УДК № 004.42

Чистый код как концепция развития программного обеспечения: теория и применение в компонентно-ориентированном программировании

М. Ю. Павлов, А. В. Боднар
Донецкий национальный технический университет, г. Донецк
кафедра программной инженерии
E-mail: vigototheroad@mail.ru

Аннотация:
Статья рассматривает понятие чистого кода и его роль в разработке ПО. Анализируются отличия чистого и «грязного» кода и их влияние на проектирование. Особое внимание уделено компонентно-ориентированному программированию как способу эффективного применения принципов чистого кода. Подход к чистому коду как к философии развития программного продукта позволяет достичь оптимального баланса между качеством архитектуры и практическими требованиями разработки.

Введение

Что делает код «чистым»? В программировании часто обсуждают принципы хорошего стиля, архитектурной чистоты и читаемости кода. Но где проходит грань между чистым и грязным кодом? Можно ли считать плохо структурированный, но работающий код действительно «грязным»?

В своей книге «Чистый код» Роберт Мартин утверждает: «Умение писать чистый код – тяжёлый труд. Оно не ограничивается знанием паттернов и принципов. Над кодом необходимо попотеть. Необходимо пытаться и терпеть неудачи» [1].

Так что же такое «чистый» код и каким критериям он должен соответствовать? Код, который не соответствует эти критериям, автоматически становится плохим? Важно понимать, что в реальных условиях программирования иногда приходится отклоняться от полного соответствия стандартам в угоду скорости разработки, требованиям бизнеса или ограничениям ресурсов.

В этой статье рассматривается, чем отличается чистый код от «грязного», в каких ситуациях «грязный» код может быть оправдан и почему фанатичное следование правилам может привести к обратному эффекту.

Чистый Код

Стоит уточнить, что само выражение «чистый код» имеет множество смыслов и трактовок, но в рамках данной работы будет выведено общее определение, которое наилучшим образом отражает его идею. В книге Роберта Мартина нет чёткого определения – вместо этого автор описывает отношение ведущих разработчиков к чистому коду и то, как, по их мнению, он должен выглядеть.

Ниже приведены некоторые цитаты из данной книги [1]:

«Вы работаете с чистым кодом, если каждая функция делает примерно то, что вы ожидали. Код можно назвать красивым, если у вас также создается впечатление, что язык был создан специально для этой задачи», – Уорд Каннингем.

«Я мог бы перечислить все признаки, присущие чистому коду, но существует один важнейший признак, из которого следуют все остальные. Чистый код всегда выглядит так, словно его автор над ним тщательно потрудился», – Майкл Физерс [1].

«Сокращение дублирования, высокая выразительность и раннее построение простых абстракций. Все это составляет чистый код в моем понимании», – Рон Джеффрис [1].

Самая лаконичная и при том информативная цитата принадлежит Бьёрну Страуструпу:

«Чистый код – это код, который легко читать и улучшать» [1].

Про способы написания чистого кода, изложенные в книге, Роберт Мартин говорит следующее:

«Мы излагаем свои мнения в виде беспрекословных истин и не извиняемся за свою категоричность. Для нас, на данном моменте наших карьер, они являются беспрекословными истинами» [1].

Хотя Роберт Мартин не даёт прямого определения чистому коду, сама книга описывает критерии, которым следуют как сам Мартин, так и разработчики, которых он приводит в пример.

Тем самым он описывает следующие характеристики чистого кода:

- понятность;
- лаконичность;
- логичность;
- работа на одном уровне абстракции;
- модульность;
- легкость в поддерживании;
- тестируемость.

Эти принципы прослеживаются в конце книги, в списке эвристических правил и «запахов» кода. Под «запахами» подразумеваются опыт и интуиция программиста, говорящие ему о том, что с кодом что-то не так. Поскольку эвристические правила зависят от конкретного программиста и его видения кода, то их существует огромное множество.

Важным является то, что вышеописанные идеи нашли отражение во многих подходах к разработке программного обеспечения. К примеру, в объектно-ориентированном программировании (ООП) принципы инкапсуляции и полиморфизма помогают писать расширяемый код, при этом не разрушая изначальную структуру и создавая управляемую конструкцию. В компонентно-ориентированном программировании (КОП) акцент делается на гибкость модулей. В нём реализация выходит за рамки конкретного компонента, позволяя взаимодействовать с любыми объектами, имеющими определенный модуль, через «системный» компонент, что так же поддерживает ранее описываемую идею.

Все вышеописанное позволяет вывести следующее определение чистого кода:

Чистый код – это не просто стиль написания, а разумный подход к разработке. Это код, который выполняет свою задачу и не мешает улучшению продукта.

Грязный код

Грязный код является противоположностью чистого. Если идти от обратного, то у такого кода должна отсутствовать хотя бы одна из характеристик чистого кода. Возникает вопрос: достаточно ли отсутствия одного критерия, чтобы код считался грязным, или он должен не соответствовать всем критериям одновременно? Как упоминалось в книге, отсутствие определенных качеств в коде может привести к проблемам. Например, неправильное именование переменной вызывает вопросы о её предназначении. Это и есть первопричина появления «не чистого» кода.

Дать прямое определение «грязному» коду довольно сложно. Можно ли считать весь код, который не соответствует критериям чистого, грязным? Сложности с определением связаны с отсутствием чистых критериев. Однако, здесь возможны два подхода.

Первый подход заключается в отрицании чистого кода: любой код, не соответствующий его критериям, автоматически считается грязным.

Второй подход предполагает описание собственного понимания грязного кода и определение границы, за которой «грязь» начинает вредить.

Грязный код как отрицание чистого кода

Со знанием критериев чистого кода становится возможным сформулировать то, чем является «грязный» код:

- неочевидность (использование непонятных имен переменных и функций, слишком длинные или вложенные конструкции, затрудняющие восприятие, применение «магических чисел» вместо осмысленных констант);
- избыточность (дублирование кода, написание избыточных проверок, которые уже предусмотрены языком, использование ненужных переменных и классов, усложняющих структуру);
- хаотичность (отсутствие четкой структуры, использование разных стилей кодирования в одном файле, изменение переменных в разных частях кода, что затрудняет отслеживание их состояния);
- смешение абстракций (одновременная работа с абстрактными и конкретными деталями);
- монолитность (весь код сосредоточен в одном огромном классе без разделения на модули, один метод выполняет несколько задач вместо их делегирования, отсутствие четких границ ответственности между частями кода);
- хрупкость (любое изменение ломает другие части системы, код зависит от множества несвязанных модулей, функции и классы разрастаются, содержат десятки параметров);
- плохая тестируемость (многие функции имеют неочевидные входные и выходные данные или выполняют дополнительные действия, влияющие не только на результат).

Если хотя бы один из этих пунктов применим к коду, его можно считать «грязным».

Однако, имеется нюанс: код не всегда бывает идеальным. Даже с учетом приведенного определения сложно представить код, который бы соответствовал всем требованиям одновременно.

Например, оптимизированный алгоритм может выглядеть запутанно, но он необходим для повышения производительности. Или в коде могут быть реализованы сложные взаимодействия, но при этом входные и выходные данные сами по себе просты.

Таким образом, в рамках этого подхода практически любой код можно считать «грязным».

Когда грязный код действительно вреден

Когда код можно считать «грязным»? Логично предположить, что решать проблему стоит тогда, когда она действительно начинает негативно влиять на разработку.

Когда код является приемлемым?

- он локализован и не разрастается по всей системе;
- он выполняет простую логику без неоднозначных выводов;
- его легко адаптировать;
- он не создает лишние зависимости;
- он оправдан скоростью работы или разработки;
- он временный и будет исправлен позже.

Когда грязный код начинает вредить?

- его сложно понять и реализовать;
- он провоцирует цепную реакцию багов при изменении;
- он требует изменение базовой логики при дальнейшей разработке;
- его операции ведут к неявным входным и выходным данным.

Таким образом, граница между неидеальным, но рабочим кодом и по-настоящему вредным кодом проходит там, где он начинает мешать развитию и поддержке продукта. Если код замедляет разработку, вносит хаос и увеличивает вероятность ошибок – это уже не компромисс, а проблема.

Второй подход кажется более практичным, особенно в условиях реальной разработки. Он несет в себе простую идею: грязный код становится проблемой только тогда, когда начинает негативно влиять на проект. Конечно, в некоторых случаях частные правила важнее глобальных, и попытка внедрить идеальные стандарты там, где они не нужны, может только навредить, запутывая логику и усложняя ранее выполненную работу.

Использование чистого кода в КОП

Понимание принципов чистого кода позволяет эффективно применять их в компонентно-ориентированном подходе. В данном разделе рассматривается, как принципы чистого кода используются в этой парадигме, каким образом их соблюдение упрощает разработку, а также какие базовые особенности написания кода в рамках КОП необходимо учитывать. Слепое заимствование практик чистого кода оказывается недостаточным – их требуется адаптировать к специфике КОП и особенностям разработки для обеспечения эффективности и гибкости создаваемых систем.

КОП активно опирается на принцип «композиция вместо наследования». Вместо построения сложных иерархий классов, где поведение объектов определяется через наследование, КОП формирует объекты путём комбинирования независимых компонентов, каждый из которых отвечает за отдельную функциональность. Это повышает гибкость системы, позволяя без значительных затрат добавлять, удалять или изменять поведение объектов. Наследование, напротив, часто приводит к жёстким связям между классами, что затрудняет модификацию системы. Например, в RPG-системе, если персонаж наследует базовый класс Character, а затем создаются подклассы Warrior, Mage и другие, изменение базового класса способно нарушить корректную работу всех производных классов. Композиция позволяет динамически добавлять компоненты по мере необходимости, что значительно упрощает настройку объектов.

Ещё одной важной характеристикой КОП является возможность добавления, удаления или изменения компонентов во время выполнения программы. Это обеспечивает адаптивность системы к изменяющимся условиям без необходимости модификации исходного кода или перекомпиляции проекта. Например, в RPG-проекте персонаж может получить новую способность в процессе выполнения квеста, что повышает интерактивность игрового мира.

КОП часто применяет системы, координирующие взаимодействие групп компонентов. Вместо прямого взаимодействия между компонентами данные обрабатываются системами, которые обновляют их состояние. Такой подход характерен для архитектуры Entity-Component-System (ECS), одной из популярных реализаций КОП. В ECS компоненты представляют собой структуры данных, а системы реализуют прикладную логику, что снижает связанность и способствует повышению производительности.

КОП также поощряет декларативный подход, при котором поведение и свойства компонентов определяются через данные, а не через жёстко закодированную логику. Это реализуется посредством сериализации данных, например через форматы JSON, XML или настройки редакторов игровых движков, таких как Unity. Такой подход упрощает процесс настройки компонентов дизайнерами без необходимости внесения изменений в код. Несмотря на возможное противоречие с принципом самодокументируемости кода, в Unity использование атрибутов, таких как [SerializeField], позволяет создавать наглядные интерфейсы редактора и задавать связи между компонентами без нарушения читаемости.

Компоненты в КОП должны быть максимально автономными, то есть их поведение не должно зависеть от состояния других компонентов, за исключением явно

определённых взаимодействий, например, через события или системы. Автономность обеспечивает переиспользуемость компонентов в различных контекстах без необходимости их модификации. Например, компонент HealthComponent может использоваться как для игроков, так и для NPC или разрушаемых объектов.

Принцип читаемости ярко выражен в КОП, поскольку каждый компонент отвечает за строго определённую функциональность. Названия компонентов и их методов должны быть интуитивно понятными. Например, вместо абстрактного Component1 предпочтительнее использовать наименования вроде PlayerMovementComponent или InventoryComponent, что упрощает восприятие кода. Лаконичность достигается за счёт того, что каждый компонент описывает только необходимые характеристики, используемые системами для реализации функциональности, минимизируя избыточный код и повышая управляемость компонентов.

Принцип единой ответственности в КОП реализуется элегантно: каждый компонент выполняет только одну функцию. Например, в RPG-системе HealthComponent управляет здоровьем персонажа, а InventoryComponent – его предметами. Это предотвращает создание так называемых «божественных объектов», объединяющих множество функций, что увеличивает сложность системы и снижает её гибкость. Зависимости между компонентами минимизируются посредством обмена данными через события, сообщения или интерфейсы, а не через прямые ссылки. Это снижает связанность и повышает модульность системы. Например, компонент атаки может генерировать событие OnAttack, обрабатываемое компонентом здоровья без знания о его внутреннем устройстве.

Для устранения дублирования кода в КОП применяются базовые компоненты или утилитные классы. Например, если несколько компонентов используют событийную модель, целесообразно создать общий EventManager, который централизованно управляет событиями, устраняя необходимость повторного написания однотипного кода.

Чистый код в КОП: синтез авторских подходов

Понимание чистого кода в компонентно-ориентированном программировании не может ограничиваться только практическими приёмами. Для полноценного применения этой концепции необходимо учитывать теоретические и методологические подходы, выработанные ведущими исследователями в области программной инженерии. В данной главе рассмотрены взгляды авторов, оказавших значительное влияние на формирование современной культуры разработки, а также проведена интерпретация их идей в контексте КОП.

В книге «Совершенный код» С. Макконнелл подчёркивает, что читаемость, локализация изменений и снижение когнитивной нагрузки – ключевые признаки качественного кода [2]. Эти принципы непосредственно применимы к КОП, где каждый компонент реализует узкоспециализированную функцию, а значит, должен быть предельно понятным и изолированным.

М. Фаулер в книге «Рефакторинг» рассматривает архитектурную чистоту как результат регулярной реорганизации кода без изменения поведения [3]. Для КОП это особенно важно, так как позволяет модифицировать поведение системы через изменение отдельных компонентов без затрагивания остальной архитектуры.

Д. Кнут рассматривает программирование как форму выражения мысли [4]. В КОП, где компоненты должны быть переиспользуемыми и инкапсулированными, ясность их интерфейсов и поведения становится критическим фактором. Концепция чистого кода как выразительного средства полностью применима к описанию поведения компонентов и систем.

Авторы «Паттернов проектирования» подчеркивают важность слабой связанности и высоко уровня абстракции [5]. Эти характеристики органично вписываются в КОП, где вместо иерархий классов применяются независимые модули, взаимодействующие через общие интерфейсы или события.

К. Бек в рамках экстремального программирования настаивает на простоте реализации и приоритете тестируемости [6]. Для КОП, где поведение системы определяется взаимодействием множества компонентов, возможность изолированного тестирования каждого из них – необходимое условие надёжности.

В «Программисте-прагматике» Томас и Хант подчёркивают значение повторного использования, отсутствия дублирования и локализации логики [7]. Эти требования совпадают с архитектурной философией КОП, в рамках которой компоненты должны быть автономными и легко заменяемыми.

М. Физерс в книге «Работа с унаследованным кодом» описывает стратегии постепенного улучшения сложных систем без полной их переработки [8]. Это напрямую применимо к КОП, так как замена или модификация конкретных компонентов позволяет эволюционно адаптировать систему к новым требованиям.

Анализ подходов признанных экспертов позволяет сделать вывод, что многие принципы чистого кода – читаемость, модульность, слабая связанность, тестируемость и минимализм – естественным образом интегрируются в компонентно-ориентированную архитектуру. Эти авторские идеи не только дополняют теоретическую базу КОП, но и обосновывают его эффективность с точки зрения инженерной практики. Таким образом, чистый код и КОП не просто совместимы – они взаимно усиливают друг друга в процессе построения гибких и сопровождаемых программных систем.

Когда используется Чистый Код

Здесь необходимо обратиться к парадигмам программирования и их предназначению. Если кратко, парадигмы программирования – это концепции, определяющие стиль написания кода в соответствии с определёнными принципами. Они помогают достичь оптимальных результатов в конкретных условиях. Многие парадигмы включают принципы и методологии, которые способствуют повышению эффективности разработки. Одним из ключевых преимуществ ООП является возможность структурировать программу с помощью классов, каждый из которых отвечает за свою область. Однако это может стать и недостатком: чрезмерное усложнение архитектуры ведёт к увеличению связей и затруднению сопровождения кода.

Используя данный подход, важно учитывать последствия архитектурных решений. Следует задуматься о глубине иерархии: не приведёт ли она к чрезмерному усложнению структуры кода? Не сделает ли избыточная иерархия базовый класс хрупким? И, конечно, не приведёт ли слепое следование парадигме к ненужному усложнению кода?

Важно понимать, что, когда разработчик пишет код, он пишет его не для компилятора, а для других людей. Когда возникает потребность в написании чистого кода? В первую очередь, каждый программист должен следить за тем, что он пишет. Когда он замечает проблему, он должен её решить. Роберт Мартин называет это «запахами кода» – ситуациями, когда разработчик интуитивно понимает, где может возникнуть проблема [1].

И здесь кроется главная особенность книги: «чистый код» – это не просто свод правил, а концепция развития кода. Это инструмент, который может как улучшить, так и ухудшить ситуацию, или философия, которая влияет на разработку, позволяя программисту использовать её не как цель, а как возможность улучшить состояние проекта.

Если относиться к чистому коду не как к набору строгих правил, а как к идее о том, что код стоит чистить и улучшать, это позволит будущим разработчикам не тратить время на попытки разобраться в том, что делает этот код и зачем он нужен. Это, в свою очередь, облегчит развитие и поддержку кода. Именно поэтому в начале книги делается большой акцент на опыте других разработчиков. Большинство правил, приведённых в книге, используются на подсознательном уровне, когда программист понимает, с какими проблемами он может столкнуться, возвращаясь к коду, или где части кода могут быть слишком запутанными, как неопытное описание автора. Но опыт и стремление сделать свой код лучше – это лучшее лекарство от таких проблем. Анализ того, почему определённый кусок кода кажется непонятным или запутанным, позволяет взглянуть на код с другой стороны, как на искусство, порождающее понимание как в себе, так и в других людях. Именно к этому отсылает фраза из начала книги:

«А как мне написать чистый код?» Бесполезно пытаться написать чистый код, если вы не знаете, что это такое! – Боб Мартин [1].

И самый большой плюс такого подхода – это простота разработки. Когда код начинает получаться простым, с понятной и проработанной структурой, не возникает вопросов о том, как получить доступ к переменной или где выделить функции для реализации определённой фичи так, чтобы это было органично. А когда приходится дополнять такой код, ранее разработанная структура начинает работать на разработчика, а не разработчик пытается втиснуть новый код в уже перегруженную систему.

Поэтому многие находят отражение программирования в разных сферах: кто-то – в кулинарии, как Кнут, кто-то – в искусстве, как Мартин [4]. И действительно, чем больше разработчик рассматривает другие области, тем больше решений он может найти для своих задач, что побуждает к развитию и изучению нового.

Ни архитектура, ни чистый код не требуют от нас совершенства – просто будьте честны и делайте все, что можете – Боб Мартин [1].

Выводы

В данной статье проведён комплексный анализ понятия чистого кода, его характеристик и противоположности – грязного кода. Показано, что понятие чистого кода выходит за рамки формальных правил и является важным аспектом культуры программирования, ориентированным на повышение читаемости, сопровождаемости и адаптивности программных решений.

Особое внимание уделено интеграции принципов чистого кода в компонентно-ориентированное программирование. Продемонстрировано, что КОП благодаря свойственным ему свойствам композиции, слабой связности и автономности компонентов,

позволяет наиболее естественным образом применять идеи чистого кода в архитектуре программных систем.

Отмечено, что эффективное использование принципов чистого кода требует осознанной адаптации к контексту конкретной задачи, а не механического следования установленным рекомендациям. Подход к чистому коду как к философии развития программного продукта позволяет достичь оптимального баланса между качеством архитектуры и практическими требованиями разработки.

Перспективы дальнейших исследований связаны с изучением применения принципов чистого кода в других парадигмах программирования, а также с разработкой методик автоматизированного анализа качества кода в рамках КОП.

Литература

1. Мартин, Р. Чистый код: создание, анализ и рефакторинг. Библиотека программиста / Р. Мартин. - СПб.: Питер. 2019. – 452 с.

2. Макконнелл, С. Совершенный код: мастер-класс по разработке программного обеспечения / С. Макконнелл. – СПб.: Питер, 2017. – 896 с.

3.Фаулер, М. Рефакторинг. Улучшение проекта существующего кода/ М. Фаулер. Пер. с англ. - СПб: Символ-Плюс, 2003. - 432 с.

4. Кнут, Д. Искусство программирования. Том 1: Основные алгоритмы / Д. Кнут. — М.: Вильямс, 2011. — 784 с.

5. Гамма, Э. Приёмы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Влиссидес. – СПб.: Питер, 2016. – 368 с.

6. Бек, К. Экстремальное программирование: разработка через ценности / К. Бек. — СПб.: Питер, 2017. — 240 с.

7. Томас, Д. Программист-прагматик: путь от подмастерья к мастеру / Д. Томас, Э. Хант. — СПб. Диалектика, 2020. — 352 с.

8. Физерс М. Работа с унаследованным кодом: как улучшить старый код / М. Физерс. — М.:Вильямс, 2009. — 400 с.

9. Heinman, G. T. Component-based software engineering : putting the pieces together / G. T. Heinman, W. T. Concill. - Boston : Addison-Wesley, 2001. – 818 с.

10. Тепляков, С. Паттерны проектирования на платформе .NET / С. Тепляков. – СПБ: Питер, 2015. - 320 с.

11. Эванс, Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем / Э. Эванс. - Москва : 000 "И.Д. Вильямс", 2011. - 448 с.

12. Медведев, В. И. NET компонентно – ориентированное программирование / В. И. Медведев. - 2-е издание – Казань: Республиканский центр мониторинга качества образования, 2013. – 248 с.

13. Кулямин, В. В. Компонентный подход в программировании/ В. В. Кулямин. - 2-е издание - М.: НОУ "Интуит" 2016. – 590 с.

Павлов М. Ю., Боднар А. В. Чистый код как концепция развития программного обеспечения: теория и применение в компонентно-ориентированном программировании. *Статья рассматривает понятие чистого кода и его роль в разработке ПО. Анализируются отличия чистого и "грязного" кода и их влияние на проектирование. Особое внимание уделено компонентно-ориентированному программированию как способу эффективного применения принципов чистого кода. Подход к чистому коду как к философии развития программного продукта позволяет достичь оптимального баланса между качеством архитектуры и практическими требованиями разработки.*

Ключевые слова*: чистый код, грязный код, компонентно-ориентированное программирование, композиция, архитектура ПО, сопровождаемость.*

Pavlov M., Bodnar A. Clean Code as a Software Development Concept: Theory and Practice in Component-Oriented Programming. *The article examines the concept of clean code and its role in software development. It analyzes the differences between clean and "dirty" code and their impact on design. Special attention is given to component-oriented programming as an effective way to apply clean code principles. The approach to clean code as a software product development philosophy allows achieving an optimal balance between the quality of architecture and the practical requirements of development.*

Keywords*: clean code, dirty code, component-oriented programming, composition, software architecture, maintainability.*

Статья поступила в редакцию 25.02.2025
Рекомендована к публикации профессором Зори С. А.