Программное дело

о разработке программ расскажет
Михаил В. Шохирев

Клуб программистов
Шадринск
2020-22

Содержание

Прагматичный подход к разработке программ.

Разработка через тестирование.

Важность правильной архитектуры.

Образцы проектирования ПО.

Знания и умения разработчика ПО.

Необходимые инструменты разработки.

Термины-синонимы
(в этом рассказе):

- *программирование* ≈ разработка ПО;
- *программа* ≈ программная система ≈ система ≈ ПО;
- *модуль* ≈ подпрограмма ≈ метод;
- *тест* ≈ test case.

Ремесло программирования

Програмирование это инженерное ремесло: результат разработки ПО – практически полезная вещь, программа. Это такое же уважаемое ремесло, как гончарное, столярное, плотницкое, горное дело или радиоинженерия.

Во всяком ремесле настоящий мастер проявляет **творчество**, постоянно развивает свой **профессионализм** и отвечает за **результат** своего дела.

Любое профессиональное ремесло построено на неукоснительном соблюдении производственных правил и этических норм, на владении приёмами и навыками работы, которые основаны на теоретических знаниях.

Это и есть профессиональная культура.

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. Постепенно у меня сложилась картина, как следует разрабатывать добротные программы.

И мне захотелось поделиться с коллегами-одноклубниками.

Почему я захотел рассказать об этом?

Разрабатывается чрезвычайно много некачественных программ.
Многие программисты не имеют опыта разработки ПО.
В разработчики идёт масса 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 осознают, что успех разработки в правильной архитектуре системы.
Не пользуются необходимыми средствами разработки.
Не воспитывают в себе профессионализм.
Не имеют профессиональной культуры.
. . .

Несколько примеров непрофессионализма

– Пусть тестируют тестировщики, а я – программист, я пишу программы.
– Как? Ты не используешь этот замечательный фреймворк?!?!
– Так ты всё ещё пишешь на этом старом языке?!
– Наше ПО не поддерживает эту СУБД.
– Это же веб-приложение, здесь всё по-другому…
– Проще вставить SQL-запрос прямо в PHP-страницу.
– Понятно, почему не работает: у Вас не тот браузер!
– Для нашей системы нужен прямой выход в Интернет, без всяких прокси.
– Сделайте полный доступ в корень диска C: – и программа установится.
– Если не установлен Excel, отчёт формироваться не будет.
– Да, наша система работает только под Windows 10.
– Программа работает? Не трогай её, чтобы не сломалась!
– Эти пользователи!!! Они снова хотят, чтобы я переделал программу!
– Ну, не знаю... У меня на компьютере всё работет!

Причины низкого профессионализма

Этот рассказ – о культуре разработки, без которой нет профессионализма.

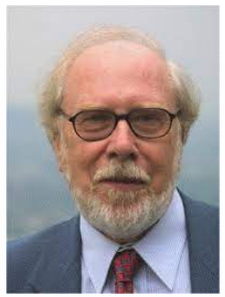
- Развитие ИТ требует большого количества разработчиков *(их количество удваивается каждые 5 лет).*
- 1/2 современных разработчиков ПО имеет опыт работы менее 5 лет.
- У них недостаточный контакт с носителями профессиональной культуры.
- Дефицит качественных образовательных ресурсов по разработке.
- Современными технологиями разработки пренебрегают в ВУЗах.
- У преподавателей недостаточно практического опыта разработки ПО.
- Коммерциализация разработки не заинтересована в качестве ПО.
- «Продаваны» управляют разработкой, не понимая её специфики.
- Пренебрежение к культуре оправдывается прагматичными целями.
- Профессионализм – это дисциплина, а её у многих как раз не хватает.
- Требуется личная мотивация для стремления к профессионализму.

Мои учителя


Brian Kernighan


Phillip Plauger


Glenford Myers


Niklaus Wirth


Dave Thomas


Andy Hunt


Kent Beck


Robert Martin

1. **Гленфорд Майерс**: «Надёжность программного обеспечения» (1980); «Искусство тестирования программ» (1982).
2. **Дэнни Ван Тассел**: «Стиль, разработка, эффективность, отладка и испытание программ» (1981).
3. **Никлас Вирт**: «Алгоритмы + структуры данных = программы» (1985).
4. **Брайан Керниган и Филип Плоджер**: «Инструментальные средства программирования на языке Pascal» (1985).
5. **Гради Буч**: «Объектно-ориентированное проектирование с примерами применения» (1992).
6. **Кент Бек**: «Экстремальное программирование» (2002).
7. **Дэйв Томас и Энди Хант**: «Программист-прагматик. Путь от подмастерья к мастеру» (2004).
8. **Роберт Мартин**: «Чистый код. Создание, анализ и рефакторинг» (2018); «Чистая архитектура. Искусство разработки программного обеспечения» (2018); «Чистый Agile. Основы гибкости» (2020); «Идеальная работа. Программирование без прикрас» (2022).

О чём пойдёт речь

Часть I.
(~20 минут)

Waterfall ~ *традиционный подход к разработке: поэтапный.*

Agile ~ *адаптивный подход к разработке: эволюционный.*

X. P. ~ *“экстремальное программирование”.*

Часть II.
(~50 минут)

Software ~ *2 взаимосвязанные ценности программного обеспечения.*

Testing ~ *важность всеобъемлющего тестирования.*

TDD ~ *разработка через тестирование.*

Refactoring ~ *необходимость постоянной реорганизации исходников.*

Architecture ~ *ценность и важность архитектуры.*

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

Часть III.
(~20 минут)

FP, SP, OOP ~ *ценность парадигм программирования.*

Design Patterns ~ *признанные образцы проектирования ПО.*

Qualification ~ *знания и умения разработчика ПО*.

Tools ~ *необходимые инструменты разработки.*

Что нужно знать и уметь разработчику

Написать программу не сложно.
Труднее сделать, чтобы она работала правильно.
Ещё сложнее поддерживать исходный код “чистым”.
Важнее всего сделать программу, которую можно развивать при использовании.

Для этого надо овладеть современными подходами к разработке:

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

Максим Шафиров, JetBrains CEO

И неотступно соблюдать профессиональные правила и требования.

Традиционный подход к разработке

Waterfall

«Водопад»:

Требования: *определить и проанализироать требования заказчика*.

Проектирование: *формализованно описать и утвердить проект*.

Программирование: *написать программы по спецификациям*.

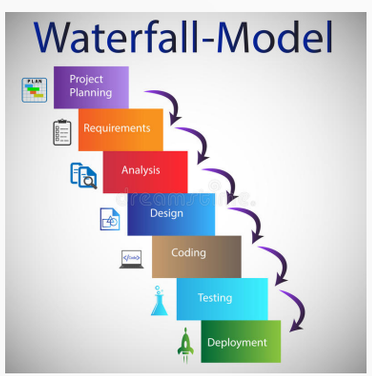
Отладка: *найти и исправить ошибки кодирования*.

Тестирование: *показать, что всё работает, как нужно*.

Документирование: *теперь можно написать её*.

Внедрение: *когда всё готово и протестировано.*

Сопровождение: *при эксплуатации.*



«Утвердить (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 на ходу*.

Отладка: *поэтому ошибки неизбежны*.

Тестирование: *недостаточное и несвоевременное*.

Документирование: *утомительное и после времени*.

Внедрение: *когда «поджимают сроки»*.

Сопровождение:

«Эта сказка хороша — начинай сначала!».

Ни зафиксировать, ни спокойно програмировать 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**.

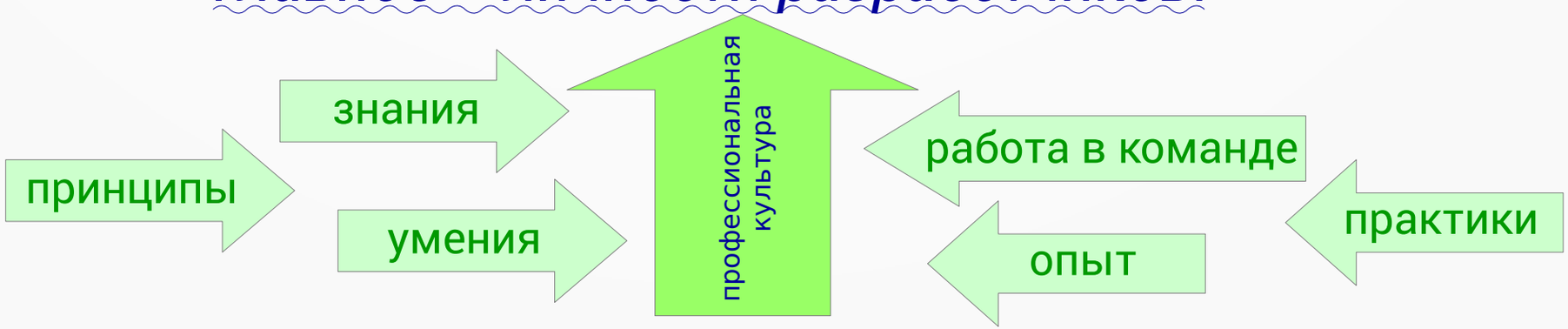
Это не просто провозглашённые принципы

— они давно и успешно применяются на практике!

Технологии: *гибкие* ←→ тяжеловесные

- *Адаптация* вместо предопределённости.
- *Наращивание* архитектуры, а не её полная проработка.
- Проектирование *эволюционное* вместо предварительного.
- Ясный *исходный код + тесты*, а не объёмная документация.
- Полезный *результат*, а не соблюдение утверждённого проекта.
- Делать только *необходимое*, а не .
- Ориентация на *человека*, а не на процесс разработки.

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 (*for web-based software*).
- Lean Software Development (LSD).
- OpenUP (← RUP).
- Rapid Application Development (RAD).
- Scrum (Sprint Continious Rugby Unified Methodology).

agile

~ “able to move quickly and easily”

- проворный, шустрый,
- быстрый;
- ловкий;
- подвижный, поворотливый;
- быстрореагирующий;
- сообразительный;
- манёвренный;
- энергичный;
- гибкий *(о разработке)*.

X. P. = eXtreme Programming*

Отдельная презентация по X. P.

4 основные положения XP :

I. **Короткий цикл обратной связи** (Fine-Scale Feedback = **FSF**): *за 1 итерацию разработки в 1-4 недели (оптимально 1-2 недели) много не напортачишь.*

II. **Непрерывный процесс** (Continuous Process = **CP**): *соблюдение правил (заказчиками и разработчиками) помогает безостановочно двигаться к цели с известной скоростью.*

III. **Понимание, разделяемое всеми** (Shared Understanding = **SU**): *минимизирует ошибки и сокращает время взаимодействия в команде.*

IV. **Социальная защищённость программиста** (Programmer Welfare = **PW**): *стабильная ритмичная разработка без авралов и переработки.*

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

X. P. = eXtreme Programming

12 основных приёмов XP :

I. **Короткий цикл обратной связи** (Fine-Scale Feedback): *за 1 итерацию разработки в 1-4 недели (оптимально 1-2 недели) много не напортачишь.*

1. **Заказчик всегда рядом** (Onsite Customer = Whole Team = **WT**): *конечный пользователь (product owner) всегда на связи для вопросов и уточнений; он одобряет план на итерацию.*
2. **Игра в планирование** (Planning Game = **PG**): *направляет разработку продукта через планирование объёма работ и установку приоритетов для пользовательских историй.*
3. **Разработка через тестирование** (Test-Driven Development = **TDD**): *весь код покрыт тестами, код качественный, ему можно доверять; требования выполнены.*
4. **Парное программирование** (Pairing = Pair Programming = **PP**): *коллеги работают совместно, знают код, заменяют друг друга ← **CCO**. → Collaborative programming.*

X. P. = eXtreme Programming

12 основных приёмов XP :

II. **Непрерывный процесс** (Continuous Process): *соблюдение правил помогает равномерно двигаться к цели по графику.*

5. **Рефакторинг** (Refactoring = Design Improvement = **DI**): *обязательная реструктуризация кода (без изменения его поведения) поддерживает исходники “в чистоте”.*
6. **Частые небольшие релизы** (Small Releases = **SR**): *помогают пользователю оценить функциональность, видеть прогресс, вносить изменения, менять приоритеты.* ← **CI**.
7. **Непрерывная интеграция** (Continuous Integration = **CI**): *частое тестирование и сборка проекта, до нескольких раз в день; готовность к развёртыванию.*

Приёмочное тестирование (Acceptance Tests = **AT**): *автоматизированное тестирование ожидаемой функциональности по тестам заказчика / аналитика.*

X. P. = eXtreme Programming

12 основных приёмов XP :

III. **Понимание, разделяемое всеми** (Shared Understanding): *минимизирует ошибки и сокращает время взаимодействия в команде и с заказчиком.*

8. **Метафора системы** (Metaphor = System Metaphor = **SM**): *система однозначно понимаемых терминов, единая для заказчиков и разработчиков* → **DDD** ← **CC**.
9. **Простота проектирования** (Simple Design = **SD**): *выбирается наиболее простой способ реализовать функциональность.*
10. **Стандарт оформления кода** (Coding Standard = Coding Conventions = **CC**): *соблюдаемые* всеми *соглашения о стиле и именованию* → **SM**.
11. **Коллективное владение кодом** (Collective Ownership = Collective Code Ownership = **CCO**): *все разработчики знают исходный код, могут заменять друг друга* ← **PP**.

X. P. = eXtreme Programming

12 основных приёмов XP :

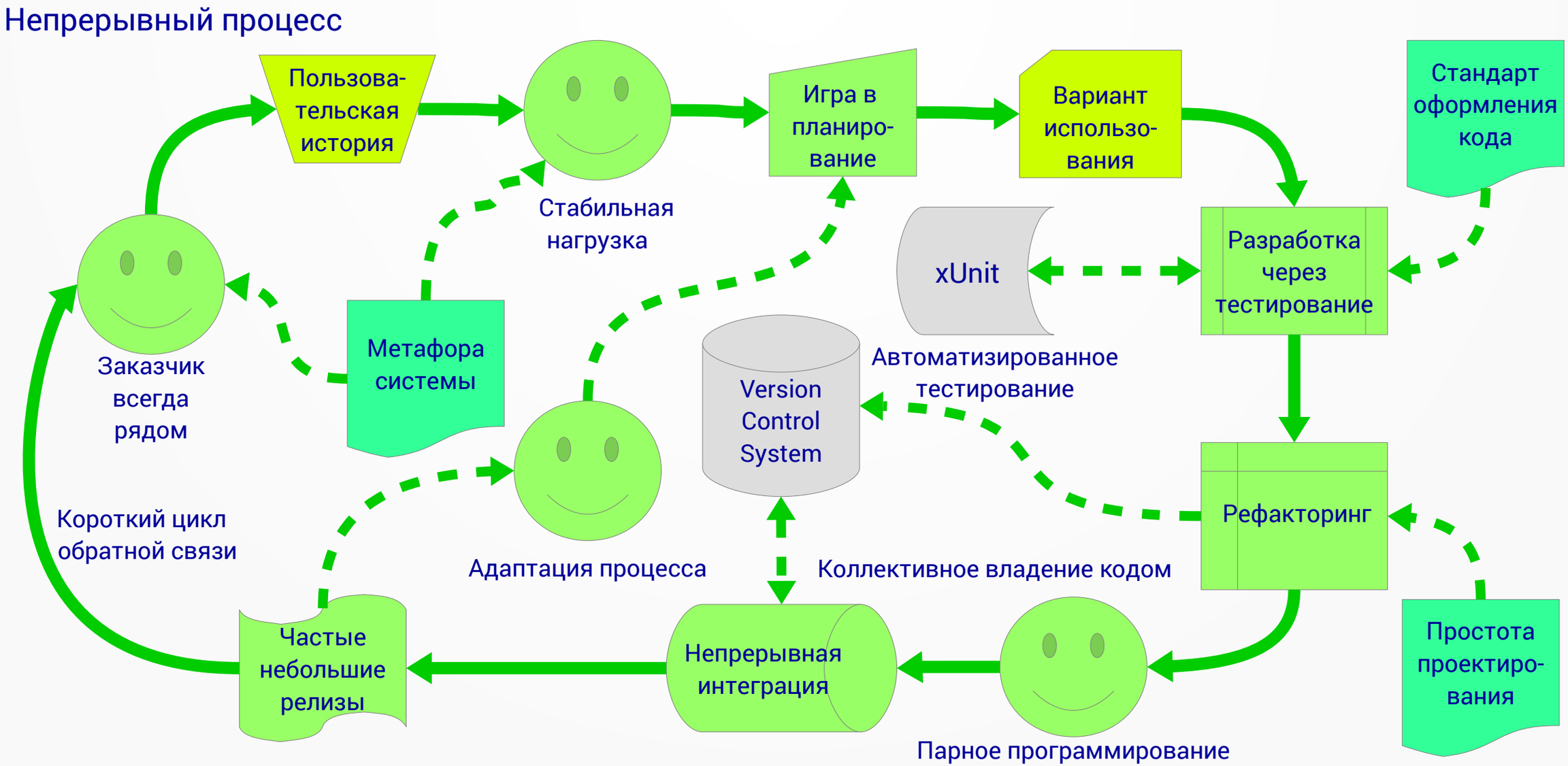
IV. **Социальная защищённость программиста** (Programmer Welfare): *стабильная ритмичная разработка без авралов и переработки.*

12. **Стабильная нагрузка** (Sustainable Pace = **SP**): *40-часовая рабочая неделя (40-hour week), стабильная работоспособность, уверенное планирование.*

Метод “вчерашняя погода”. Команда говорит: “За последнее время мы делали 4-6 фич в неделю, какие 4-6 фич мы будем делать на следующей неделе?” Владелец продукта (product owner) должен выбрать, какие именно пользовательские истории будут реализованы на этой неделе.

Ограничение количества задач метадами WIP-лимитирования: «Сделай больше в будущем, делая меньше прямо сейчас».

Экстремальное программирование



Польза от X.P. *(очевидная)*

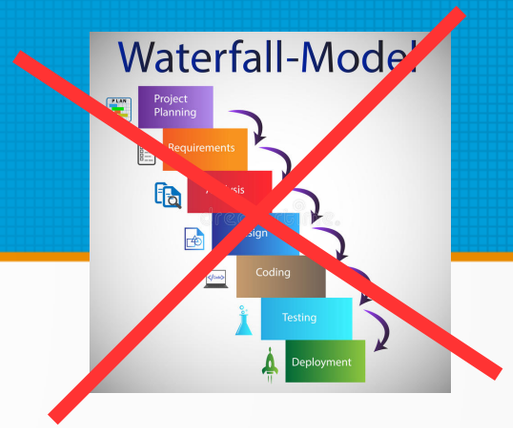
Для заказчика / руководства

- Стабильный темп разработки.
- Уверенное планирование.
- Соблюдение сроков разработки.
- Время модификации и стоимость изменений пропорциональны объёму требований.
- Эффективное управление процессом разработки: пользователь видит прогресс, оценивает новую функциональность, оперативно вносит изменения.

Для разработчиков

- Уверенность в качестве кода.
- Полноценная замена временно отсутсвующих разработчиков.
- Готовность кода для развёртывания.
- Социальная защищённость программиста.
- Ритмичная разработка без авралов и переработки.
- Стабильная нагрузка на работников.
- Условия для работоспособности и творчества.
- 40-часовая рабочая неделя.

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 при TDD.*
- Тестирование: *приёмочное — показать, что требования выполнены.*
- *Минимальное* документирование: *т. к. есть “чистый код” и тесты.*
- Внедрение: *в конце каждой итерации продукт готов к работе.*

Сопровождения *нет – есть только разработка!*

Заказчик хочет развития программы – разработчик её адаптирует!

Адаптивный подход к разработке

Суть Agile **в том,**
чтобы воспитать *команду разработчиков*,
вместе способных эффективно создавать продукт,
адаптируя непрерывный процесс разработки
к изменяющимся условиям и требованиям
(как это и всегда бывает).

Agile – это средство разработчика,
а не средство руководства.

Профессионалы в команде

1. Делать именно то, что нужно

Столкнувшись с запутанными проблемами, ценные сотрудники думают о потребностях своей команды. Они выходят за пределы назначенной им задачи и берутся за ту работу, которую надо сделать на самом деле, сосредоточивать усилия там, где это наиболее полезно..

2. Вести и уступать

Когда ясно, что надо что-то делать, но неясно, кто за это отвечает, ценные сотрудники вмешиваются и возглавляют процесс, а когда задача выполнена, уступают руководство и столь же легко следуют указаниям. Готовность и вести за собой, и подчиняться создаёт в организации культуру смелости, инициативности и гибкости.

3. Доводить до конца

Ценные сотрудники не уступают и полностью завершают работу без постоянного надзора, даже если становится сложно и путь усеян непредвиденными препятствиями.

4. Спрашивать и корректировать

Ценные сотрудники быстрее коллег адаптируются к изменчивым условиям, потому что видят в новых правилах и целях шанс учиться и развиваться с учётом критики. Попутно они способствуют обучению и новаторству в коллективе, помогают организации не отстать от жизни, приобретают репутацию легко обучаемых людей, которые совершенствуются сами и служат примером для всей команды.

5. Облегчать работу другим

Они помогают другим нести груз, но не потому, что берут на себя чужие задачи, а потому, что с ними легко работается. Благодаря им в коллективе появляются жизнерадостность и невозмутимость, становится меньше драматизма, политиканства и стресса, дело спорится. Они создают позитивную, продуктивную рабочую атмосферу, укрепляют культуру сотрудничества, становятся людьми, с которым хочется иметь дело.

Soft + ware

2 ценности программного обеспечения:

2 **soft-**
«изменчивый»

Структура – возможность свободно модифицировать систему при её адаптации и развитии;

реализуется в структурном проекте программной системы,
в её архитектуре.

«программу легко изменить»

1 **-ware**
«продукт»

Поведение – функциональность, позволяющая получить нужный результат;

реализовано в алгоритмах и взаимодействии программных модулей системы при выполнении.

«программа работает правильно»

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

Программа жива, если может меняться

*Заказчики хотят менять **всё**:*

- *выходные отчёты;*
- *форматы данных;*
- *структуру информации;*
- *функциональность программы;*
- *алгоритмы обработки;*
- *используемые 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

“В начале разработки вы можете быть уверенными только в одном – что вы делаете совсем не тот продукт.

Ни один продукт не выдерживает столкновения с пользователем. Как только вы дадите продукт в руки пользователя, вы обнаружите, что сделанный вами продукт не устраивает его по сотне разных причин. И если вы не можете изменить его, не превратив в кашу, то вы обречены”.

Роберт Мартин

Практически изменяемая программа

Программа, котрую **можно изменять**

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

Отдельный разговор про оценку сложности.

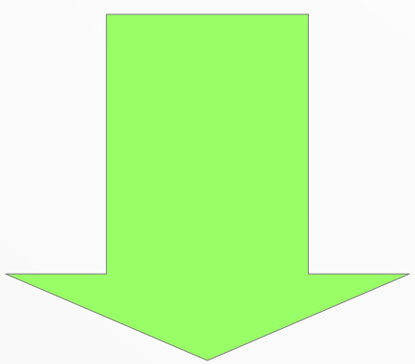
Для пользователя критически важна ***стоимость внесения изменений*** в программную систему (время, люди, деньги).

Продолжительность модификации и стоимость изменений должны быть пропорциональны объёму требований пользователя (соответствовать его ожиданиям)*.

* Заказчик ожидает, что незначительные изменения в программе не должны выполняться долго и стоить дорого.

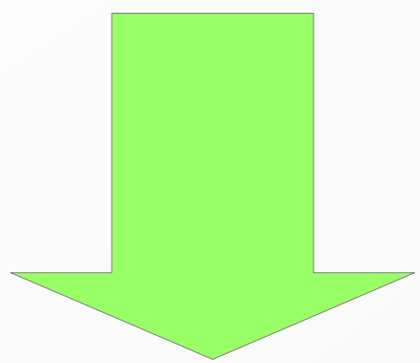
Условия для изменяемости

Уверенность, что программа продолжит правильно работать после изменения.



Надёжный набор тестов

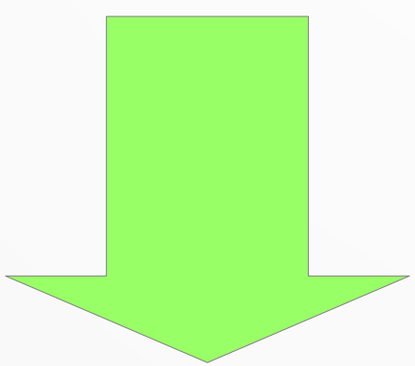
Программа имеет структуру, которая позволяет легко вносить изменения.



“Чистая архитектура”

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** на разработку.
... формальная **спецификация** разрабатываемой системы.
... лучшая низкоуровневая **документация** на программу.
... один из **способов использования** программной системы.
... **примеры** реального применения разрабатываемого кода.
... особый **API** к разрабатываемой системе.
... неотъемлемая **часть архитектуры** системы.

Эффективно тестировать возможно только, если тестирование автоматизтровано (automated testing).

Виды тестов *(основные)*

низкоуровневый тест

unit test

(модульный тест)

проверяет реализацию технического требования или решения разработчика;

*создаётся **программистом**, выполняется автоматически*

высокоуровневый тест

acceptance test

(приёмочный тест)

проверяет реализацию функционального требования заказчика;

разрабатывается тестировщиком, выполняется автоматически*

** пользователем, бизнес-аналитиком (BA) или специалистом по контролю качества QA (Quality Assurance)*

Польза и важность тестирования

Отдельная презентация по TDD

Имея надёжный набор (модульных) тестов, ...

... можно рефакторить программу без опаски её испортить.

... можно реорганизовать программу в любое время.

... можно смело править программные ошибки.

... можно спокойно модифицировать программу.

... можно менять программу любому из разработчиков.

... можно уверенно развивать функциональность программы.

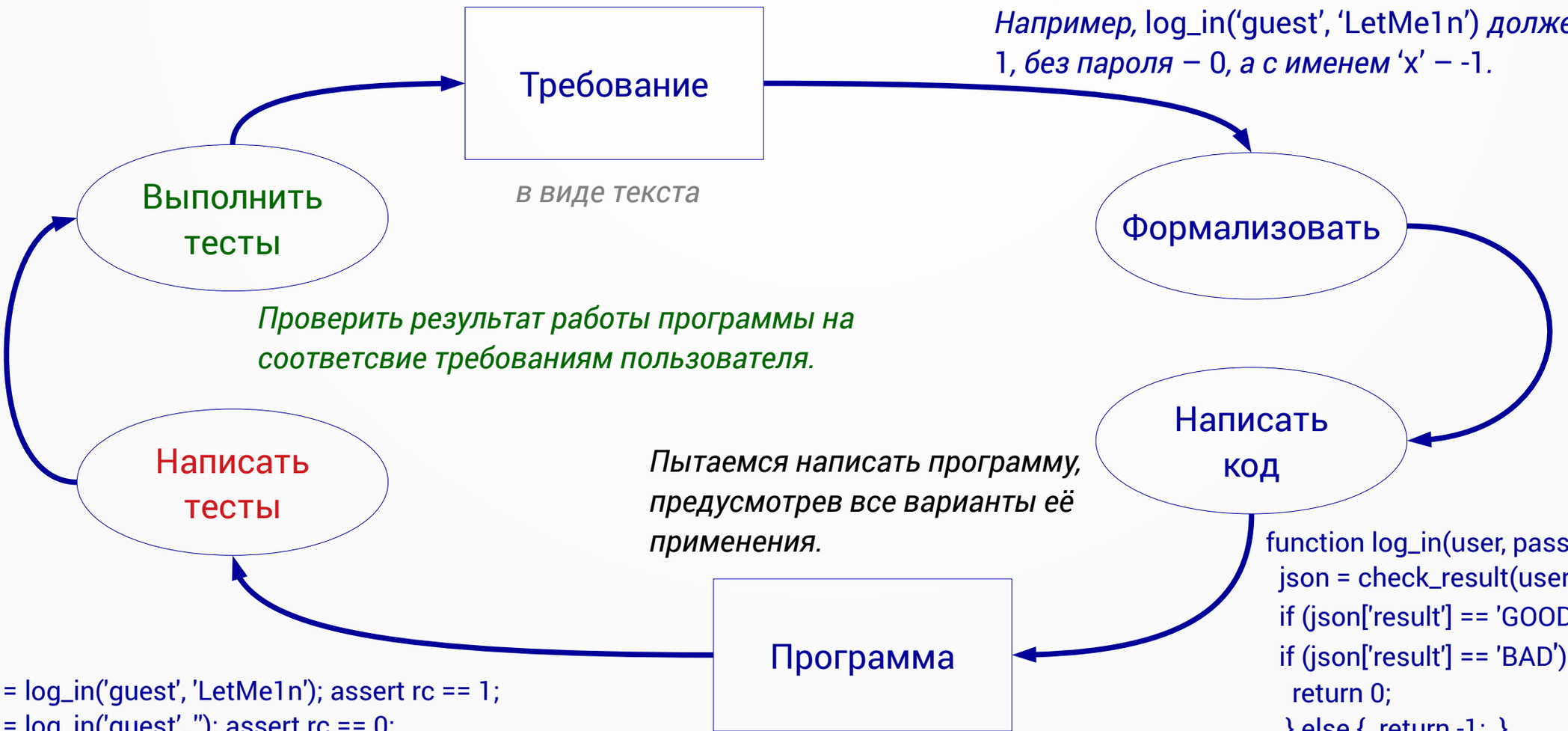
Имея полный набор тестов, можно даже воссоздать утерянный исходный текст программы *(ведь в тестах записаны все требования к тому, что она должна делать)*.

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

Code First ~ *обычная разработка*

“Пользователя нужно аутентифицировать по существующему имени и правильному паролю.”

Вызвать log_in() *с параметрами* user *и* password*, которая вернёт один из кодов возврата*: -1, 0, 1. *Например,* log_in(‘guest’, ‘LetMe1n’) *должен вернуть* 1, *без пароля* – 0, *а с именем* ‘x’ – -1.




```
function log_in(user, passwd) {
  json = check_result(user, passwd);
  if (json['result'] == 'GOOD') return 1;
  if (json['result'] == 'BAD') {
    return 0;
  } else {  return -1;  }
}
```

```
rc = log_in('guest', 'LetMe1n'); assert rc == 1;
rc = log_in('guest', ''); assert rc == 0;
rc = log_in('x', 'LetMe1n'); assert rc == -1;
```

~~Code First~~

Тестирование выполняется *после кодирования* (как что-то дополнительное и необязательное), поэтому программисты не осознают необходимости тестирования.

Непонятно, достаточно ли тестов написано и все ли требования проверены.

Можно только *надеяться* на сознательность, добросовестность и дисциплинированность программистов (что они протестируют хорошо).

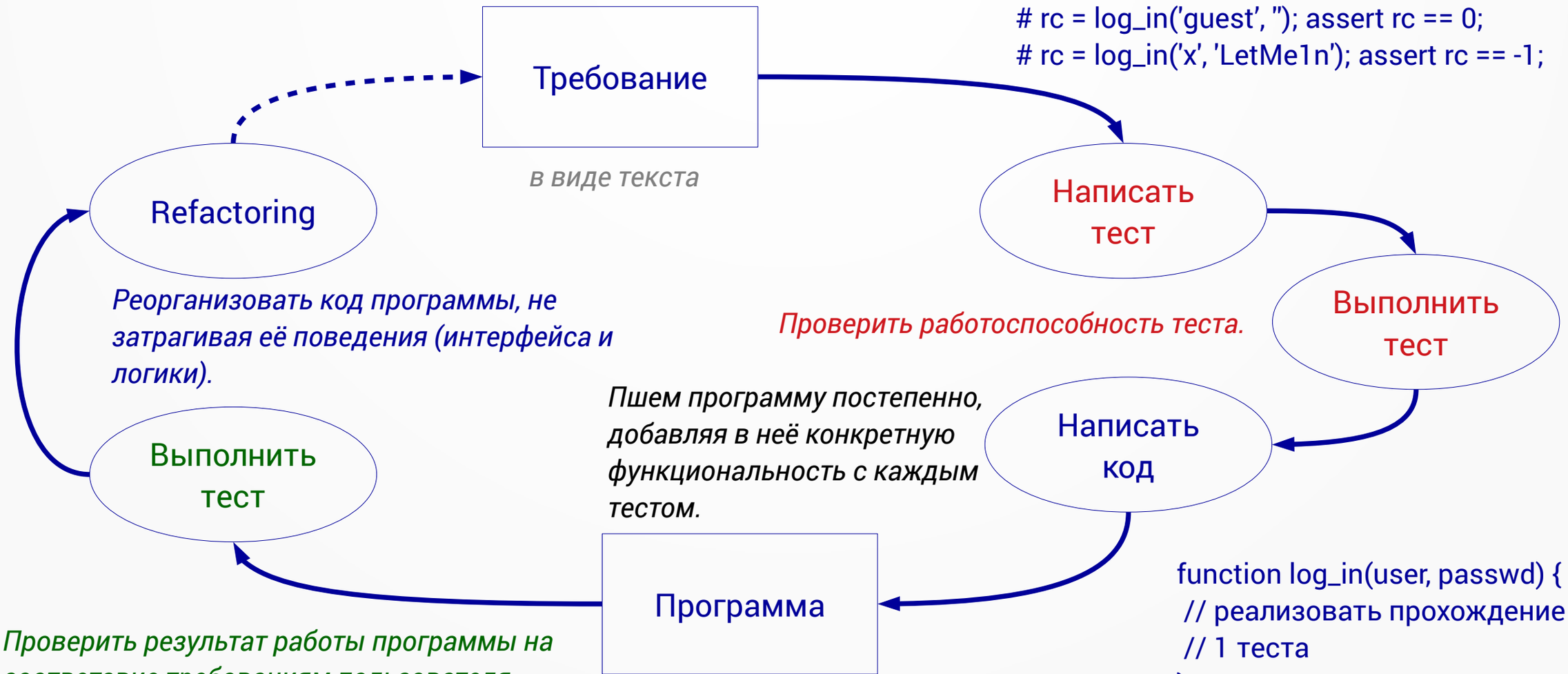
В результате набор тестов получается *недостаточный*, на него нельзя положиться при модификации системы.

При этом фактически тестирование проводят пользователи «на своей шкуре», когда сталкиваются с ошибками в работе программы!

Test First

“Пользователя нужно аутентифицировать по существующему имени и правильному паролю.”

```
# пишем тест для проверки 1 условия
rc = log_in('guest', 'LetMe1n'); assert rc == 1;
# rc = log_in('guest', ''); assert rc == 0;
# rc = log_in('x', 'LetMe1n'); assert rc == -1;
```




```
function log_in(user, passwd) {
 // реализовать прохождение
 // 1 теста
}
```

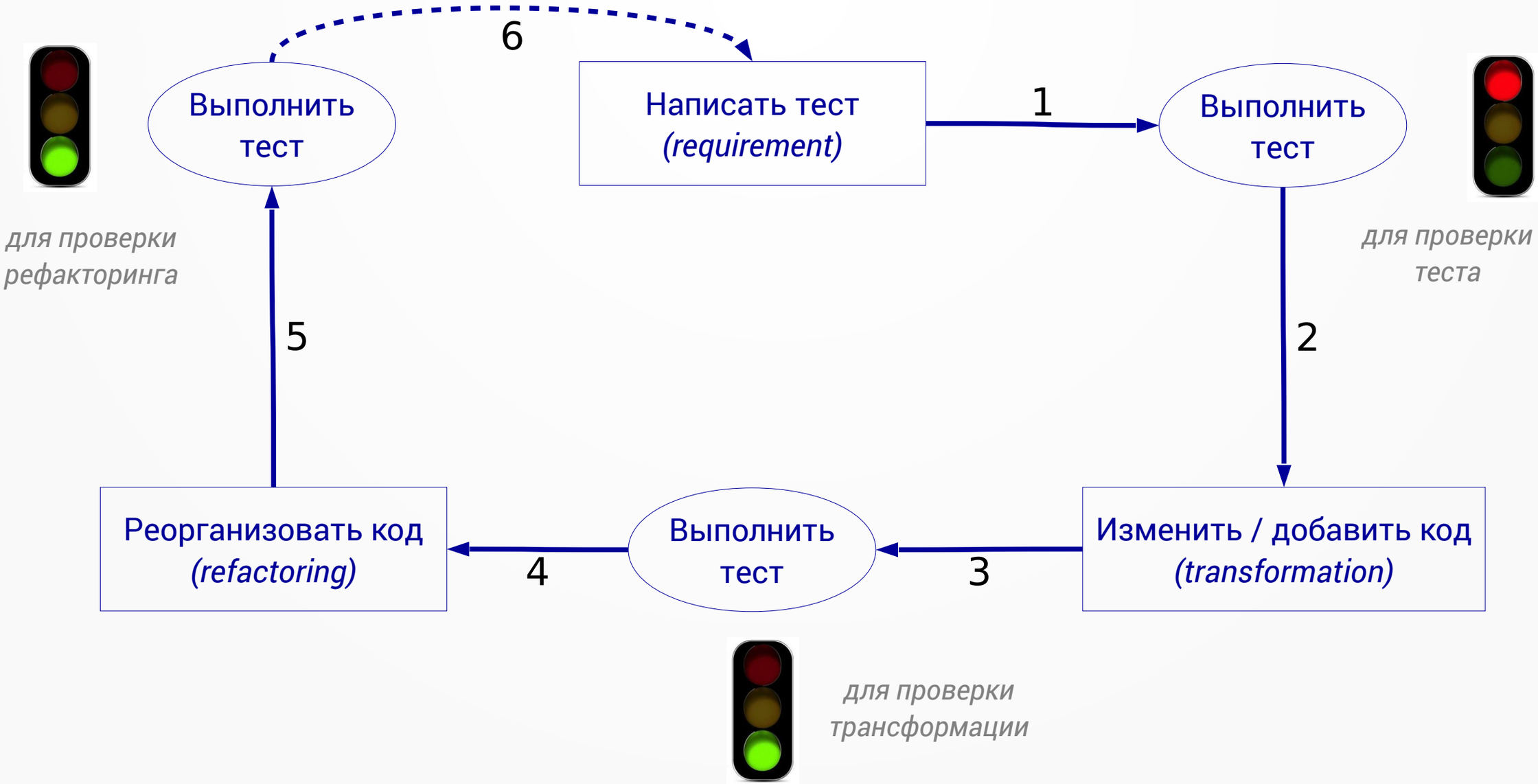
TDD = “Разработка, движимая тестами”

Сформулированное пользователем нечёткое общее требование (user story / use case), скорее всего, придётся разбить на несколько «атомарных» технических требований, более чётко сформулированных.

1. Разработка начинается, когда вы *записываете требование к функционалу* в формализованном и исполняемом виде: пишете **модульный тест** для подпрограммы (= модуля).
2. Разрабатывая тест, вы *определяете интерфейс тестируемой подпрограммы* (имя, параметры, возвращаемое значение).
3. Для проверки выполнения требования вы предусматриваете в тесте *сравнение возвращаемого результата с ожидаемым значением* (задаёте критерий правильности выполнения).
4. Выполнив тест для несуществующей подпрограммы, вы убеждаетесь, что ошибок в самом тесте нет. (Он отрабатывает, хотя показывает ошибочный результат, т. к. код ещё не написан.)
5. Затем вы пишете *минимальное тело тестируемой подпрограммы*, в котором реализуется это проверяемое требование (“How should we make that pass?”).
6. Выполнив тест после этого, вы убеждаетесь, что *реализация проходит проверку* (соответствует критерию).
7. Возвращаясь к тексту подпрограммы, вы *улучшаете его, не меняя интерфейса* (refactoring).
8. А затем *снова выполняете тест*, чтобы убедиться, что реорганизация не испортила подпрограмму.

9. Потом вы *пошагово пишете тесты* для каждого «атомарного» требования и трансформируете первоначальный исходник, чтобы постепенно *реализовать полный функционал*, соответствующий общему требованию, выполняя тесты на каждом шаге разработки.

Разработка, основанная на тестах



“Мантра” TDD *(уточнённая)*

0 1 2

Test → Red → *Transform* → Green → *Refactor* → Green

0. Сначала пишется тест для проверки требования:
   разработчик знает требование (какой должен быть результат).
   Прогон этого теста закончится неудачей, но тест уже разработан: требование записано в виде исполняемого кода (и интерфейс модуля определён).
1. Потом пишется *минимум* исходного кода:
   только, чтобы реализовать это требование.
   Готовый тест прогоняется до удачного окончания: это значит, код рабочий.
2. Затем написанный код реорганизуется:
   Начальный вариант кода структурно улучшается без изменения его поведения.
   Новый прогон теста должен показать, что ничего не сломалось.

Цикл повторяется снова… Набор тестов постепенно пополняется.

→ *Тестами охвачены все изменения.*

Test First = TDD = Test-Driven Development

Создание тестов является 1-м шагом в итерационной **разработке** модуля: сначала – определением его интерфейса, затем – записями требований к вариантам его использования. При этом *кодирование поведения* производится в ответ на сформулированное требование, записанное в виде теста.

TDD основано на сознательном неотступном соблюдении технологической дисциплины разработчиками (как двойная запись в бухучёте).

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

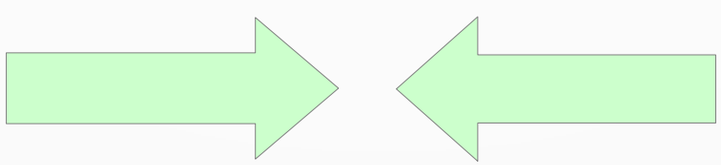
Применение TDD основано на автоматизированных модульных тестах (xUnit).

Refactoring & Transformation

2 вида преобразований кода при разработке программы:

Refactoring ~ последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований, изменение внутренней структуры программы, не затрагивающее её интерфейса, с целью облегчить понимание её работы, улучшить выразительность кода.

Transformation ~ последовательность изменений в поведении программы, не затрагивающих её интерфейс, с целью пошагово добавлять требуемую функциональность (которая даст результат записанный в тесте в виде ожидания).



Рефакторинг – это «чистка» кода.

Трансформация – это развитие кода.

Refactoring = Design Improvement

Отдельный разговор про виды рефакторинга

Refactoring = перепроектирование кода = переработка кода = равносильное преобразование алгоритмов ~ изменение внутренней структуры программы, не затрагивающее её поведения (интерфейса) 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 выразительности (clean code).

Рефакторинг — обязательная часть разработки и постоянная деятельность программиста.

Без регулярной «чистки» исходников
жёсткость, хрупкость, хаотичность изменяемого кода возрастает.

Transformation

Отдельный разговор про виды трансформации

Transformation ~ изменение поведения программы, не затрагивающее её интерфейс. Трансформация – это противоположность рефакторинга.
Путём пошаговых трансформаций “заглушка” эволюционирует до полноценной подпрограммы с требуемой функциональностью.
Разработка может идти по пути пошагового преобразования: сначала возвращается константа, затем – переменная, принимающая значение в зависимости от условия, один оператор может разбиться на несколько, один оператор заменяется вызовом подпрограммы, другой – рекурсивным вызовом, условие может преобразиться в выбор или цикл.

Любое изменение в исходном коде – это либо *трансформация* (изменение поведения от частного к более общему), либо *рефакторинг* (улучшение без изменения поведения).

Чем специфичнее становятся тесты, тем более обобщенным получается код.

Хороший тест

- Проверяет только одно требование.
- Тестирует логику через *интерфейс* модуля.
- Короткий (3 высказывания – ***Given-When-Then*** = AAA):

  Если *у меня имеется пустой стек,*
  Когда *я затолкну в него что-нибудь,*
  Тогда *его размер станет равен 1.*

```
Stack stack = makeEmptyStack();
stack.push(0);
assertThat(stack.size(), equalTo(1));
```

- Быстро выполняется.
- Не зависит от других тестов.
- Выполняется автоматически.

Тесты пишутся для проверки работы модуля с корректными и ошибочными данными (в штатной и нештатной ситуациях).

Для ошибочных данных должны быть свои технические требования (“Что делать в случае этой ошибки?”).

Критика TDD *(и возражения)*

- Писать тест до разработки рабочего кода – это нонсенс.
- Применение TDD очень замедляет разработку!
- TDD усложняет работу, сковывает творчество.
- TDD невозможно применять при работе над большими проектами.
- Руководство не одобряет TDD, не даёт на него дополнительное время.
- TDD неприменимо при разработке кода в области безопасности данных и взаимодействия между процессами.

- Создание теста – это уже начало разработки рабочего кода.
- TDD – важный этап работы, гарантирующий постоянный темп разработки.
- TDD организует деятельность разработчика и дисциплинирует его.
- TDD для разработки модулей дополняет технологии для создания архитектуры.
- TDD – одна из обязательных технологий разработки, не требующая одобрения.
- TDD при этом тоже применимо для разработки функциональности.
  Одно TDD не позволяет механически продемонстрировать адекватность такого кода; нужно дополнить его другими способами тестирования.

Преимущества применения TDD

- Разработка идёт путём реализации только 1 требования за раз.
- Каждое требование однозначно фиксируется в виде исполняемого теста.
- Набор тестов проверяет работу программы через её интерфейс, поэтому изменяется не часто.
- Разрабатывается достаточное количество тестов, чтобы проверить все требования – и не больше.
- Разработчик уверен, что каждое изменение в программе покрыто тестом.
- Такой набор тестов доказывает, что разработка идёт в соответсвии с требованиями.
- Тесты проверяют поведение модулей при корректных и ошибочных входных данных.
- После успешного прогона накопленного набора тестов программа находится в готовом состоянии.
- На такой набор тестов можно положиться при модификации и развитии программы.

Трудности при освоении TDD

непривычно

- Программисту психологически трудно перестроить своё сознание с привычного подхода «Code First» на технологию «Test First».

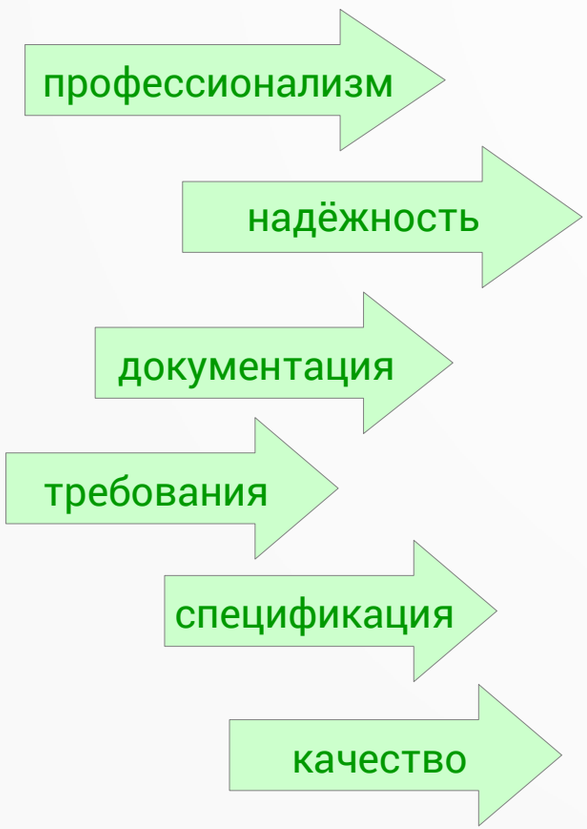
не постоянно

- Трудно неотступно практиковать TDD, особенно первые месяцы.

не осознано

- Разработчиков можно только убедить применять TDD, заставить – нельзя. Как и нельзя заставить писать хороший код. Никакие административные меры не помогут, а вызовут лишь озлобление.

Мотивация для применения TDD



- профессионализм не позволяет перекладывать тестирование на пользователей;
- нет другого способа получить надёжный набор тестов;
- набор тестов — средство документировать программу для всей команды;
- набор тестов — это формализованная запись всех требований к коду;
- это постоянный *контроль качества* для разработчиков;
- уверенность всей команды в качестве разрабатываемой программы;

Советы по освоению TDD

- TDD можно осваивать при отработке упражнений / этюдов / Code Kata.
- Внедрять TDD лучше всего на новом проекте (можно на учебном).
- Можно внедрять TDD, начиная с отдельных классов / модулей / библиотек.
- Не рекомендуется начинать внедрение TDD с больших коммерческих проектов.
- Намного проще внедрять TDD без унаследованного кода (legacy code).
- Идеально, если на этапе внедрения TDD нет жёстких ограничений по срокам.
- При разработке тестов составляется список требований / вариантов использования, из которого последовательно вычёркиваются уже реализованные.

TDD — это итеративная разработка: непрерывное, параллельное написание тестов и рабочего кода, с обязательными фазами рефакторинга для его чистки.

Тесты внешних интерфейсов системы

высокоуровневый тест

acceptance test

(приёмочный тест)

проверяет реализацию функционального требования заказчика;

разрабатывается тестировщиком, выполняется автоматически*

** пользователем, бизнес-аналитиком (BA) или специалистом по контролю качества QA (Quality Assurance)*

Тестирование

Структура любого теста – AAA (Arrange, Act, Assert):

- подготовка контекста выполнения, входных данных;
- выполнение тестовой программы;
- проверка соответствия результата критерию правильности.

Инструменты для автоматизации приёмочного тестирования:

FitNesse, JBehave, RSpec, SpecFlow, Cucumber, . . .

Табличное представление теста:

Требование	**Условие**	**Входные данные**	**Ожидаемый результат**	**Выходные данные**	**Проверка соответствия**
(что проверяется)	(контекст выполнения)	(параметры)	(эталон)	(результат выполнения)	(ожидание подтвердилось?)

BDD для приёмочного тестирования

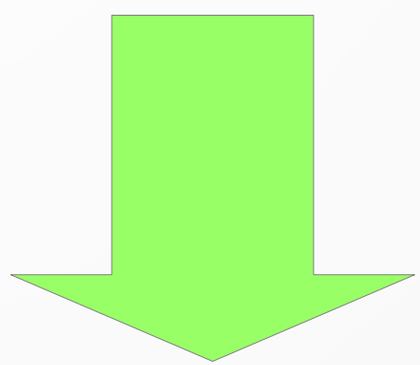
Для разработки приёмочных тестов (acceptance test) успешно применяется BDD (Behavior-Driven Development) – разработка, определяемая поведением.

BDD позволяет привлекать к разработке тестов специалистов по предметной области (заказчиков, пользователей, экспертов, аналитиков).

Инструментарий BDD (xBehave, xSpec, Cucumber, …) предлагает средства записи требований на специализированном языке (DSL), понятном пользователю, из которых автоматически генерируются приёмочные тесты.

Условия для изменяемости

Программа имеет структуру, которая позволяет лекго вносить изменения.



“Чистая архитектура”

Архитектура = design ≠ дизайн

Архитектура выражает замысел ПО, его назначение (*система заказов, учётная система, система мониторинга, игра и т. п.*) в виде структуры и взаимосвязей компонентов.

Функциональность строится на основе вариантов использования (use case ← user story) ПО (*размещение заказа пользователем, обработка заявки диспетчером, отбор объектов оператором, просмотр результатов игроком и т. п.*).

Помимо функциональности для программы проектируется её **структура**: состав компонентов, их взаимосвязи и способы взаимодействия.

Цель архитектуры программного обеспечения — уменьшить трудозатраты разработчиков при создании и последующем развитии системы (фундамент для изменений).

Что не является архитектурой ПО

~~Subscribe - Publish~~ = алгоритм взаимодействия.

~~SOAP~~ = протокол обмена.

~~REST~~ = архитектурный стиль = способ взаимодействия (I/O).

~~Framework~~ = инструмент / компонента ПО.

~~MVC (Model-View-Controller)~~ = образец проектирования.

~~Client — Server~~ = сетевая архитектура = организация взаимодействия.

~~Database Server~~ = хранилище = устройство ввода-вывода (I/O).

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

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

~~Microservices~~ = вариант SOA (I/O).

Архитектура: Simple Design

4 правила простого проектирования (Simple Design) Кента Бека:

1. **Covered by Tests**: *код покрыт тестами, все тесты проходят.*
2. **Minimize Duplication**: *устранено дублирование в исходниках.*
3. **Maximize Expression**: *код и тесты выражают намерения разработчика.*
4. **Minimize Size**: *классы и методы минимальны по размеру.*

Кент Бек часто повторял:

“First make it work, then make it right, then make it small and fast”.

<https://blog.cleancoder.com/uncle-bob/2013/09/23/Test-first.html>

Simple Design (X. P.)

1. **Covered by Tests** = Покрыть весь код тестами.

   Надо стремиться к 100% покрытию строк (line coverage) и покрытию ветвлений (branch coverage). Пригодный для тестирования код — это несвязанный код. Фактически тесты проверяют не только поведение, но и степень несвязанности.

2. **Minimize Duplication** = Минимизировать дублирование кода.

   Чем больше дублируется код, тем выше риск хрупкости. Надо исключать дублирование и управлять непреднамеренным дублированием (accidental duplication).

3. **Maximize Expression** = Максимизировать выразительность.

   В красноречивом (и простом) коде назначение переменных, функций и типов понятно по их именам, в нём легко увидеть структуру алгоритма. Код в совокупности с тестами показывает функцию каждого элемента системы и способ его применения.

4. **Minimize Size** = Минимизировать размер.

   После того, как прошли все тесты, максимально красноречиво проявлено предназначение кода и минимизировано дублирование, нужно поработать над уменьшением размеров кода без нарушения трех других принципов.

Архитектура: простота

Лучшая архитектура будет у самого простого проекта, который поддерживает всю необходимую функциональность, одновременно обеспечивая наибольшую гибкость для изменений.

> «Простой» означает «без запутанных взаимосвязей», «без неправильных зависимостей» между уровнями программной системы. → **S. O. L. I. D.**

Прагматичные принципы:

Делай проще, глупец (KISS = Keep It Simple, Stupid): выбирай наиболее простой способ реализовать функциональность.

Тебе это не нужно (YAGNI = You Ain't Gonna Need It): реализовать только необходимое, не закладывать излишнюю ("на будущее") функциональность.

"Сначала сделай, чтобы программа заработала, потом сделай её как положено": это 2 неразрывных действия в одной транзакции разработки (transform + refactor).

Плохая архитектура

Плохая структура делает систему жесткой, хрупкой и неподвижной. Это признаки плохого проекта (design smells).

Жёсткость (Rigidity): из-за незначительных изменений приходится повторно выполнять компиляцию, сборку и развертывание больших частей системы. Систему называют жесткой, когда усилия, затрачиваемые на интеграцию изменения, намного превосходят стоимость самого изменения.

Хрупкость (Fragility): незначительные изменения в поведении системы вызывают изменения в большом количестве модулей. В такой ситуации мы рискуем, меняя одно поведение, одновременно изменить какое-то другое. Фактически это означает отсутствие контроля над программным обеспечением — вы понятия не имеете, к чему приведут те или иные ваши действия.

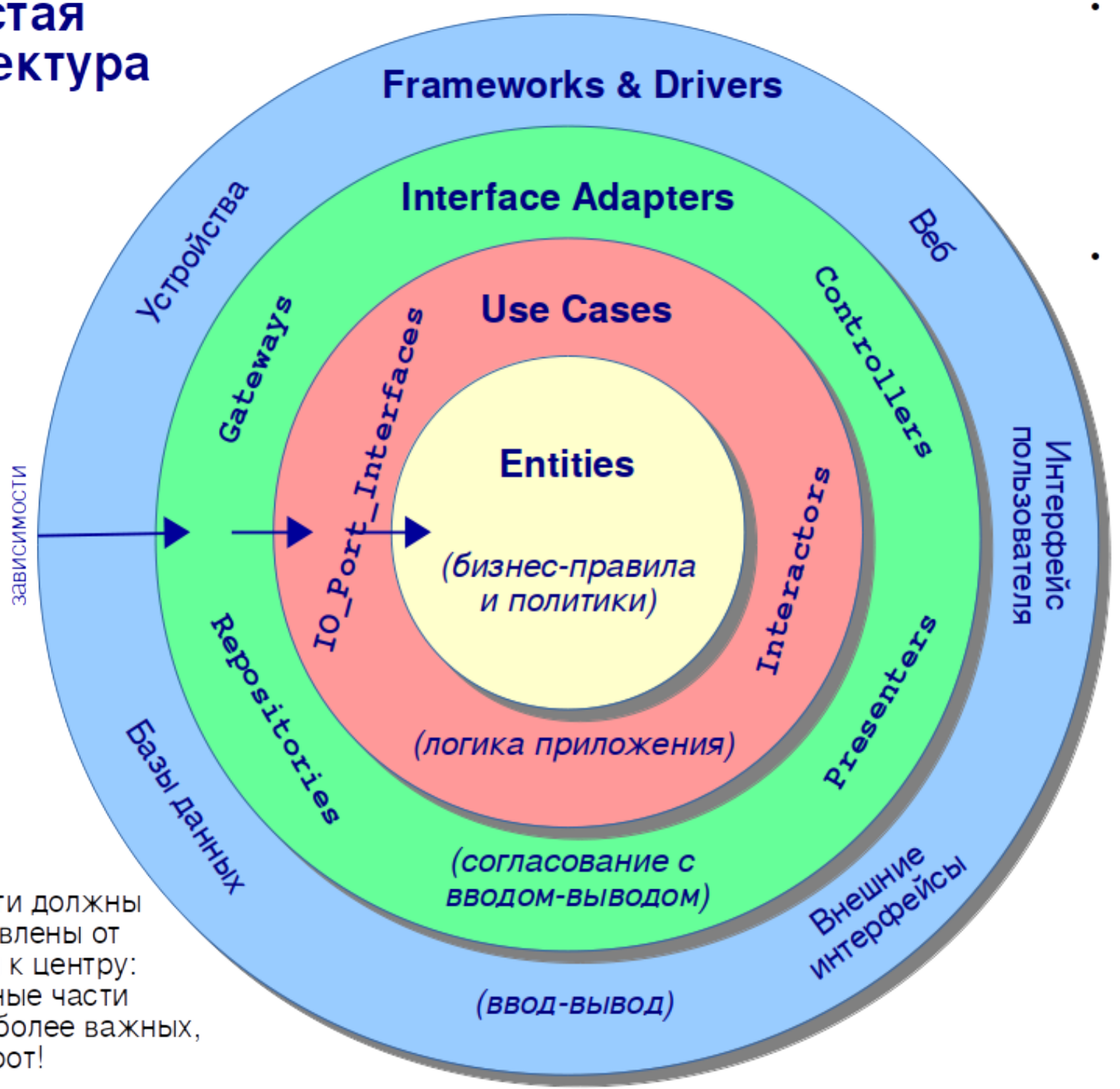
Неподвижность (Immobility): из-за запутанного кода вы не можете извлечь модуль с необходимым поведением для его использования в другой системе.

Clean Architecture

“Чистая архитектура” программной системы – такая, в которой предусмотрено **управление зависимостями** между основными уровнями:

- бизнес-правила и политики – то, что требуется, независимо от программной реализации (*как действия должны быть выполнены в в реальной жизни*);
- варианты использования – логика работы программной системы (*как действия выполняются с помощью программы*);
- согласование с вводом-выводом – интерфейсы и адаптеры (*как организовать предоставление 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.

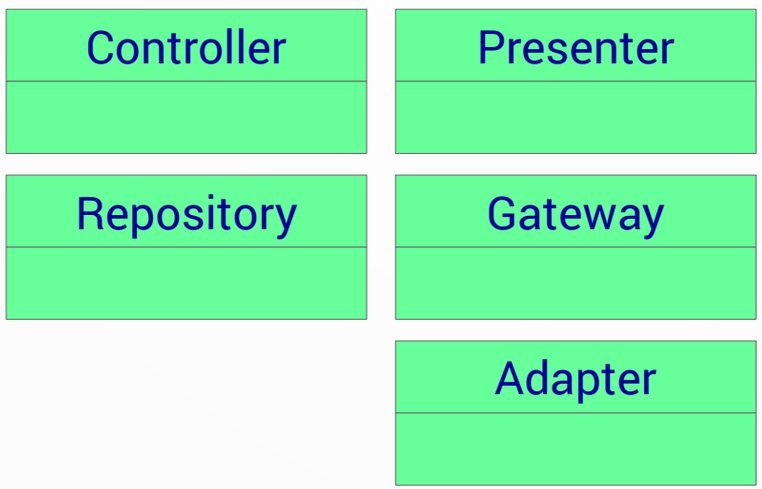
Зависимости должны быть направлены от периферии к центру: менее важные части зависят от более важных, а не наоборот!

Виды архитектурных компонентов

Ввод-вывод

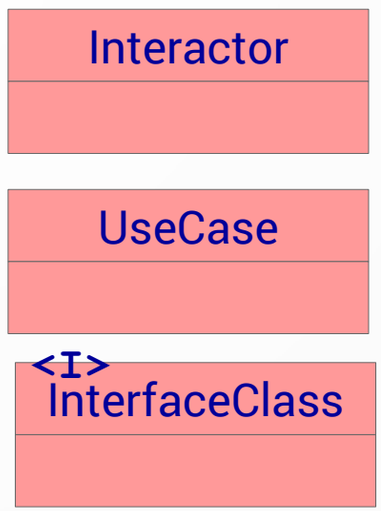
Классы для объектов, реализующих механизм **доставки данных**: источники и приёмники данных, средства взаимодействия с человеком и внешним ПО.

Согласование с вводом-выводом



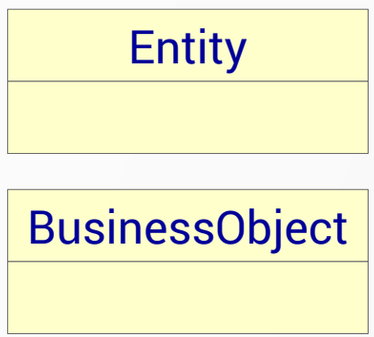
Классы для объектов, в которых реализуются интерфейсные программы и адаптеры, отвечающие за **представление данных** в нужном виде.

Варианты использования



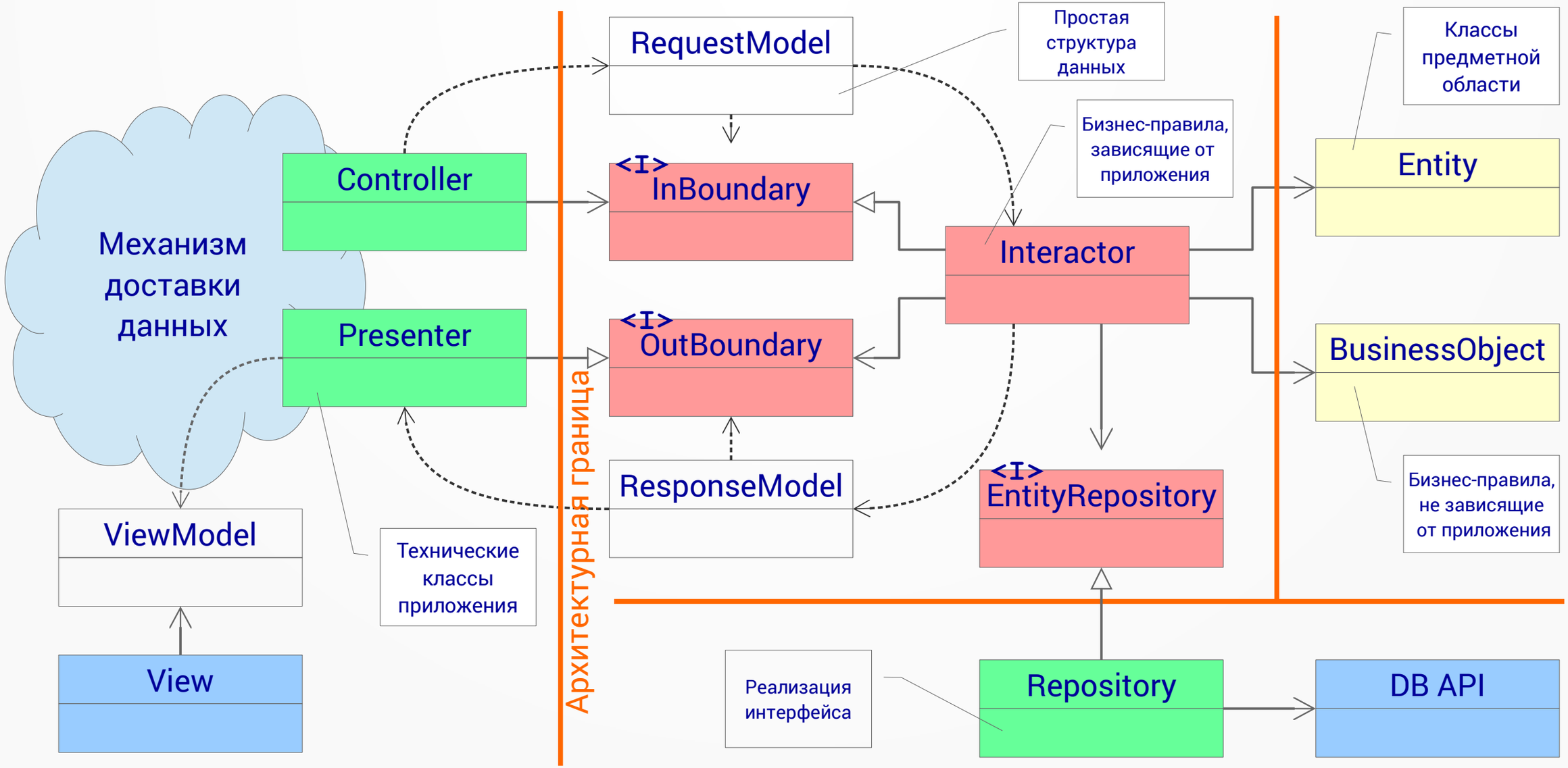
Классы для объектов, в которых содержатся алгоритмы **обработки данных** для реализации “пользовательских историй”.

Бизнес-правила и политики



Классы для бизнес-объектов предметной области, внешних или внутренних политик и правил обработки данных, не зависящих от программной реализации.

Взаимодействие компонентов



OOD = Object-Oriented Design

Компоненты должны правильно располагаться на нужных уровнях программной системы.

Именно OOD лучше всего может обеспечить для разрабатываемых компонентов:

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

Успешное **объектно-ориентированное проектирование** основывается на 5 основных принципах, известных как S. O. L. I. D.

S. O. L. I. D. – принципы OOD

Отдельная презентация по S.O.L.I.D.

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».

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 разные требования).

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

OCP ← S. O. L. I. D.

O (OCP = 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.*

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

При нарушении OCP компоненты потребуется постоянно изменять каждый раз, когда требуется добавить новую функциональность.

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.

I (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 нежелательные изменения в зависимых высокоуровневых компонентах.

Архитектурные границы

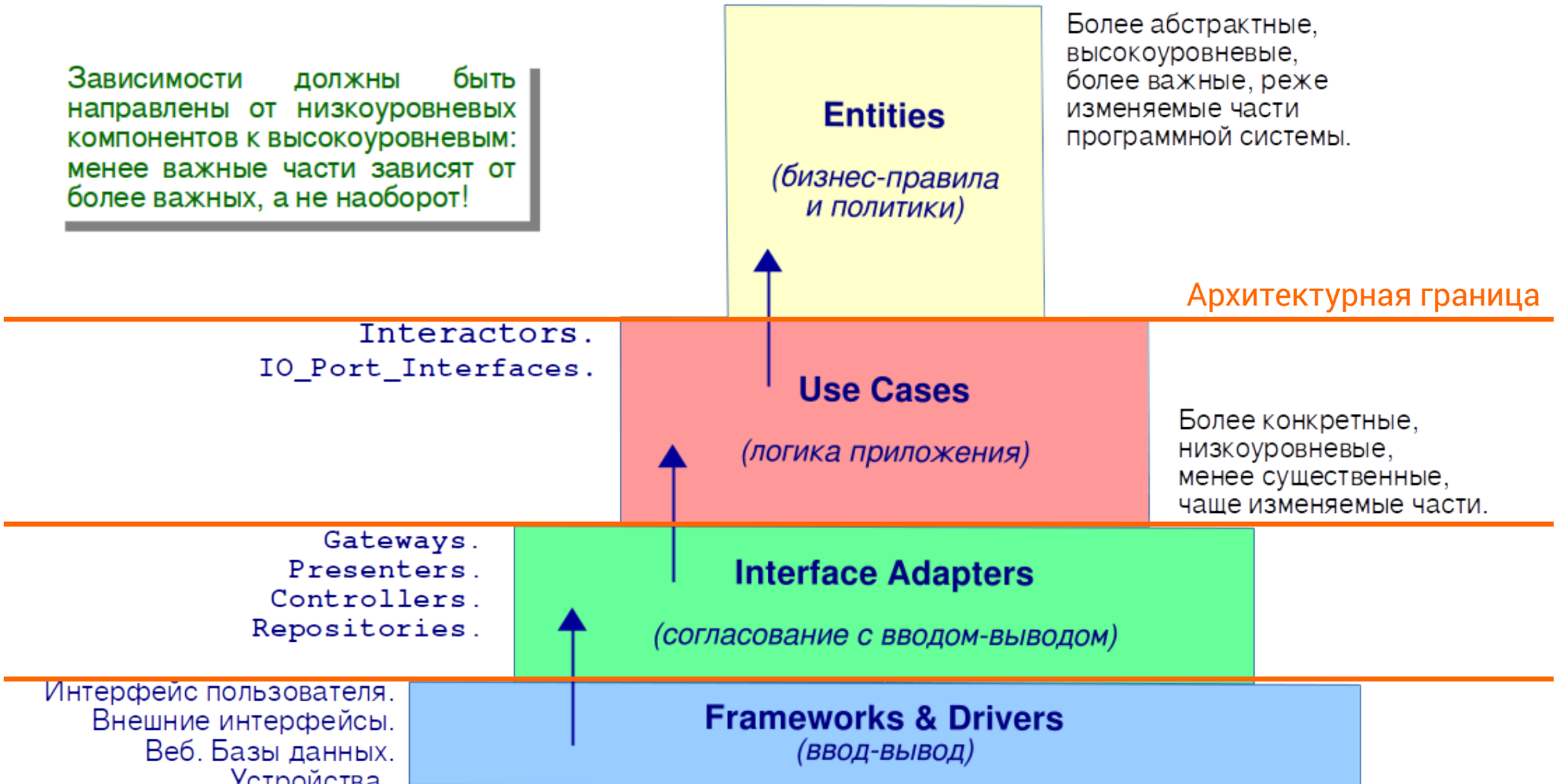
Архитектурная граница (components boundary) проходит между компонентами разных уровней программной системы.

Разделение системы
на *уровни с управляемыми зависимостями*
позволяет:

- ограничить их влияние друг на друга (influence);
- независимо разрабатывать (develop);
- независимо тестировать (test);
- независимо заменять (replace / upgrade).
- независимо разворачивать (deploy).
- независимо использовать (use).

Чистая архитектура

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



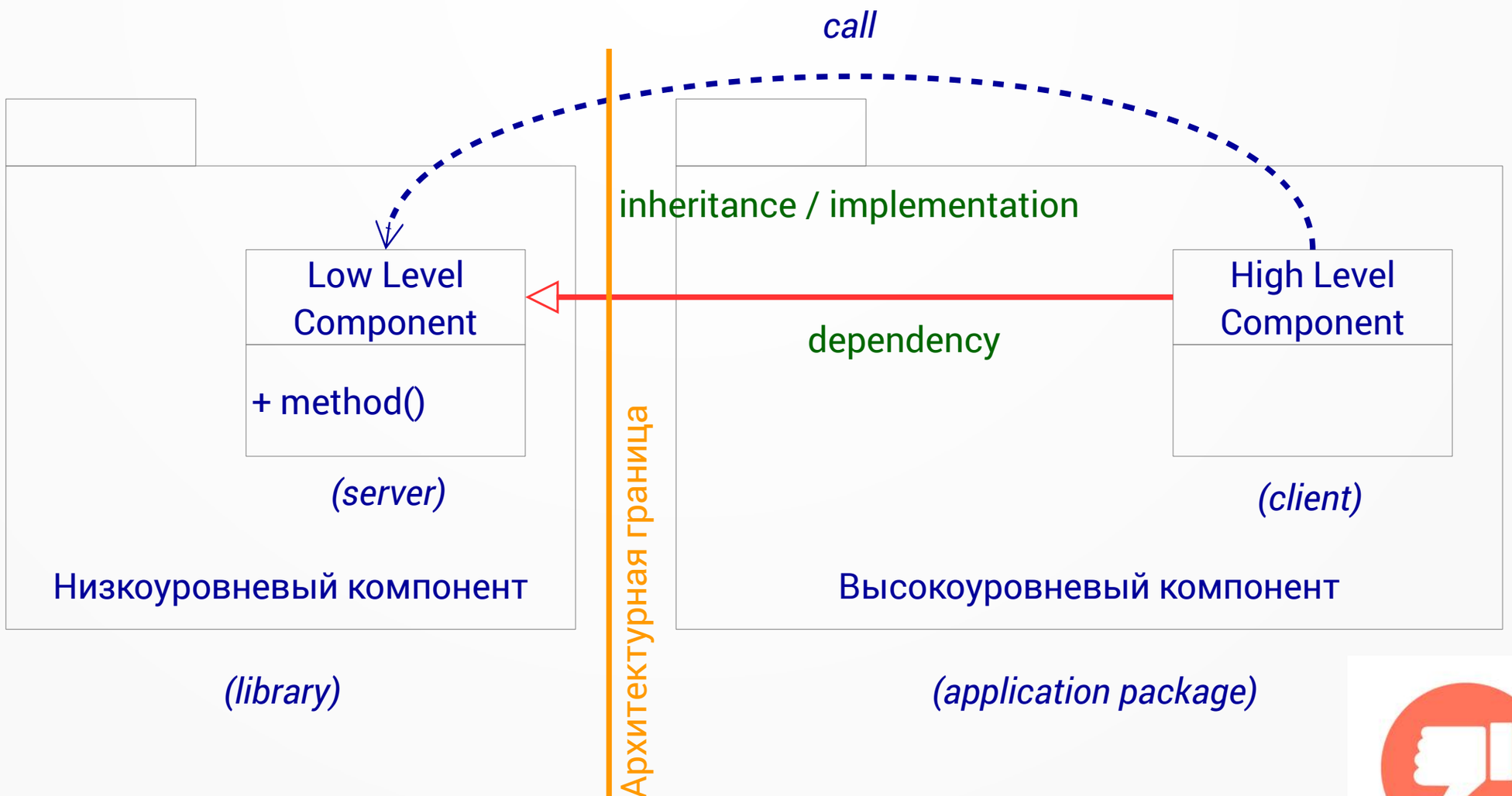
Зависимости

Зависимости (dependency):

- **исходного кода** *(между включающими и подключаемыми файлами)*;
- **потока управления** *(между вызывающими и вызываемыми модулями)*;
- **между классами** (*наследование, реализация*);
- **между объектами** классов (*ассоциация, агрегация, композиция*);
- **в единицах развёртывания** *(совместно хранимые модули)*;

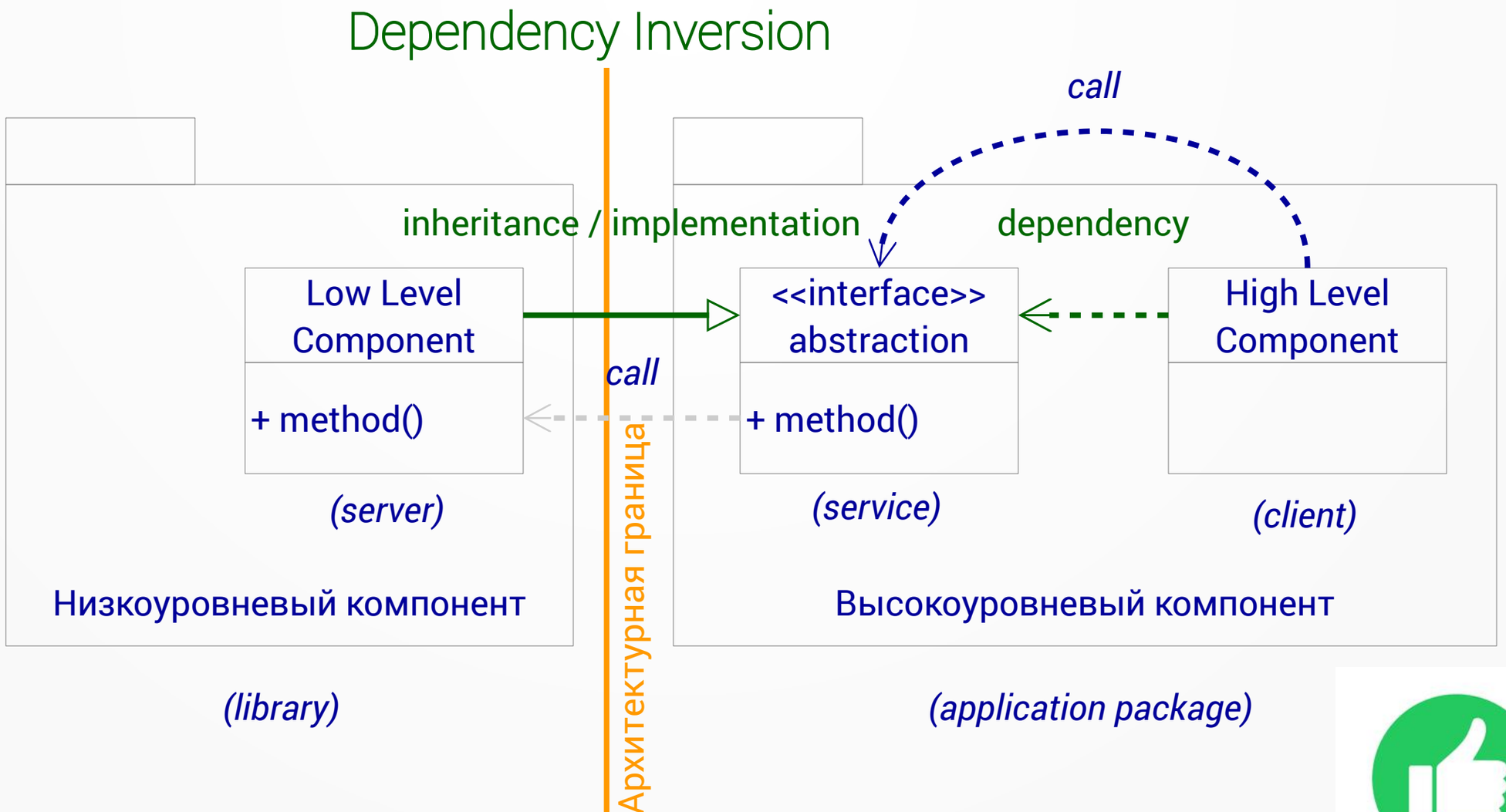
Главное правило построения надёжной модифицируемой системы: зависимости должны быть направлены от низкоуровневых компонентов к высокоуровневым: менее важные части должны зависеть от более важных, а не наоборот!

Зависимость исходного кода: плохая



Заисимость направлена от высокого уровня к низкому.

Инвертирование зависимости (D. I.)



Заисимость направлена от низкого уровня к высокому.

Парадигмы и их ценности

Структурное программирование **даёт возможность** ***управлять сложностью***.

Объектно-ориентированное программирование **даёт возможность** ***управлять зависимостями***.

Функциональное программирование **даёт возможность** ***управлять состоянием***.

Разные парадигмы вводят свои ограничения, за счёт чего дают новые возможности.

Ценность S. P.

Структурное программирование [1968]:

- ограничение на прямую передачу управления (GOTO);
- запись всех программ 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 компоненты разных уровней.

Модульность – основа разработки.

Ценность F. P.

Функциональное программирование [1958]:

- ограничение на изменение состояния (присваивание);
- чистые функции без побочных эффектов;
- рекурсия;
- функции высших порядков.

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

Не просто модули – функции без побочных эффектов.

Ценность О. О. Р. для архитектуры

Объектно-ориентированное программирование [1980]:

- 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).

Целенаправленное программирование интерфейсов для модулей.

Взгляды на архитектуру

Статический

- Физическое расположение:
  - Каталоги и файлы, где хранятся программы.
  - Исходный, объектный и исполняемый код.
  - Внешние библиотеки.
  - Установочные архивы.
- Логическая структура:
  - Связи между компонентами.
  - Подключаемые модули.
  - Иерархия классов.
  - Пакеты.
- . . .

Динамический

- Узлы, на которых выполняются программы.
- Исполняемые программы в ОЗУ.
- Процессы: process, thread, fiber, . . .
- Объекты со своим жизненным циклом.
- Взаимодействие объектов (вызов методов).
- Синхронная и асинхронная активация.
- События, сигналы, прерывания.
- Источники и приёмники данных.
- . . .

Проектирование

Выполнение

Тексты. Описания. Расположение.
Имена. Логические связи.

Исполнение. Экземпляры. Время. Ресурсы.
Адреса. Физические связи. Размеры.

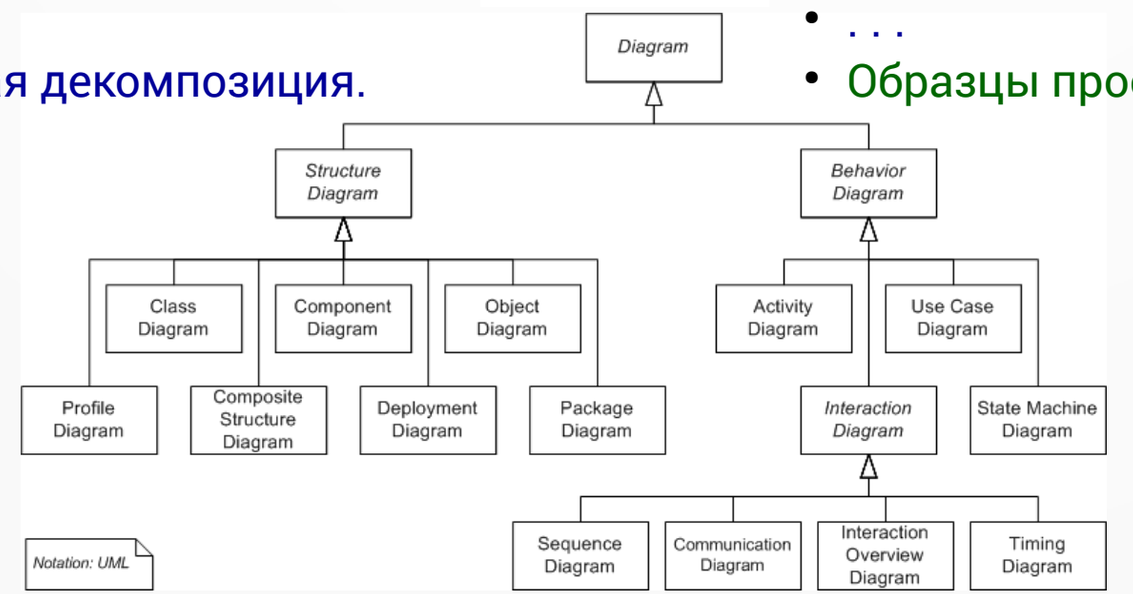
Static ← Program → Dynamic

Статика

- Структура.
- Исходный код.
- Алгоритм.
- Структуры данных.
- Иерархия **классов**.
- Программые компоненты.
- Типы связей между компонентами.
- . . .
- Функциональная декомпозиция.

Динамика

- Поведение.
- Исполняемая программа.
- Ход выполения программы.
- Обрабатываемые данные.
- Объекты со своим жизненным циклом.
- Взаимодействие **объектов**.
- Динамические связи между объектами.
- . . .
- Образцы проектирования.



Статическое и динамическое представления программы кардинально отличаются!

Dynamic: Образцы проектирования

Design Pattern – *именованное описание (в унифицированной форме) способа взаимодействия объектов и классов, адаптированного для решения типичной задачи проектирования в конкретном контексте*.

Design Patterns :

- независимые от языка структуры с известным поведением;
- проверенные способы решения типичных архитектурных задач;
- механизмы, из которых строится поведение ОО-систем;
- приёмы управления зависимостями при разработке;
- наборы технических классов и объектов в системе;
- воплощение обобщённого опыта разработки ОО-систем;

Классические образцы

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

Creational / Порождающие:

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

Structural / Структурные:

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

Behavioral / 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.

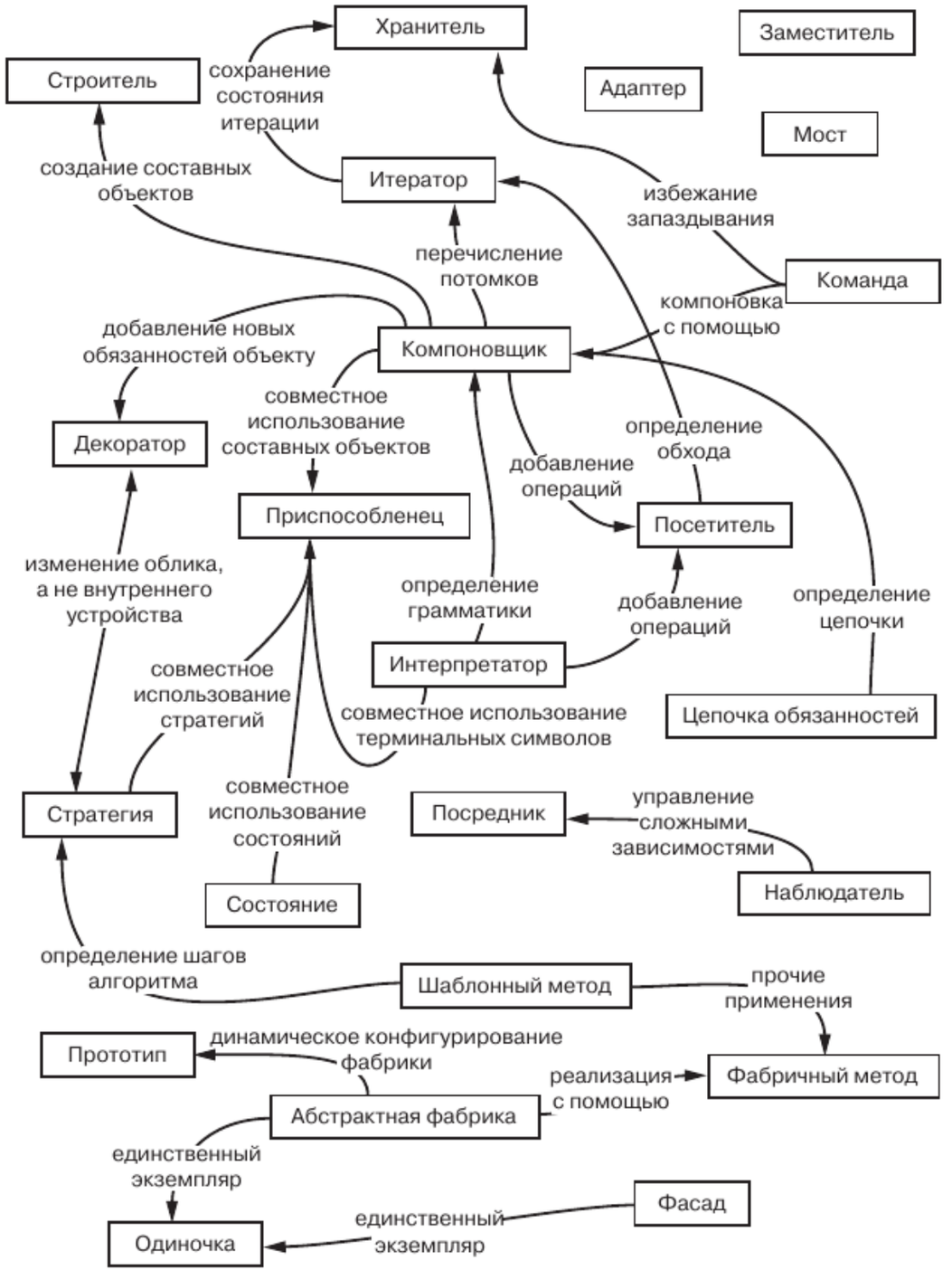


Рис. 1.1. Отношения между паттернами проектирования

Каталог из 23 классических образцов описан в книге «банды четырёх» - Gang of Four (GoF):
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.

Дополнительные образцы рассмотрены в книгах:
Фаулер М. Шаблоны корпоративных приложений = **Patterns of Enterprise Application Architecture**. — М.: Вильямс, 2016.

МакКоннелл С. Совершенный код. Практическое руководство по разработке программного обеспечения = **Code Complete: A Practical Handbook of Software Construction**. — М.: Издательство «Русская редакция», 2010.

Таблица 1.2. Изменяемые паттернами элементы дизайна

Назначение	Паттерн проектирования	Аспекты, которые можно изменять
Порождающие паттерны	Абстрактная фабрика	Семейства порождаемых объектов
	Одиночка	Единственный экземпляр класса
	Прототип	Класс, из которого инстанцируется объект
	Строитель	Способ создания составного объекта
	Фабричный метод	Инстанцируемый подкласс объекта
Структурные паттерны	Адаптер	Интерфейс к объекту
	Декоратор	Обязанности объекта без порождения подкласса
	Заместитель	Способ доступа к объекту, его местоположение
	Компоновщик	Структура и состав объекта
	Мост	Реализация объекта
	Приспособленец	Накладные расходы на хранение объектов
	Фасад	Интерфейс к подсистеме
Паттерны поведения	Интерпретатор	Грамматика и интерпретация языка
	Итератор	Способ обхода элементов агрегата
	Команда	Время и способ выполнения запроса
	Наблюдатель	Множество объектов, зависящих от другого объекта; способ, которым зависимые объекты поддерживают себя в актуальном состоянии
	Посетитель	Операции, которые можно применить к объекту или объектам, не меняя класса
	Посредник	Объекты, взаимодействующие между собой, и способ их кооперации
	Состояние	Состояние объекта
	Стратегия	Алгоритм
	Хранитель	Закрытая информация, хранящаяся вне объекта, и время ее сохранения
	Цепочка обязанностей	Объект, выполняющий запрос
	Шаблонный метод	Шаги алгоритма

Применение образцов проектирования (Design Patterns) даёт возможность заложить в разрабатываемую систему гибкие решения: каким образом организовать её функционирование, как построить систему, чтобы легче изменять её в будущем.

Design Patterns помогают легче понимать, как работает сложная система.

Ценность образцов проектирования

Design Patterns

- дают возможность обсуждать задачи на более высоком уровне абстракции;
- улучшают взаимопонимание при взаимодействии между программистами в ходе разработки;
- помогают выбирать наиболее подходящий вариант проектирования для реализации задачи;
- дают разработчикам возможность строить систему из крупных ахитектурных блоков;
- способствуют проектировать повторно используемый код;
- помогают разрабатывать, предусматривая возможность будущих изменений;
- помогают лучше понять работу модифицируемой системы;
- экономят время разработчиков.

Требования к разработчику

Что должен знать, уметь и практиковать
хороший программист?

Квалификация

Разработчик ПО должен знать и соблюдать **принципы** разработки, владеть современными технологиями, владеть **инструментами** разработки, применять прогрессивные **практики**, воспитывать в себе профессиональную **культуру** и умение работать **в команде** – всё это позволит стабильно разрабатывать программное обеспечение, которое будет надёжно работать, которое можно легко модифицировать и развивать на предсказуемом уровне затрат.

Это – основы профессионализма раработчиков.

Как и любому профессионалу, программисту нужно учиться всю жизнь.

Знания *(основы)*

Языки программирования:

- **Компилируемый (OOP):** Kotlin ← Java | C# / C++ / ObjectPascal | FreePascal / Swift . . .
- **Скриптовый:** Ruby / Python / . . .
- Web-приложения: JavaScript (← Kotlin) | ActionScript ← ECMAScript
- **Командный язык ОС:** sh | bash / cmd | PowerShell . . .
- **Язык СУБД:** SQL / noSQL DBMS API

Алгоритмы:

линейные, циклические, рекурсивные, . . ., (классы сложности)

Структуры данных:

array, structure, hash, tree, graph, queue, stack, . . .

Форматы данных:

JSON, YAML, XML, CSV, KWP, . . .

OO-проектирование (OOD):

design patterns, UML, clean architecture, SOLID, . . .

Web-технологии:

HTTP, REST, cookie, sockets, . . .

Знания (*дополнительные, специальные*)

Языки программирования:

- **Функциональный:** Scala, Haskell, . . .
- **Для параллельных вычислений:** Erlang, Go, Occam, Modula-3, Scala, . . .
- **Для статистической обработки данных:** R, . . .

Алгоритмы:

гибкие (стохастические, эвристические, . . .), . . .

Библиотеки:

специализированные: GIS, ML, IoT, . . .

Базы данных:

SQLite, PostgreSQL, MySQL, Oracle, IBM DB2, MS SQL Server, Firebird, Interbase, . . .

Физическое программирование (IoT) на MCU и SBC:

RTC, ADC, GPIO, PWM, UART, протоколы (RS-232, SPI, I^2C / TWI, I^2S, modbus), . . .

Параллельные и распределённые вычисления:

OpenMP, PVM, MPI, mltithreading, multitasking, RPI, . . .

Практики

Практики – неотступное применение принципов и технологий, которое присуще профессионалам:

- Extreme Programming – адаптивный подход к разработке.
- Effective Estimation – оценка сложности / сроков выполнения работ.
- Planning Game – планирование по экспертным оценкам (Story Points).
- Test First – применение TDD.
- Refactoring – привычка к реорганизации исходников.
- Simple Design – правила простого проектирования.
- Code Kata – тренировка навыков програмирования.

Практики, как и юридические законы, имеет смысл только, если их применяют ВСЕГДА, без исключений.

Инструменты разработчика

Документирование:

User Story ≈ Use Case, UML, структурограммы (NSD), соглашения по оформлению исходников, средства генерирования документации из исходников.

Редактор / интегрированная среда разработки (IDE):

Code::Blocks, Eclipse, Geany, IntelliJ IDEA, KDevelop, NetBeans, MS VS CE / VS Code, …

Система управления версиями исходного кода (VCS):

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

Средство автоматизации тестирования:

FitNesse, Robot Framework, Selenium, **xUnit**, xSpec/xBehave, …

Система непрерывной интеграции (CIS), сборки (CIB), развёртывания (CID):

CruiseControl, GitLab, Jenkins, Travis CI, Vexor; Jenkins, Buildbot, Travis, …

Личность разработчика

В конечном счёте, всё зависит от личных качеств разработчика:

– соблюдает ли он профессиональные требования и принципы?
– обладает ли он культурой разработки?
– соблюдает ли он профессиональную этику?
– хорошо ли он овладел навыками и приёмами работы?
– как глубоко он освоил теоретические основы профессии?
– насколько он сведущ в предметной области?
– мржет ли он работать в составе команды?
– стремится ли он стать Мастером своего дела?
– . . .

Это сложная задача: воспитать стремление к профессионализму.

Но многого можно добиться атмосферой в коллективе разработчиков, при обучении новичков, примером личности Мастера.

Обучение + воспитание

Это необходимо осваивать каждому.

Это нужно постоянно практиковать.

Этому должен учить новичков их наставник.

Этому учатся при практической работе над проектом в команде.

Это бесконечный процесс для любого...

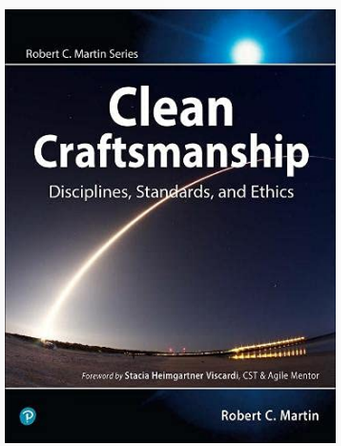
道 - Пути совершенствоания

Есть очень много разработчиков,
которые служат примерами для других программистов
и с которыми можно даже пообщаться по Сети.
На их опыте, изложенном в книгах, публикациях, видео,
можно учиться самим
и рекомендовать эти материалы другим.

Попрактиковаться работе в команде можно,
примкнув к проекту с открытым кодом.

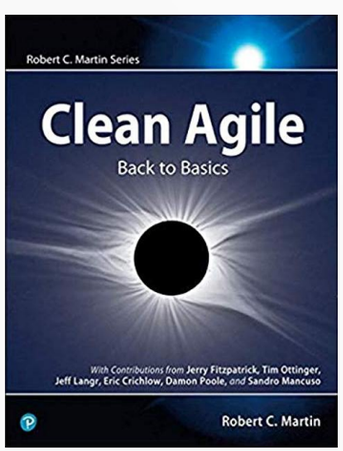
Никто не мешает самосовершенствоваться.

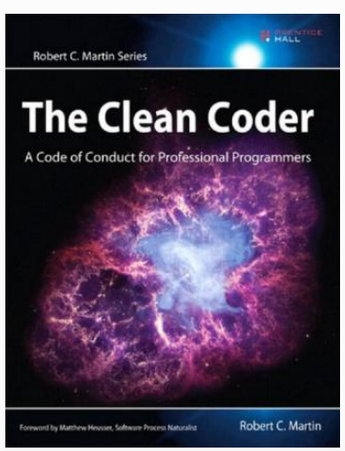
Книги «дядюшки Боба»

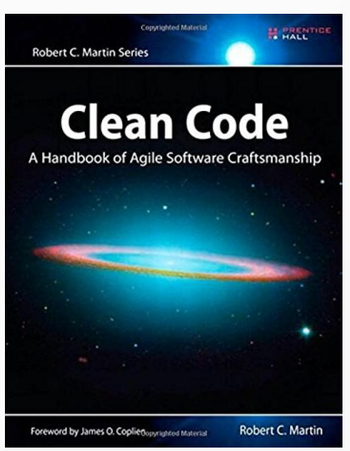


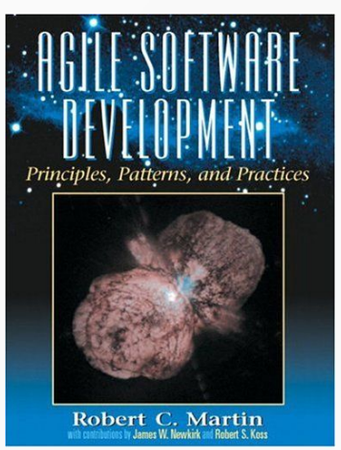
Robert "Uncle Bob" Martin

с 1970 – профессионально разрабатывает ПО, с 1990 – международный консультант в этой области, с 1996 по 1999 – главный редактор журнала "C++ Report", в 2001 организовал встречу группы, которая положила начало гибким методологиям разработки программ.









Книги, которые нужно прочитать *(IMHO)*

1. Бек К. Экстремальное программирование = **Extreme Programming Explained: Embrace Change** (2004) — СПб: Питер, 2002.
2. Бек К. Экстремальное программирование. Разработка через тестирование = **Test-Driven Development by Example** (2002) — 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** (1994) — СПб: Питер, 2015.
4. Мартин Р. Быстрая разработка программного обеспечения = **Agile Software Development, Principles, Patterns, and Practices** (2002) — М.: Вильямс, 2004.
5. Мартин Р. Идеальный программист = **The Clean Coder. A Code of Conduct for Professional Programmers** (2011) — СПб: Питер, 2012.
6. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения = **Clean Architecture. A Craftsman's Guide to Software Structure and Design** (2017) — СПб: Питер, 2018.
7. Мартин Р. Чистый код. Создание, анализ и рефакторинг = **Clean Code. A Handbook of Agile Software Craftsmanship** (2009) — СПб: Питер, 2018.
8. Мартин Р. Чистый Agile. Основы гибкости = **Clean Agile. Back to Basics** (2020) — СПб.: Питер, 2020.
9. Мартин Р. Идеальная работа. Программирование без прикрас = **Clean Craftsmanship: Disciplines, Standards, and Ethics** (2021) — СПб.: Питер, 2022.

Книги, которые стоит прочитать *(IMHO)*

10. Метц С. Ruby. Объектно-ориентированное проектирование = **Practical Object-Oriented Design. An Agile Primer Using Ruby** (2012) — СПб.: Питер, 2017.
11. Фаулер М. Рефакторинг. Улучшение проекта существующего кода = **Refactoring. Improving the Design of Existing Code**. — М.: Диалектика, 2019.
12. Томас Д., Хант Э. Программист-прагматик. Путь от подмастерья к мастеру = **The Pragmatic Programmer: From Journeyman to Master**. — М.: Лори, 2004.
13. Майерс Г. Надёжность программного обеспечения = **Software Reliability: Principles and Practices**. — М.: Мир, 1980.
14. Майерс Г. Искусство тестирования программ = **The Art Of Software Testing**. — М. Финансы и статистика, 1982; 3-е изд. — М., СПб.: Диалектика, 2019.
15. Ван Тассел Д. Стиль, разработка, эффективность, отладка и испытание программ = **Program Style, Design, Efficiency, Debuggind And Testing**.— М.: Мир, 1981.
16. Вирт Н. Алгоритмы + структуры данных = программы = **Algorithms + Data Structures = Programs**. — М.: Мир, 1985.
17. Керниган Б., Плоджер Ф. Инструментальные средства программирования на языке Pascal = **Software Tools in Pascal**. — М.: Радио и связь, 1985.
18. Буч Г. Объектно-ориентированное проектирование с примерами применения = **Object-Oriented Design with Applications**. — М.: Конкорд, 1992.

Полезные ссылки

- https://basecamp.com/gettingreal # “Getting Real” @ 37 Signals
- https://it.wikireading.ru/37243 # “Getting Real” на русском
- https://en.wikipedia.org/wiki/Software_design_pattern # Образцы проектирования
- https://blog.cleancoder.com/uncle-bob/2014/06/30/ALittleAboutPatterns.html
- https://en.wikipedia.org/wiki/Kata_(programming)
- https://habr.com/ru/post/171883/ #Стартап-ловушка
- https://medium.com/@pablo127/effective-estimation-review-of-uncle-bobs-presentation-a2150f5f68ac
- https://codingjourneyman.com/2014/10/06/the-clean-coder-estimation/ # PERT
- https://ru.wikipedia.org/wiki/Покер_планирования # Относительное планирование
- https://8thlight.com/blog/micah-martin/2012/11/17/transformation-priority-premise-applied.html
- https://habr.com/ru/company/piter/blog/427853/ # Размышления о TDD
- https://habr.com/ru/post/206828/ # Test-Driven Development — телега или лошадь?
- https://software-testing.ru/library/5-testing/72---tdd-- # Ошибки начинающих TDD-практиков
- https://habr.com/ru/post/459620/ # Всё, что вы хотите узнать о Driven Development
- https://habr.com/ru/company/edison/blog/313410/ # Как объяснить бабушке, что такое Agile
- https://habr.com/ru/post/131926/ # Почему Agile вам не подходит
- https://habr.com/ru/post/695554/ # У вас не Agile
- https://worksection.com/blog/work-in-progress.html # WIP-лимит в жизни, в работе, в семье
- https://habr.com/ru/post/352282/ # Continuous Integration для новичков
- https://habr.com/ru/company/southbridge/blog/691782/ # CI/CD: как, зачем, для чего
- https://lifehacker.ru/cennyj-sotrudnik/ # 5 установок, которые отличают ценных сотрудников от обычных

Словарик терминов и сокращений

- agile software development ~ гибкие методологии разработки ПО.
- AT = acceptance test ~ приёмочный тест.
- BDD = Behaviour-Driven Development ~ разработка, определяемая поведением.
- CCC = Collaboration, Coordination, Communication ~ agile practices
- CCO = collective code ownership ~ совместрое владение кодом (X.P.).
- CIS = continuous integration system ~ система непрерывной интеграции.
- DDD = Domain-Driven Design ~ предметно-/проблемно-ориентированное проектирование.
- DI = dependency inversion ~ инвертирование зависимостей.
- DSL = domain-specific language ~ специализированный язык для конкретной предметной области.
- FP = functional programming ~ функциональное программирование.
- GoF = Gang of Four ~ «банда четырёх».
- I/O = input/output ~ ввод-вывод.
- IDE = integrated development environment ~ интегрированная среда разработки.
- OOD = object-oriented design ~ объектно-ориентированного проектирование.
- OOP = object-oriented programming ~ объектно-ориентированное программирование.
- PP = pair programming ~ парное программирование.
- QA = quality assurance ~ контроль качества.
- S.O.L.I.D. ~ 5 основных принципов OOP и OOD.
- SP = structured programming ~ структурное программирование.
- TDD = Test-Driven Development ~ разработка через тестирование, тест-ориентированная разработка.
- TPP = Transformation Priority Premise ~ предположения о приоритетах преобразований.
- user stories ~ пользовательские истории – неформальное описание требований к разрабатываемой программе.
- UML = Unified Modeling Language ~ унифицированный язык моделирования.
- Uncle Bob = Robert Cecil Martin ~ Роберт “дядюшка Боб” Мартин.
- UT = unit test ~ модульный тест.
- VCS = version control system ~ система управления версиями исходного кода.
- WIP-limit = work in progress limit ~ ограничение числа незавершенных задач

Готов ответить на вопросы

???