Программное дело

о разработке программ расскажет
Михаил В. Шохирев

Клуб программистов
Шадринск
2020-21

Ремесло программирования

Програмирование – это инженерное ремесло, потому что результат разработки – практически полезная вещь, программа. Это такое же уважаемое ремесло, как кузнецкое дело или радиоинженерия.

Конечно, во всяком ремесле настоящий мастер проявляет творчество, развивает себя и своё дело.

Всякое ремесло построено на неукоснительном собдюдении важных правил и на овладении приёмами и навыками работы, которые основаны на теоретических знаниях.
И тот, кто не следует этому, не станет мастером своего дела.

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.

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 языке?!
– Наше ПО не поддерживает эту СУБД.
– Это же веб-приложение, здесь всё по-другому…
– 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!

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 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 Ван Тассел** (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** и **Филип Плоджер** (1985).
6. «Объектно-ориентированное проектирование с примерами применения» — **Гради Буч** (1992).
7. «Экстремальное программирование» — **Кент Бек** (2002).
8. «Программист-прагматик. Путь от подмастерья к мастеру» — **ДэйвТомас, Энди Хант** (2004).
9. «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 программ.***

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.***

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.
Сложно разработать программу, которую можно развивать при использовании.

Для этого надо овладеть современными подходами к разработке:

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

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 при кодировании*.

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: *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**.

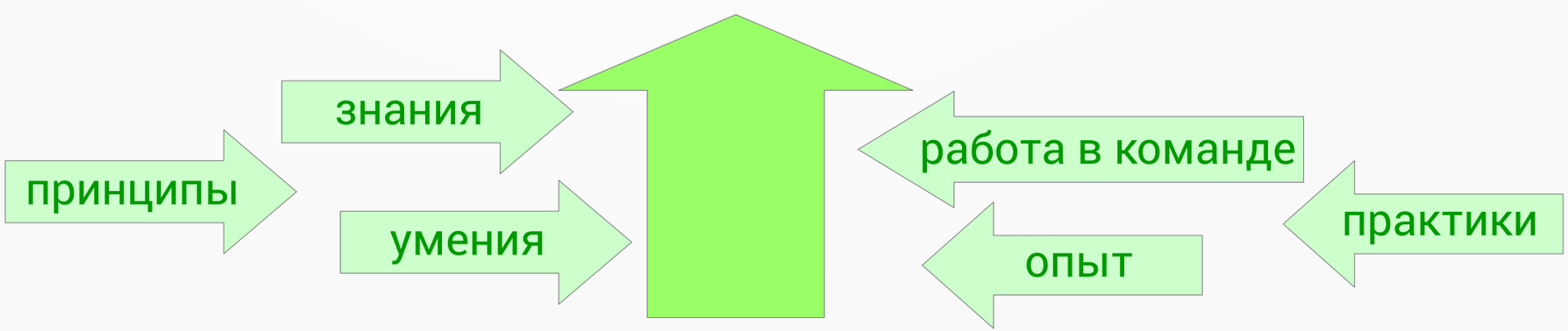
Это не просто провозглашённые принципы

— они давно и успешно применяются на практике!

Технологии: *гибкие* ←→ тяжеловесные

- *Адаптация* вместо предопределённости.
- *Наращивание* архитектуры, а не её 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 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 (web-interface).
- Lean Software Development (LSD).
- OpenUP (← RUP).
- Rapid Application Development (RAD).
- Scrum (Sprint Continious Rugby Unified Methodology).

X. P. = eXtreme Programming*

4 основные положения XP :

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

II. **Непрерывный процесс** (Continuous Process): *соблюдение правил (заказчиками и рпзработчиками) помогает безостановочно двигаться к цели по графику.*

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

IV. **Социальная защищённость программиста** (Programmer Welfare): *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.*

X. P. = eXtreme Programming*

12 основных приёмов XP :

I. **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 (product owner) всегда на связи для вопросов.*
2. **Игра в планирование** (Planning Game): *направляет разработку продукта через планирование итераций и установку приоритетов для пользовательских историй.*
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* ←**CCO**.

X. P. = eXtreme Programming*

12 основных приёмов XP :

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

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

X. P. = eXtreme Programming*

12 основных приёмов XP :

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

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* ← **PP**.

X. P. = eXtreme Programming*

12 основных приёмов XP :

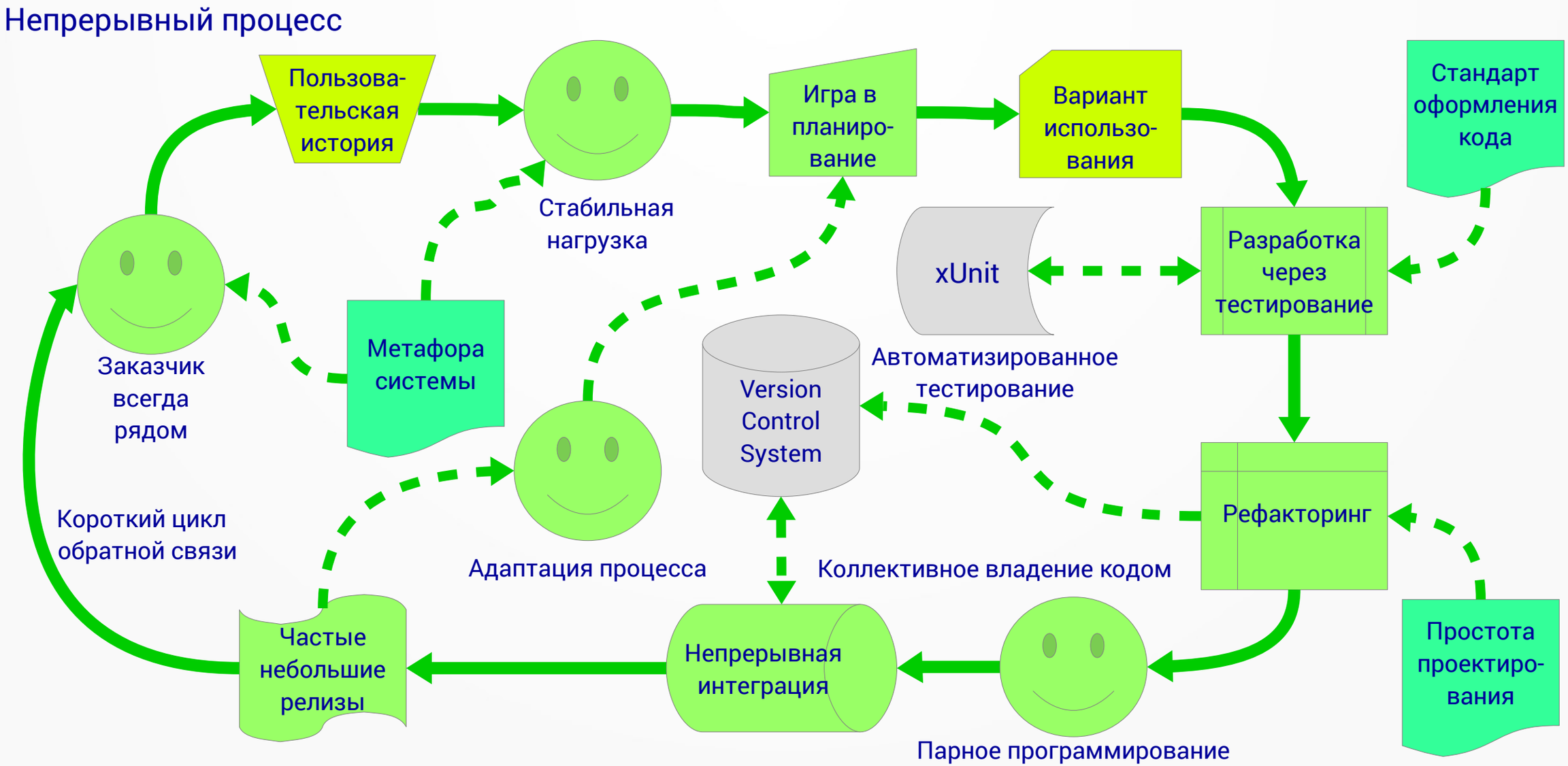
IV. **Социальная защищённость программиста** (Programmer Welfare): *разработка без авралов и переработки.*

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

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?” Владельц продукта должен выбрать, какие именно пользовательские истории будут реализованы на этой неделе.

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

Экстремальное программирование



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!

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»

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, котрую можно изменять

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

Заказчики хотят менять всё:

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

И это нормально!

Для пользователя критически важна *стоимость внесения изменений* в программную систему.

Время и стоимость изменений должны быть пропорциональны объёму требований пользователя (соответствовать его ожиданиям).

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

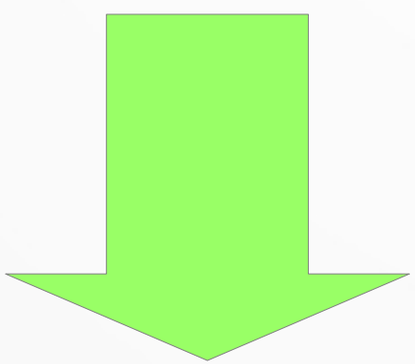
Эд Йордан

"Процесс автоматизации изменяет взгляд пользователя на объект автоматизации".

Гленфорд Майерс

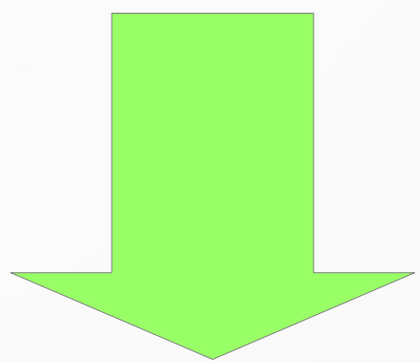
Условия для изменяемости

Уверенность, что программа продолжит правильно работать после изменения.



Надёжный набор тестов

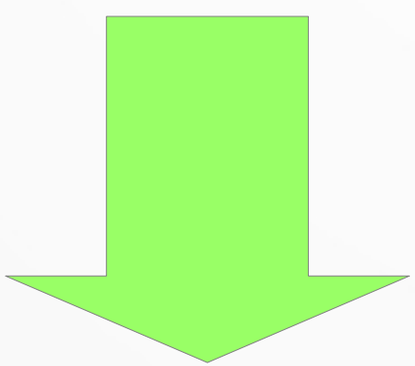
Программа имеет структуру, которая позволяет легко вносить изменения.



“Чистая архитектура”

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 кода.
... особый API к разрабатываемой системе.
... неотъемлемая часть архитектуры системы.

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

Виды тестов *(основные)*

низкоуровневый тест

unit test

(модульный тест)

проверяет реализацию технического требования или решения разработчика;

создаётся и выполняется программистом

высокоуровневый тест

acceptance test

(приёмочный тест)

проверяет реализацию функционального требования заказчика;

*разрабатывается и выполняется тестировщиком**

** специалистом QA (Quality Assurance) или руководителем разработки / архитектором*

Польза и важность тестирования

Имея надёжный набор тестов, ...

... можно рефакторить программу без опаски.

... можно смело править программные ошибки.

... можно спокойно модифицировать программу.

... можно уверенно развивать функционал программы.

Имея полный набор тестов, можно даже воссоздать потерянную программу.

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

Test First = TDD = Test-Driven Development

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

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

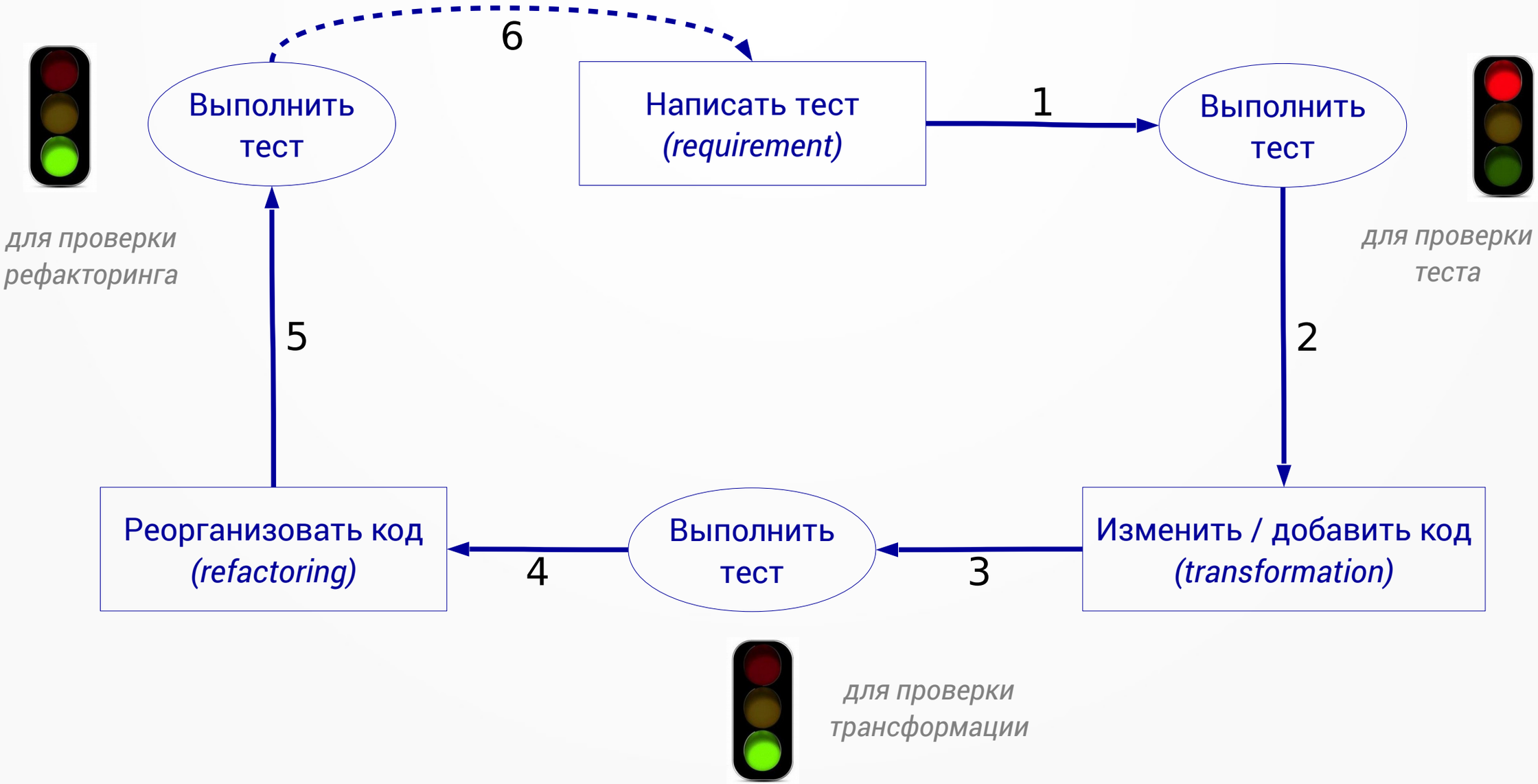
TDD – сознательное неотступное соблюдение технологической дисциплины разработчиками (как двойная запись в бухучёте).

Применение TDD основано на автоматизированных модульных тестах.

“Разработка, движимая тестами”

1. Разработка начинается, когда вы *записываете требование к функционалу* в формализованном и исполняемом виде – пишете модульный тест.
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*.
4. Выполнив тест для несуществующей подпрограммы, вы убеждаетесь, что ошибок в самом тесте нет.
5. Затем вы пишете минимальное тело тестируемой подпрограммы, в котором реализуется это требование (“How should we make that pass?”).
6. Выполнив тест после этого, вы убеждаетесь, что *реализация проходит проверку*.
7. Возвращаясь к тексту подпрограммы, вы *улучшаете его, не меняя интерфейса*.
8. А затем снова выполняете тест, чтобы убедиться, что реорганизация не испортила подпрограмму.
9. Потом вы пошагово трансформируя первоначальный исходник, *реализуете полный функционал*, соответсвующий этому требованию. Выполняя тесты после каждого шага.

Разработка, основанная на тестах



“Мантра” TDD *(уточнённая)*

0 1 2

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 на будущее!
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.

Refactoring / Design Improvement

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 (clean code).

Рефакторинг — обязательная часть разработки и постоянная деятельность программиста.

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

Transformation

Transformation ~ изменение поведения программы, не затрагивающее её интерфейс. Трансформация – это противоположность рефакторинга.
Путём пошаговых трансформаций “заглушка” эволюционирует до полноценной подпрограммы с требуемым функционалом.
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 switch 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 интерфейс модуля / подпрограммы.
- Короткий (3 высказывания - ***Given-When-Then***):

  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.
- Выполняется автоматически.

Тесты пишутся для проверки работы модуля с корректными и ошибочными данными.

Критика TDD

- Писать тест до разработки рабочего кода – это нонсенс.
- Применение TDD замедляет разработку!
- TDD усложняет работу, сковывает творчество.
- TDD не работатет для больших проектов.
- 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 выявляет пробелы в знаниях разработчика и дисциплинирует его.
- TDD для разработки модулей дополняет технологии для создания архитектуры.
- TDD – одна из обязательных технологий разработки, не требующая одобрения.
- Да, только 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.
- 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

- Психологически трудно перестроить своё сознание на эту технологию.
- Трудно неотступно практиковать TDD, особенно первые месяцы.
- Разработчиков можно только убедить применять TDD, заставить – нельзя. Как и нельзя заставить писать хороший код. Никакие административные меры не помогут, а вызовут лишь озлобление.

Советы по освоении TDD

- Внедрять TDD лучше всего на новом проекте (можно учебном).
- Можно внедрять TDD, начиная с отдельных классов.
- Не рекомендуется начинать внедрение TDD с больших коммерческих проектов.
- Намного проще внедрять TDD без унаследованного кода (legacy code).
- Идеально, если на начальном этапе внедрения TDD нет жёстких ограничений по срокам.
- При разработке тестов составляйте список, из которого последовательно вычёркивайте уже сделанные.

TDD — это процесс итеративного, непрерывного, параллельного написания тестов и рабочего кода, с обязательными фазами рефакторинга.

BDD для приёмочного тестирования

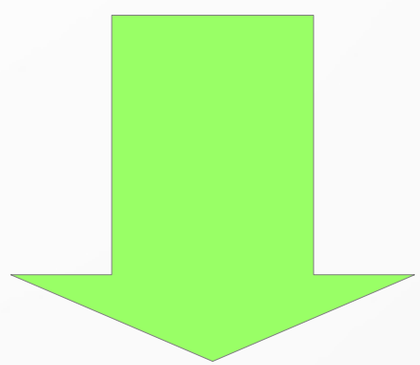
Для разработки приёмочных тестов (acceptance test) успешно применяется BDD (Behavior-Driven Development) – разработка, определяемая поведением.

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

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 = дизайн

Архитектура выражает замысел ПО, его назначение (*система заказов, учётная система, система мониторинга, игра и т. п.*).
Она строится на основе вариантов использования (use cases) ПО (*размещение заказа пользователем, обработка заявки диспетчером, отбор результатов оператором, просмотр результатов игрокомм и т. п.*).

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

Что не является архитектурой ПО

~~Subscribe - Publish~~ = алгоритм взаимодействия.

~~SOAP~~ = протокол обмена.

~~REST~~ = архитектурный стиль.

~~Framework~~ = инструмент / компонента.

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

~~Client — Server~~ = сетевая архитектура = конфигурация узлов.

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

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

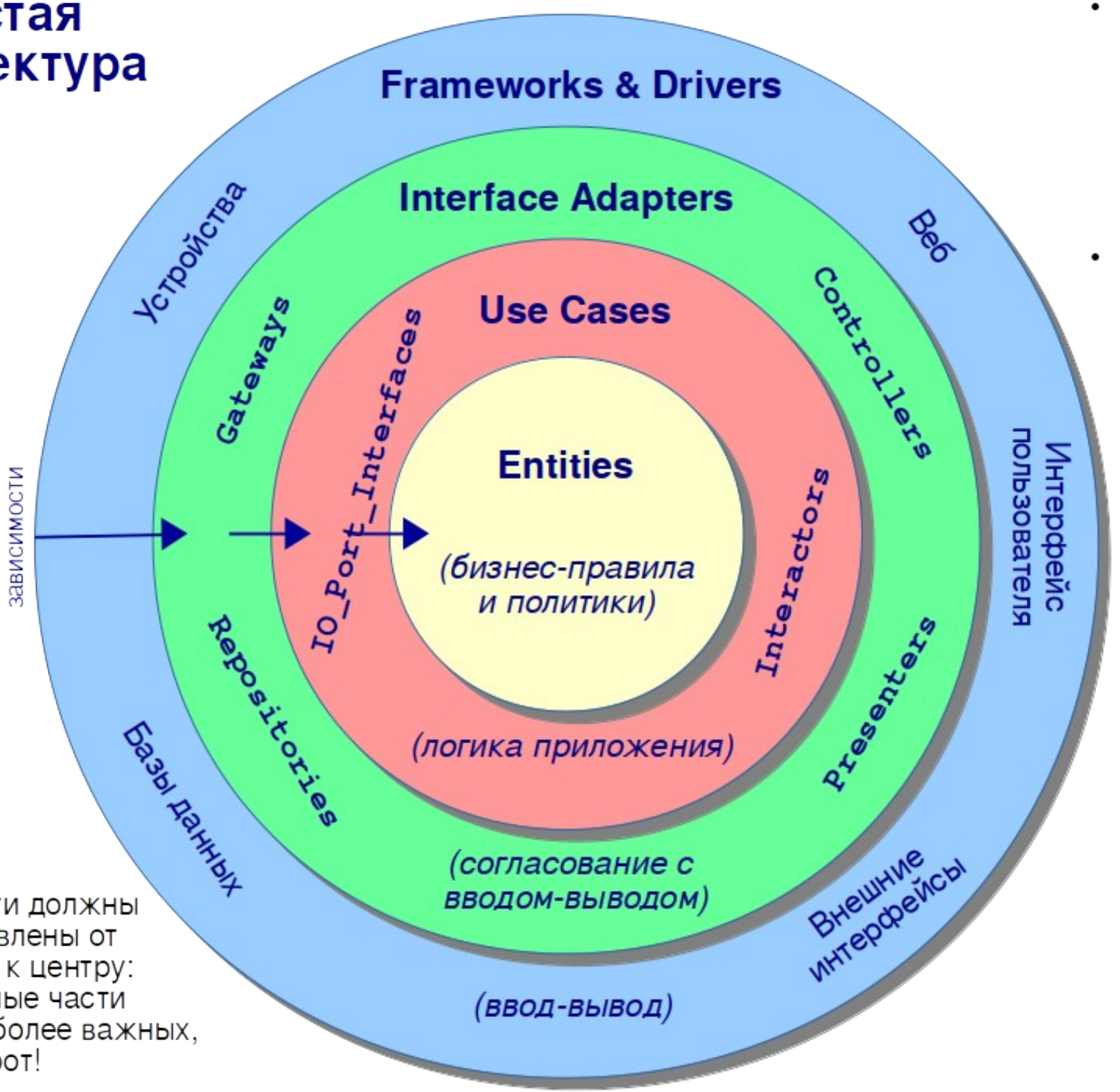
~~SOA (service-oriented architecture)~~ = способ организации I/O.

Clean Architecture

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

- бизнес-правила и политики – то, что не зависит от программной реализации (*как действия выполняются без всякой автоматизации*);
- 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

UI

DataStore

API

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

Согласование с вводом-выводом

Controller

Presenter

Repository

Gateway

Классы для объектов, реализующих интерфейсы и адаптеры, которые организуют предоставление данных в нужном виде.

Варианты использования

Interactor

UseCase

Классы для объектов программной системы, реализующих автоматизированную обработку данных, зависящие от бизнес-правил и политик.

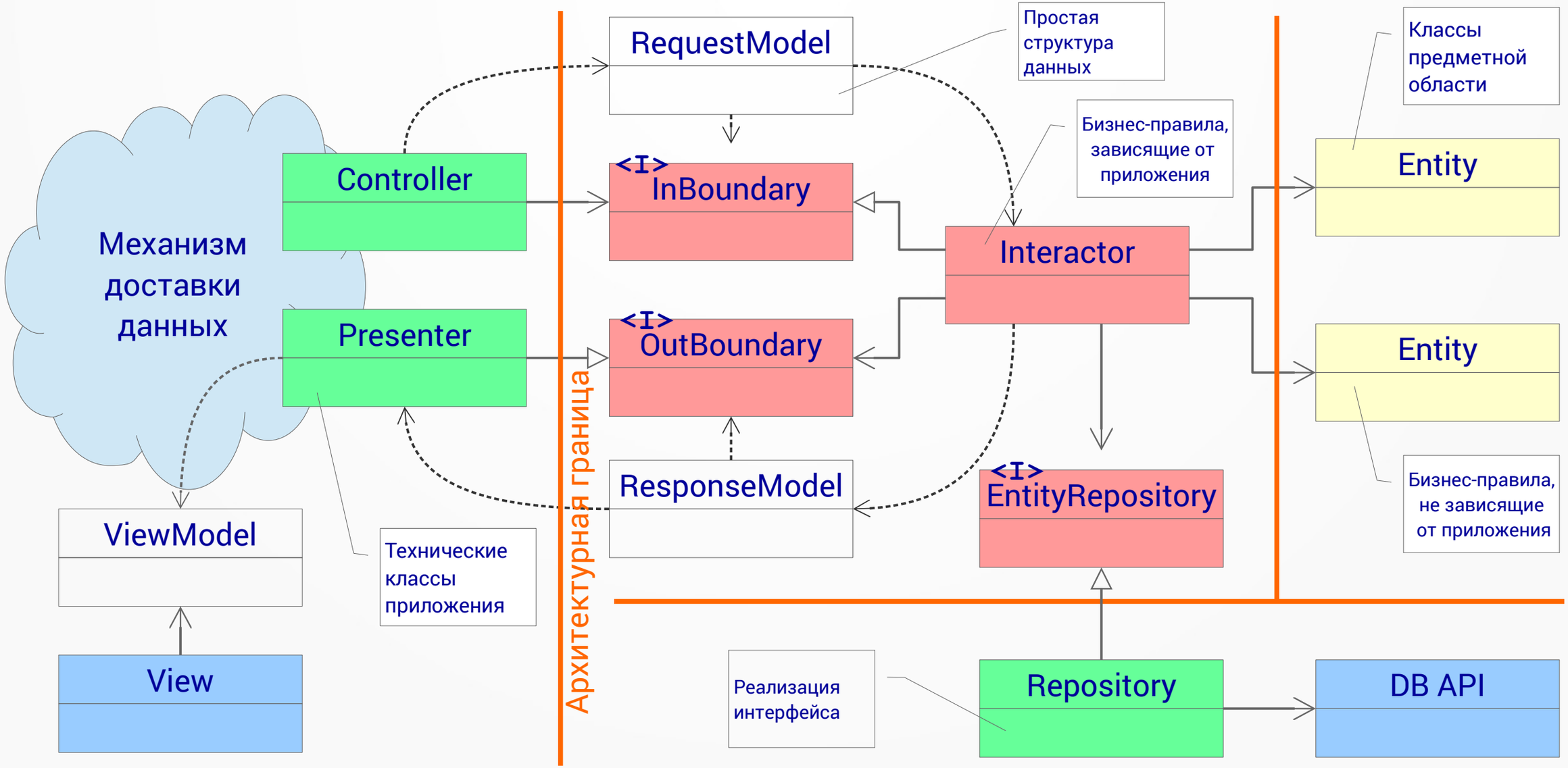
Бизнес-правила и политики

Entity

BusinessObject

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 разных 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

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 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 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.

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

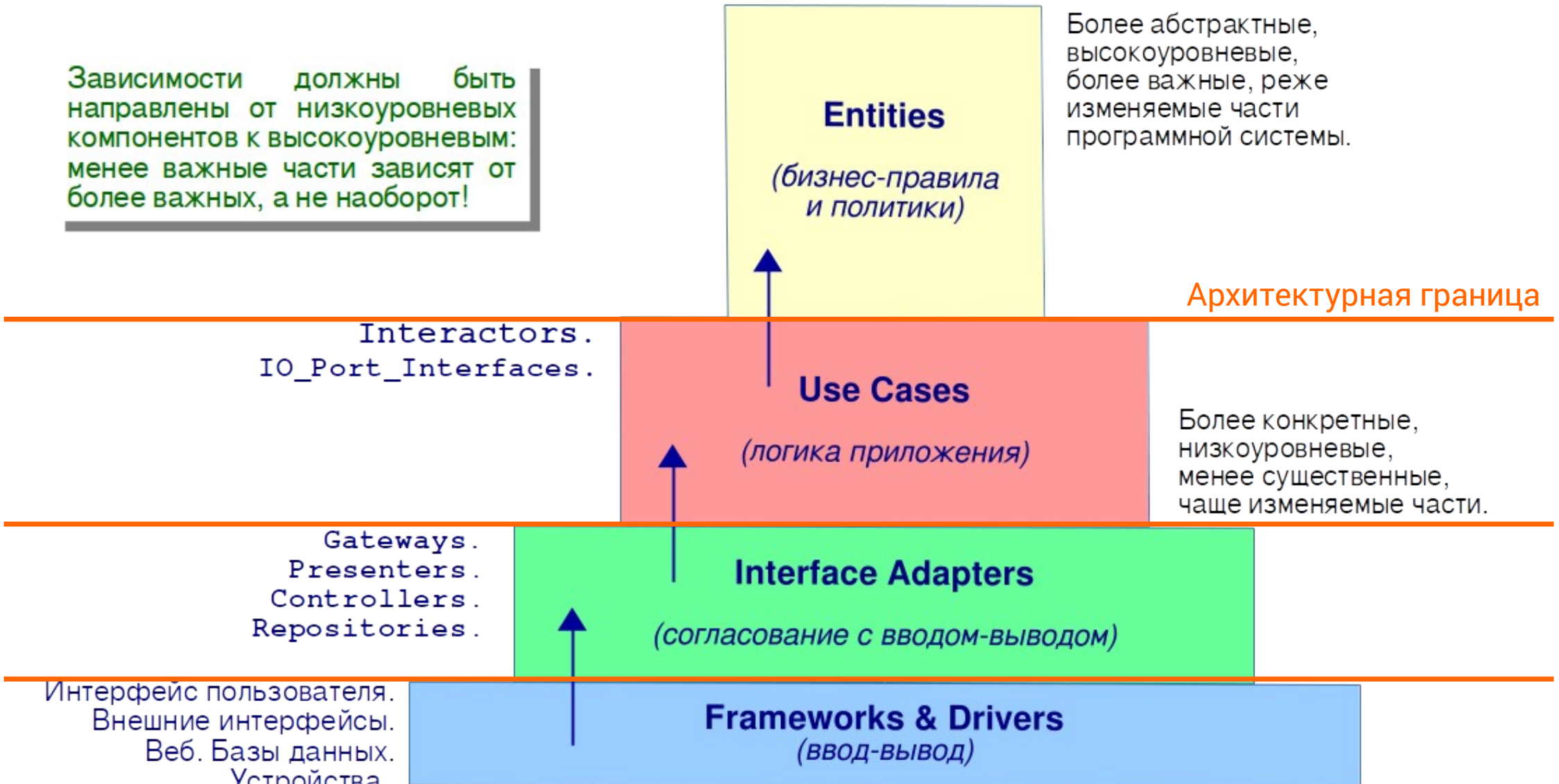
Архитектурная граница проходит между компонентами разных уровней программной системы.

Разделение системы на *уровни с управляемыми зависимостями* позволяет:

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

Чистая архитектура

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



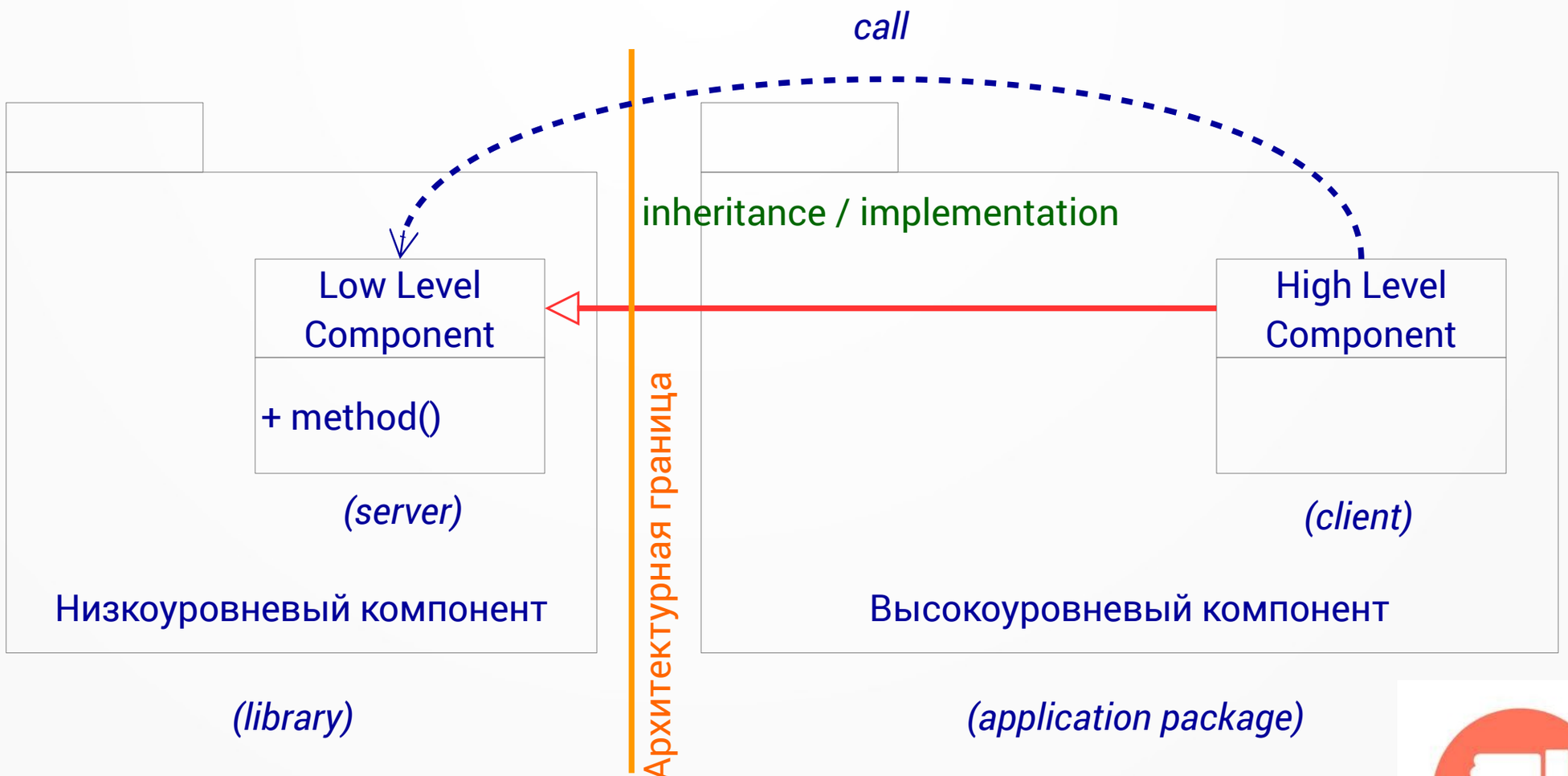
Зависимости

Зависимости:

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

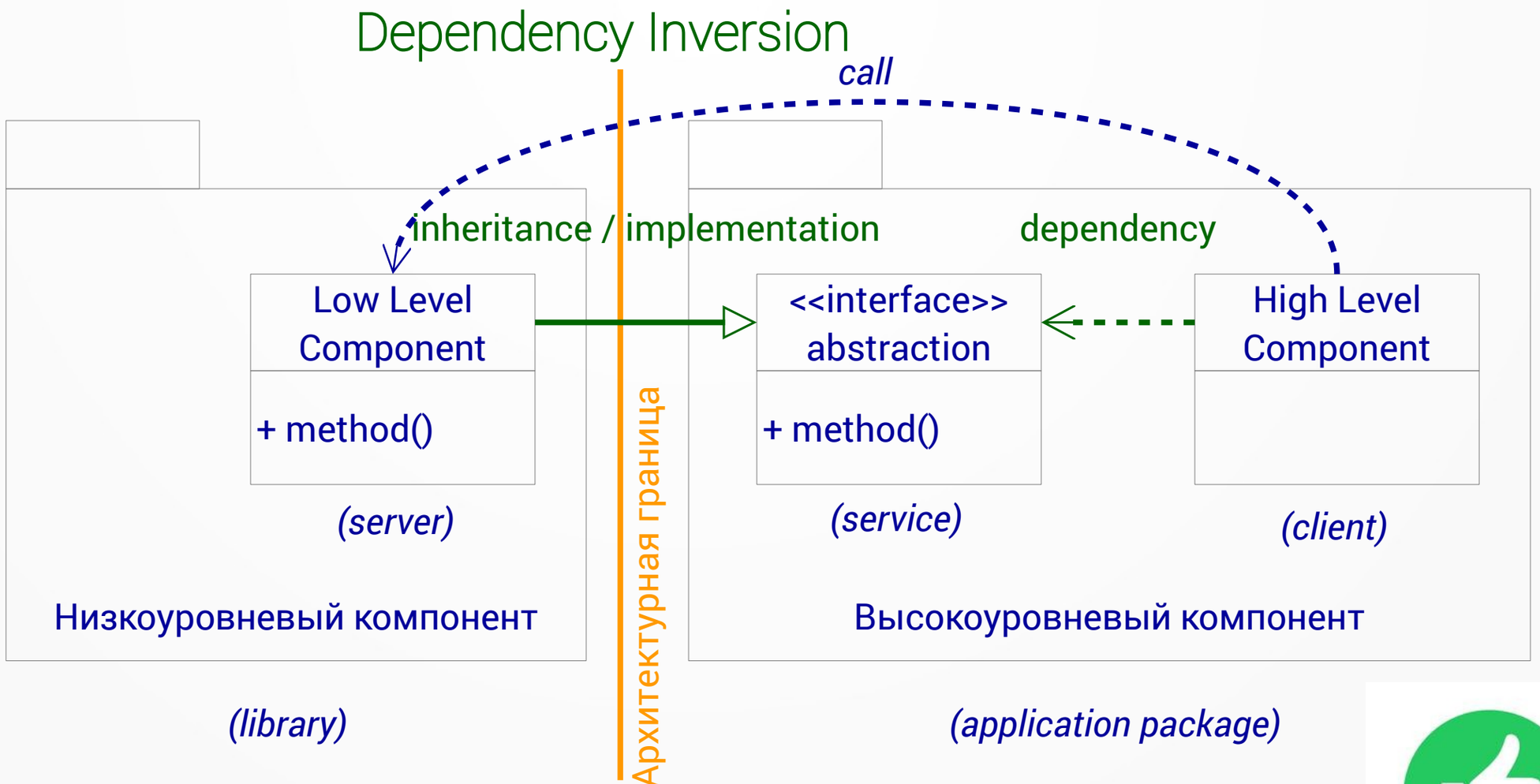
Главное правило построения надёжной модифицируемой системы: зависимости должны быть направлены от низкоуровневых компонентов к высокоуровневым: менее важные части должны зависеть от более важных, а не наоборот!

Зависимость исходного кода



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***.

Ценность S. P.

Структурное программирование:

- 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 должна быть «в крови» у разработчика.

Ценность F. P.

Функциональное программирование:

- 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:

- 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.

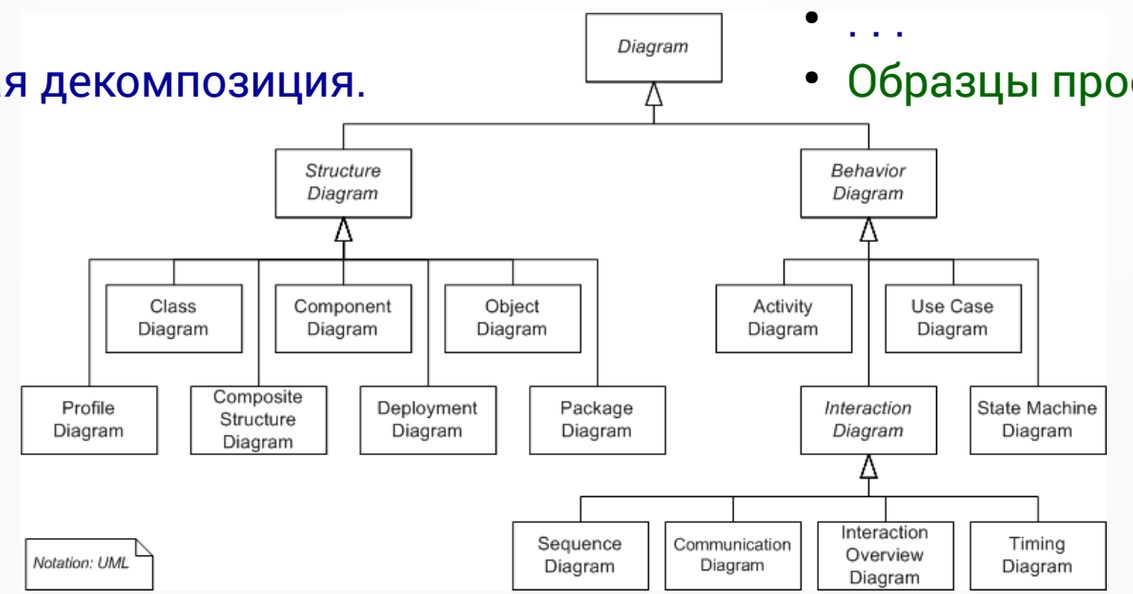
Static ← Program → Dynamic

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

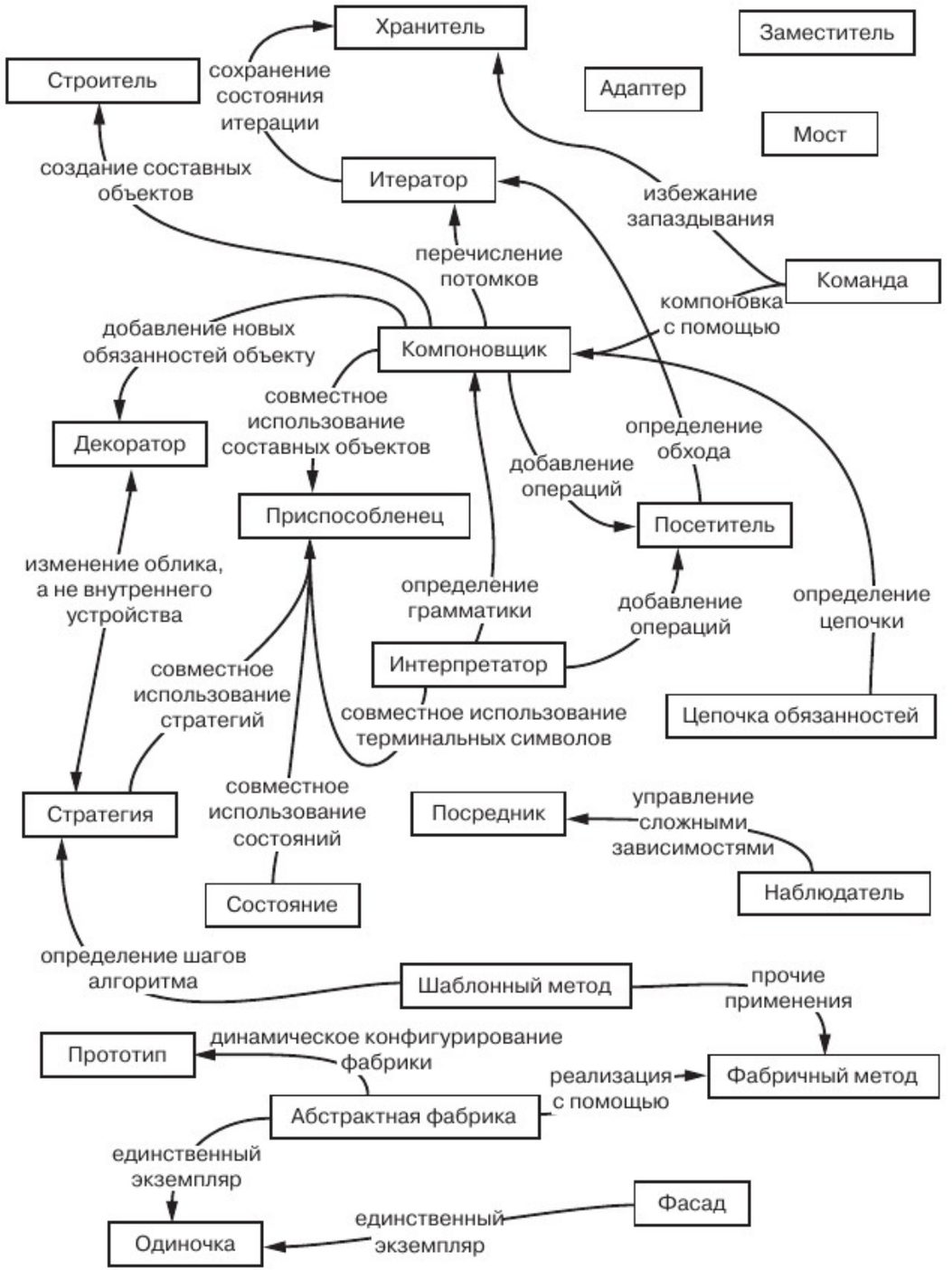


Рис. 1.1. Отношения между паттернами проектирования

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 разрабатываемой системе гибкие решения: каким образом организовать её функционирование, как построить систему, чтобы легче изменять её в будущем.

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;
- помогают 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 применять
хороший программист?

Квалификация

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

Это – основы профессионализма раработчиков.

Знания *(основы)*

Языки программирования (ООР):

- **Компилируемый:** Java | C# / C++ / Pascal / . . .
- **Скриптовый:** JavaScript / Ruby / Python / . . .
- **Командный язык ОС:** sh | bash / cmd | PowerShell . . .
- **Язык СУБД:** SQL / noSQL DBMS API

Алгоритмы:

линейные, циклические, рекурсивные, . . .

Структуры данных:

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

Форматы данных:

JSON, YAML, XML, CSV, . . .

ОО-проектирование (OOD):

design patterns, UML, clean architecture, . . .

Сетевые технологии:

HTTP, REST, sockets, . . .

Практики

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

- Effective Estimation – **оценка сроков выполнения работ.**
- Planning Game – **планирование по экспертным оценкам.**
- Test First – **применение** TDD.
- Refactoring – **привычка к реорганизации исходников.**
- 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, соглашения по оформлению исходников, средства генерирования документации из исходников.

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**, xSpec/xBehave, …

Система непрерывной интеграции (CIS):

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

Личность разработчика

В конечном счёте, всё зависит от личных качеств разработчика:
– соблюдает ли он профессиональные правила и принципы?
– обладает ли он культурой разработки?
– овладел ли он навыками и приёмами работы?
– как глубоко он освоил теоретические основы професии?
– насколько он сведущ в предметной области?
– стремится ли он стать Мастером своего дела?
– . . .

Это сложная задача: воспитать стремление к профессионализму.
Но многого можно добиться требованиями в коллективе разработчиков, при обучении новичков, примером личности Мастера.

Обучение + воспитание

Это необходимо осваивать самим.
Это нужно постоянно практиковать.

Этому нужно учить новичков.

Этому учатся при практической работе над проектом в команде.

Причины низкого профессионализма

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

- Развитие ИТ требует большого количества разработчиков.
- 1/2 разработчиков ПО имеет опыт работы менее 5 лет.
- У них недостаточный контакт с реальной культурой разработки.
- Дефицит образовательных ресурсов по обсуждаемым темам.
- Этими темами пренебрегают в университетах.
- Коммерциализация разработки не заинтересована в качестве ПО.
- Пренебрежение к культуре оправдывается прагматичными целями.
- Профессионализм – это дисциплина, а её у многих как раз не хватает.
- Требуются личная мотивация для стремления к профессионализму.

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.

Книги, которые нужно прочитать *(IMHO)*

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.

Полезные ссылки

- https://en.wikipedia.org/wiki/Software_design_pattern
- 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/
- https://ru.wikipedia.org/wiki/Покер_планирования
- 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://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

Готов ответить на вопросы

???

Дополнения

- Словарик терминов и сокращений.
- Software values paradox: *исходный текст на англйском.*
- Extreme Programming: *исходная схема на англйском.*
- Simple Design – *4 правила Кента Бека*.
- Refactorings = *реорганизации.*
- Transformations = *преобразования.*
- Темы для дальнейших разговоров.

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.
- 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 ~ .
- 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

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.

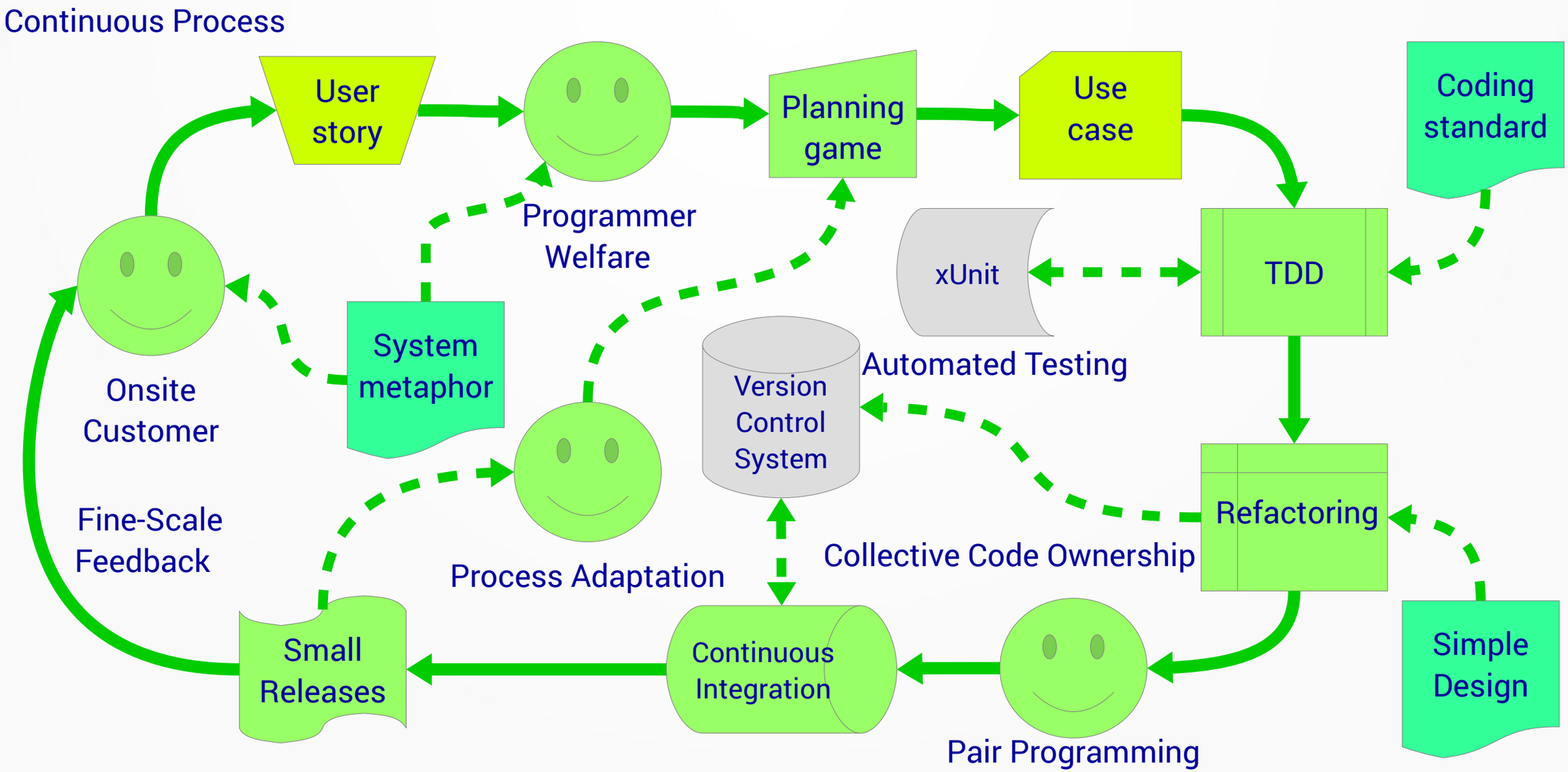
Therefore the program
will remain continually useful.

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

eXtreme Programming



Simple Design

Years ago Ron Jeffries codified Kent Beck's rules of simple design. They are, in order of priority:

1. *All the tests pass.*
2. *There is no duplication.*
3. *The code expresses the intent of the programmer.*
4. *Classes, and methods are minimized.*

Over the years we have used this as a guide for writing our code. Indeed, Kent would often say:

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>

Refactorings = Реорганизации

Реорганизации – это простые операции, которые изменяют структуру кода, не меняя его поведения.
Наиболее часто употребимые методы рефакторинга в ООП:

- Измененить сигнатуру метода (change method signature).
- Инкапсулировать поле (encapsulate field).
- Выделить класс (extract class).
- Выделить интерфейс (extract interface).
- Выделить локальную переменную (extract local variable).
- Выделить метод (extract method).
- Обобщить тип (generalize type).
- Встроить (inline).
- Ввести фабрику (introduce factory).
- Ввести параметр (introduce parameter).
- Подять метод (pull up method).
- Спустить метод (push down method).
- Переименовать метод (rename method).
- Переместить метод (move method).
- Заменить условный оператор полиморфизмом (replace conditional with polymorphism).
- Заменить наследование делегированием (replace inheritance with delegation).
- Заменить код типа подклассами (replace type code with subclasses).

Transformations = Преобразования

У реорганизаций есть противоположности – преобразования. **Преобразования** – это простые операции, которые изменяют поведение кода. Всегда нужно стремиться к простейшему варианту (в верхней части списка очередности преобразований):

1. ({} → nil) *заменить отсутствие кода на код, использующий нулевое значение (failing implementation).*
2. (nil → constant) *заменить нулевое значение константой.*
3. (constant → constant+) *заменить простую константу более сложной.*
4. (constant → scalar) *заменить константу на переменную или аргумент.*
5. (statement → statements) *добавить безусловный оператор (break, continue, return и т. п.).*
6. (unconditional → if) *разбить последовательность выполнения с помощью условного оператора.*
7. (scalar → array) *заменить переменную или аргумент массивом.*
8. (array → container) *заменить массив более сложным контейнером.*
9. (statement → tail-recursion) *заменить оператор "хвостовой" рекурсией.*
10. (if → while) *заменить условный оператор циклом.*
11. (statement → recursion) *заменить выражение рекурсией.*
12. (expression → function) *заменить выражение функцией или алгоритмом.*
13. (variable → assignment) *заменить значение переменной присваиванием нового.*
14. (case) *добавить case или else к имеющемуся switch или if.*

"As the tests get more specific, the code gets more generic."

<https://blog.cleancoder.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html>

Темы для дальнейших разговоров

- XP
- TDD
- BDD
- SOLID
- Design Patterns
- VCS