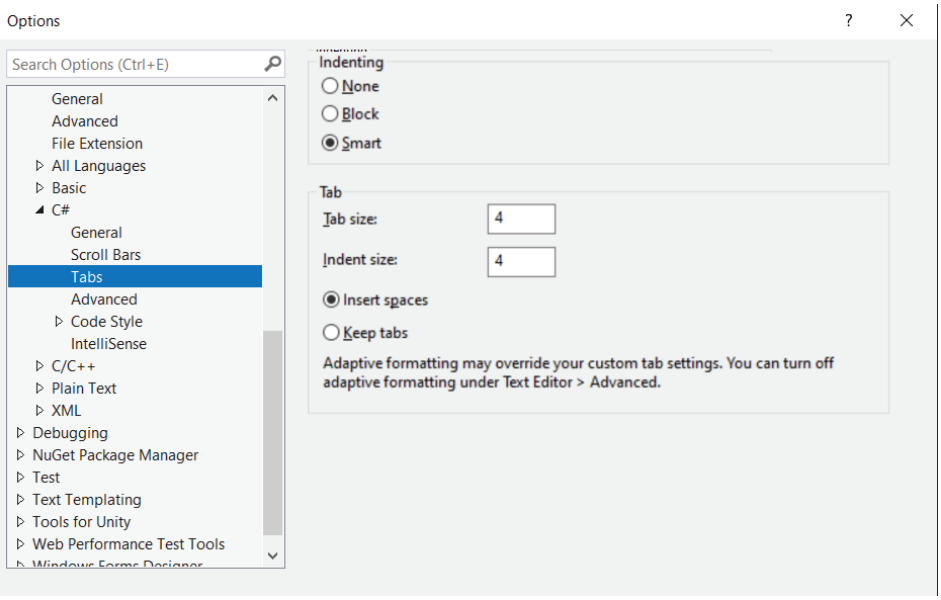
```
// 例：オールマン（BSD）スタイルでは、開き波括弧を新規行に配置します。

void DisplayMouseCursor(bool showMouse)
{
     if (!showMouse)
     {
          Cursor.lockState = CursorLockMode.Locked;
          Cursor.visible = false;
     }
     else
     {
          Cursor.lockState = CursorLockMode.None;
          Cursor.visible = true;
     }
}

// 例：K&R スタイルでは、開き波括弧を前の行に配置します。

void DisplayMouseCursor(bool showMouse){
     if (!showMouse) {
          Cursor.lockState = CursorLockMode.Locked;
          Cursor.visible = false;
     }
     else {
          Cursor.lockState = CursorLockMode.None;
          Cursor.visible = true;
     }
}
```

これらのインデントスタイル にも、さらにバリエーションがあります。このガイドの例では、Microsoft Framework Design Guidelines のオールマンスタイルを使用しています。どれを選ぶかにかかわらず、チームのすべてのメンバーが同じインデントスタイルと波括弧スタイルに従うようにしてください。

以下のテクニックを試してみてください。

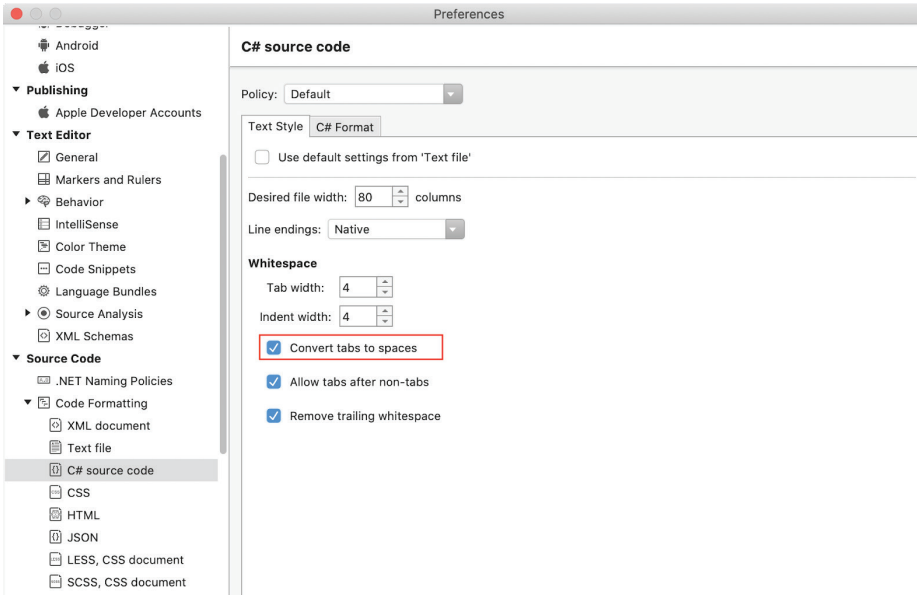
- **統一のインデントを決める：**これは通常、4 つから 2 つのスペースにします。タブかスペースかで対立することは避け、チームの全員が同意できるエディター設定を決めてください。Visual Studio には、タブをスペースに変換するオプションがあります。

  Visual Studio（Windows）で、**「ツール」>「オプション」>「テキストエディター」>「C#」>「タブ」の順に移動します。**



Visual Studio の「Tabs」の設定

Visual Studio for Mac で、**「Preferences」>「Source Code」>「C# Source Code」**の順に移動します。「Text Style」を選択して設定を調整します。



タブをスペースに変換して、インデントスタイルを統一します。

— **1 行のステートメントでも、可能な限り括弧は省略しない：**これにより一貫性が向上し、コードの読み取りとメンテナンスが容易になります。この例では、波括弧によってアクション（DoSomething）をループから明確に分離しています。

後で Debug 行を追加したり、DoSomethingElse を実行したりする必要が生じた場合も、波括弧がすでに付いた状態になります。句を個別の行に配置することで、ブレークポイントを簡単に追加できるようになります。

© 2022 Unity Technologies

```
// 例：わかりやすくするために波括弧を維持する ...

for (int i = 0; i < 100; i++) { DoSomething(i); }

// … また、句は個別の行に分ける
for (int i = 0; i < 100; i++)
{
    DoSomething(i);
}

// 悪い例：波括弧を省略する

for (int i = 0; i < 100; i++) DoSomething(i);
```

— **ネストされた複数行のステートメントから波括弧を削除しない：**この場合、波括弧を削除してもエラーにはなりませんが、わかりにくくなる恐れがあります。省略可能な場合でも、わかりやすくするために波括弧を使いましょう。

```
// 例：わかりやすくするために波括弧をそのままにしておく

for (int i = 0; i < 10; i++)
{
        for (int j = 0; j < 10; j++)
     {
                ExampleAction();
     }
}
// 悪い例：ネストされた複数行のステートメントから波括弧を削除する

for (int i = 0; i < 10; i++)
        for (int j = 0; j < 10; j++)
                ExampleAction();
```

— **switch ステートメントを標準化する：**フォーマットがばらつく可能性があるので、チームとしての方針をスタイルガイドで文書化しましょう。以下に示すのは、case ステートメントをインデントする場合の例です。

```
// 例：switch ステートメントの case をインデントする
switch (someExpression)
{
   case 0:
      DoSomething();
      break;
   case 1:
      DoSomethingElse();
      break;
   case 2:
      int n = 1;
      DoAnotherThing(n);
      break;
}
```

© 2022 Unity Technologies

EditorConfig とは？

複数の開発者が異なるエディターや IDE を使って同じプロジェクトに取り組んでいる場合は、**EditorConfig** ファイルを使うことを検討してください。

EditorConfig ファイルは、チーム全体に対応したコーディングスタイルを定義するのに役立ちます。これは多くの IDE（Visual Studio や Rider など）でネイティブサポートされているので、別途プラグインを追加する必要がありません。

EditorConfig ファイルは読みやすく、バージョン管理システムで使用できます。サンプルファイルはこちらから確認できます。EditorConfig のコードスタイルはコードと一緒に移動されるので、Visual Studio 以外でも同じコードスタイルを適用できます。

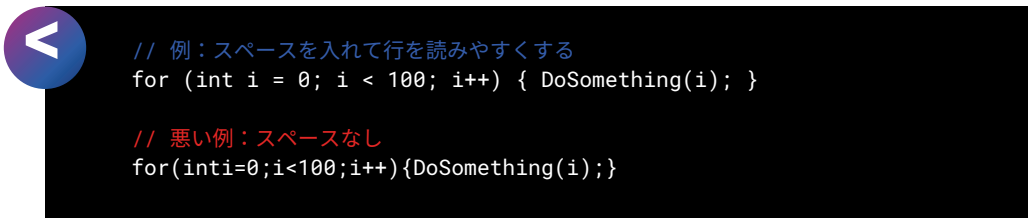
EditorConfig の設定は、Visual Studio のグローバルなテキストエディター設定よりも優先されます。**.editorconfig** ファイルなしでコードベースの作業を行う場合や、特定の設定が **.editorconfig** ファイルによってオーバーライドされない場合は、個人のエディター設定が適用されます。

GitHub リポジトリで実際のサンプルを参照してください。

水平方向のスペーシング

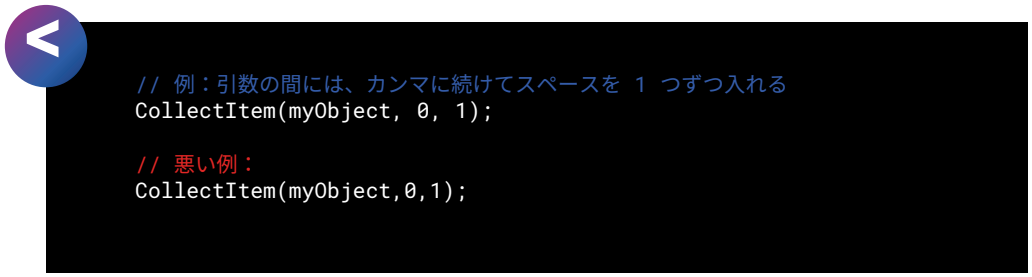
スペースのようなシンプルなものでも、画面上でのコードの見やすさを向上させることができます。フォーマットの好みは人によって異なる場合もありますが、読みやすさを向上させるために以下の推奨事項を試してみてください。

— **スペースを追加してコードの密度を下げる：**空白を追加することで、行内の特定の部分と別の部分に、視覚的な分離感を持たせることができます。



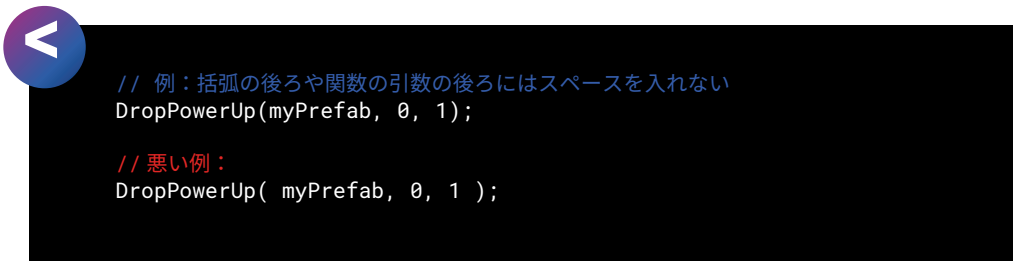
```
// 例：スペースを入れて行を読みやすくする
for (int i = 0; i < 100; i++) { DoSomething(i); }

// 悪い例：スペースなし
for(inti=0;i<100;i++){DoSomething(i);}
```

— **関数の引数の間には、カンマに続けてスペースを 1 つずつ入れる。**



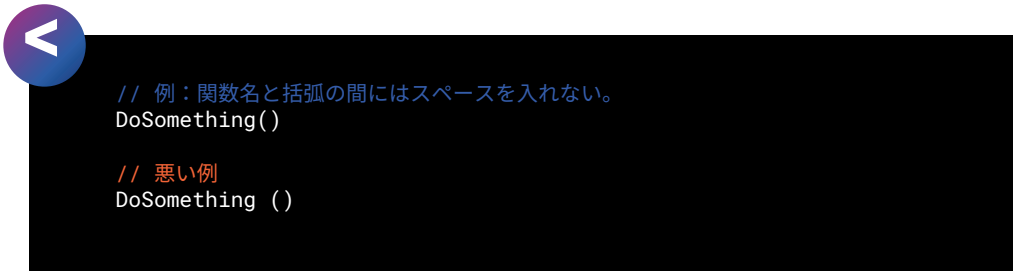
```
// 例：引数の間には、カンマに続けてスペースを 1 つずつ入れる
CollectItem(myObject, 0, 1);

// 悪い例：
CollectItem(myObject,0,1);
```

— **括弧の後ろや関数の引数の後ろにはスペースを追加しない。**



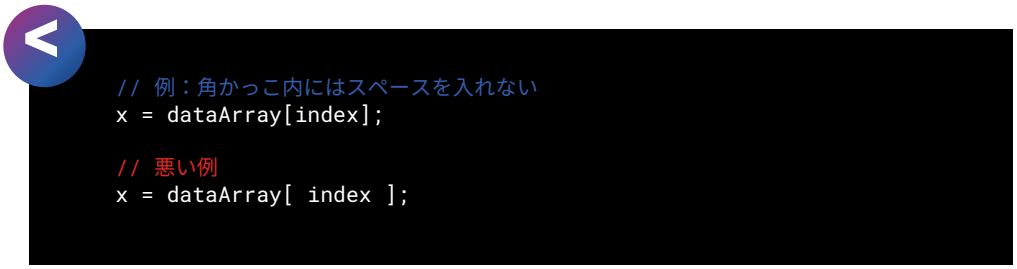
```
// 例：括弧の後ろや関数の引数の後ろにはスペースを入れない
DropPowerUp(myPrefab, 0, 1);

// 悪い例：
DropPowerUp( myPrefab, 0, 1 );
```

— **関数名と括弧の間にはスペースを入れない。**



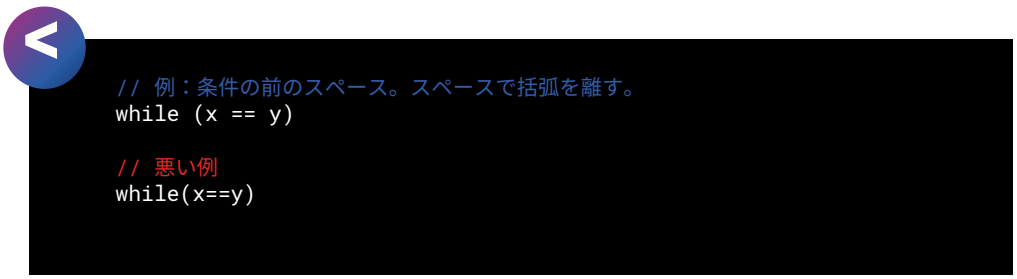
```
// 例：関数名と括弧の間にはスペースを入れない。
DoSomething()

// 悪い例
DoSomething ()
```

— **角かっこの内側にはスペースを入れない。**



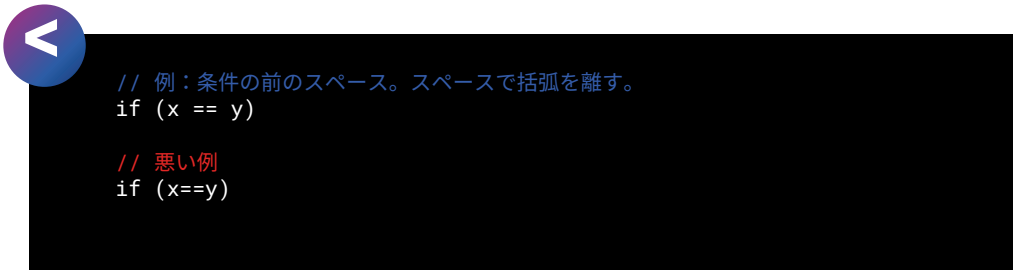
```
// 例：角かっこ内にはスペースを入れない
x = dataArray[index];

// 悪い例
x = dataArray[ index ];
```

— **フロー制御条件の前にはスペースを 1 つ入れる：**フロー比較演算子と括弧の間にはスペースを 1 つ入れましょう。



```
// 例：条件の前のスペース。スペースで括弧を離す。
while (x == y)

// 悪い例
while(x==y)
```

— **比較演算子の前後にはスペースを 1 つ入れる。**



```
// 例：条件の前のスペース。スペースで括弧を離す。
if (x == y)

// 悪い例
if (x==y)
```

© 2022 Unity Technologies

- **行を短く収める。水平方向の空白について検討する**：標準の行幅（80 文字から 120 文字）を決めましょう。行が長くなる場合は、より短いステートメントに分割して、行幅を超えないようにしてください。
- **インデント / 階層を維持する**：読みやすくするためにコードをインデントしましょう。
- **読みやすくするために必要な場合を除き、列の位置合わせは行わない**：この種のスペーシングを行うと、変数の位置は揃いますが、型と名前のペアがわかりにくくなります。

  ただし、データ量の多いビット単位の式や構造体の場合は、列の位置合わせが役に立つ場合もあります。とは言え、項目数が増えてくると、列の位置合わせを維持するための作業が増える可能性があるので注意してください。また、一部の自動フォーマッターでは、列のどの部分を揃えるかが変わる場合もあります。

```
// 例：型と名前の間にスペースを 1 つ入れる

    public float Speed = 12f;
    public float Gravity = -10f;
    public float JumpHeight = 2f;

    public Transform GroundCheck;
    public float GroundDistance = 0.4f;
    public LayerMask GroundMask;

// 悪い例：列の位置合わせ

    public float                Speed = 12f;
    public float                Gravity = -10f;
    public float                JumpHeight = 2f;
    public Transform            GroundCheck;
    public float                GroundDistance = 0.4f;
    public LayerMask            GroundMask;
```

縦方向のスペーシング

縦方向のスペーシングも効果的に利用できる場合があります。スクリプト内の関連する部分をまとめたい場合、空白の行を使うと便利です。コードを縦方向で整理するには、以下の推奨事項を実践してみてください。

- **依存関係のあるメソッドや類似するメソッドをグループ化する**：コードは論理的で一貫性のあるものにする必要があります。同じ働きをするメソッドを並べて配置しておけば、ロジックを読んでいる人がファイル内をあちこち移動して回らなくて済みます。
- **クラス内の異なる部分を区切るには、縦方向の空白を使う**：たとえば、以下の要素の間には 2 つの空白行をはさむと効果的です。
  - 変数宣言とメソッド
  - クラスとインターフェース
  - `if-then-else` ブロック（それによって読みやすくなる場合）

これの使用範囲は最小限に抑え、どのような場合に適用できるかをスタイルガイドに明記するようにしてください。

領域

`#region` ディレクティブを使用すると、C# ファイル内のコードセクションを折りたたんだり非表示にしたりできるので、大きなファイルの管理が容易になり、読みやすくなります。

ただし、このガイドで説明している、クラスについて一般的なアドバイスに従っていれば、クラスのサイズはそれほど大きくならないはずなので、`#region` ディレクティブを使うのはやりすぎです。コードブロックを領域から隠すよりも、コードをより小さなクラスに分割するようにしましょう。ソースファイルが短ければ、領域を追加する必要性も減ります。

注：多くの開発者は、領域が「コードの臭い」またはアンチパターンだと考えています。どちらの考えに与するかについては、チームで結論を出してみてください。

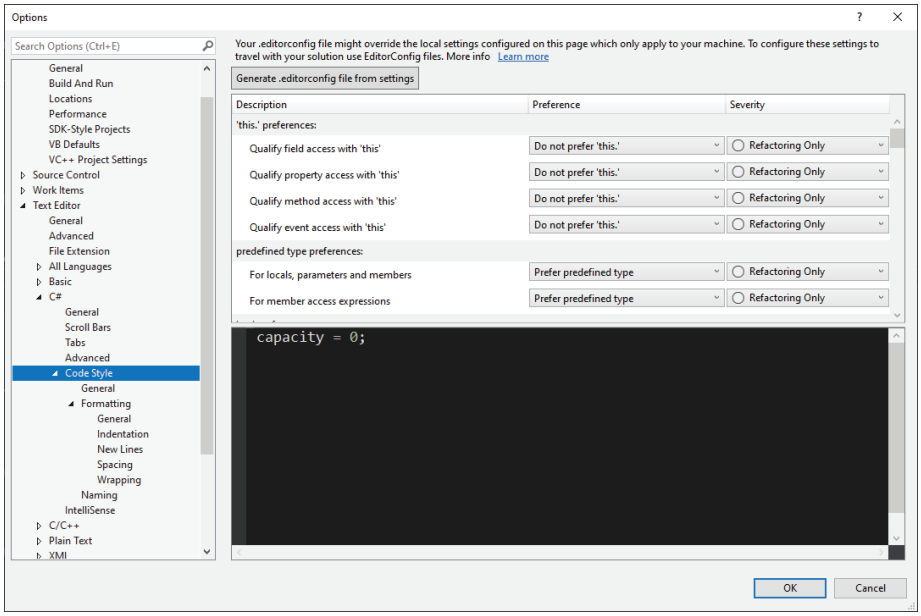
Visual Studio でのコードフォーマット

これらのフォーマット規則の適用が途方もない作業に思えても、あきらめてはいけません。最新の IDE を使用すれば、フォーマットを効率的に設定し適用できます。フォーマット規則のテンプレートを作成し、プロジェクトファイルを一気に変換できるのです。

スクリプトエディター用のフォーマット規則を設定するには：

— Visual Studio（Windows）で、**「Tools」>「Options」の順に移動します。「Text Editor」>「C#」>「Code Style Formatting」を探します。**

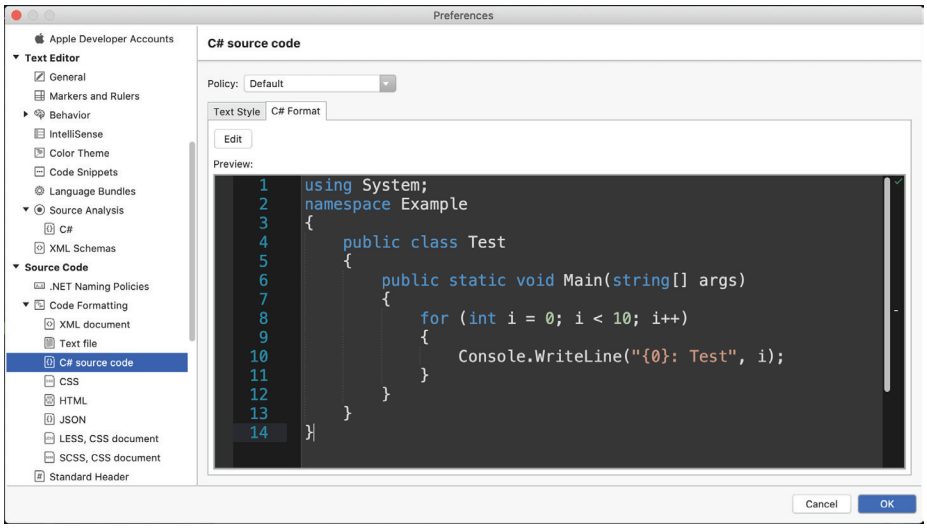
設定を使って、「General」、「Indentation」、「New Lines」、「Spacing」、および「Wrapping」オプションを変更します。



コードスタイルのフォーマットオプション

— Visual Studio for Mac で、**「Visual Studio」>「Preferences」を選択し、「Source Code」>「Code Formatting」>「C# source code」の順に移動します。**

上部の「Policy」を選択します。その後、「Text Style」タブでスペーシングとインデントを設定します。「C# Format」タブで、「Indentation」、「New Lines」、「Spacing」、および「Wrapping」の設定を調整します。



「Preview」ウィンドウにスタイルガイドの選択結果が表示されます。

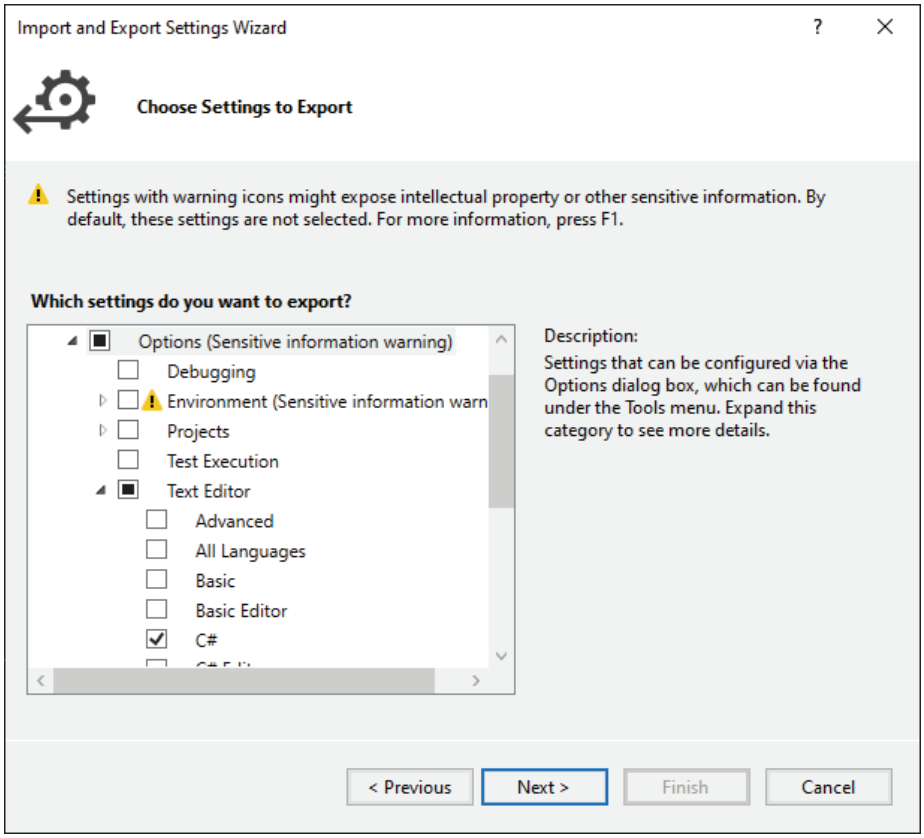
スクリプトファイルを強制的にスタイルガイドに準拠させたい場合は、次の手順を実行します。

— Visual Studio（Windows）で、**「Edit」>「Advanced」>「Format Document」（ホットキーコードは Ctrl + K、Ctrl + D）**の順に移動します。スペースとタブの位置合わせだけをフォーマットしたい場合は、エディターの下部にある「Run Code Cleanup」（**Ctrl + K、Ctrl + E**）を使用することもできます。

— Visual Studio for Mac で、**「Edit」>「Format Document」（ホットキーは Ctrl + I）**の順に移動します。

Windows では、**「Tools」>「Import and Export Settings」**からエディター設定を共有することもできます。スタイルガイドの C# コードフォーマットを使ったファイルをエクスポートし、すべてのチームメンバーにそのファイルをインポートさせてください。

© 2022 Unity Technologies



共有する C# コードフォーマットのエクスポート

Visual Studio を使用すると、スタイルガイドに簡単に準拠できます。ホットキーを使うのと同じくらい簡単に、フォーマットを適用できます。

注：Visual Studio の設定をインポートしたりエクスポートしたりする代わりに、EditorConfig ファイル（上記を参照）を設定することもできます。これを使えば、さまざまな IDE 間でフォーマットをより簡単に共有できるようになりますし、バージョン管理の面でもメリットが得られます。詳細については、.NET コードスタイルルールのオプションを参照してください。

クリーンコードに限定した話ではありませんが、Visual Studio を使って Unity でのプログラミングワークフローを迅速化するための 10 の方法も必ずチェックしてください。これらの生産性のテクニックを適用すると、クリーンコードのフォーマットとリファクタリングがはるかに簡単になります。

© 2022 Unity Technologies

4 クラス

“コンピューティングの短い歴史の中で、完璧なソフトウェアを書いた人は１人もいません。あなたがその１人目になる可能性も低いでしょう。」

– Andy Hunt（『《村上雅章》訳）『《達人プログラマー（"The Pragmatic Programmer"》』著者）

Robert C. Martin 氏の『*Clean Code*』によると、クラスに関する第一のルールは、小さくなくてはならないということです。そして第二のルールは、それよりもさらに小さくするべきだということです。

サイズを制限することで、それぞれのクラスがより集中的で、まとまりのあるものになります。既存のクラスに内容を追加し続けていくと、機能の範囲はすぐに広がりすぎてしまいます。そうならないよう、クラスはなるべく短くまとめるように心がけましょう。クラスが肥大化すると可読性が低下し、トラブルシューティングも難しくなります。

新聞というメタファー

クラスのソースコードをニュース記事だと思ってみてください。最初から読み始めると、まず見出しと署名記事が目を引きます。導入段落では大まかな概要が示され、下に進むにつれてさらに詳しい内容へと掘り下げられていきます。

ジャーナリストはこれを、逆ピラミッドと呼びます。最も注目すべき項目は、大きな文字で冒頭に記載されます。その後、末尾に向かって読み進めるにつれて、詳細な内容が伝えられていきます。

クラスも、この基本パターンに従うのが効果的です。上から下に向かって内容を整理しながら、複数の関数で階層を形成していくものだと考えてください。上層のレベルから全体を構成基盤を提供するようなメソッドは最初に配置し、その後に下位レベルの関数を配置して、実装の詳細を指定していきます。

たとえば、`SetInitialVelocity` および `CalculateTrajectory` というメソッドを参照する、`ThrowBall` というメソッドを作成する場合、メインのアクションを説明するのは `ThrowBall` なので、それを最初に配置します。その後、その下にサポートメソッドを追加します。

それぞれのニュース記事は短くても、新聞やニュースのウェブサイトにはそのような記事がたくさん掲載されます。それらの記事が集まることで、全体としての統一的な機能が構成されるのです。Unity プロジェクトについても、同じように考えてください。多数のクラスが集まることで、より大きな、それでいて一貫性のあるアプリケーションが形成されるようにしましょう。

クラスの整理

いずれのクラスにも、何らかの標準化が必要です。クラスメンバーは、以下のセクションへとグループ化して整理しましょう。

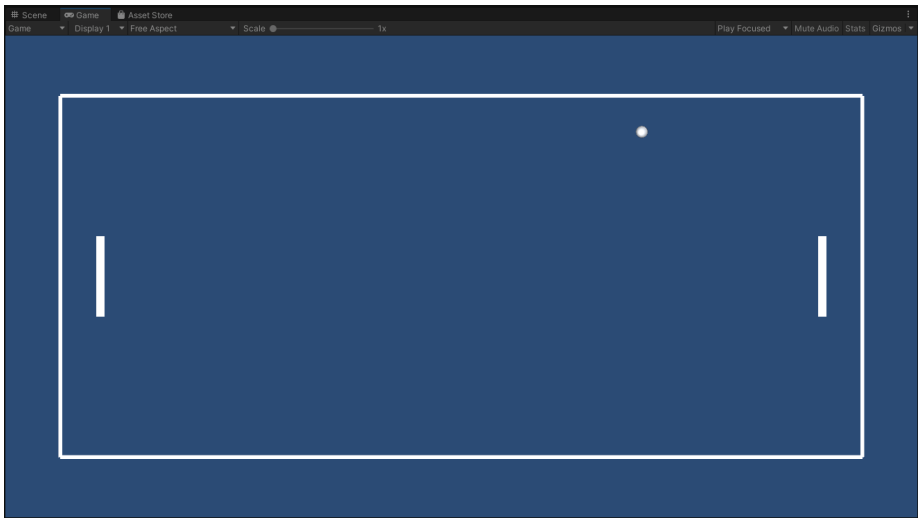
- フィールド
- プロパティ
- イベント / デリゲート
- Monobehaviour メソッド（Awake、Start、OnEnable、OnDisable、OnDestroy など）
- public メソッド
- private メソッド

Unity が推奨するクラス命名規則を思い出してください。ソースファイル名は、ファイル内の Monobehaviour の名前と一致する必要があります。ファイル内に他の内部クラスがある場合でも、Monobehaviour は 1 つのファイルにつき 1 つだけ存在するようにしてください。

単一責任の原則

先にも述べたように、目指すべきことは、各クラスを短くすることです。ソフトウェア設計において、シンプル化を目指すうえでの指針となるのが、単一責任の原則です。

これは、それぞれのモジュール、クラス、関数に、1 つの役割を担当させるという考え方です。たとえば、*ピンポン*のゲームを作りたい場合、あなたはまず、ラケット、ボール、壁などのクラスから作り始めるでしょう。

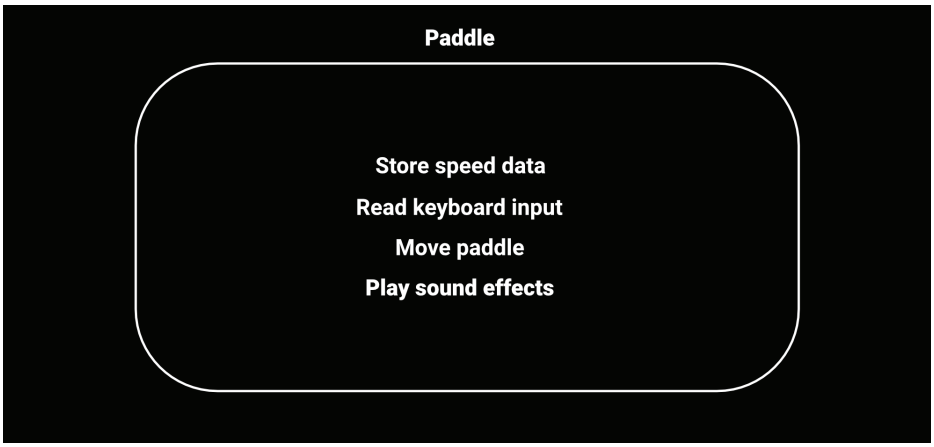


ピンポンのゲームを作る場合

Paddle（ラケット）クラスでは、次のことを行う必要があります。

- ラケットの移動速度に関する基本データを保存する
- キーボード入力をチェックする
- 入力に応じてラケットを移動させる
- ボールにぶつかったときに音を再生する

ゲームデザインがシンプルなので、これらすべてを基本の Paddle クラスに組み込むこともできます。実際、Monobehaviour を 1 つ作成して、必要なすべてのことをそれで実行することは可能です。



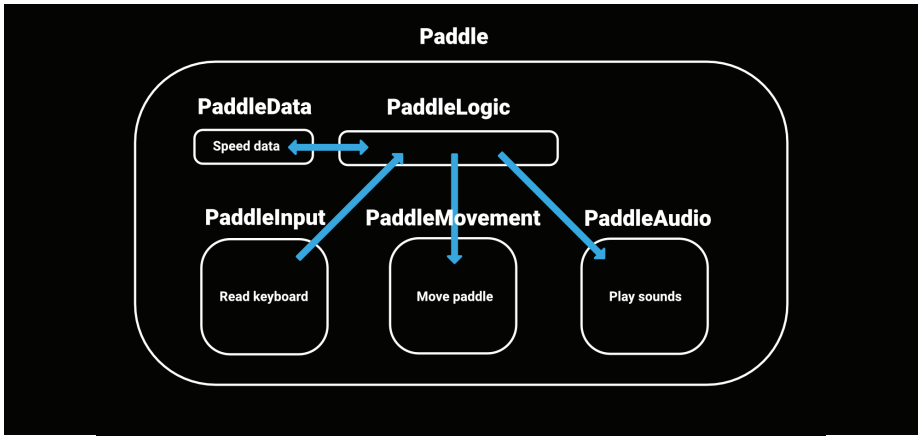
1 つの Monobehaviour ですべてのことをする

© 2022 Unity Technologies

しかし、たとえ小さなクラスであっても、すべてを 1 つのクラスに含めると、役割が混在して設計が複雑になります。データと入力が密接に結びついて、それら両方にロジックを適用しなければならなくなります。KISS の原則に反し、シンプルな機能を複数盛り込むことで、それらが絡み合ってしまうのです。

これを回避するには、Paddle クラスを複数の小さなクラスに分け、それぞれに 1 つの役割を持たせるようにしましょう。データを専用の PaddleData クラスに分離するか、ScriptableObject を使用します。その後、他のすべての要素を、PaddleInput クラス、PaddleMovement クラス、および PaddleAudio クラスへとリファクタリングします。

PaddleLogic クラスは、PaddleInput からの入力を処理します。PaddleData からの速度情報を適用しながら、PaddleMovement を使ってラケットを移動させます。最後に、PaddleLogic は、ボールがラケットに当たったときに音を再生するよう、PaddleAudio に通知します。



Paddle クラスを単一責任へとリファクタリングする

この再設計によって、各クラスがそれぞれ 1 つだけの役割を持つようになり、理解しやすく小分けされたパーツになってくれます。コードを読むために、複数の画面をスクロールする必要がなくなります。

いまだ Paddle スクリプトが必要となりますが、その仕事は 1 つだけで、こういった他のクラス同士を結びつけるだけになります。機能の大部分は、他のクラスに分割されています。

なお、クリーンコードとは、必ずしもコードを極力コンパクトにすることではありません。短いクラスを使用するようにしたとしても、リファクタリングによって全体の行数が増えることはあります。ただし、個々のクラスの可読性は高まります。デバッグを行ったり新機能を追加したりした場合でも、このように構造が簡素化されていれば、すべての機能を元の場所のまま維持することができます。

リファクタリングの例

シンプルなプロジェクトのリファクタリングについてさらに詳しく知りたい場合は、「プロジェクトの規模が拡大するのに合わせてコードを設計する方法」を参照してください。この記事では、単一責任の原則を使って、大きな Monobehaviour を小さなパーツへと分解する方法を説明しています。

また、Unite Berlin で Mikael Kalms 氏が行ったオリジナルプレゼンテーション『From *Pong* to 15-person project（ピンポンから 15 名規模のプロジェクトまで）』を視聴することもできます。

© 2022 Unity Technologies

5 メソッド

“ルーチンを読んで、それらが思ったとおりの内容であったとき、あなたはクリーンコードに取り組んでいる成果を実感するでしょう。”

– Ward Cunningham（Wiki の発明者であり、eXtreme Programming の共同創始者）

クラスと同様にメソッドも、役割を 1 つに絞って小さくする必要があります。各メソッドでは、1 つのアクションを記述するか、1 つの質問に答えるようにしましょう。両方を兼ねることは避けてください。

メソッドには、そのメソッドが何をするのかがわかる名前を付けましょう。たとえば、`GetDistanceToTarget` などとすれば、その目的がはっきりとわかります。

カスタムクラスのメソッドを作成する際には、以下の推奨事項に従うようにしてください。

- **引数はなるべく少なくする：**引数を増やすと、メソッドの複雑さが増します。引数の数を減らせば、メソッドが読みやすくなり、テストも行いやすくなります。
- **過度なオーバーロードを避ける：**メソッドのオーバーロードについては、置換が無限に生成される可能性があります。メソッドの呼び出し方法を反映するものをいくつか選び、それらを実装するようにしてください。メソッドをオーバーロードする場合は、各メソッドシグネチャの引数の数がそれぞれ違うことを確認して、混乱を防ぐようにしてください。
- **副作用を回避する：**メソッドでは、その名前が表していることだけを実行する必要があります。スコープ外のことを変更するのは避けてください。可能な場合は、参照ではなく値で引数を渡してください。`out` または `ref` キーワードを使って結果を返す場合は、それがそのメソッドでの目的どおりの動作であることを確認してください。

  特定のタスクにおいて副作用は有用ですが、意図しない結果を招く可能性もあります。副作用のないメソッドを記述して、予期しない動作を減らすようにしましょう。
- **フラグを渡す代わりに、別のメソッドを作成する：**フラグに基づいて 2 つの異なるモードで動作するようにメソッドを設定することは避けてください。異なる名前で 2 つのメソッドを作成するようにしましょう。たとえば、フラグ設定に基づいて角度やラジアンを返す `GetAngle` メソッドを作成することは避けてください。代わりに、`GetAngleInDegrees` と `GetAngleInRadians` の各メソッドを作成してください。

引数としての Boolean フラグは無害に見えますが、実装が複雑になったり、単一責任が破られたりする可能性があります。

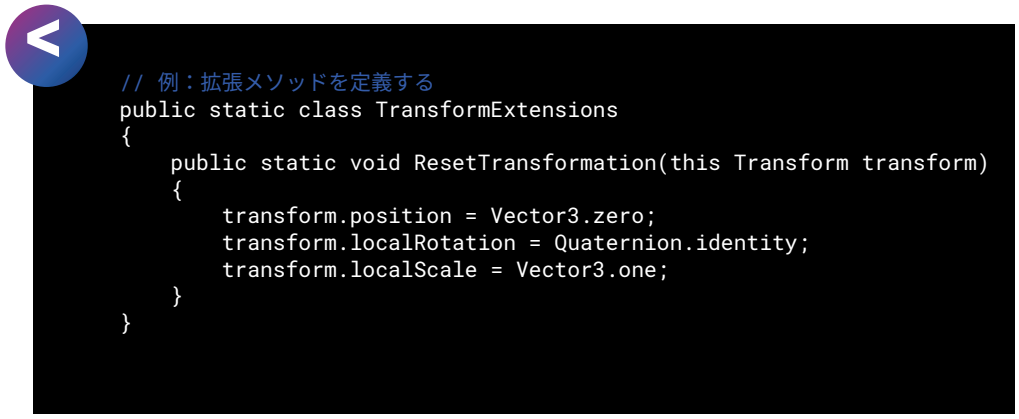
拡張メソッド

拡張メソッドを使うと、それ以外の場合には封印される可能性があるクラスに、機能を追加することができます。これは、UnityEngine API を拡張するクリーンな方法としも利用できます。

拡張メソッドを作成するには、静的メソッドを作成し、最初の引数の前に this キーワードを置きます。これが、拡張したい型になります。

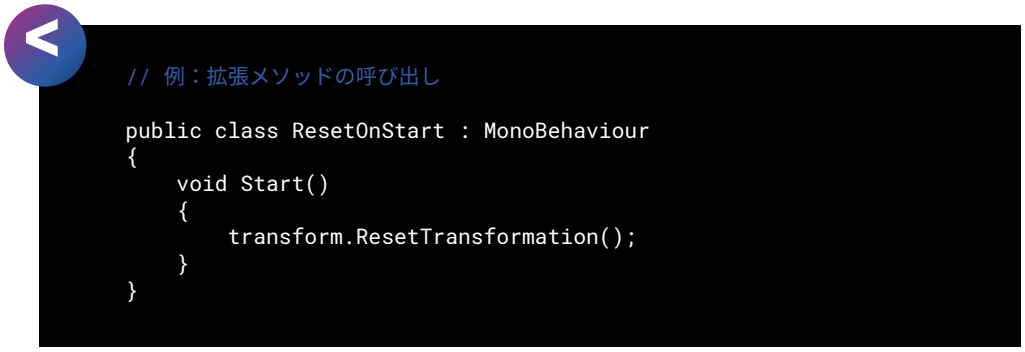
たとえば、ResetTransformation というメソッドを作成して、スケーリング、回転、または変換をゲームオブジェクトから削除したいとします。

静的メソッドは、this キーワードを使って最初の引数の Transform を渡すことで作成できます。



```
// 例：拡張メソッドを定義する
public static class TransformExtensions
{
    public static void ResetTransformation(this Transform transform)
    {
        transform.position = Vector3.zero;
        transform.localRotation = Quaternion.identity;
        transform.localScale = Vector3.one;
    }
}
```

その後、それを使用したいときには、ResetTransformation メソッドを呼び出します。ResetOnStart クラスは、Start 時に現在の Transform でこれを呼び出します。



```
// 例：拡張メソッドの呼び出し

public class ResetOnStart : MonoBehaviour
{
    void Start()
    {
        transform.ResetTransformation();
    }
}
```

コードを整理するために、拡張メソッドは静的クラス内で定義してください。たとえば、Transform を拡張するメソッドの場合は TransformExtensions というクラスを、Vector3s を拡張する場合は Vector3Extensions というクラスを作成します。

拡張メソッドを使えば、Monobehaviour をさらに作成することなく、多くの有用なユーティリティを構築できます。自分のゲーム開発手法にこれらを加えたい場合は、「Unity Learn: Extension Methods」を参照してください。

© 2022 Unity Technologies

DRY 原則：Don’t repeat yourself（繰り返しを避ける）

（《村上雅章》訳）『《達人プログラマー 熟達に向けたあなたの旅（"The Pragmatic Programmer"》） で、Andy Hunt 氏と Dave Thomas 氏は DRY 原則（「don’t repeat yourself」）を考案しました。ソフトウェアエンジニアリングの世界で頻繁に言及されるこの原則は「プログラマーよ。ロジックの重複や反復は避けよ」というものです。

そうしていればバグ修正やメンテナンスのコストを削減することができます。単一責任の原則に従っている場合、クラスやメソッドを変更する際に、関連のないコードを変更する必要は生じないはずです。DRY プログラムで論理的なバグをなくせば、これをあらゆる場所で阻止することができます。

DRY の反対は WET です（「we enjoy typing（タイピングは楽しい）」または「write everything twice（すべて 2 回ずつ記述する）」の略）。コードに不必要な繰り返しがある場合、そのプログラミングは WET です。

たとえば、2 つの ParticleSystem（explosionA と explosionB）と、2 つの AudioClip（soundA と soundB）があるとします。各 ParticleSystem はそれぞれのサウンドで再生される必要があるのですが、これは以下のようなシンプルなメソッドで実現できます。

```
// 例：WRITE EVERYTHING TWICE

    private void PlayExplosionA(Vector3 hitPosition)
    {
        explosionA.transform.position = hitPosition;
        explosionA.Stop();
        explosionA.Play();

        AudioSource.PlayClipAtPoint(soundA, hitPosition);
    }

    private void PlayExplosionB(Vector3 hitPosition)
    {
        explosionB.transform.position = hitPosition;
        explosionB.Stop();
        explosionB.Play();

        AudioSource.PlayClipAtPoint(soundB, hitPosition);
    }
```

この例では、各メソッドが Vector3 の位置を取得し、ParticleSystem を再生用の位置に移動します。まず、（パーティクルがすでに再生されている場合もあるため）パーティクルを停止し、シミュレーションを再生します。その後、AudioSource の静的 PlayClipAtPoint メソッドが、同じ場所にサウンドエフェクトを作成します。

一方のメソッドは、もう一方のメソッドを切り取って貼り付けたバージョンで、テキストを少し置き換えたものです。これでも機能はしますが、爆発を作成するたび新規メソッドを作らなければなりません。しかも、重複したロジックまで含んだまま…。

© 2022 Unity Technologies

代わりに、以下のような 1 つの PlayFXWithSound メソッドへとリファクタリングしましょう。

```
// 例：リファクタリング後

   private void PlayFXWithSound(ParticleSystem particle,
AudioClip clip, Vector3 hitPosition)
   {
       particle.transform.position = hitPosition;
       particle.Stop();
       particle.Play();

       AudioSource.PlayClipAtPoint(clip, hitPosition);
   }
```

ParticleSystem と AudioClip を追加すれば、これと同じメソッドを使ってそれらを同時に再生できます。

なお、DRY 原則に違反していなくても、コードが重複することはありえます。ロジックを重複させないということが、肝要です。

この例では、PlayFXWithSound メソッドにコア機能を抽出しました。ロジックを調整する必要が生じた場合は、`PlayExplosionA` と `PlayExplosionB` の両方ではなく、1 つのメソッドで変更を済ませることができます。

© 2022 Unity Technologies

6 コメント

“コードはユーモアと似ています。説明が必要な場合、それは上質とは言えません。」

– Cory House（ソフトウェアアーキテクト、執筆者）

コメントを適切に配置すると、コードの読みやすさが向上します。しかし、過剰なコメントや無駄なコメントがあると、逆の効果をもたらすことがあります。あらゆる要素と同様に、コメントもバランスをとるように努めてください。

KISS の原則に従い、コードをわかりやすい論理パーツに分割すれば、ほとんどの場合、コードにコメントは必要ありません。変数や関数は、適切な名前を付ければ、別途説明は不要です。

要素の目的を説明するのではなく、足りない情報を補ったり、理由を説明したりするのが、有用なコメントだと言えます。何かを決定するための根拠がすぐには見つからない場合や、ややこしいロジックについて明確に説明する必要がある場合には、有用なコメントを添えることで、コード自体からは得られない情報を示すことができます。

以下に示すのは、コメントに関するいくつかの注意点です。

— **わかりにくいコードを説明するためにコメントを追加することは避ける：**複雑に絡み合ったロジックについて説明が必要な場合は、コメントを追加するのではなく、わかりやすくなるようにコードを再構成してください。そうすれば、コメントは必要なくなります。

— **クラス、変数、メソッドに適切な名前を付ければ、コメントの代わりになる：**コードを見ただけで内容が理解できるようになっていれば、説明は余計なので、コメントは控えましょう。

```
// 悪い例：邪魔で冗長なコメント

// 撃つ標的
Transform targetToShoot;
```

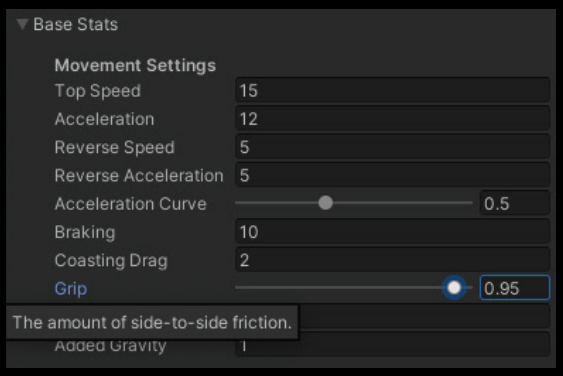
— **可能な場合は、コード行の末尾ではなく、別の行にコメントを配置する：**ほとんどの場合は、わかりやすくするために、各コメントを専用の行に配置しましょう。

— **ほとんどの場合は、ダブルスラッシュ（//）のコメントタグを使用する：**先頭にたくさんの行を配置することはせず、説明するコードの近くにコメントを配置しましょう。近くに配置すれば、説明とロジックの結びつきが読む人にとってわかりやすくなります。

© 2022 Unity Technologies

— **シリアル化されたフィールドには、コメントの代わりにツールチップを使用する：**インスペクター内のフィールドに説明が必要な場合は、ツールチップ属性を追加し、個別のコメントは省きましょう。ツールチップで 2 つの役割を果たせます。

```
// 例：コメントの代わりにツールチップ（ツールヒント）を使う

[Tooltip( “The amount of side-to-side friction.” )]
public float Grip;
```



インスペクターのツールのテクニック。

— **パブリックのメソッドや関数の前では、サマリー XML タグを使うこともできる：**Visual Studio では、たくさんの一般的な XML 形式コメントに対応した、IntelliSense を利用できます。

```
// 例：
// これはよくあるコメントです。
// 意図、論理フロー、およびアプローチを示すために使用します。

// サマリー XML タグも使用できます。
//
/// <summary>
/// 武器を発射する
/// </summary>
public void Fire()
{
    ...
}
```

© 2022 Unity Technologies

- **コメントデリミター（//）とコメントテキストの間にはスペースを 1 つ挿入する。**

- **法的免責事項を追加する：**ライセンス情報や著作権情報には、コメントを使用するのが適切です。ただし、訴訟事件への適用書面（Legal brief）全体をコード内に挿入することは避けてください。代わりに、対象の法的な事由がすべて記載されている外部ページへのリンクを記載しましょう。

- **コメントのスタイルを決める：**コメントの外観を統一しましょう（たとえば、各コメントを大文字で開始し、ピリオドで終わるなど）。チームで決定したことをスタイルガイドに盛り込み、それに従うようにしてください。

- **コメントの周りにアスタリスクや特殊文字のフォーマット済みブロックを作成することは避ける：**これを挿入すると、可読性が低下し、コードが散らかる原因になります。

- **コメントアウトされたコードを削除する：**テスト時や開発時にステートメントをコメントアウトをするのは普通のことですが、コメントアウトされたコードをそのまま放置することはしないでください。ソース管理を信頼し、思い切ってその 2 行のコードを削除してください。

- **TODO コメントを最新に維持する：**タスクを完了したら、リマインダーとして残した TODO コメントを忘れずに削除してください。古いコメントは邪魔になります。

  TODO に名前と日付を追加すると、説明や状況把握がしやすくなります。

  実利から考える癖をつけましょう。5 年前の TODO をコード内に残していても、それを参照することはもうないでしょう。YAGNI 原則を思い出してください。TODO コメントは、有効性がなくなるまでに削除しましょう。

- **日記として使わない：**コメントは、開発日記のための場所ではありません。新しいクラスを開始する際に、今やっていることをすべてコメントに記録する必要はありません。ソース管理を適切に使用していれば、それは必要ないのです。

- **アトリビューションを避ける：**ソース管理を使用している場合は特に、バイライン（たとえば、// added by devA や devB など）を追加する必要はありません。

© 2022 Unity Technologies

7 よくある落とし穴

“デバッグがソフトウェアのバグを除去するプロセスであると言うならば、プログラミングは言わばそれらを組み込むプロセスと言えるでしょう。」

— Edsger W. Dijkstra（コンピューターサイエンスのパイオニア）

クリーンコードは偶然に得られるものではありません。チーム作業の視点に立った考え方やコーディングを心がける個人が、意図的に作っていくものです。

もちろん、すべてが計画どおりにいくわけではありません。どんなに努力しても、クリーンでないコードは必然的に発生します。そこで、それを探す必要性が出てくるのです。

コードの臭いは、プロジェクト内に厄介なコードが潜んでいる可能性があることを示す兆候です。以下の兆候は必ずしも根本的な問題を示すものではありませんが、見つかった場合は調査する価値があります。

- **不可解な名前：**謎に満ちたミステリーは多くの人に好まれますが、コーディング標準に関しては、それは当てはまりません。クラス、メソッド、および変数には、無駄を排した、わかりやすい名前を付ける必要があります。
- **不要な複雑さ：**オーバーエンジニアリングは、クラスに対するあらゆるニーズを想定しようとした場合に発生します。これは、多くのことをやりすぎようとする長いメソッドや大きなクラスを含んだ、ゴッドオブジェクトとして現れることがあります。大規模なクラスは小規模な専門パーツに分割し、それぞれに独自の役割を持たせるようにしましょう。
- **柔軟性の欠如：**小さな変更の場合、通常は他の場所で複数の変更を行う必要はありません。ただし、そのような複数の変更が発生した場合は、単一責任の原則に違反していないかどうかを再確認してください。

  複数の役割を与えると、すべてのケースを予測することが難しくなるので、不具合が発生しやすくなります。役割が 1 つだけのメソッドを更新した場合、更新後のロジックが正常に機能していれば、他のコードもその後正常に機能し続けると考えることができます。
- **脆弱性：**小さな変更を加えた後にすべてが機能しなくなった場合は、多くの場合、問題があると考えられます。
- **固定性：**異なるコンテキストでも再利用できるコードを作成することは、よくあることです。ただし、他の場所に展開するために多くの依存関係が必要になる場合は、ロジックの動作を分離しましょう。
- **重複するコード：**コードを切り取って貼り付けたことが目立つ場合は、リファクタリングをしましょう。コアロジックを専用の関数に抽出し、他の関数からそれを呼び出すようにしましょう。コードをコピーして貼り付けると、変更があるたびに複数の場所でロジックを更新する必要が生じるため、メンテナンスが困難になります。
- **過剰なコメント：**コメントは、直感的にわかりにくいコードを説明するのに役立ちます。ただし、開発者がそれらを使い過ぎる場合もあります。すべての変数やステートメントにコメントを付けるのは不必要です。先にも述べたように、最良のコメントとは、メソッドやクラスに適切な名前を付けることです。また、ロジックを小さいパーツに分割すれば、コードスニペットが短くなり、説明も少なくて済むようになります。

© 2022 Unity Technologies

8 まとめ

“プログラミングはゼロサムゲームではありません。同僚のプログラマーに何かを教えても、その知識があなたから奪われるわけではないのです。」

— John Carmack（id Software 共同設立者）

クリーンコーディングの原則に関する本ガイドを楽しんでいただけたでしょうか。

ここで紹介したテクニックは、具体的なルールセットというよりも、一連の習慣というべきものです。そしてそれらは、すべての習慣と同様、日常的な実践を通じて自ら発見していく必要があります。

本ガイドで前述したように、Unity 開発者向けのこちらの C# スタイルシートをコピーして、ご自身のガイドの開始点として使用してください。

コードは、モジュール型の小さなパーツに分割することで、拡張性の高いコードにすることができます。開発というマラソンを続けていく間には、コードの書き直しに何度も遭遇することでしょう。その生産活動は、要件がたびたび変わっていく困難なプロセスになる可能性もあります。しかし幸いなことに、一人で取り組む必要はありません。

グループでコーディングを行えば、ゲーム開発はソロレースというよりも、リレーに近いものになります。チームメイトと作業を分担し、コースを区間に分けて担当することができます。

担当業務に専念し、バトンを渡しながら仕事を進めていけば、皆で一緒にゴールを迎えることができるでしょう。

コードをクリーンアップする方法についてサポートを受けたい場合は、Unity のプロフェッショナルサービスチーム、Accelerate Solutions までお問い合わせください。このチームは、Unity の最上級ソフトウェア開発者で構成されています。Accelerate Solutions は、パフォーマンスの最適化、開発の迅速化、ゲームプランニング、イノベーションなどを専門とし、あらゆる規模のゲームスタジオに向けてカスタムコンサルティングや開発ソリューションを提供しています。

Accelerate Solutions が提供するサービスの 1 つとして、CAP（コード、アセット、パフォーマンス）があります。この 2 週間のコンサルティングエンゲージメントでは、まずお客様のコードとアセットを 3 日間にわたって詳細に調査し、パフォーマンスの問題の根本原因を明らかにします。これには、ベストプラクティスの推奨事項を含む実用的で詳細なレポートが含まれます。Unity Accelerate Solutions が提供するこのサービスや、その他のサービスの詳細については、Unity の担当者までお問い合わせください。

参照情報

このガイドは、コンピューティングで使用される一連のベストプラクティスを短くまとめたものです。詳細については、このドキュメントの全体的なスタイルガイドとなっている、Microsoft Framework Design Guidelines を参照してください。

また、これまでに書かれたクリーンコード関連の幅広い書籍からも、詳細を学ぶことができます。皆さんの理解を深めていただくために、私たちのお勧めの書籍をいくつか紹介します。

Clean Code: A Handbook of Agile Software Craftsmanship.Robert C. Martin, 2008. Prentice Hall.ISBN 978-0132350884.

The Pragmatic Programmer, 20th Anniversary Edition.David Thomas and Andrew Hunt, 2019, Addison Wesley, ISBN 978-0135957059.

9 付録：スクリプトテンプレート

“説明はいいから、
コードを見せなさい。」

— Linus Torvalds（Linux および Git の生みの親）

スタイルガイドのフォーマットルールを設定したら、次はスクリプトテンプレートを設定しましょう。これらのテンプレートは、C# スクリプト、シェーダー、マテリアルなどのスクリプトアセットのための、空白の開始用ファイルを生成します。

Unity の事前設定済みのスクリプトテンプレートは、以下の場所にあります。

Windows: C:\Program Files\Unity\Editor\Data\Resources\ScriptTemplates

Mac: /Applications/Unity/Unity.app/Contents/Resources/ScriptTemplates

macOS で、Unity.app パッケージの内容を表示して、Resources サブディレクトリを表示します。

このパス内には、デフォルトのテンプレートが表示されます。

81-C# Script-NewBehaviourScript.cs.txt

82-Javascript-NewBehaviourScript.js.txt

83-Shader__Standard Surface Shader-NewSurfaceShader.shader.txt

84-Shader__Unlit Shader-NewUnlitShader.shader.txt

「Create」メニューから「Project」ウィンドウで新しいスクリプトアセットを作成した場合、Unity ではこれらのテンプレートのいずれかが使用されます。

81-C# Script-NewBehaviourScript.cs.txt という名前のファイルをテキストエディターで開いた場合は、以下のような画面が表示されます。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class #SCRIPTNAME# : MonoBehaviour
{
    // 最初のフレームが更新される前に Start を呼び出します
    void Start()
    {
        #NOTRIM#
    }

    // 1 フレームに 1 回 Update が呼び出される
    void Update()
    {
        #NOTRIM#
    }
}
```

© 2022 Unity Technologies

以下のキーワードをメモします。

— **#SCRIPTNAME#：**これは、あなたがスクリプト用に指定した名前です。名前をカスタマイズしなかった場合は、デフォルトの名前が使用されます（NewBehaviourScript など）。

— **#NOTRIM#：**これによりスペースが確保され、波括弧の間に行が 1 つ表示されます。

スクリプトテンプレートはカスタマイズ可能です。たとえば、名前空間を追加したり、デフォルトの Update メソッドを削除したりすることができます。テンプレートに変更を加えることで、これらのいずれかのスクリプトアセットを作成するたびに、キーストロークを数回減らすことができます。

スクリプトテンプレートのファイル名は、以下のパターンに従っています。

PriorityNumber–MenuPath–DefaultName.FileExtension.txt

ダッシュ（-）文字は、名前の各部分を区切っています。

— **PriorityNumber** は、「Create」メニューでのスクリプトの表示順序です。数値が小さいほど、優先順位が高くなります。

— **MenuPath** を使用すると、「Create」メニューでのファイルの表示方法をカスタマイズできます。ダブルアンダースコア（__）でカテゴリを作成できます。

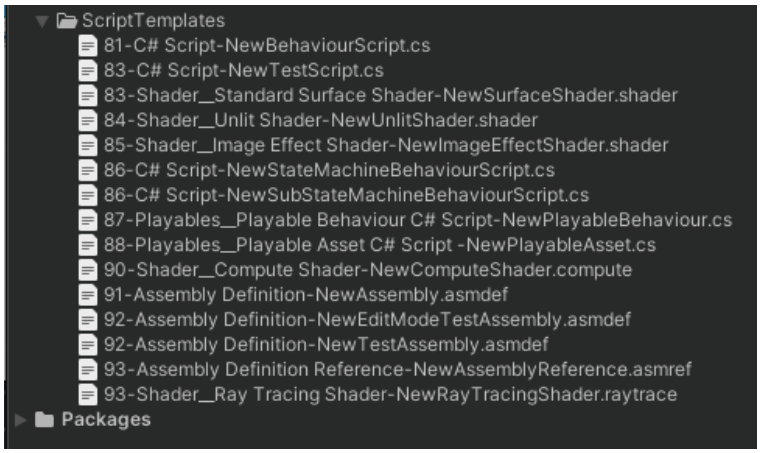
— たとえば、「CustomScript__Misc__ScriptableObject」とすると、**「Create」>「CustomScript」>「Misc」**メニューに、メニュー項目「ScriptableObject」が作成されます。

— **DefaultName** は、名前を指定しなかった場合にアセットに付けられるデフォルトの名前です。

— **FileExtension** は、アセット名に付加されるファイル拡張子です。

なお、各スクリプトテンプレートでは、FileExtension の後に .txt も付加されていることに注意してください。

スクリプトテンプレートを特定の Unity プロジェクトに適用したい場合は、ScriptTemplates フォルダー全体をプロジェクトの「Assets」の直下にコピーして貼り付けます。



Unity プロジェクトにコピーされた ScriptTemplates。

© 2022 Unity Technologies

次に、新しいスクリプトテンプレートを作成するか、希望の設定に合わせて元のスクリプトを変更します。スクリプトテンプレートを変更しない場合は、プロジェクトからすべてのスクリプトテンプレートを削除します。

たとえば、ScriptableObject 用の空白のスクリプトテンプレートを作成する場合は、ScriptTemplates フォルダーの下に、次の名前の新しいテキストファイルを作成します。

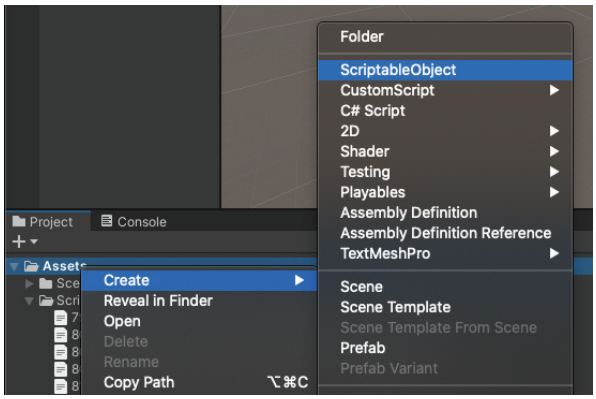
```
80-ScriptableObject-NewScriptableObject.cs.txt
```

テキストを以下のように編集します。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(fileName = “#SCRIPTNAME#” , menuName =
“#SCRIPTNAME#” )]
public class #SCRIPTNAME# : ScriptableObject
{
    #NOTRIM#
}
```

これにより、CreateAssetMenu 属性を含んだ、空の ScriptableObject スクリプトが作成されます。

スクリプトテンプレートを保存したら、エディターを再起動します。次回起動すると、「Create」メニューに追加のオプションが表示されます。



カスタムスクリプトテンプレートによって、「Create」メニューに新しいメニュー項目が追加されます。

「Create」メニューから、新しい ScriptableObject スクリプト（および対応する ScriptableObject アセット）を作成します。

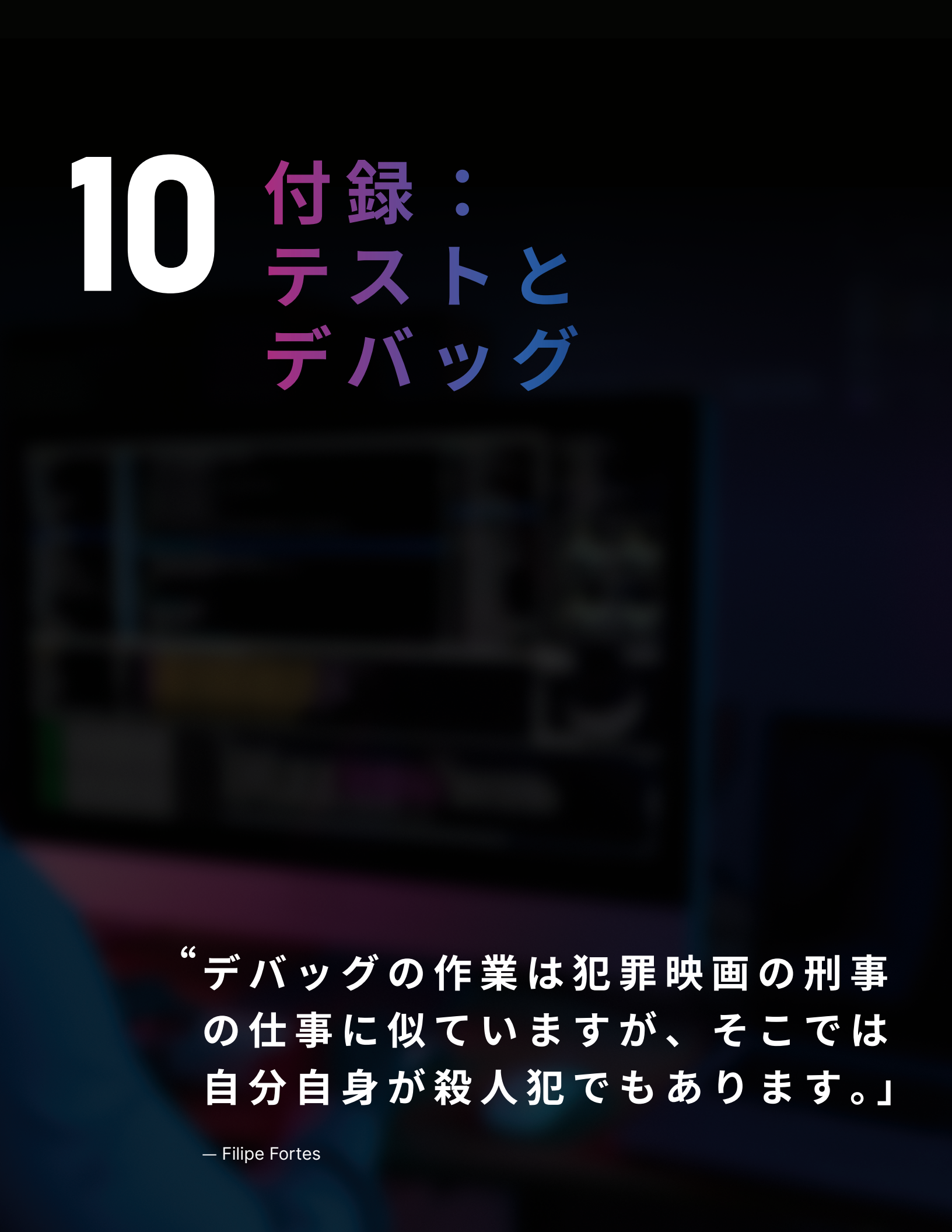
カスタマイズされたスクリプトテンプレートと元のテンプレートの両方をバックアップしてください。変更されたテンプレートを Unity が認識できなかった場合、ファイルを復元する必要があります。

お好みのスクリプトテンプレートのセットができたら、ScriptTemplates フォルダーを新しいプロジェクトにコピーし、自分のニーズに合わせてそれらをカスタマイズします。アプリケーションリソースにある元のスクリプトテンプレートを変更することもできますが、注意してください。これは、そのバージョンの Unity を使用するすべてのプロジェクトに影響します。

スクリプトテンプレートのカスタマイズについて詳しくは、こちらのサポート記事を参照してください。また、いくつかの追加的なスクリプトテンプレートの例については、添付のプロジェクトを確認してください。

© 2022 Unity Technologies

10 付録：テストとデバッグ



“デバッグの作業は犯罪映画の刑事の仕事に似ていますが、そこでは自分自身が殺人犯でもあります。」

— Filipe Fortes

自動テストは、コードの品質を高めるとともに、バグ修正にかかる時間を短縮することができる、効果的なツールです。テスト駆動開発（TDD：Test-Driven Development）は、ソフトウェアを開発しながら単体テストを作成する開発方法論です。実際、開発者は特定の機能を作成する前に、各テストケースを定期的に作成します。

ソフトウェア開発を進めながら、自動化されたプロセスのテストスイート全体に対して、テストを繰り返し実行していくことになります。これは、まずソフトウェアを作成し、後でテストケースを作成するのとはまったく対照的です。TDD では、コーディング、テスト、およびリファクタリングが織り混ぜられるように行われます。

以下に示すのは、Kent Beck 氏の『*《テスト駆動開発（"Test-Driven Development by Example"》*』で紹介されている基本的な考え方です。

1. **単体テストを 1 つ追加する**：これによって、アプリケーションに追加する 1 つの新機能を記述します。何を実行する必要があるかは、チームまたはユーザーベースから特定します。
2. **テストを実行する**：まだ新機能をプログラムに実装していないので、テストは失敗します。さらに、これにより、テスト自体が有効かどうかが検証されます。常にデフォルトで合格するわけではありません。
3. **新しいテストに合格できる、最もシンプルなコードを記述する**：新しい単体テストに合格するための必要十分なロジックを記述します。この時点では、クリーンコードである必要はありません。単体テストに合格さえできれば、洗練されてない構造や、ハードコードされたマジックナンバーなどを使用してもかまいません。
4. **すべてのテストに合格することを確認する**：完全な自動テストスイートを実行します。過去の単体テストにはすべて合格するはずです。新しいコードは、新しいテスト要件と古い要件の両方を満たしていることになります。

   そうでない場合は、すべてのテストに合格するまで、新しいコード（だけ）を変更していきます。
5. **リファクタリングする**：戻って新しいコードをクリーンアップします。スタイルガイドを使って、すべて適合していることを確認してください。

   コードを移動させて、論理的に整理します。類似するクラスやメソッドを並べて配置するなどしてください。重複するコードがあれば削除します。また、識別子の名前を変更して、コメントの必要性を最小限に減らしてください。長すぎるメソッドやクラスは分割します。

   リファクタリングするごとに、自動テストスイートを実行します。
6. **繰り返す**：新機能を追加するたびに、このプロセスを実行します。各ステップは、小さな段階的変化となります。ソース管理の下で頻繁にコミットしてください。デバッグ時には、単体テストごとに少量の新規コードを調べるだけで済みます。これにより、作業の範囲がシンプルになります。他のすべてが失敗した場合は、前のコミットにロールバックして再び開始します。

以上が要点です。この方法を使ってソフトウェアを開発すれば、KISS の原則を必然的に守るようになっていきます。機能を一度に 1 つずつ追加し、その都度テストしながら、テストごとに継続的にリファクタリングをしていくので、コードをクリーニングすることが常に意識されるようになります。

クリーンコードの原則の大部分と同様に、TDD では短期間でたくさんの作業が発生しますが、多くの場合は長期的なメンテナンス効率と可読性が向上していきます。

© 2022 Unity Technologies

Unity Test Framework

Unity Test Framework（UTF）（旧称 Unity Test Runner）は、Unity 開発者に標準のテストフレームワークを提供します。UTF では、.NET 言語向けのオープンソーステストライブラリである NUnit が使用されます。

Unity Test Framework では、エディター（**編集モード**または**再生モード**を使用）およびターゲットプラットフォーム（スタンドアロン、Android、iOS など）で単体テストを実行できます。UTF は Package Manager を通じてインストールします。使用を始める際には、オンラインドキュメントが役立ちます。

Unity Test Framework の一般的なワークフローは以下のとおりです。

- **テストアセンブリと呼ばれる新しいテストスイートを作成する**：Test Runner UI を使用すると、このプロセスが簡素化され、プロジェクトにフォルダーが作成されます。
- **テストを作成する**：Test Runner UI を使用すると、作成する C# スクリプトを単体テストとして管理するのに便利です。Test Assembly フォルダーを選択し、**「Assets」>「Create」>「Testing」>「C# Test Script」**に移動します。このスクリプトを編集して、テストのロジックを追加します。
- **テストを実行する：**Test Runner UI を使用して、すべての単体テストを実行するか、選択した単体テストを実行します。JetBrains Rider を使用して、スクリプトエディターから直接 UTF を実行することもできます。
- **エディターで再生モードのテストを追加するか、スタンドアロンとして追加する**：デフォルトのテストアセンブリは編集モードで動作します。単体テストを実行時に動作させたい場合は、再生モードで別のアセンブリを作成します。これをスタンドアロンビルド用にも設定します（テストの結果がエディターに表示された状態で）。

Test Framework で、エディター内にスタンドアロンビルドの結果が表示されます。

UTF の使用を開始するための詳細情報については、Test Framework のマイクロサイトを参照してください。

© 2022 Unity Technologies


Unity®