There's a lot of teamwork and ingenuity that goes into making a game like Destiny. We have talented people across all disciplines working together to make the best game that we can. However, achieving the level of coordination needed to make Destiny isn’t easy.

При создании такой игры как Destiny, требуется много командой работы и изобретательности. В нашей команде имеются люди, талантливые в разных областях, работающие вместе для создания лучшей игры. Однако, достижение такой скоординированности для создания - непростая задача.

It's like giving a bunch of people paintbrushes but only one canvas to share between them and expecting a high-quality portrait at the end. In order to make something that isn't pure chaos, some ground rules need to be agreed upon. Like deciding on the color palette, what size brushes to use in what situations, or what the heck you’re trying to paint in the first place. Getting that alignment amongst a team is incredibly important.

Это схоже с тем, если выдать куче людей по кисточке и ожидать от них портрет высоко качества на одном общем холсте. Чтобы из этого не вышел полный хаос, необходимо обозначить общепринятые базовые правила. Например, использовать одну палитру цветов, какой размер кисточки использовать в зависимости от ситуации, и первую очередь, что вы пытаетесь нарисовать. Достижение этой слаженности в команде невероятно важно.

One of the ways that we achieve that alignment over in engineering land is through coding guidelines: rules that our engineers follow to help keep the codebase maintainable. Today, I'm going to share how we decide what guidelines we should have, and how they help address the challenges we face in a large studio.

Один из способов добиться слаженности среди разработчиков - руководства по написанию кода, которые диктуют правила для поддержки работоспособности исходного кода. Сейчас я расскажу Вам какие правила мы выбираем и как эти правила помогают нам справится с препятствиями в крупной компании.

The focus of this post will be on the game development side of things, using the C++ programming language, but even if you don't know C++ or aren't an engineer, I think you'll still find it interesting.

Основное внимание в этой статье будет уделено деталям разработки игр и использование языка программирования C++. Думаю если вы не знаете C++ или не являетесь программистом вам будет интересно.

**WHAT'S A CODING GUIDELINE?**

A coding guideline is a rule that our engineers follow while they're writing code. They're commonly used to mandate a particular format style, to ensure proper usage of a system, and to prevent common issues from occurring. A well-written guideline is clearly actionable in its wording, along the lines of "Do X" or "Don't do Y" and explains the rationale for its inclusion as a guideline. To demonstrate, here’s a couple examples from our C++ guidelines:

Что такое "руководство по написанию кода"?

Руководство по написанию кода это свод которым следуют наши разработчики во время написания кода. В основном эти правила используются для описания стилистики кода, рационального использования системных ресурсов и для предотвращения частых ошибок. Правильно написанное руководство должно сводиться к "Делай А" или "Не делай Б" и почему ты должен так делать. Для демонстрации вот пару примеров из нашего руководства:

##### DON'T USE THE STATIC KEYWORD DIRECTLY

*The "static" keyword performs a bunch of different jobs in C++, including declaring incredibly dangerous static function-local variables.   
  
You should use the more specific wrapper keywords in cseries\_declarations.h, such as static\_global, static\_local, etc. This allows us to audit dangerous static function-locals efficiently.*

##### BRACES ON THEIR OWN LINES

*Braces are always placed on a line by themselves.   
  
There is an exception permitted for single-line inline function definitions.*

НЕ ИСПОЛЬЗУЙ КЛЮЧЕВОЕ СЛОВО STATIC БЕЗРАСУДНО

Ключевое слово "static" исполняет множество задач в C++ в которые входит объявление особо опасных локальных переменных.

Необходимо использовать более утонченные ключевые слова описанные в библиотеке cseries\_declarations.h, такие как static\_global, static\_local и т.д. Это позволит нам эффективнее обратить внимание на опасные переменные.

ФИГУРНЫЕ СКОБКИ НА ИНДИВИДУАЛЬНЫХ СТРОКАХ

Фигурные скобки всегда должны быть единственным символом в строке. Однако есть исключение для функций определенных в одну строку.

Notice how there’s an exception called out in that second guideline? Guidelines are expected to be followed most of the time, but there's always room to go against one if it results in better code. The reasoning for that exception must be compelling though, such as producing objectively clearer code or sidestepping a particular system edge case that can't otherwise be worked around. If it’s a common occurrence, and the situation for it is well-defined, then we’ll add it as an official exception within the guideline.

Обратите внимание на исключение из второго правила. От руководства ожидается бесспорное исполнение правил, хотя всегда находится место для бреши если она ведет к улучшенному коду. Причина исключений должна быть довольно серьезной, такая как создание более чистого кода или упор в особенности системы что нельзя обойтись без них. Если это частое явление и ситуация четко обозначена - мы добавляем офицальное исключение в руководство.

Now, even if you have a lot of well-written, thought-out guidelines, how do you make sure people follow them? At Bungie, our primary tool for enforcing our guidelines is through code reviews. A code review is where you show your code change to fellow engineers, and they’ll provide feedback on it before you share it with the rest of the team. Kind of like how this post was reviewed by other people to spot grammar mistakes or funky sentences I’d written before it was shared with all of you. Code reviews are great for maintaining guideline compliance, spreading knowledge of a system, and giving reviewers/reviewees the opportunity to spot bugs before they happen, making them indispensable for the health of the codebase and team.

Теперь если у вас есть прописанное руководство, как заставить людей следовать ему? В Bungie основным инструментом является обзор кода. Обзор кода это когда вы обмениваетесь своим кодом с коллегами, которые укажут на недочеты прежде чем вы отправите код на общий доступ. Эта статья также был выдана на обзор другим людям на поиск грамматических ошибок или странных предложений перед тем как поделиться с вами. Обзоры кода это отлчиное решение для всеобщего согласия с руководством, деление знаниями, и возможность для обзорщиков найти баги делая их незаменимыми для работоспособности исходного кода и команды.

You can also have a tool check and potentially auto-fix your code for any easily identifiable guideline violations, usually for ones around formatting or proper usage of the programming language. We don't have this setup for our C++ codebase yet unfortunately, since we have some special markup that we use for type reflection and metadata annotation that the tool can't understand out-of-the-box, but we're working on it!

Вы также можете сделать инструмент для проверки и автоматически исправлять свой код для простых правил из руководства, например для стилистики кода или его правильности. К сожалению у нас пока нет такого инструмента из-за специальных разметок которые мы используем для отражения типов данных и метаданных. Но мы работаем над этим!

Ok, that pretty much sums up the mechanics of writing and working with guidelines. But we haven't covered the most important part yet: making sure that guidelines provide value to the team and codebase. So how do we go about figuring out what's valuable? Well, let's first look at some of the challenges that can make development difficult and then go from there.

Хорошо, это вполне дает понимание о механизме написания и работы с руководством. Но мы еще не затронули одну из важнейших деталей - руководство должно иметь какое-то влияние на команду и исходный код. Так как же нам понять что имеет ценность? Ну давайте посмотрим на некторые проблемы которые могут возникнуть и продолжим.

The first challenge is the programming language that we’re using for game development: C++. This is a powerful high-performance language that straddles the line between modern concepts and old school principles. It’s one of the most common choices for AAA game development to pack the most computations in the smallest amount of time. That performance is mainly achieved by giving developers more control over low-level resources that they need to manually manage. All of this (great) power means that engineers need to take (great) responsibility, to make sure resources are managed correctly and arcane parts of the language are handled appropriately.

Первая проблема это то что мы используем язык программирование С++ для создание игр. Это мощный язык который совмещает себе как старые так и новые решения. С++ один из самых популярных выборов при создании ААА игр который совершает огромное количество операций за наименьший промежуток времени. Такая мощность достигается возможностью разработчиков работать на более низком уровне системы. Вся эта огромная сила влечет за собой огромную ответственность для уверенности что ресурсы и тонкие части языка корректно использованы.

Our codebase is also fairly large now, at about 5.1 million lines of C++ code for the game solution. Some of that is freshly written code, like the code to support Cross Play in Destiny. Some of it is 20 years old, such as the code to check gamepad button presses. Some of it is platform-specific to support all the environments we ship on. And some of it is cruft that needs to be deleted. Changes to long-standing guidelines can introduce inconsistency between old and new code (unless we can pay the cost of global fixup), so we need to balance any guideline changes we want to make against the weight of the code that already exists.

Наш код уже довольно огромен, где-то около 5.1 миллиона строк С++ кода игры. Один код был написан недавно, например поддержка кросс-платформенной игры в Destiny. А другой уже 20-ти летней давности, напрмер проверка нажата ли кнопка на геймпаде. Другие расчитаны на определенную платформу и должны оставаться с нами, а другие вообще должны быть удалены. Изменения в руководстве могут создать несостыковки меджу поколениями кода ( если конечно мы не готовы переписывать все сначала ), поэтому мы пытаемся сбалансировать новые правила чтобы они были совместимыми с уже имеющимся кодом.

Finally, Bungie is a large company; much larger than a couple college students hacking away at games in a dorm room back in 1991. We're 150+ engineers strong at this point, with about 75 regularly working on the C++ game client. Each one is a smart, hardworking individual, with their own experiences and perspectives to share. That diversity is a major strength of ours, and we need to take full advantage of it by making sure code written by each person is accessible and clear to everyone else.   
  
Now that we know the challenges that we face, we can derive a set of principles to focus our guidelines on tackling them. At Bungie, we call those principles our C++ Coding Guideline Razors.

В итоге, Bungie огромная компания, всяко больше чем студентов взамывающие игры в общежитии в 1991. У нас уже более 150 разработчиков 75 из которых постоянно работают на C++. Каждый из них умен, трудолюбив, с неповторимым опытом и потенциалом. Наша основная сила в разнообразии и нужно брать от нее все в то же время держать код понятным для каждого.

Теперь когда мы знаем об основных проблемах и как с ними справлятся, мы можем взглянуть на явные случаи указанные в нашем руководстве. В Bungie мы зовем их Тонкости Руководства по C++.

## RAZORS? LIKE FOR SHAVING?

Well, yes. But no. The idea behind the term razor here is that you use them to "shave off" complexity and provide a sharp focus for your goals (addressing the challenges we went through above). Any guidelines that we author are expected to align with one or more of these razors, and ones that don't are either harmful or just not worth the mental overhead for the team to follow.   
  
I'll walk you through each of the razors that Bungie has arrived at and explain the rationale behind each one, along with a few example guidelines that support the razor.

ТОНКИЕ? КАК ЛЕЗВИЯ?

Ну да, но вообще нет. Идея термина "тонкость" в данном контексте говорит о бритве способной "сбрить" все сложности и предоставить четкое понимание того что ты делаешь ( к слову о вышеупомянутых проблемах ). В наших руководствах мы ожидаем от вас уложиться в эти тонкости и те у кого не получается либо неуважительны либо недостойны головной боли всей команды.

Я проведу вас по этим тонкостям через которые прошли в Bungie и объясню принципы каждый из них в примерах.

### #2 AVOID DISTINCTION WITHOUT DIFFERENCE

*When possible without loss of generality, reduce mental tax by proscribing redundant and arbitrary alternatives.*

This razor and the following razor go hand in hand; they both deal with our ability to spot differences. You can write a particular behavior in code multiple ways, and sometimes the difference between them is unimportant. When that happens, we'd rather remove the potential for that difference from the codebase so that readers don't need to recognize it. It costs brain power to map multiple things to the same concept, so by eliminating these unnecessary differences we can streamline the reader's ability to pick up code patterns and mentally process the code at a glance.

#2 ИЗБЕГАЙТЕ РАЗЛИЧИЙ БЕЗ РАЗЛИЧИЙ

Когда это возможно без потери общности, уменьшите ментальный налог, запретив избыточные и произвольные альтернативы.

Эта и следующая тонкости идут рука об руку; обе они имеют дело с нашей способностью замечать различия. Вы можете записать определенное поведение в коде несколькими способами, и иногда разница между ними несущественна. Когда это произойдет, мы бы предпочли исключить потенциальную возможность такого различия из кодовой базы, чтобы читателям не нужно было его распознавать. Сопоставление нескольких объектов с одной и той же концепцией требует больших мозговых усилий, поэтому, устраняя эти ненужные различия, мы можем упростить способность читателя улавливать шаблоны кода и мысленно обрабатывать код с первого взгляда.  
  
An infamous example of this is "tabs vs. spaces" for indentation. It doesn't really matter which you choose at the end of the day, but a choice needs to be made to avoid code with mixed formatting, which can quickly become unreadable.   
  
Some Bungie coding guidelines that support this razor are:

* Use American English spelling (ex "color" instead of "colour").
* Use post increment in general usage (
* Miscellaneous whitespace rules and high-level code organization within a file.

### Печально известным примером этого является "табуляция против пробелов" для отступов. На самом деле не имеет значения, что вы выберете в конце концов, но выбор необходимо сделать, чтобы избежать кода со смешанным форматированием, который может быстро стать нечитаемым.

### Вот некоторые рекомендации по кодированию bungie, которые поддерживают эту тонкость:

### • используйте американский английский (например, "color" вместо "colour").

### • используйте post increment в обычном использовании (index++ поверх ++index).

### • \* и & идут рядом с именем переменной вместо имени типа (int32 \*my\_pointer вместо int32\* my\_pointer).

### • различные правила использования пробелов и высокоуровневая организация кода внутри файла.

### #3 LEVERAGE VISUAL CONSISTENCY

*Use visually-distinct patterns to convey complexity and signpost hazards*

The opposite hand of the previous razor, where now we want differences that indicate an important concept to really stand out. This aids code readers while they're debugging to see things worth their consideration when identifying issues.   
  
Here's an example of when we want something to be really noticeable. In C++ we can use the preprocessor to remove sections of code from being compiled based on whether we're building an internal-only version of the game or not. We'll typically have a lot of debug utilities embedded in the game that are unnecessary when we ship, so those will be removed when we compile for retail. We want to make sure that code meant to be shipped doesn’t accidentally get marked as internal-only though, otherwise we could get bugs that only manifest in a retail environment. Those aren't very fun to deal with.

#3 ИСПОЛЬЗУЙТЕ ВИЗУАЛЬНУЮ СОГЛАСОВАННОСТЬ

Используйте визуально различимые узоры, чтобы передать сложность и обозначить опасности

Противоположная сторона предыдущей razor, где теперь мы хотим, чтобы различия, указывающие на важную концепцию, действительно выделялись. Это помогает читателям кода во время отладки видеть вещи, заслуживающие внимания при выявлении проблем.

Вот пример того, когда мы хотим, чтобы что-то было действительно заметным. В C++ мы можем использовать препроцессор для удаления фрагментов кода из компиляции в зависимости от того, создаем ли мы версию игры только для внутреннего использования или нет. Обычно в игру встроено множество утилит отладки, которые не нужны при отправке, поэтому они будут удалены при компиляции для розничной продажи. Мы хотим убедиться, что код, предназначенный для отправки, случайно не будет помечен как предназначенный только для внутреннего использования, в противном случае мы могли бы получить ошибки, которые проявляются только в розничной среде. С ними не очень-то весело иметь дело.  
  
We mitigate this by making the C++ preprocessor directives really obvious. We use all-uppercase names for our defined switches, and left align all our preprocessor commands to make them standout against the flow of the rest of the code. Here's some example code of how that looks:

Code

Some Bungie coding guidelines that support this razor are:

* Braces should always be on their own line, clearly denoting nested logic.
* Uppercase for preprocessor symbols (eg
* No space left of the assignment operator, to distinguish from comparisons (eg
* Leverage pointer operators (

### Мы смягчаем это, делая директивы препроцессора c++ действительно очевидными. Мы используем имена в верхнем регистре для наших определенных переключателей и выравниваем все наши команды препроцессора по левому краю, чтобы они выделялись на фоне остального кода.

### Вот некоторые рекомендации по кодированию bungie, которые поддерживают эту тонкость:

### • фигурные скобки всегда должны располагаться на отдельной строке, четко обозначая вложенную логику.

### • верхний регистр для символов препроцессора (например, #ifdef PLATFORM\_WIN64).

### • после оператора присваивания не осталось пробела, чтобы отличать его от сравнения (например, my\_number= 42 против my\_number == 42).

### • используйте операторы указателя (\*/&/->) для объявления косвенного обращения к памяти вместо ссылок

### #4 AVOID MISLEADING ABSTRACTIONS.

*When hiding complexity, signpost characteristics that are important for the customer to understand.*

We use abstractions all the time to reduce complexity when communicating concepts. Instead of saying, "I want a dish with two slices of bread on top of each other with some slices of ham and cheese between them", you're much more likely to say, "I want a ham and cheese sandwich". A sandwich is an abstraction for a common kind of food.  
  
Naturally we use abstractions extensively in code. Functions wrap a set of instructions with a name, parameters, and an output, to be easily reused in multiple places in the codebase. Operators allow us to perform work in a concise readable way. Classes will bundle data and functionality together into a modular unit. Abstractions are why we have programming languages today instead of creating applications using only raw machine opcodes.

#4 избегайте вводящих в заблуждение абстракций.

Скрывая сложность, указывайте характеристики указателей, которые важны для понимания клиента.

Мы постоянно используем абстракции, чтобы уменьшить сложность при передаче концепций. Вместо того чтобы сказать: "я хочу блюдо с двумя ломтиками хлеба, положенными друг на друга, и несколькими ломтиками ветчины и сыра между ними", вы, скорее всего, скажете: "я хочу бутерброд с ветчиной и сыром". Бутерброд - это абстракция для обычного вида пищи.

Естественно, мы широко используем абстракции в коде. Функции заключают в себе набор инструкций с именем, параметрами и выводом, которые можно легко повторно использовать в нескольких местах кодовой базы. Операторы позволяют нам выполнять работу в сжатой и удобочитаемой форме. Классы будут объединять данные и функциональность в единое модульное устройство. Абстракции - вот почему сегодня у нас есть языки программирования вместо того, чтобы создавать приложения, используя только необработанные машинные коды операций.

An abstraction can be misleading at times though. If you ask someone for a sandwich, there's a chance you could get a hot dog back or a quesadilla depending on how the person interprets what a sandwich is. Abstractions in code can similarly be abused leading to confusion. For example, operators on classes can be overridden and associated with any functionality, but do you think it'd be clear that

* Use standardized prefixes on variables and types for quick recognition.
  + eg:
  + eg:
* No operator overloading for non-standard functionality.
* Function names should have obvious implications.
  + eg:
  + eg:
  + eg:

### Однако абстракция иногда может вводить в заблуждение. Если вы попросите у кого-нибудь сэндвич, есть шанс, что вам вернут хот-дог или кесадилью, в зависимости от того, как человек интерпретирует, что такое сэндвич. Аналогичным образом можно злоупотреблять абстракциями в коде, что приводит к путанице. Например, операторы в классах могут быть переопределены и связаны с любой функциональностью, но, как вы думаете, было бы ясно, что m\_game\_simulation++ соответствует вызову функции обновления для каждого кадра в состоянии моделирования? Нет! Это запутанная абстракция, и вместо этого должно быть что-то вроде m\_game\_simulation.update(), чтобы ясно сказать, каково намерение.

### Цель этой тонкости состоит в том, чтобы избежать использования нетрадиционных абстракций, делая при этом те абстракции, которые у нас есть, понятными по своему назначению. Мы делаем это с помощью таких рекомендаций, как следующие:

### • используйте стандартизированные префиксы для переменных и типов для быстрого распознавания.

### • например: c\_ для типов классов, e\_ для перечислений.

### • например: m\_ для переменных-членов, k\_ для констант.

### • отсутствие перегрузки оператора из-за нестандартной функциональности.

### • имена функций должны иметь очевидный смысл.

### • например: get\_blank() должен иметь тривиальную стоимость.

### • например: try\_to\_get\_blank() может завершиться ошибкой, но сделает это изящно.

### • например: ожидается, что compute\_blank() или query\_blank() будут иметь нетривиальную стоимость.

### #5 FAVOR PATTERNS THAT MAKE CODE MORE ROBUST.

*It’s desirable to reduce the odds that a future change (or a conflicting change in another branch) introduces a non-obvious bug and to facilitate finding bugs, because we spend far more time extending and debugging than implementing.*

Just write perfectly logical code and then no bugs will happen. Easy right? Well... no, not really. A lot of the challenges we talked about earlier make it really likely for a bug to occur, and sometimes something just gets overlooked during development. Mistakes happen and that's ok. Thankfully there's a few ways that we can encourage code to be authored to reduce the chance that a bug will be introduced.

#5 отдавайте предпочтение шаблонам, которые делают код более надежным.

Желательно уменьшить вероятность того, что будущее изменение (или конфликтующее изменение в другой ветке) приведет к неочевидной ошибке, и облегчить поиск ошибок, потому что мы тратим гораздо больше времени на расширение и отладку, чем на реализацию.

Просто напишите идеально логичный код, и тогда никаких ошибок не произойдет. Легко, не так ли? Ну... Нет, не совсем. Многие проблемы, о которых мы говорили ранее, действительно повышают вероятность возникновения ошибки, и иногда что-то просто упускается из виду во время разработки. Ошибки случаются, и это нормально. К счастью, есть несколько способов, которыми мы можем поощрять создание кода, чтобы снизить вероятность появления ошибки.

One way is to increase the amount of state validation that happens at runtime, making sure that an engineer's assumptions about how a system behaves hold true. At Bungie, we like to use asserts to do that. An assert is a function that simply checks that a particular condition is true, and if it isn't then the game crashes in a controlled manner. That crash can be debugged immediately at an engineer’s workstation, or uploaded to our TicketTrack system with the assert description, function callstack, and the dump file for investigation later. Most asserts are also stripped out in the retail version of the game, since internal game usage and QA testing will have validated that the asserts aren't hit, meaning that the retail game will not need to pay the performance cost of that validation.

Один из способов - увеличить количество проверок состояния, которые выполняются во время выполнения, чтобы убедиться, что предположения инженера о том, как ведет себя система, верны. В Bungie нам нравится использовать утверждения для этого. Assert - это функция, которая просто проверяет, верно ли определенное условие, и если это не так, то игра выходит из строя контролируемым образом. Этот сбой может быть немедленно устранен на рабочей станции инженера или загружен в нашу систему tickettrack с описанием assert, стеком вызовов функций и файлом дампа для последующего расследования. Большинство утверждений также удаляются в розничной версии игры, поскольку внутреннее использование игры и тестирование качества подтвердят, что утверждения не были выполнены, а это означает, что розничной игре не нужно будет оплачивать затраты на производительность, связанные с этой проверкой.

Another way is to put in place practices that can reduce the potential wake a code change will have. For example, one of our C++ guidelines is to only allow a single

Другой способ - внедрить методы, которые могут уменьшить потенциальный эффект, который может вызвать изменение кода. Например, одно из наших рекомендаций по C++ заключается в том, чтобы разрешить существование в функции только одного оператора return. Опасность наличия нескольких операторов return заключается в том, что добавление новых операторов return к существующей функции потенциально может привести к пропуску требуемой части логики, которая была настроена ниже в функции. Это также означает, что будущим инженерам необходимо понимать все точки выхода функции, вместо того чтобы полагаться на вложенные условные обозначения с отступами для визуализации потока выполнения функции. Разрешая только один оператор return в нижней части функции, инженер вместо этого должен создать условие, чтобы показать разветвление логики внутри функции, и тогда с большей вероятностью рассмотрит код, заключенный в условное выражение, и влияние, которое оно окажет.

Some Bungie coding guidelines that support this razor are:

* Initialize variables at declaration time.
* Follow const correctness principles for class interfaces.
* Single
* Leverage asserts to validate state.
* Avoid native arrays and use our own containers.

Вот некоторые рекомендации по кодированию Bungie, которые поддерживают эту тонкость:

• Инициализируйте переменные во время объявления.

• Следуйте принципам корректности const для интерфейсов классов.

• Одиночный оператор return в нижней части функции.

• \* Используйте активы для проверки состояния.

• \* Избегайте собственных массивов и используйте свои собственные контейнеры.

### #6 CENTRALIZE LIFECYCLE MANAGEMENT.

*Distributing lifecycle management across systems with different policies makes it difficult to reason about correctness when composing systems and behaviors. Instead, leverage the shared toolbox and idioms and avoid managing your own lifecycle whenever possible.*

When this razor is talking about lifecycle management, the main thing it's talking about is the allocation of memory within the game. One of the double-edged swords of C++ is that the management of that memory is largely left up to the engineer. This means we can develop allocation and usage strategies that are most effective for us, but it also means that we take on all of the bug risk. Improper memory usage can lead to bugs that reproduce intermittently and in non-obvious ways, and those are a real bear to track down and fix.   
  
Instead of each engineer needing to come up with their own way of managing memory for their system, we have a bunch of tools we've already written that can be used as a drop-in solution. Not only are they battle tested and stable, they include tracking capabilities so that we can see the entire memory usage of our application and identify problematic allocations.   
  
Some Bungie coding guidelines that support this razor are:

* Use engine-specified allocation patterns.
* Do not allocate memory directly from the operating system.
* Avoid using the Standard Template Library for game code.

## #6 централизуйте управление жизненным циклом.

## Распределение управления жизненным циклом между системами с различными политиками затрудняет обоснование правильности при составлении систем и моделей поведения. Вместо этого используйте общий набор инструментов и идиомы и по возможности избегайте управления собственным жизненным циклом.

## Когда эта тонкость говорит об управлении жизненным циклом, главное, о чем она говорит, - это распределение памяти внутри игры. Одним из обоюдоострых преимуществ c++ является то, что управление этой памятью в значительной степени остается на усмотрение инженера. Это означает, что мы можем разработать стратегии распределения и использования, которые являются наиболее эффективными для нас, но это также означает, что мы берем на себя весь риск возникновения ошибок. Неправильное использование памяти может привести к ошибкам, которые воспроизводятся с перерывами и неочевидным образом, и их очень сложно отследить и исправить.

## Вместо того, чтобы каждому инженеру приходилось придумывать свой собственный способ управления памятью для своей системы, у нас есть набор инструментов, которые мы уже написали и которые можно использовать в качестве готового решения. Они не только протестированы в боевых условиях и стабильны, но и включают в себя возможности отслеживания, так что мы можем видеть все использование памяти нашим приложением и выявлять проблемные распределения.

## Вот некоторые рекомендации по кодированию bungie, которые поддерживают эту тонкость:

## • используйте шаблоны распределения, заданные движком.

## • не выделяйте память непосредственно из операционной системы.

## • избегайте использования стандартной библиотеки шаблонов для игрового кода.

## RECAP PLEASE

Alright, let's review. Guideline razors help us evaluate our guidelines to ensure that they help us address the challenges we face when writing code at scale. Our razors are:

* Favor understandability at the expense of time-to-write
* Avoid distinction without difference
* Leverage visual consistency
* Avoid misleading abstractions
* Favor patterns that make code more robust
* Centralize lifecycle management

Also, you may have noticed that the wording of the razors doesn't talk about any C++ specifics, and that’s intentional. What's great about these is that they're primarily focused on establishing a general philosophy around producing maintainable code. They're mostly applicable to other languages and frameworks, whereas the guidelines that are generated from them are specific to the target language, project, and team culture. If you're an engineer, you may find them useful when evaluating the guidelines for your next project.

Подведите итог, пожалуйста

Хорошо, давайте рассмотрим. Руководства по тонкостям помогают нам оценить наши рекомендации, чтобы убедиться, что они помогают нам решать проблемы, с которыми мы сталкиваемся при написании масштабного кода. Наши тонкости - это:

• повышайте понятность за счет сокращения времени на написание

• избегайте различий без различий

• повышайте визуальную согласованность

• избегайте вводящих в заблуждение абстракций

• отдавайте предпочтение шаблонам, которые делают код более надежным

• централизовать управление жизненным циклом

Кроме того, вы, возможно, заметили, что формулировка тонкость не говорит о какой-либо специфике c++, и это намеренно. Что в них замечательного, так это то, что они в первую очередь направлены на формирование общей философии создания поддерживаемого кода. Они в основном применимы к другим языкам и фреймворкам, в то время как рекомендации, которые формируются на их основе, специфичны для целевого языка, проекта и культуры команды. Если вы инженер, они могут оказаться полезными при оценке рекомендаций для вашего следующего проекта.

- Ricky Senft