

# Программное дело

*о разработке программ расскажет  
Михаил В. Шохирев*

Клуб программистов  
Шадринск  
2020

## Почему я захотел рассказать об этом?

Разрабатывается очень много некачественных программ.  
Многие программисты не имеют опыта разработки ПО.  
В разработчики идёт много неквалифицированных людей.  
Учат не разрабатывать программное обеспечение, а «писать программы».  
Многие «программисты» не знают про современные подходы к разработке.  
Их не учат разрабатывать ПО в эволюционном процессе работы в команде.  
Игнорируется, что тестирование — неотъемлемая часть разработки.  
Пренебрегают рефакторингом: не понимают, не умеют и не практикуют.  
Не понимают, что успех разработки в правильной архитектуре системы.  
Не пользуются необходимыми средствами разработки.

*«Велосипеды изобретены» — надо научиться на них ездить.*

## Личная причина

*“Amazing grace! How sweet the sound  
That saved a wretch like me.  
I once was lost, but now am found,  
Was blind but now I see.”*

*John Newton, 1772*

Всю свою программистскую жизнь стремлюсь стать профессионалом.  
Не так давно у меня окончательно сложилась картина, как следует  
разрабатывать добротные программы.

И мне захотелось поделиться с коллегами-одноклубниками.

# Мои учителя

1. «Надёжность программного обеспечения» — Гленфорд Майерс (1980)
2. «Стиль, разработка, эффективность, отладка и испытание программ» — Дэнни ВанТассел (1981)
3. «Искусство тестирования программ» — Гленфорд Майерс (1982)
4. «Алгоритмы + структуры данных = программы» — Никлас Вирт (1985)
5. «Инструментальные средства программирования на языке Pascal» — Брайан Керниган, Филип Плоджер (1985)
6. «Экстремальное программирование» — Кент Бек (2002)
7. «Программист-прагматик. Путь от подмастерья к мастеру» — ДэйвТомас, Энди Хант (2004)
8. «Чистая архитектура. Искусство разработки программного обеспечения» — Роберт Мартин (2018)

# О чём пойдёт речь

Software ~ взаимосвязанные ценности программного обеспечения.

Traditional ~ традиционный подход к разработке.

Agile ~ гибкие технологии разработки.

X. P. ~ экстремальное программирование.

TDD ~ разработка через тестирование.

Testing ~ важность тестирования.

Refactoring ~ необходимость реорганизации исходников.

Architecture ~ ценность и важность архитектуры.

S. O. L. I. D. ~ принципы построения программных систем.

FP, SP, OOP ~ ценность парадигм программирования.

Design Patterns ~ признанные образцы проектирования ПО.

Tools ~ необходимые инструменты разработки.

Qualification ~ знания и умения разработчика ПО.

# Что нужно знать и уметь разработчику

*Написать программу не сложно.*

*Труднее сделать, чтобы она работала правильно.*

*Сложно разработать программу, которую можно развивать при использовании.*

Для этого надо овладеть современными подходами к разработке:

- знаниями,
- принципами,
- технологиями,
- практиками,
- инструментами.



# Программное обеспечение

2 ценности программного обеспечения:

2

soft-  
«ИЗМЕНЧИВЫЙ»

возможность свободно  
модифицировать систему при её  
адаптации и развитии;  
реализуется в структурном  
проекте программной системы,  
в её архитектуре

**«программу легко изменить»**

1

-ware  
«ПРОДУКТ»

функциональное поведение,  
позволяющее получить нужный  
результат;  
реализовано в алгоритмах и  
взаимодействии программных  
модулей системы

**«программа работает правильно»**

# Software values paradox

If you give me a program  
that does not work  
but is easy to change,  
then I can make it work,  
and keep it working  
as requirements change.

Therefore the program  
will remain continually useful.

If you give me a program  
that works perfectly  
but is impossible to change,  
then it won't work  
when the requirements change,  
and I won't be able to make it work.

Therefore the program  
will become useless.

*Robert „Uncle Bob“ Martin*



# Ценности ПО: парадокс

Если программа *работает неправильно*,  
но **легко поддаётся изменению**,  
вы сможете исправить её и  
поддерживать её работоспособность  
по мере изменения требований.

То есть  
программа постоянно  
будет оставаться полезной.

Если программа *работает правильно*,  
но её **практически нельзя изменить**,  
она перестанет работать правильно,  
когда изменятся требования,  
и у вас не получится её исправить.

То есть  
программа станет бесполезной.

*Роберт „Дядюшка Боб“ Мартин*

# Практически изменяемая программа

Программа, которую можно изменять

- в требуемом объёме;
- в запланированное время;
- в рамках имеющегося бюджета;
- без дополнительного персонала;
- без искажения работы системы.

Для пользователя критически важна стоимость внесения изменений в программную систему.

*Заказчики хотят менять всё:*

- *структуру данных;*
- *набор функционала;*
- *алгоритмы обработки;*
- *подключаемые устройства;*
- *пользовательский интерфейс;*
- *взаимодействие с внешним ПО;*
- *...*

И это нормально!

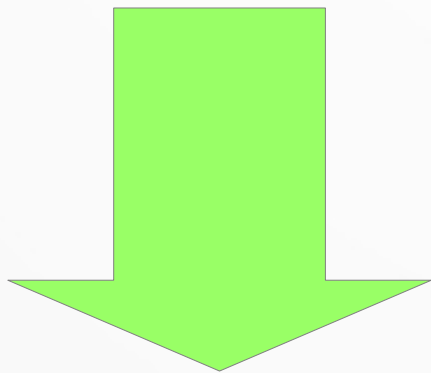
*"Пользователь не знает, чего он хочет, пока не увидит то, что он получил".  
Эд Йордан*

*"Процесс автоматизации изменяет взгляд пользователя на объект автоматизации".*

*Гленфорд Майерс*

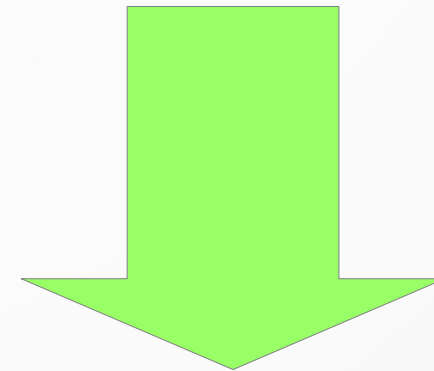
# Условия для изменяемости

Уверенность, что программа продолжит правильно работать после изменения.



Надёжный набор тестов.

Программа имеет структуру, которая позволяет легко вносить изменения.



“Чистая архитектура”.

# Традиционный подход к разработке

## Waterfall

### «Водопад»:

Требования: определить и проанализировать требования заказчика.

Проектирование: формализованно описать и утвердить проект.

Программирование: написать программы по спецификациям.

Отладка: найти и исправить ошибки при кодировании.

Тестирование: показать, что всё работает, как нужно.

Документирование: теперь можно написать её.

Внедрение: когда всё готово и протестировано.

Сопровождение: при эксплуатации.

*Зафиксировать проект – и спокойно программировать...*

# Традиционный подход *не работает!*

## ~~Waterfall~~

### «Водопад»:

Требования: *всё время меняются.*

Проектирование: *невозможно закончить полностью и в срок.*

Программирование: *по неполному проекту, решения по ходу.*

Отладка: *поэтому ошибки неизбежны.*

Тестирование: *недостаточное и несвоевременное.*

Документирование: *после времени.*

Внедрение: *когда «поджимают сроки».*

Сопровождение:

*«Эта сказка хороша — начинай сначала!».*

*Ни зафиксировать, ни спокойно программировать не удаётся...*



# «Agile Manifesto» (2001)

«Мы постоянно открываем для себя более совершенные методы разработки программного обеспечения, непосредственно занимаясь разработкой и помогая в этом другим. Благодаря проделанной работе мы смогли осознать, что:

**Люди и взаимодействие** важнее *процессов и инструментов*.

**Работающий продукт** важнее *исчерпывающей документации*.

**Сотрудничество с заказчиком** важнее *согласования условий контракта*.

**Готовность к изменениям** важнее *следования первоначальному плану*.

То есть, не отрицая важности того, что *справа*,  
мы всё-таки больше ценим то, что **слева**.»

**Kent Beck**, Mike Beedle, Arie van Bennekum, **Alistair Cockburn**, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, **Andrew Hunt**, Ron Jeffries, Jon Kern, Brian Marick, **Robert C. Martin**, Steve Mellor, Ken Schwaber, Jeff Sutherland, **Dave Thomas** + многие другие позже.



# Принципы манифеста Agile

1. **Удовлетворение потребностей заказчика** путём ранней и регулярной поставки значимого ПО.
2. **Изменения требований приветствуется**, даже на поздних стадиях разработки.
3. **Работающий продукт выпускается часто**, с периодичностью в недели, а не месяцы.
4. **Разработчики и представители бизнеса тесно работают вместе** на протяжении всего проекта.
5. **Над проектом работают мотивированные профессионалы**, которым заказчик доверяет.
6. **Непосредственное общение — лучший способ взаимодействия** с командой и внутри команды.
7. **Работающий продукт — основной показатель прогресса** разработки.
8. **Поддерживать устойчивый процесс разработки в постоянном ритме.**
9. **Постоянное внимание к техническому совершенству и качеству проектирования.**
10. **Простота — искусство минимизации лишней работы** — крайне необходима.
11. **Наилучшие проектные решения создаются самоорганизующимися командами.**
12. **Команда систематически анализирует пути улучшения и корректирует стиль своей работы.**

*Это не просто провозглашённые принципы*

*— они давно и успешно применяются на практике!*

# Технологии: гибкие vs тяжеловесные

- Адаптация вместо **предопределённости**.
- Наращивание архитектуры, а не её **полное проектирование**.
- Проектирование эволюционное вместо **предварительного**.
- Ясный исходный код, а не **объёмная документация**.
- Полезный результат, а не **соблюдение утверждённого проекта**.
- Ориентация на человека, а не на **процесс**.

Главное – личности разработчиков!

# Agile = гибкие технологии разработки

- Adaptive Software Development (ASD).
- Agile Unified Process (AUP).
- Crystal Method.
- Disciplined Agile Delivery (DAD).
- Dynamic System Development Method (DSDM).
- Extreme Programming (XP).
- Feature-Driven Development (FDD).
- Getting Real (web-interface).
- Lean Software Development (LSD).
- OpenUP (← RUP).
- Rapid Application Development (RAD).
- Scrum (Sprint Continuous Rugby Unified Methodology).

# X. P. = eXtreme Programming\*

## 12 основных приёмов XP :

**Короткий цикл обратной связи** (Fine-Scale Feedback): *за итерацию разработки в 1-4 недели много не напортачишь.*

1. **Заказчик всегда рядом** (Onsite Customer = Whole Team): *конечный пользователь всегда на связи для вопросов.*
2. **Игра в планирование** (Planning Game): *направляет разработку продукта через планирование итераций.*
3. **Разработка через тестирование** (Test-Driven Development): *весь код покрыт тестами, код качественный.*
4. **Парное программирование** (Pair Programming): *коллеги работают совместно, знают код, заменяют друг друга ← ССО.*

**Непрерывный процесс** (Continuous Process): *соблюдение правил помогает безостановочно двигаться к цели.*

5. **Рефакторинг** (Refactoring = Design Improvement): *реструктуризация кода без изменения его поведения.*
6. **Частые небольшие релизы** (Small Releases): *помогают пользователю видеть прогресс, вносить изменения.*
7. **Непрерывная интеграция** (Continuous Integration): *тестирование и сборка проекта несколько раз в день.*

**Понимание, разделяемое всеми** (Shared Understanding): *минимизирует ошибки взаимодействия.*

8. **Метафора системы** (System Metaphor): *система понятий, единая для заказчиков и разработчиков → DDD.*
9. **Простота проектирования** (Simple Design): *выбирается наиболее простой способ реализовать функциональность.*
10. **Стандарт оформления кода** (Coding Standard = Coding Conventions): *соглашения о стиле и образцах.*
11. **Коллективное владение кодом** (Collective Code Ownership): *все разработчики знают исходный код ← РР.*

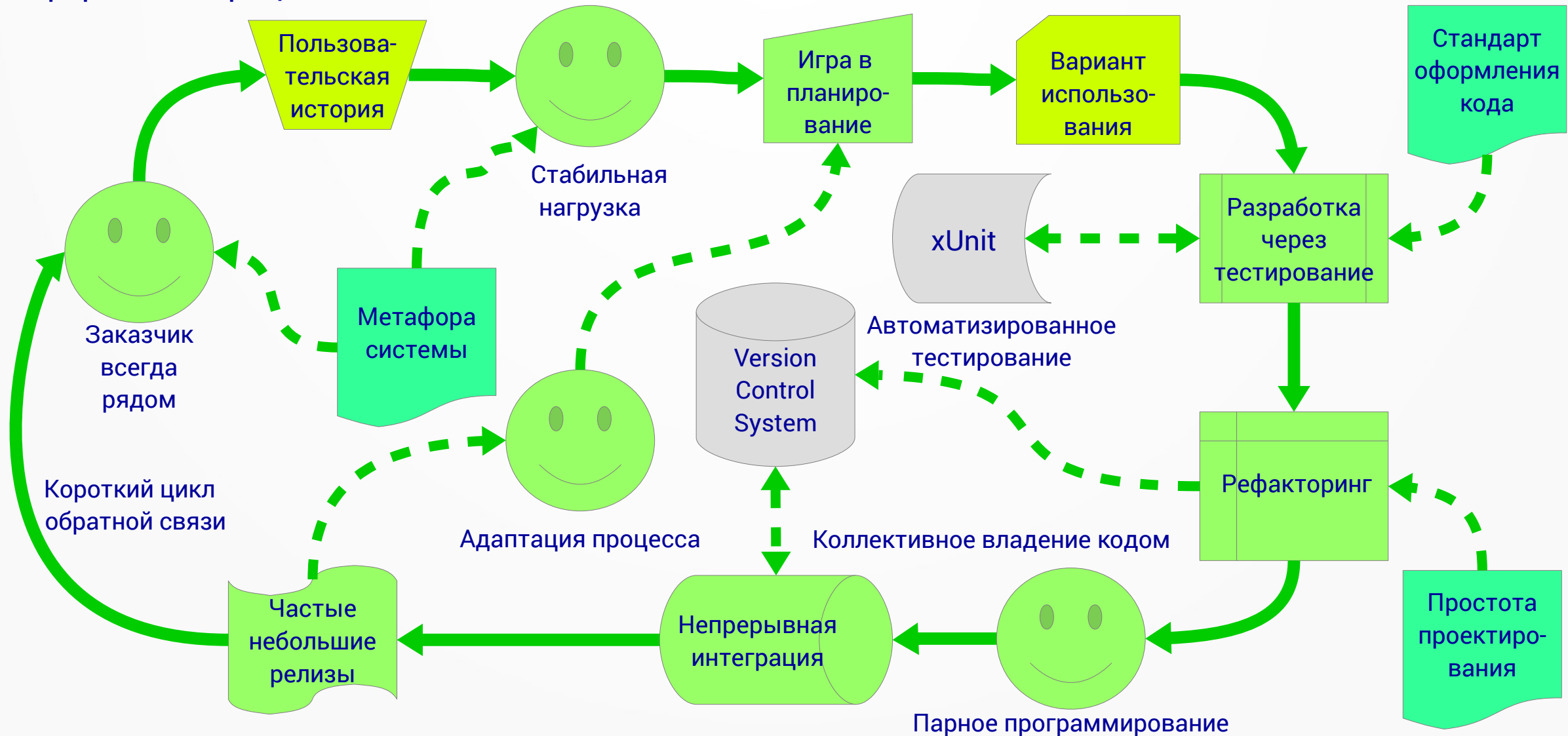
**Социальная защищённость программиста** (Programmer Welfare): *разработка без авралов и переработки.*

12. **Стабильная нагрузка** (Sustainable Pace): *40-часовая рабочая неделя (40-hour week).*

\* Название методологии исходит из идеи применить полезные традиционные методы и практики разработки программного обеспечения, подняв их на новый «экстремальный» уровень.

# Экстремальное программирование

## Непрерывный процесс





# Гибкий подход к разработке

## Agile

**Короткие итерации разработки**, во время которых делаются:

Требования: «истории пользователя» только на эту итерацию.

Проектирование: наиболее простая реализация «user stories».

Парное программирование через тестирование.

Отладка: практически не требуется после TDD.

Тестирование: приёмочное — показать, что требования выполнены.

Минимальное документирование: т. к. есть “чистый код” и тесты.

Внедрение: в конце каждой итерации.

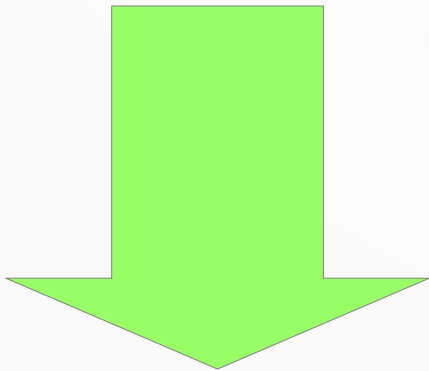
Сопровождения нет — есть только разработка!

*Заказчик хочет развития программы — разработчик это легко делает!*



# Условия для изменяемости

Уверенность, что программа  
продолжит правильно работать  
после изменения.



Надёжный набор тестов.

# Тестирование

Ошибка выполнения — это тест, который не был написан.  
Только исходный код говорит правду, а тесты — это исходники.

Тесты — это...

- ... исполняемое описание требований.
- ... лучшая низкоуровневая документация на программу.
- ... один из способов использования программной системы.
- ... особый API к разрабатываемой системе.
- ... неотъемлемая часть архитектуры системы.

Эффективно тестировать возможно, только когда тестирование автоматизировано (automated testing).

# Виды тестов

## unit test

(модульный тест)

проверяет реализацию  
технического требования или  
решения разработчика;  
*создаётся и выполняется  
программистом*

## acceptance test

(приёмочный тест)

проверяет реализацию  
функционального требования  
заказчика;  
*разрабатывается и выполняется  
тестировщиком = специалистом QA  
(Quality Assurance) или  
архитектором или руководителем  
разработки*

# Польза и важность тестирования

Имея надёжный набор тестов, ...

- ... можно рефакторить программу без опаски.

- ... можно смело править программные ошибки.

- ... можно уверенно изменять и развивать программу.

Имея полный набор тестов, можно даже воссоздать потерянную программу.

*Каким же образом можно получить надёжный набор тестов,  
которому можно полностью доверять?*

# Почему TDD?

Без TDD тестирование выполняется *после кодирования*, программисты не осознают его необходимости, в результате набор тестов получается недостаточный, на него нельзя положиться при модификации системы.

Только неуклонно практикуя TDD, разработчики получают достаточный набор тестов, которому можно доверять при модификации системы.

TDD – сознательное соблюдение технологической дисциплины разработчиками (как двойная запись в бухчёте).

Применение TDD основано на автоматизированных модульных тестах.

# TDD = test-driven development = test 1st

0            1            2  
Red → Green → Refactor!

0. Сначала пишется тест для проверки требования:  
разработчик знает требование (какой должен быть результат).  
*Прогон этого теста закончится неудачей, но тест разработан на будущее!*
1. Потом пишется минимум исходного кода:  
только, чтобы реализовать это требование.  
*Прогон этого теста закончится удачно, подтвердит: этот код рабочий.*
2. Затем написанный код реорганизуется:  
1-й вариант кода структурно улучшается без изменения сути.  
*Новый прогон теста* должен показать, что ничего не сломалось.

*Потом цикл повторяется снова... И набор тестов постепенно пополняется.*



# Refactoring

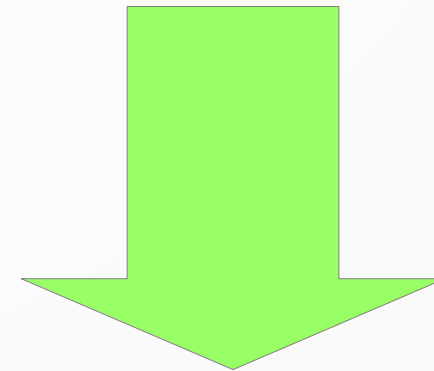
Refactoring = перепроектирование кода = переработка кода = равносильное преобразование алгоритмов ~ изменение внутренней структуры программы, не затрагивающий её внешнего поведения с целью облегчить понимание её работы. Рефакторинг выполняется как последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Поскольку каждое преобразование маленькое, программист может проследить за его правильностью. А вся последовательность преобразований приводит к существенной внутренней перестройке программы и улучшению её согласованности и чёткости.

Рефакторинг — обязательная часть разработки и постоянная деятельность программиста.

*Без регулярной «чистки» исходников  
сложность и хаотичность изменяемого кода возрастает.*

# Условия для изменяемости

Программа имеет структуру,  
которая позволяет легко  
вносить изменения.



“Чистая архитектура”.

# Архитектура = дизайн

Архитектура выражает замысел ПО, его назначение (*система заказов, учётная система, система мониторинга и т. п.*).

Она строится на основе вариантов использования ПО (*размещение заказа пользователем, обработка заказа диспетчером и т. п.*).

Цель архитектуры программного обеспечения — уменьшить программистские трудозатраты при создании и последующем развитии системы.

# Что на является архитектурой ПО

~~Framework~~ = инструмент / компонента.

~~Database~~ = хранилище = устройство ввода-вывода.

~~Web~~ = механизм доставки данных = устройство ввода-вывода.

~~SOA (service-oriented architecture)~~ = I/O device.

~~“Клиент — сервер”~~ = сетевая архитектура = конфигурация узлов.

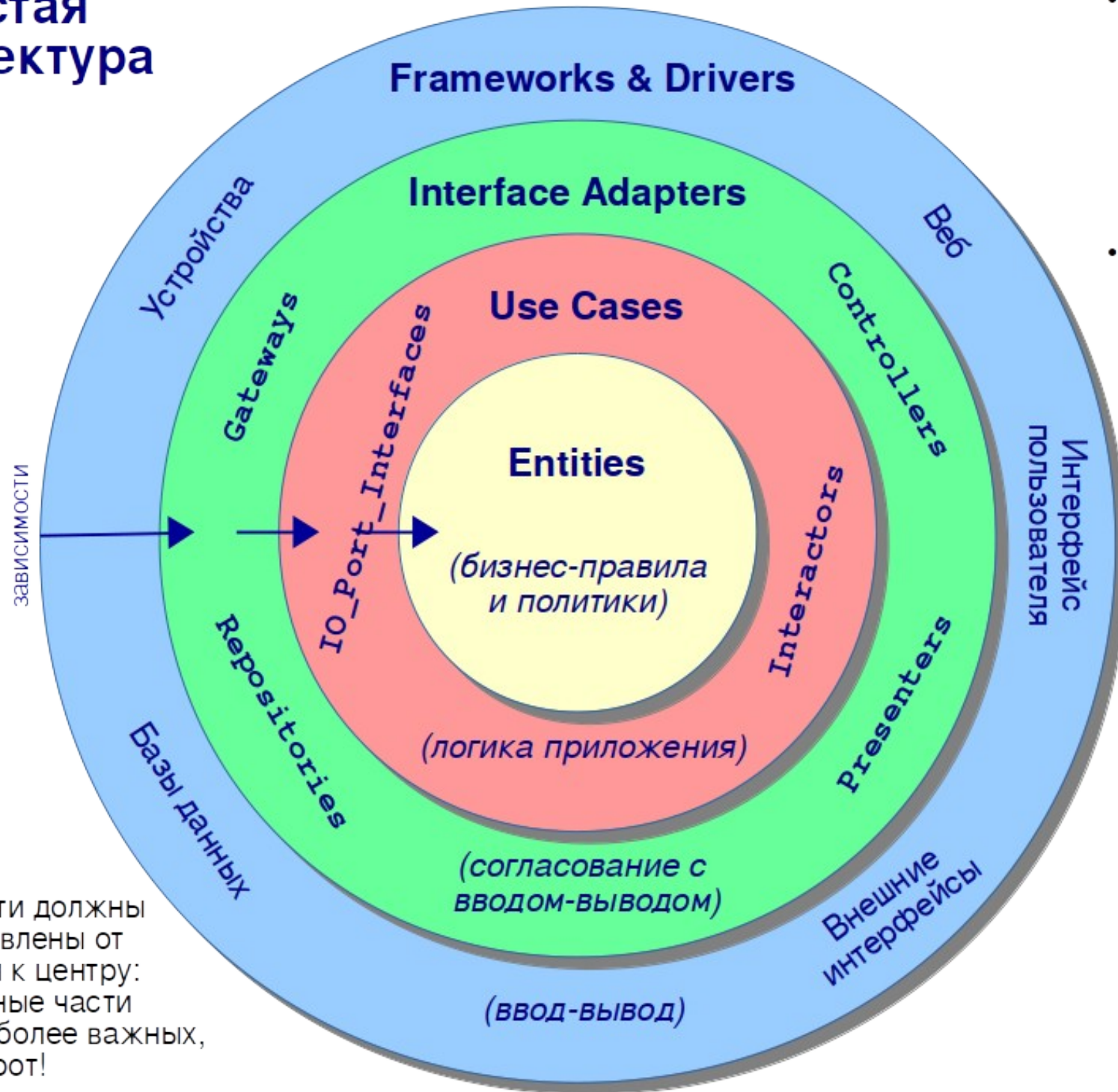
# Clean Architecture

“Чистая архитектура” программной системы – такая, в которой реализована контролируемая зависимость между основными уровнями:

- бизнес-правила и политики – то, что не зависит от программной реализации (*как делается без всякой автоматизации*);
- варианты использования – логика работы программной системы (*как бизнес-правила будут выполняться с помощью программы*);
- согласование с вводом-выводом – интерфейсы и адаптеры (*как организовать доставку данных в нужном виде*);
- ввод-вывод – механизм доставки данных (*источники и приёмники данных, средства взаимодействия с человеком и внешним ПО*).



# Чистая архитектура



- Ближе к центру — более абстрактные, высокоуровневые, более важные, реже изменяемые части программной системы.
- Ближе к периферии — более конкретные, менее существенные, чаще изменяемые части.

Зависимости должны быть направлены от периферии к центру: менее важные части зависят от более важных, а не наоборот!



# OOD = Object-Oriented Design

Именно OOD обеспечивает для разрабатываемых компонент:

- независимость разработки;
- независимость тестирования;
- независимость развертывания;

Успешное **объектно-ориентированное проектирование** основывается на нескольких основных принципах, известных как S. O. L. I. D.

# S. O. L. I. D. – принципы OOD

**S** (SRP = Single Responsibility Principle) «Принцип единственной ответственности»: «каждый класс должен иметь одну и только одну причину для изменений».

**O** (OCP = Open / Closed Principle) «Принцип открытости / закрытости»: «программные сущности ... должны быть открыты для расширения, но закрыты для модификации»ю.

**L** (LSP = Liskov Substitution Principle) «Принцип подстановки Барбары Лисков»: «объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы». Наследующий класс должен дополнять, а не изменять базовый.

**I** (ISP = Interface Segregation Principle) «Принцип разделения интерфейса»: «много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения».

**D** (DIP = Dependency Inversion Principle) «Принцип инверсии зависимостей»: «зависеть от абстракций, а не от конкретики».

# SRP ← S. O. L. I. D.

**S** (SRP = Single Responsibility Principle) «Принцип единственной ответственности»: «каждый класс должен иметь одну и только одну причину для изменений». *Действительное следствие закона Конвея: лучшей является такая структура программной системы, которая формируется в основном под влиянием социальной структуры организации, использующей эту систему, поэтому каждый программный модуль имеет одну и только одну причину для изменения.*

**SRP** предписывает разделять компоненты, которые могут изменяться по разным внешним причинам (разные роли пользователей выдвигают требования).

При нарушении **SRP** создаются компоненты, которые потребуют изменения по любой из причин, что грозит нарушением их работоспособности.

# OSCP ← S. O. L. I. D.

**O** (OSCP = Open Closed Principle) «Принцип открытости/закрытости»: «программные сущности ... должны быть открыты для расширения, но закрыты для модификации». *Этот принцип был сформулирован Бертраном Мейером в 1980-х годах. Суть его сводится к следующему: простая для изменения система должна предусматривать простую возможность изменения ее поведения добавлением нового, но не изменением существующего кода.*

**OSCP** предписывает создавать неизменяемые компоненты, которые могут расширяться, если необходимо добавить новый функционал: система развивается путём дописывания нового кода, а не модификации старого.

При нарушении **OSCP** компоненты потребуются постоянно изменять каждый раз, когда требуется добавить новый функционал.

# LSP ← S. O. L. I. D.

L (LSP = Liskov Substitution Principle) «Принцип подстановки Барбары Лисков»: «объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы». Наследующий класс должен дополнять, а не изменять базовый. *Из определения подтипов Барбары Лисков, известного с 1988 года, следует, что для создания программных систем из взаимозаменяемых частей эти части должны соответствовать контракту, который позволяет заменять эти части друг другом.*

LSP предписывает не создавать подтипы, которые изменяют поведение надтипа.

При нарушении LSP создаются компоненты, кажущиеся взаимозаменяемыми, но фактически ведущие к трудно обнаруживаемым ошибкам.



# ISP ← S. O. L. I. D.

| (ISP = Interface Segregation Principle) «Принцип разделения интерфейса»: «много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения». *Этот принцип призывает разработчиков программного обеспечения избегать зависимости от всего, что не используется.*

ISP предписывает разрабатывать компоненты со специализированными интерфейсами, каждый из которых будет изменяться независимо.

При нарушении ISP создаются хрупкие компоненты, которые неизбежно потребуют изменения при необходимости добавить поведение, что нарушит совместимость с существующими вариантами его использования.



# DIP ← S. O. L. I. D.

**D** (DIP = Dependency Inversion Principle) «Принцип инверсии зависимостей»: «зависеть от абстракций, а не от конкретики». *Код, реализующий высокоуровневую политику, не должен зависеть от кода, реализующего низкоуровневые детали. Напротив, детали должны зависеть от политики.*

**DIP** предписывает управлять зависимостями между компонентами, направляя их от (чаще изменяемых) низкоуровневых к высокоуровневым (менее подверженным изменениям), что минимизирует необходимость в модификации системы.

При нарушении **DIP** изменения в малозначительных низкоуровневых компонентах неизбежно повлекут изменения в зависимых высокоуровневых компонентах.

# Парадигмы и их ценности

Структурное программирование даёт возможность *управлять сложностью*.

Объектно-ориентированное программирование даёт возможность *управлять зависимостями*.

Функциональное программирование даёт возможность *управлять состоянием*.

# Ценность S. P.

Структурное программирование:

- ограничение на прямую передачу управления (GOTO);
- все программы можно записать с помощью 3-х конструкций (последовательность, ветвление, цикл);
- модульное программирование.

Структурное программирование даёт возможность *управлять сложностью*, применяя функциональную декомпозицию: «разделять и властвовать», разбивая программу на модули разных уровней.

*Модульность должна быть «в крови» у разработчика.*

# Ценность F. R.

Функциональное программирование:

- функции без побочных эффектов;
- ограничение на изменение состояния (присваивание).

Функциональное программирование даёт возможность *управлять состоянием* для надёжного программирования, особенно при параллельных вычислениях.

# Ценность О. О. Р. для архитектуры

Объектно-ориентированное программирование:

- ограничение на косвенную передачу управления;
- инкапсуляция для управления видимостью;
- наследование для повторного использования;
- полиморфизм для управления зависимостями;

Объектно-ориентированное программирование даёт возможность *управлять зависимостями* (не зависеть от иерархии вызовов, инвертируя зависимости при помощи интерфейсов и полиморфизма).

# Архитектурные границы

Архитектурная граница проходит между компонентами разных уровней программной системы.

Разделение системы на уровни с управляемыми зависимостями позволяет:

- ограничить их влияние друг на друга (influence);
- независимо разрабатывать (develop);
- независимо тестировать (test);
- независимо разворачивать (deploy).



# Чистая архитектура

Зависимости должны быть направлены от низкоуровневых компонентов к высокоуровневым: менее важные части зависят от более важных, а не наоборот!

## Entities

(бизнес-правила  
и политики)

Более абстрактные,  
высокоуровневые,  
более важные, реже  
изменяемые части  
программной системы.

Архитектурная граница

Interactors.  
IO\_Port\_Interfaces.

## Use Cases

(логика приложения)

Более конкретные,  
низкоуровневые,  
менее существенные,  
чаще изменяемые части.

Gateways.  
Presenters.  
Controllers.  
Repositories.

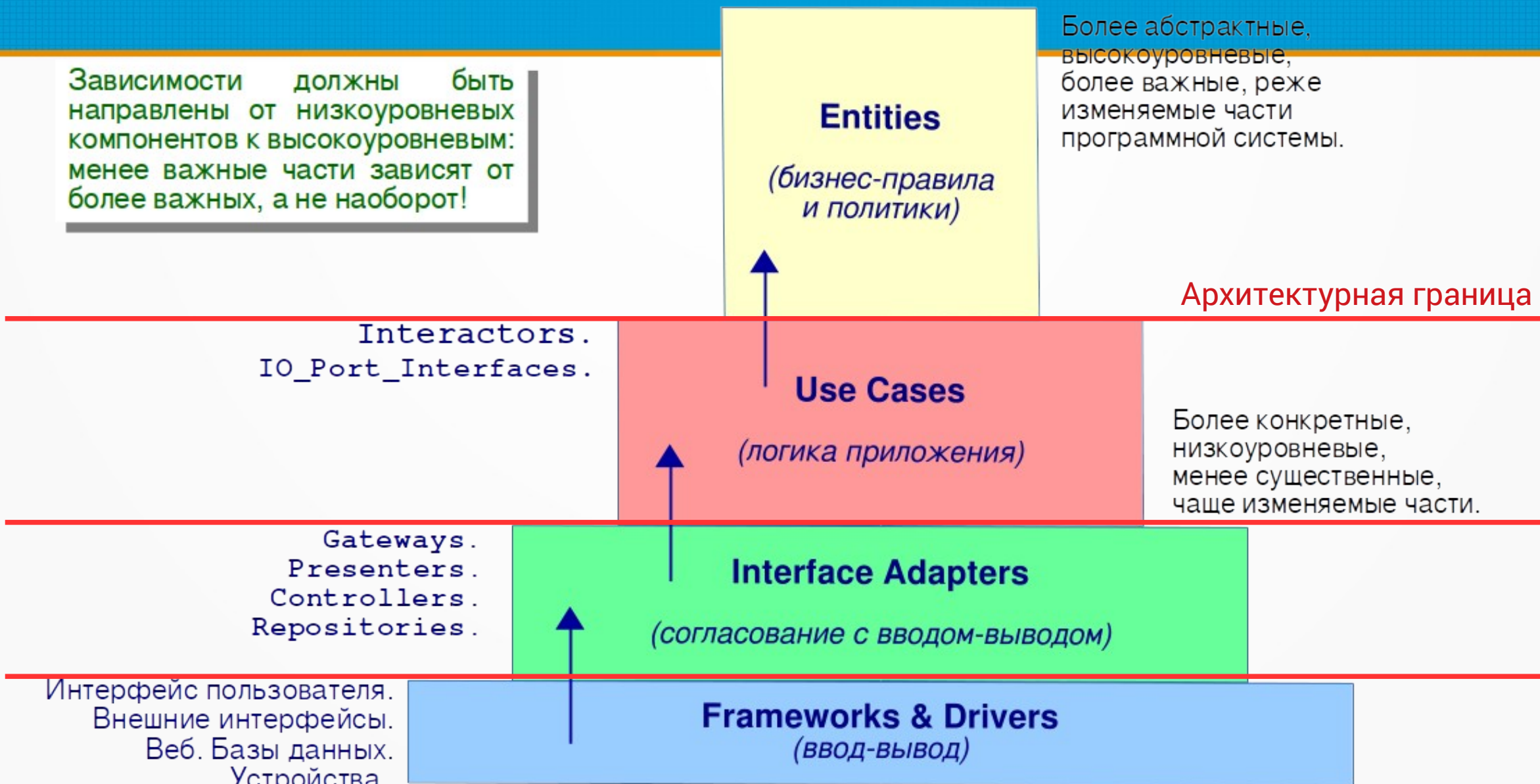
## Interface Adapters

(согласование с вводом-выводом)

Интерфейс пользователя.  
Внешние интерфейсы.  
Веб. Базы данных.  
Устройства.

## Frameworks & Drivers

(ВВОД-ВЫВОД)



# Зависимости

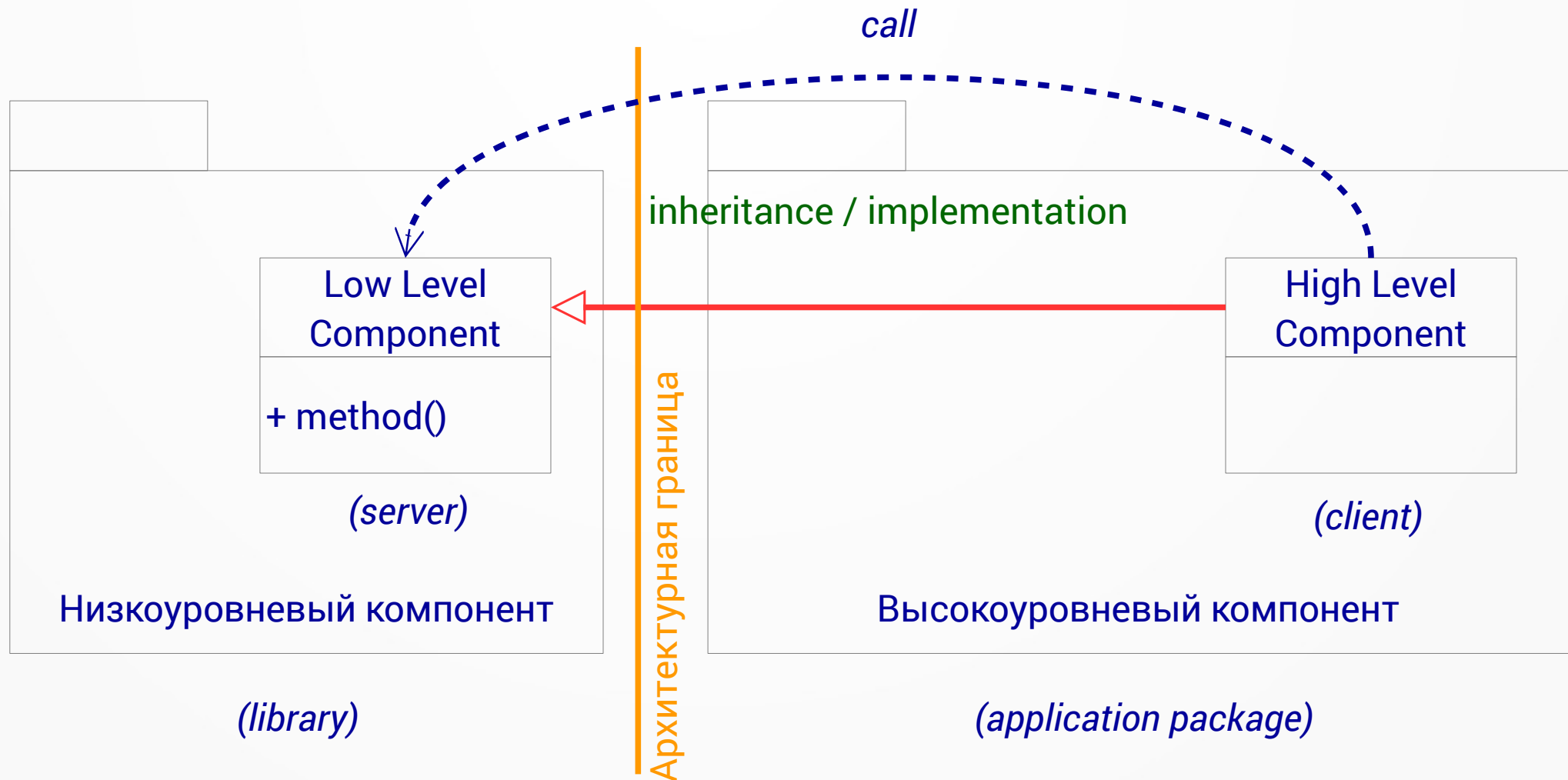
Зависимости:

- исходного кода (*между включающими и подключаемыми файлами*);
- потока управления (*между вызывающими и вызываемыми модулями*);
- между классами (*наследование, реализация*);
- в единицах развёртывания (*совместно расположенные модули*);
- 

Главное правило построения надёжной модифицируемой системы:

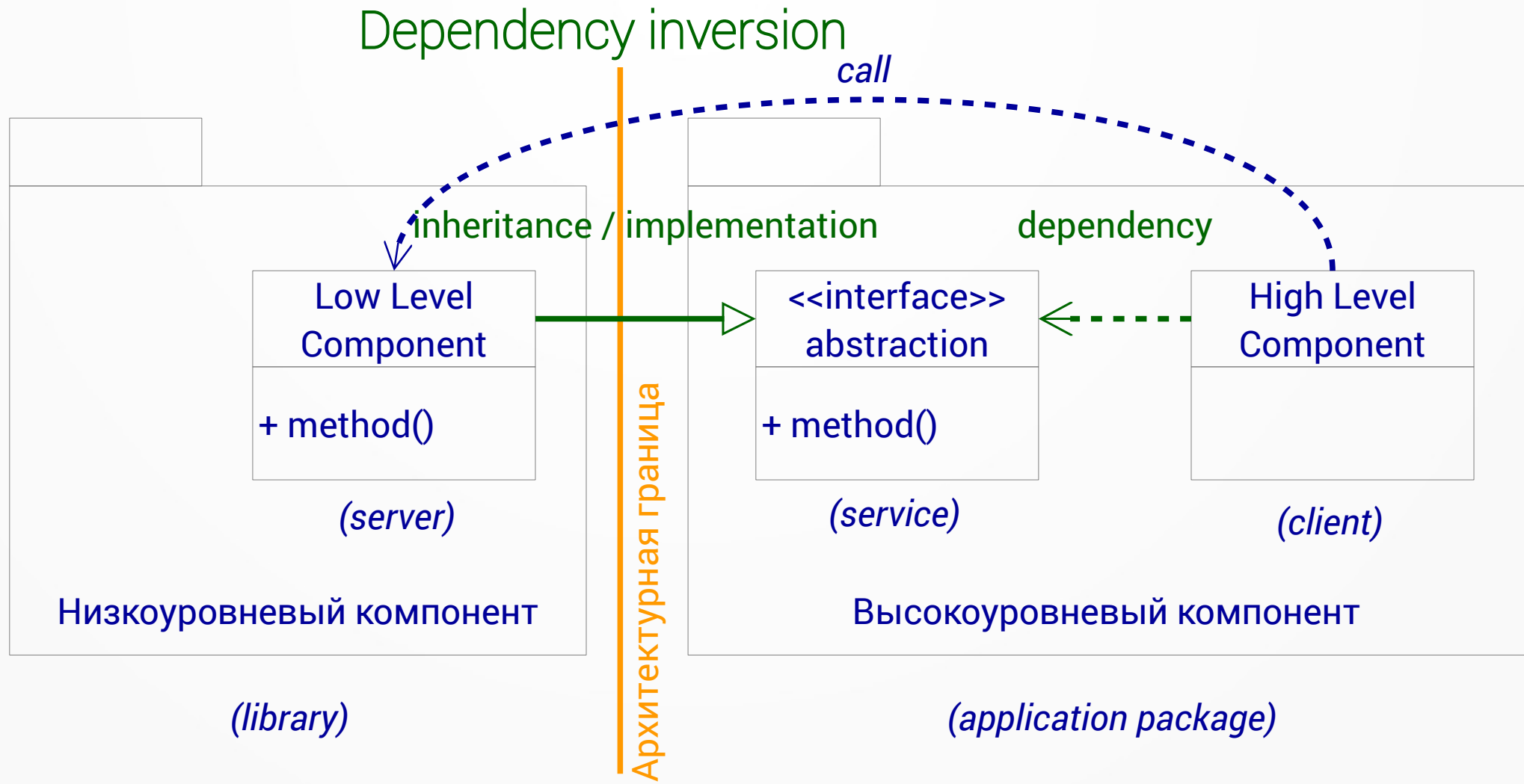
зависимости должны быть направлены от низкоуровневых компонентов к высокоуровневым: менее важные части должны зависеть от более важных, а не наоборот!

# Зависимость исходного кода



*Зависимость направлена от высокого уровня к низкому.*

# Инвертирование зависимости (D. I.)



*Зависимость направлена от низкого уровня к высокому.*

# Образцы проектирования

## Design Patterns

Образцы проектирования дают возможность обсуждать задачу на более высоком уровне абстракции и выбирать наиболее подходящий вариант проектирования для её реализации (пользуясь независимыми от языка структурами с известным поведением). Это примеры решения задач, которые можно использовать в подходящих ситуациях.

Их применение

- улучшает взаимопонимание при взаимодействии между программистами при разработке;
- даёт разработчикам возможность строить систему из крупных архитектурных блоков;
- помогает проектировать повторно используемый код.



# Классические образцы

## Классические образцы проектирования

из книги «*Design Patterns: Elements of Reusable Object-Oriented Software*»

### Порождающие:

1. Abstract Factory — Абстрактная фабрика
2. Builder — Строитель
3. Factory Method — Фабричный метод
4. Prototype — Прототип
5. Singleton — Одиночка

### Структурные:

6. Adapter — Адаптер
7. Bridge — Мост
8. Composite — Компоновщик
9. Decorator — Декоратор
10. Facade — Фасад
11. Flyweight — Приспособленец
12. Proxy — Заместитель

### Поведенческие:

13. Chain of Responsibility — Цепочка обязанностей
14. Command — Команда
15. Interpreter — Интерпретатор
16. Iterator — Итератор
17. Mediator — Посредник
18. Memento — Хранитель
19. Observer — Наблюдатель
20. State — Состояние
21. Strategy — Стратегия
22. Template Method — Шаблонный метод
23. Visitor — Посетитель

# Инструменты разработчика

## UML-схемы

для представления диаграмм в рабочей внутренней документации.

## Редактор / интегрированная среда разработки (IDE):

Code::Blocks, Eclipse, Geany, IntelliJ IDEA, Kdevelop, NetBeans, ...

## Система управления версиями исходного кода (VCS):

BitKeeper, Fossil, **Git**, GNU Bazaar, Mercurial, RCS, Subversion (SVN), ...

## Средство автоматизации тестирования:

FitNesse, Robot Framework, Selenium, **xUnit**, ...

## Система непрерывной интеграции (CI):

CruiseControl, GitLab, Jenkins, Travis CI, Vexor, ...

# Квалификация

Разработчик ПО должен знать и соблюдать принципы, владеть современными технологиями и инструментами, применять прогрессивные практики – всё это позволит стабильно разрабатывать программное обеспечение, которое будет надёжно работать, которое можно модифицировать и развивать на предсказуемом уровне затрат.

Этому надо учить при практической работе над проектом в команде.

*Нужно закладывать основы профессионализма при обучении новичков.*

# КНИГИ, которые нужно прочитать

1. Бек К. Экстремальное программирование = **Extreme Programming Explained: Embrace Change**. — СПб: Питер, 2002.
2. Бек К. Экстремальное программирование. Разработка через тестирование = **Test-Driven Development by Example**. — СПб: Питер, 2017.
3. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. Приемы объектно-ориентированного проектирования. Паттерны проектирования = **Design Patterns: Elements of Reusable Object-Oriented Software**. — СПб: Питер, 2015.
4. Метц С. Ruby. Объектно-ориентированное проектирование = **Practical Object-Oriented Design. An Agile Primer Using Ruby**. — СПб.: Питер, 2017.
5. Мартин Р. Быстрая разработка программного обеспечения = **Agile Software Development, Principles, Patterns, and Practices**. — М.: Вильямс, 2004.
6. Мартин Р. Идеальный программист = **The Clean Coder. A Code of Conduct for Professional Programmers**. — СПб: Питер, 2012.
7. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения = **Clean Architecture. A Craftsman's Guide to Software Structure and Design**. — СПб: Питер, 2018.
8. Мартин Р. Чистый код. Создание, анализ и рефакторинг = **Clean Code. A Handbook of Agile Software Craftsmanship**. — СПб: Питер, 2019.
9. Фаулер М. Рефакторинг. Улучшение проекта существующего кода = **Refactoring. Improving the Design of Existing Code**. — М.: Диалектика, 2019.

# Пожелания

Это необходимо осваивать.  
Этому нужно учить.

Даже, если сначала только некоторые вещи  
из того, о чём говорилось,  
будут преподаваться —  
качество подготовки разработчиков возрастёт!



# ГОТОВ ОТВЕТИТЬ НА ВОПРОСЫ

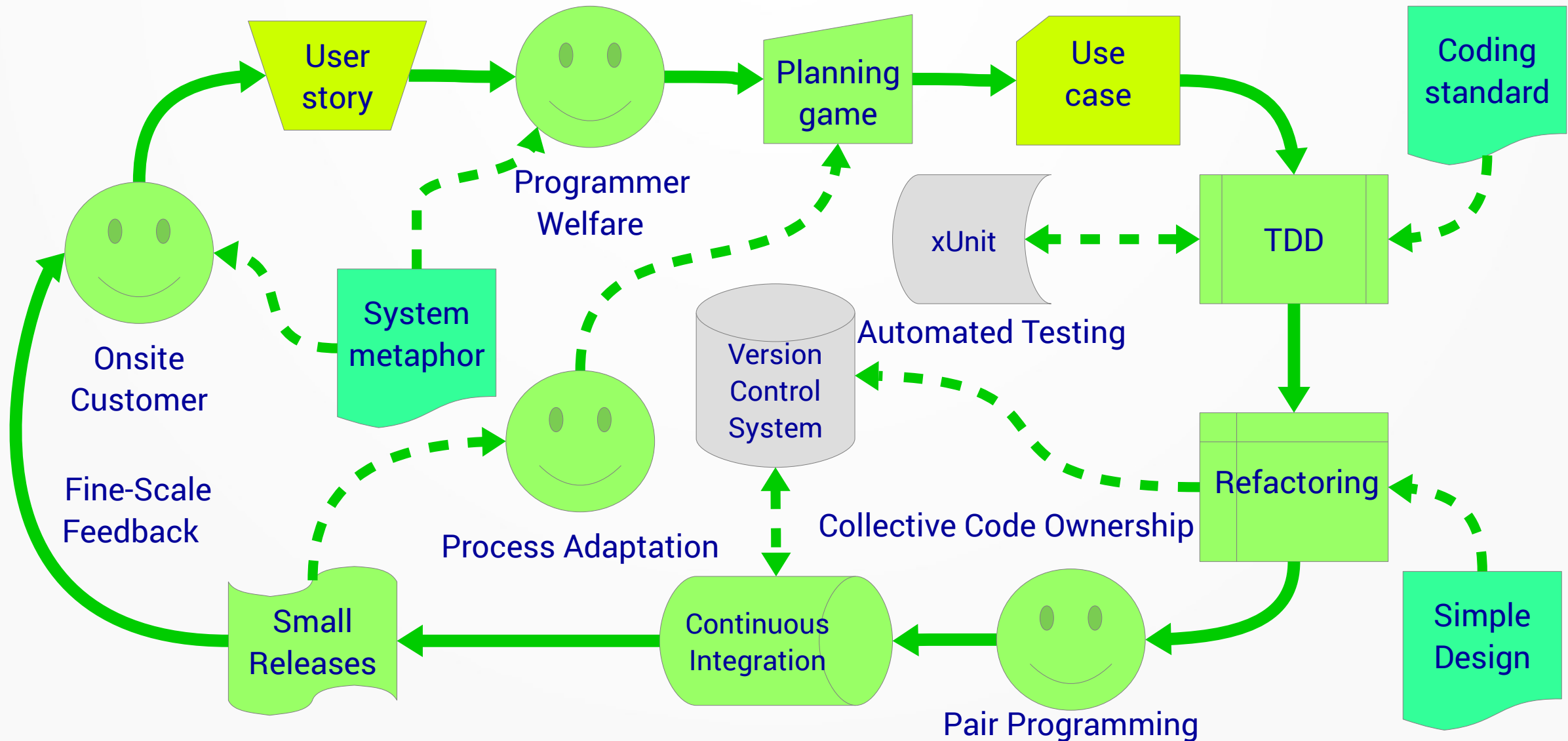
???

# Дополнения

- Схема Х. Р. на английском.
-

# eXtreme Programming

## Continuous Process



?

Что бы вы ещё желали обсудить?