Программное дело

о разработке программ расскажет
Михаил В. Шохирев

Клуб программистов
Шадринск
2020

Почему я захотел рассказать об этом?

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

«Велосипеды изобретены» — надо научиться на них ездить.

Личная причина

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

John Newton, 1772

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

И мне захотелось поделиться с коллегами-одноклубниками.

Мои учителя

1. «Надёжность программного обеспечения» — **Гленфорд Майерс** (1980)
2. «Стиль, разработка, эффективность, отладка и испытание программ» — **Дэнни 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. «Экстремальное программирование» — **Кент Бек** (2002)
7. «Программист-прагматик. Путь от подмастерья к мастеру» — **ДэйвТомас, Энди Хант** (2004)
8. «Чистая архитектура. Искусство разработки программного обеспечения» — **Роберт Мартин** (2018)

О чём пойдёт речь

Software ~ ***взаимосвязанные ценности программного обеспечения.***

Traditional ~ ***традиционный подход к разработке.***

Agile ~ ***гибкие технологии разработки.***

X. P. ~ ***экстремальное программирование.***

TDD ~ ***разработка через тестирование.***

Testing ~ ***важность тестирования.***

Refactoring ~ ***необходимость реорганизации исходников.***

Architecture ~ ***ценность и важность архитектуры.***

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

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

Design Patterns ~ ***признанные образцы проектирования ПО.***

Tools ~ ***необходимые инструменты разработки.***

Qualification ~ ***знания и умения разработчика ПО***.

Что нужно знать и уметь разработчику

Написать программу не сложно.
Труднее сделать, чтобы она работала правильно.
Сложно разработать программу, которую можно развивать при использовании.

Для этого надо овладеть современными подходами к разработке:

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

Программное обеспечение

2 ценности программного обеспечения:

2 soft-
«изменчивый»

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

«программу легко изменить»

1 -ware
«продукт»

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

«программа работает правильно»

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 работы системы.

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.

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

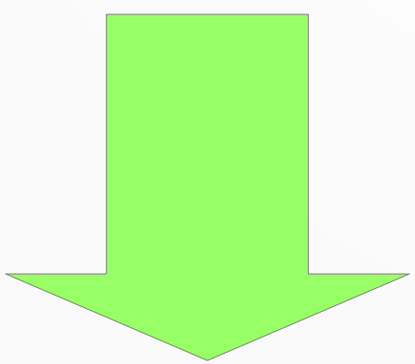
Эд Йордан

"Процесс автоматизации изменяет взгляд пользователя на объект автоматизации".

Гленфорд Майерс

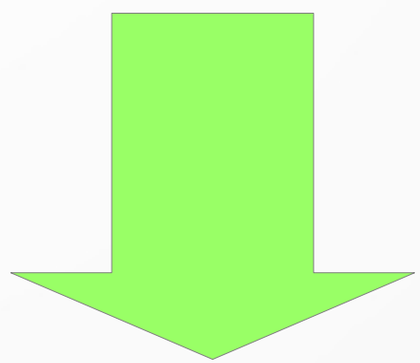
Условия для изменяемости

Уверенность, что программа продолжит правильно работать после изменения.



Надёжный набор тестов

Программа имеет структуру, которая позволяет легко вносить изменения.



“Чистая архитектура”

Традиционный подход к разработке

Waterfall

«Водопад»:

Требования: *определить и 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.*

«Зафиксировать проект – и спокойно программировать...»

Традиционный подход *не работает!*

~~Waterfall~~

«Водопад»:

Требования: *всё время меняются*.

Проектирование: *невозможно закончить полностью и в срок*.

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

Отладка: *поэтому ошибки неизбежны*.

Тестирование: *недостаточное и несвоевременное*.

Документирование: *после времени*.

Внедрение: *когда «поджимают сроки»*.

Сопровождение:

«Эта сказка хороша — начинай сначала!».

Ни зафиксировать, ни спокойно 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**.

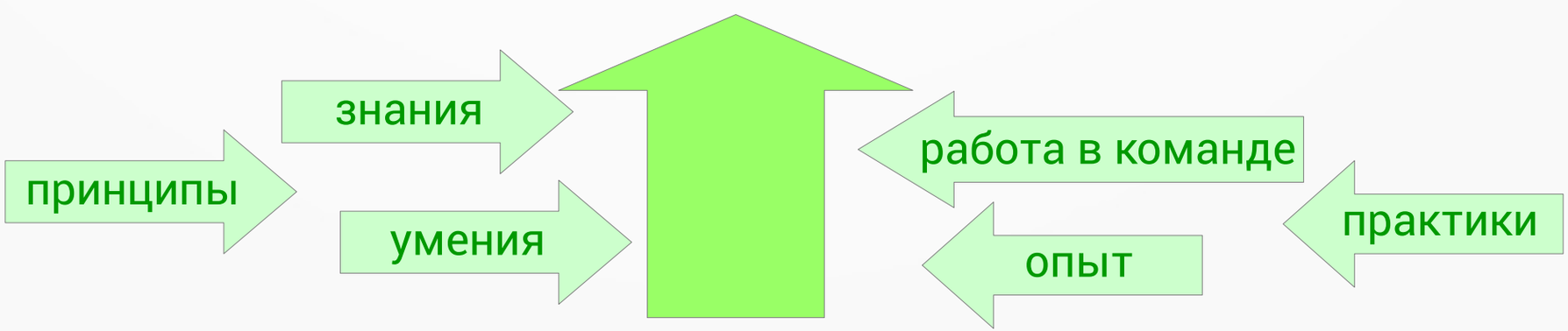
Это не просто провозглашённые принципы

— 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.

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*

12 основных приёмов XP :

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

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

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

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

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

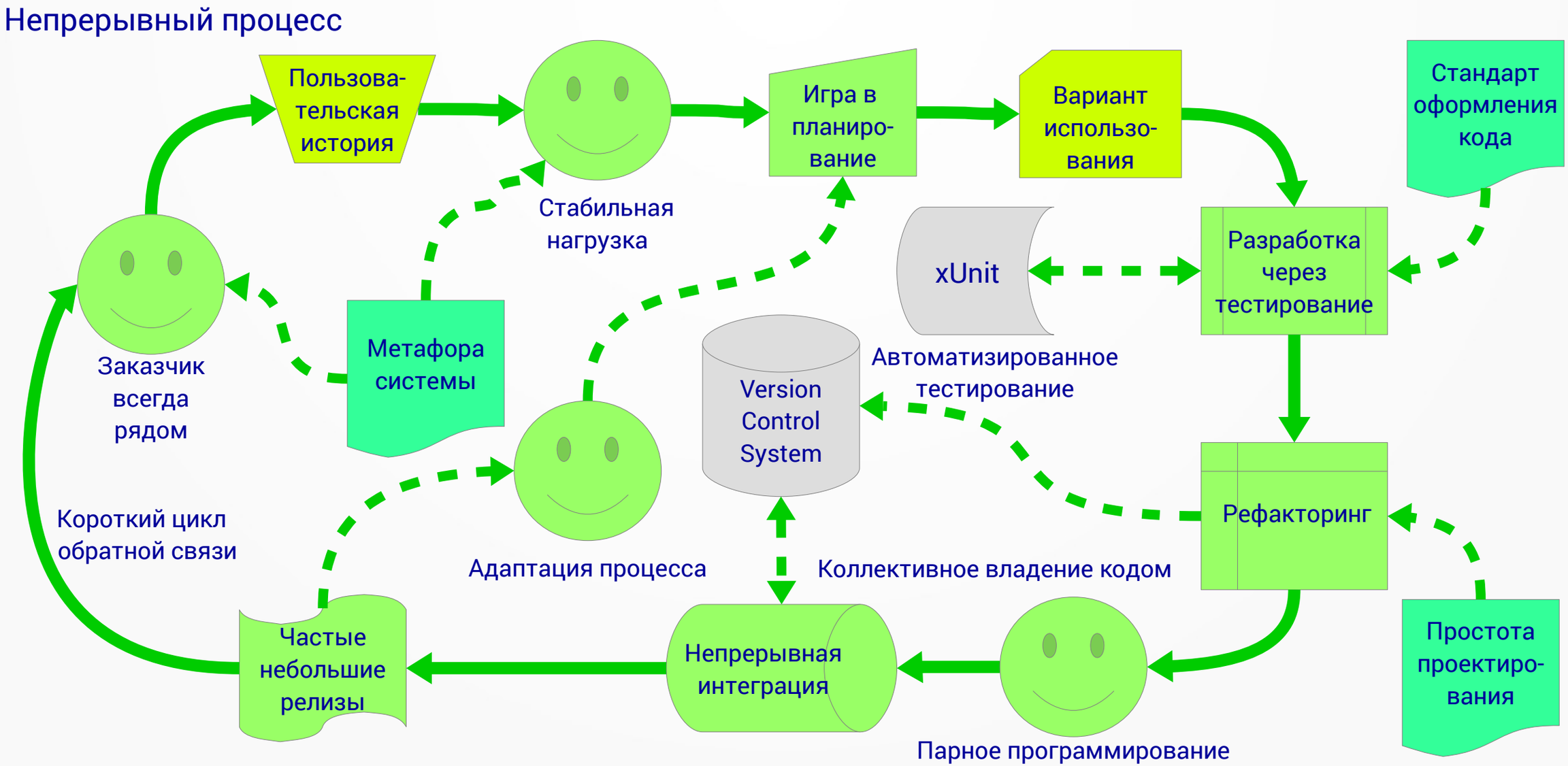
8. **Метафора системы** (System Metaphor): *система понятий, единая для заказчиков и разработчиков* → **DDD**.
9. **Простота проектирования** (Simple Design): *выбирается наиболее простой способ реализовать функциональность.*
10. **Стандарт оформления кода** (Coding Standard = Coding Conventions): *соглашения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

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

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

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

Экстремальное программирование



Гибкий подход к разработке

Agile

Короткие итерации разработки, во время которых делаются:

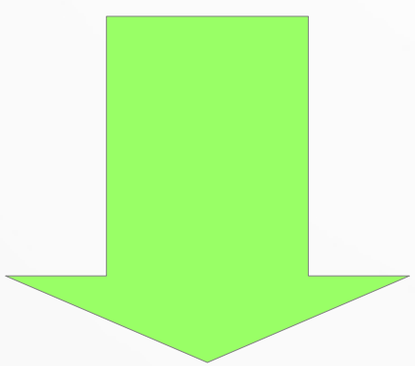
- Требования: *«истории пользователя» только на эту итерацию.*
- Проектирование: *наиболее простая реализация «user stories».*
- *Парное* программирование *через* тестирование.
- Отладка: *практически не требуется после TDD.*
- Тестирование: *приёмочное — показать, что требования выполнены.*
- *Минимальное* документирование: *т. к. есть “чистый код” и тесты.*
- Внедрение: *в конце каждой итерации.*

Сопровождения *нет – есть только разработка!*

Заказчик хочет развития программы – разработчик это легко делает!

Условия для изменяемости

Уверенность, что программа продолжит правильно работать после изменения.



Надёжный набор тестов

Тестирование

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

Тесты — это...

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

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

Виды тестов

unit test
(модульный тест)

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

acceptance test
(приёмочный тест)

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

Польза и важность тестирования

Имея надёжный набор тестов, ...

... можно рефакторить программу без опаски.

... можно смело править программные ошибки.

... можно уверенно изменять и развивать программу.

Имея полный набор тестов, можно даже воссоздать потерянную программу.

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

Test First = TDD

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 соблюдение технологической дисциплины разработчиками (как двойная запись в бухучёте).

Применение TDD основано на автоматизированных модульных тестах.

TDD = test-driven development = test 1st

0 1 2

Red → Green → Refactor!

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

Потом цикл повторяется снова… И набор тестов постепенно пополняется.

Хороший тест

- Проверяет одно требование.
- Тестирует интерфейс модуля.
- Короткий (3 строки - *Given-When-Then*):

  Если *у меня имеется пустой стек,*

  Когда *я затолкну в него что-нибудь,*

  Тогда *его размер станет равен 1.*
- Быстро выполняется.
- Выполняется автоматически.

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

Refactoring

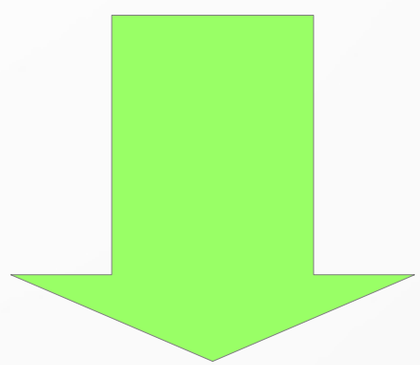
Refactoring = перепроектирование кода = переработка кода = равносильное преобразование алгоритмов ~ изменение внутренней структуры программы, не затрагивающий её внешнего поведения с целью облегчить понимание её работы. Рефакторинг выполняется как последовательность небольших эквивалентных (то есть сохраняющих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

Рефакторинг — обязательная часть разработки и постоянная деятельность программиста.

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

Условия для изменяемости

Программа имеет структуру, которая позволяет лекго вносить изменения.



“Чистая архитектура”

Архитектура = дизайн

Архитектура выражает замысел ПО, его назначение (*система заказов, учётная система, система мониторинга и т. п.*).
Она строится на основе вариантов использования ПО (*размещение заказа пользователем, обработка заказа диспетчером и т. п.*).

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

Что не является архитектурой ПО

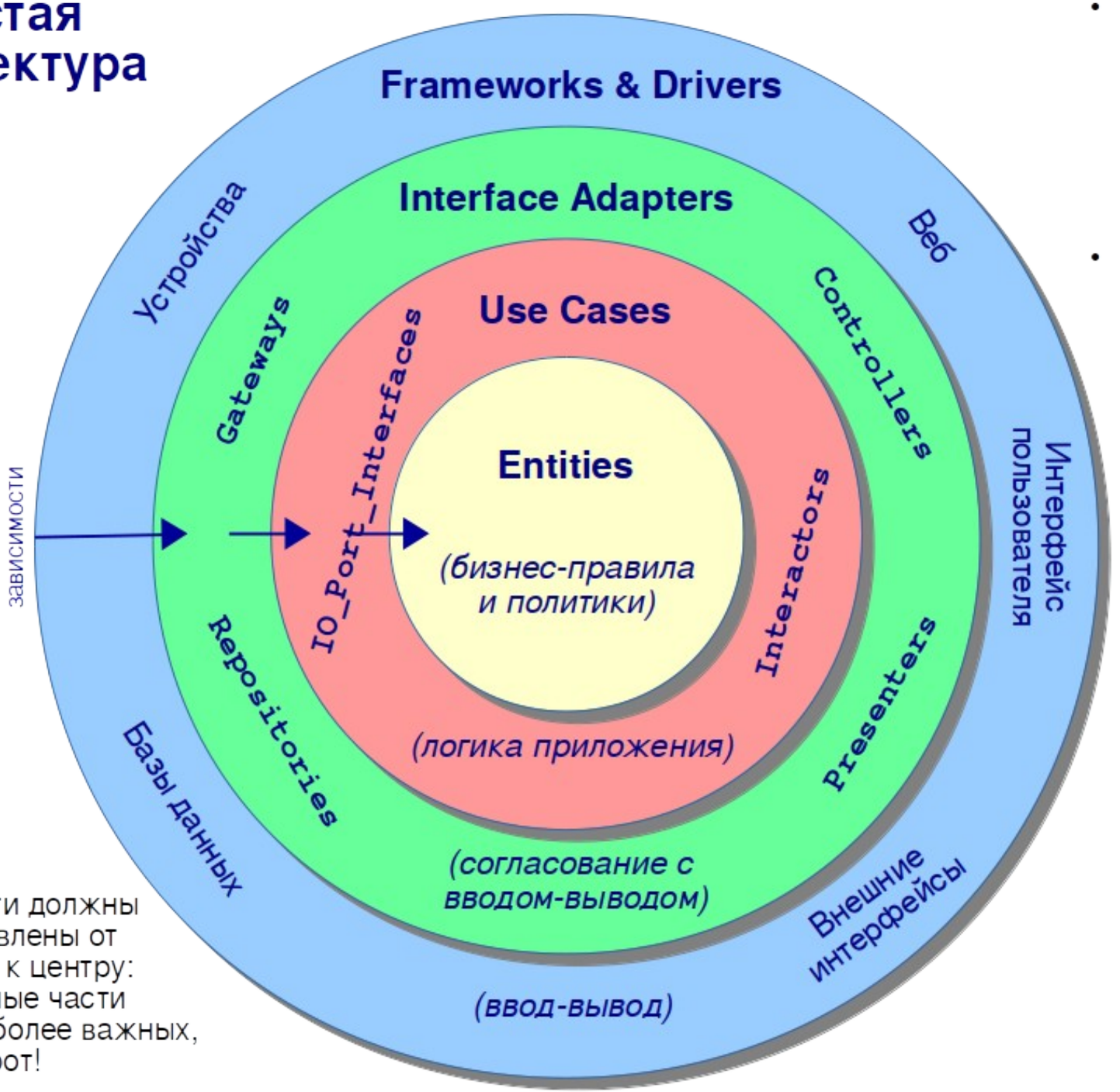
~~Subscribe - Publish~~ = протокол обмена.
~~Framework~~ = инструмент / компонента.
~~MVC (Model-View-Controller)~~ = образец проектирования.
~~Client – Server~~ = сетевая архитектура = конфигурация узлов.
~~Database~~ = хранилище = устройство ввода-вывода.
~~Web~~ = механизм доставки данных = устройство ввода-вывода.
~~SOA (service-oriented architecture)~~ = = устройство ввода-вывода.

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 делается без всякой автоматизации*);
- варианты использования – логика работы программной системы (*как обработка данных будет выполняться 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.

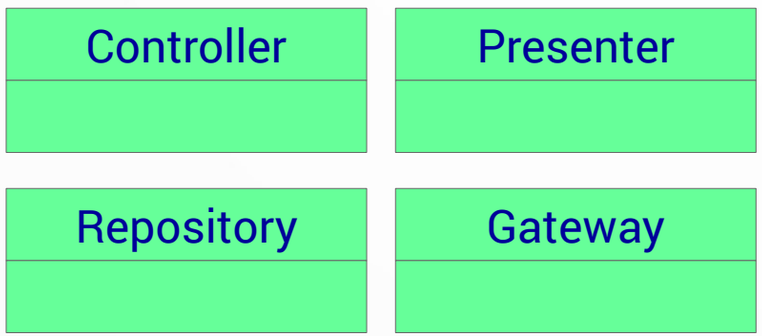
Зависимости должны быть направлены от периферии к центру: менее важные части зависят от более важных, а не наоборот!

Виды архитектурных компонентов

Ввод-вывод

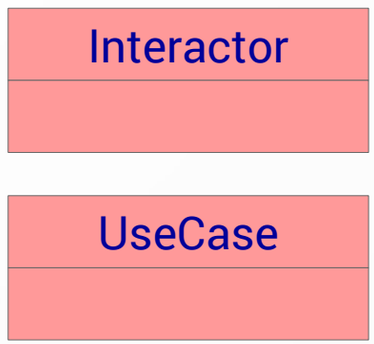
Классы для объектов, реализующих механизм доставки данных: источники и приёмники данных, средства взаимодействия с человеком и внешним ПО.

Согласование с вводом-выводом



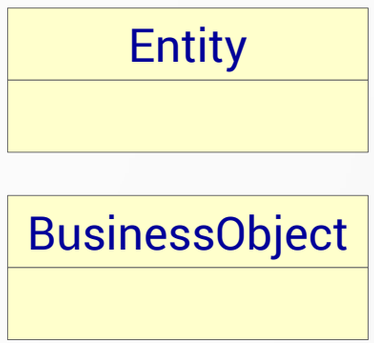
Классы для объектов, реализующих интерфейсы и адаптеры, которые организуют предоставление данных в нужном виде.

Варианты использования



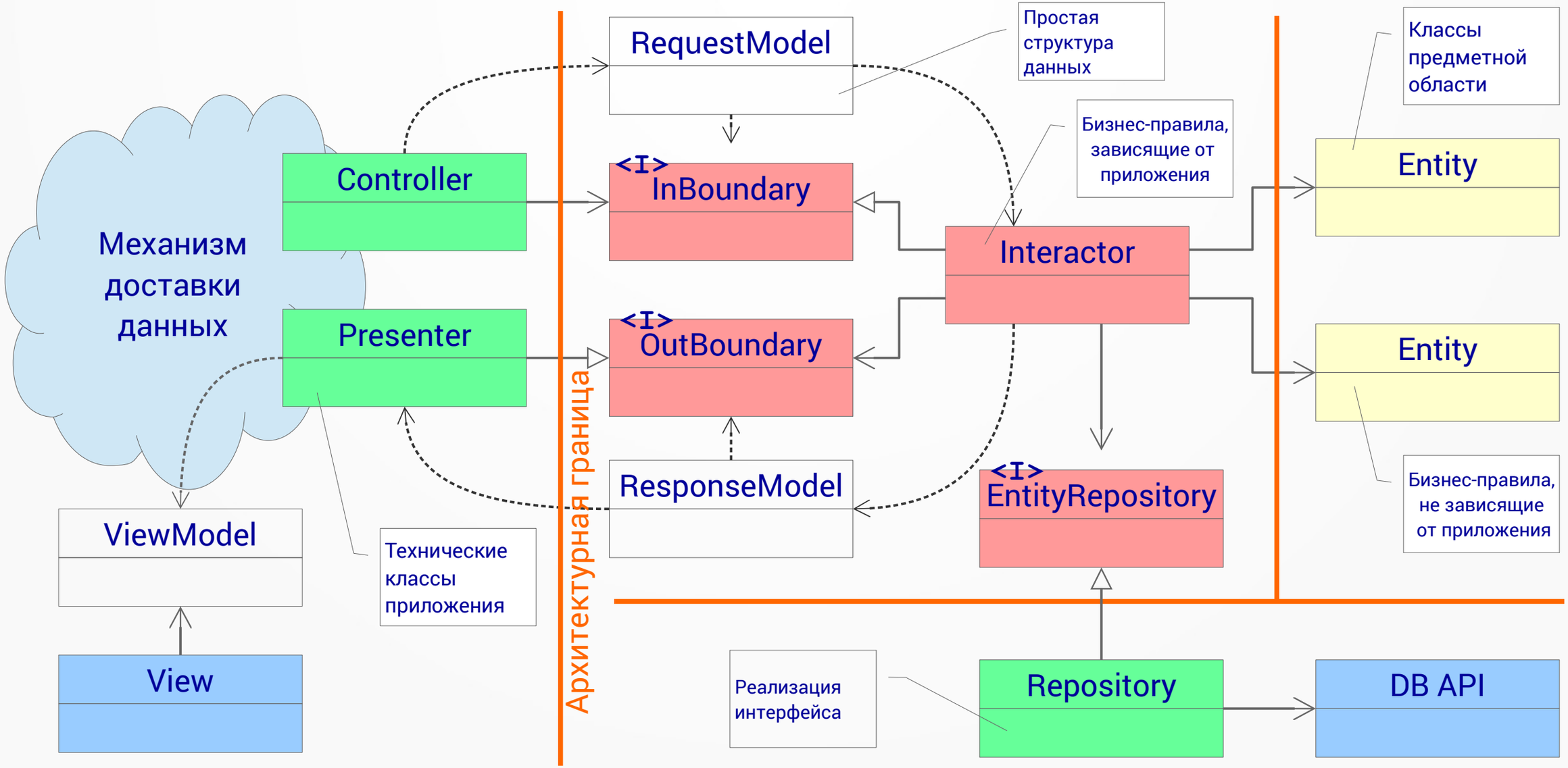
Классы для объектов программной системы, реализующих автоматизированную обработку данных, зависящие от бизнес-правил и политик.

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 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 нескольких основных принципах, известных как S. O. L. I. D.

S. O. L. I. D. – принципы OOD

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

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

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

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

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

SRP ← S. O. L. I. D.

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

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

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

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

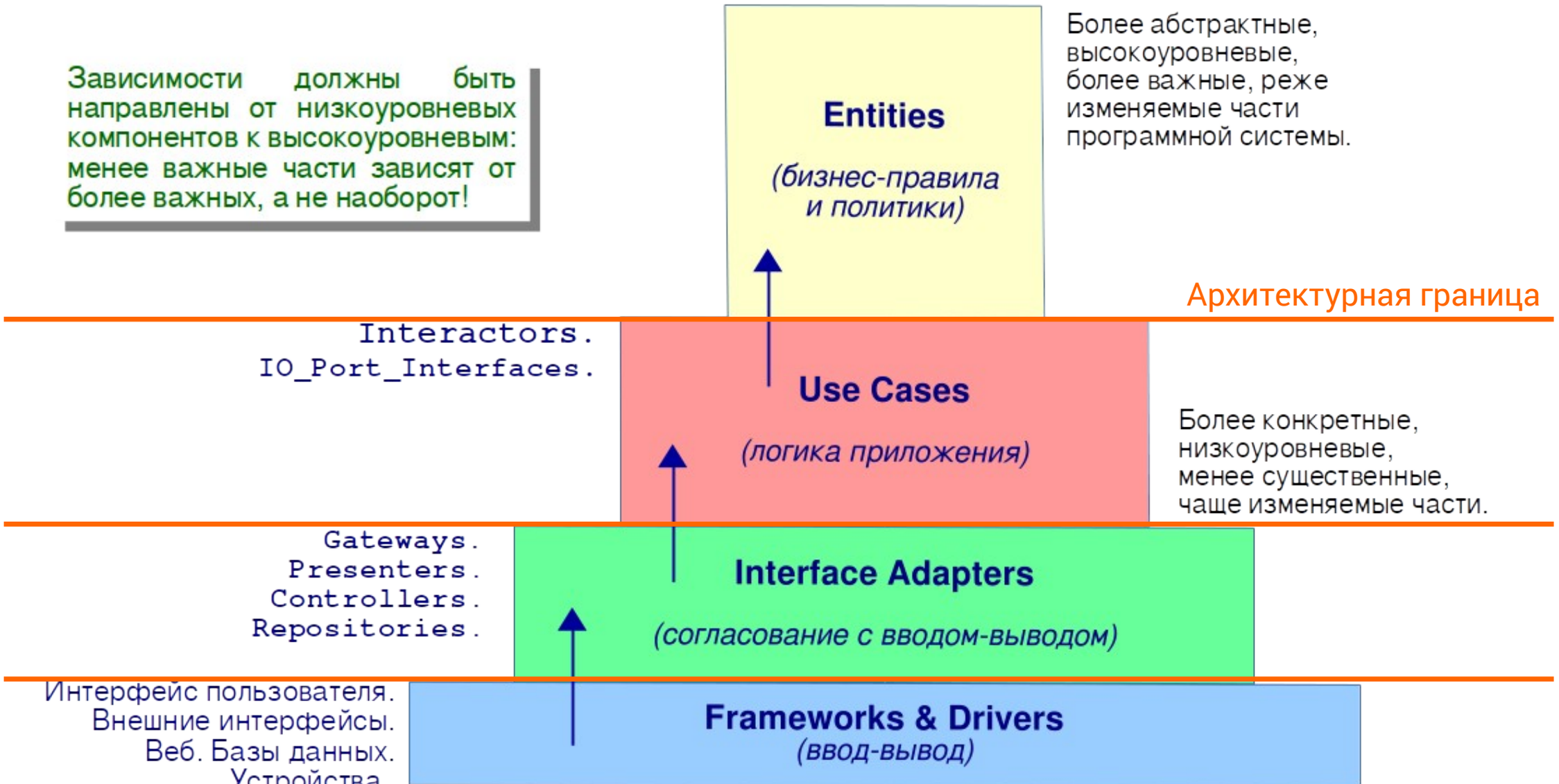
Архитектурная граница проходит между компонентами разных уровней программной системы.

Разделение системы на *уровни с управляемыми зависимостями* позволяет:

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

Чистая архитектура

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



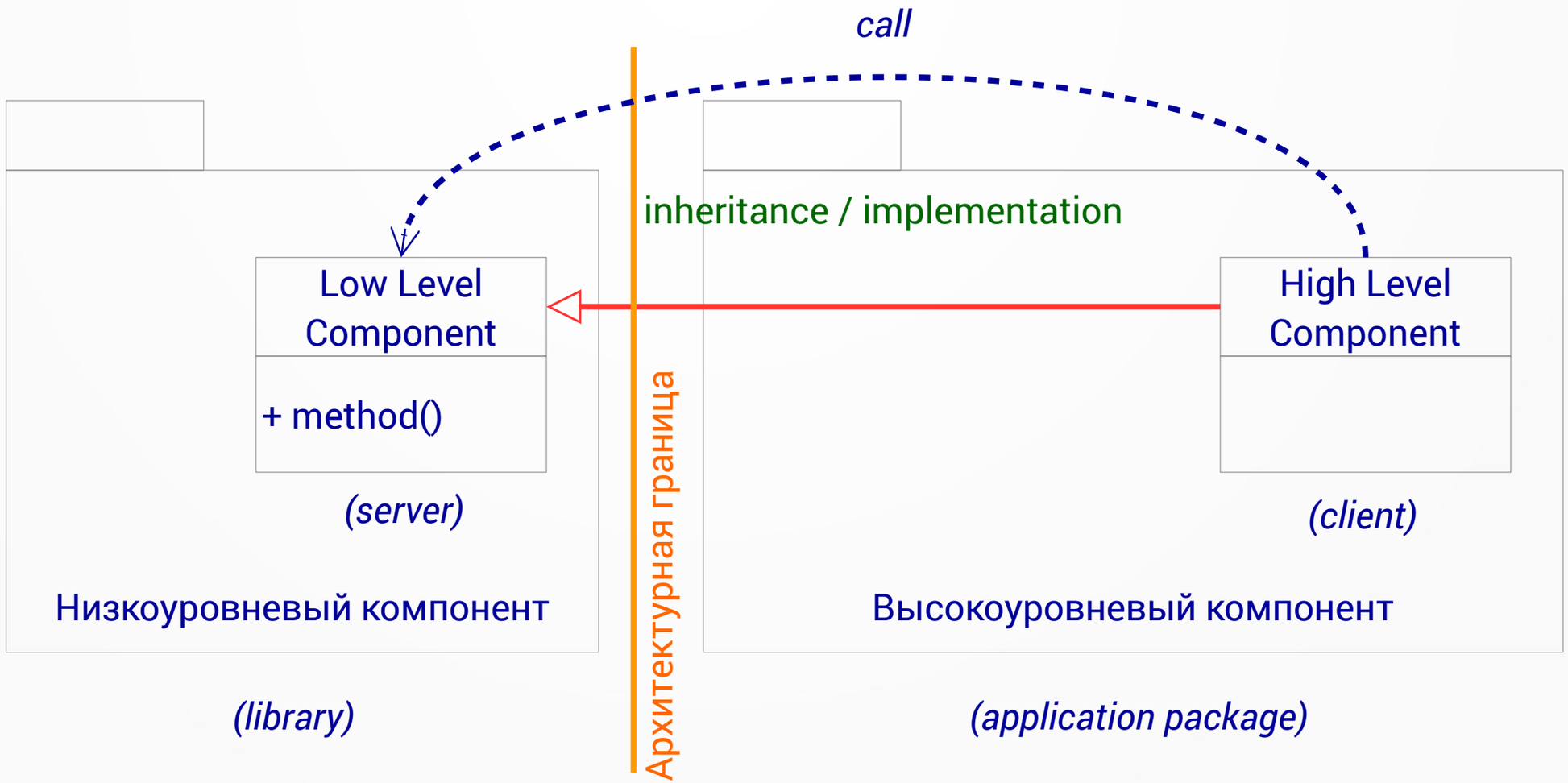
Зависимости

Зависимости:

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

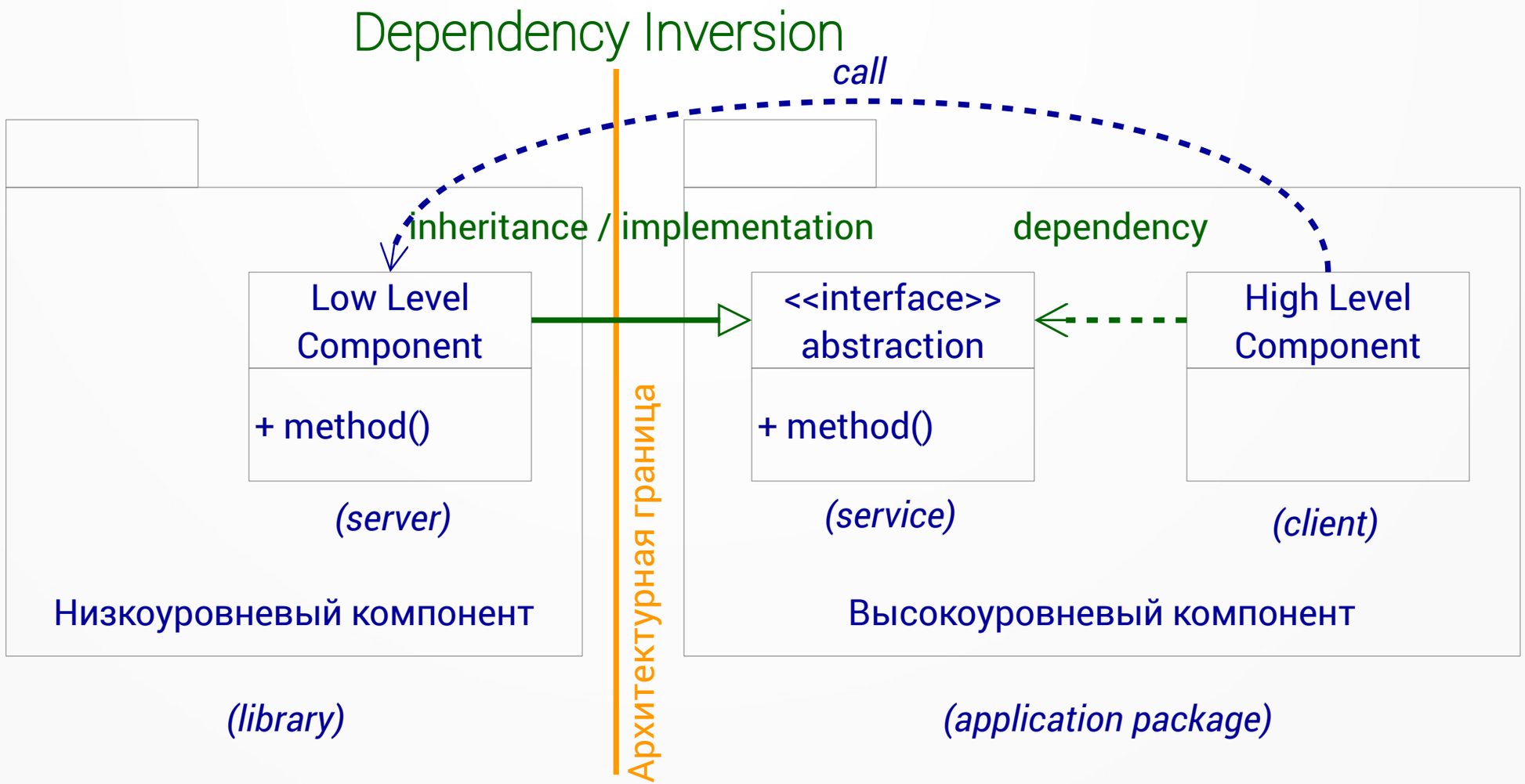
Главное правило построения надёжной модифицируемой системы:
зависимости должны быть направлены от низкоуровневых компонентов к высокоуровневым: менее важные части должны зависеть от более важных, а не наоборот!

Зависимость исходного кода



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);
- все программы можно записать с помощью 3-х конструкций (последовательность, ветвление, цикл);
- модульное программирование.

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

Модульность должна быть «в крови» у разработчика.

Ценность F. P.

Функциональное программирование:

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

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

Ценность О. О. Р. для архитектуры

Объектно-ориентированное программирование:

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

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

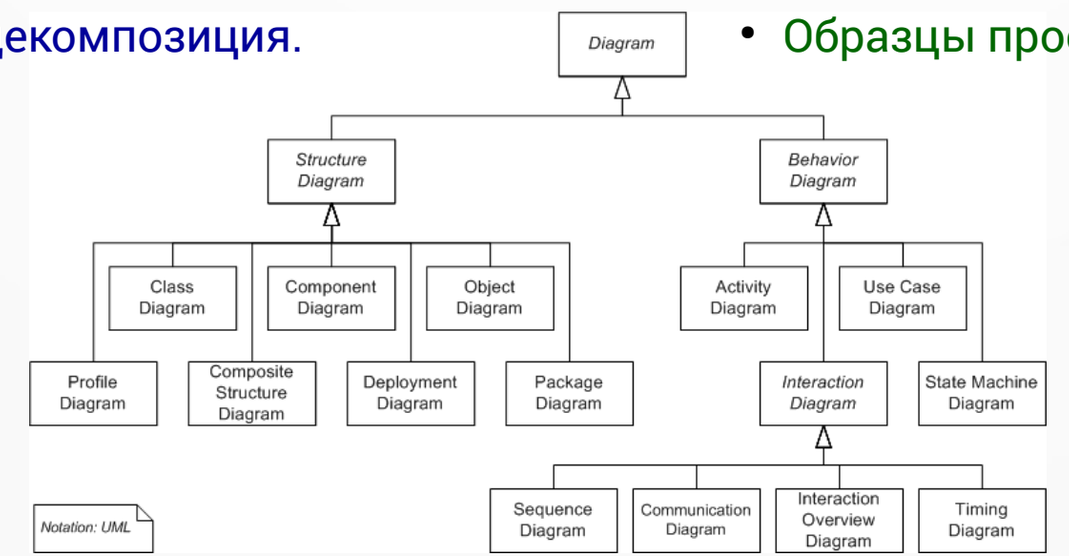
Static ← Program → Dynamic

Статика

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

Динамика

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



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

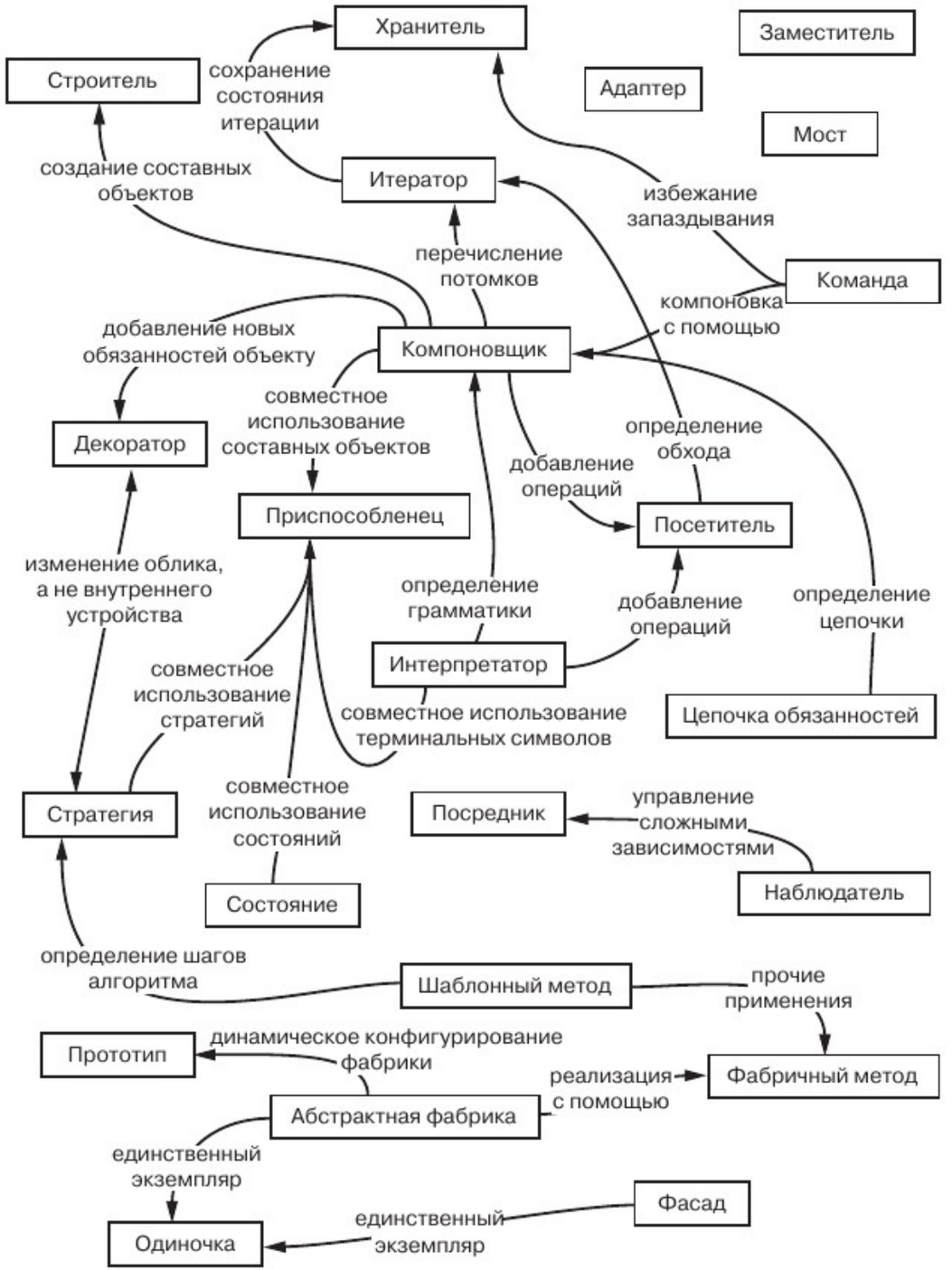


Рис. 1.1. Отношения между паттернами проектирования

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 Design Patterns даёт возможность заложить в разрабатываемой системе гибкие решения, которые позволят 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;
- экономят время разработчиков.

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

Языки программирования (OOP):

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

Алгоритмы:

. . .

Структуры данных:

массив, хэш, дерево, . . .

ОО-проектирование (OOD):

clean architecture, design patterns, . . .

Сетевые технологии:

HTTP, REST, sockets, . . .

Практики

Практики – неуклонное 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.

Большинство практик имеет смысл
только, если их применяют ВСЕГДА, без исключений.

Инструменты разработчика

Документирование:

User Story = Use Case, UML, оформление 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**, ...

Система непрерывной интеграции (CIS):

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

Обучение

Это необходимо осваивать самим.
Этому нужно учить новичков.

Этому учатся при практической работе над проектом в команде.

Книги, которые нужно прочитать

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

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://en.wikipedia.org/wiki/Planning_poker
- https://8thlight.com/blog/micah-martin/2012/11/17/transformation-priority-premise-applied.html

Готов ответить на вопросы

???

Дополнения

- Software values paradox
- Extreme Programming
- Simple Design
- Refactorings.
- Transformations.

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.

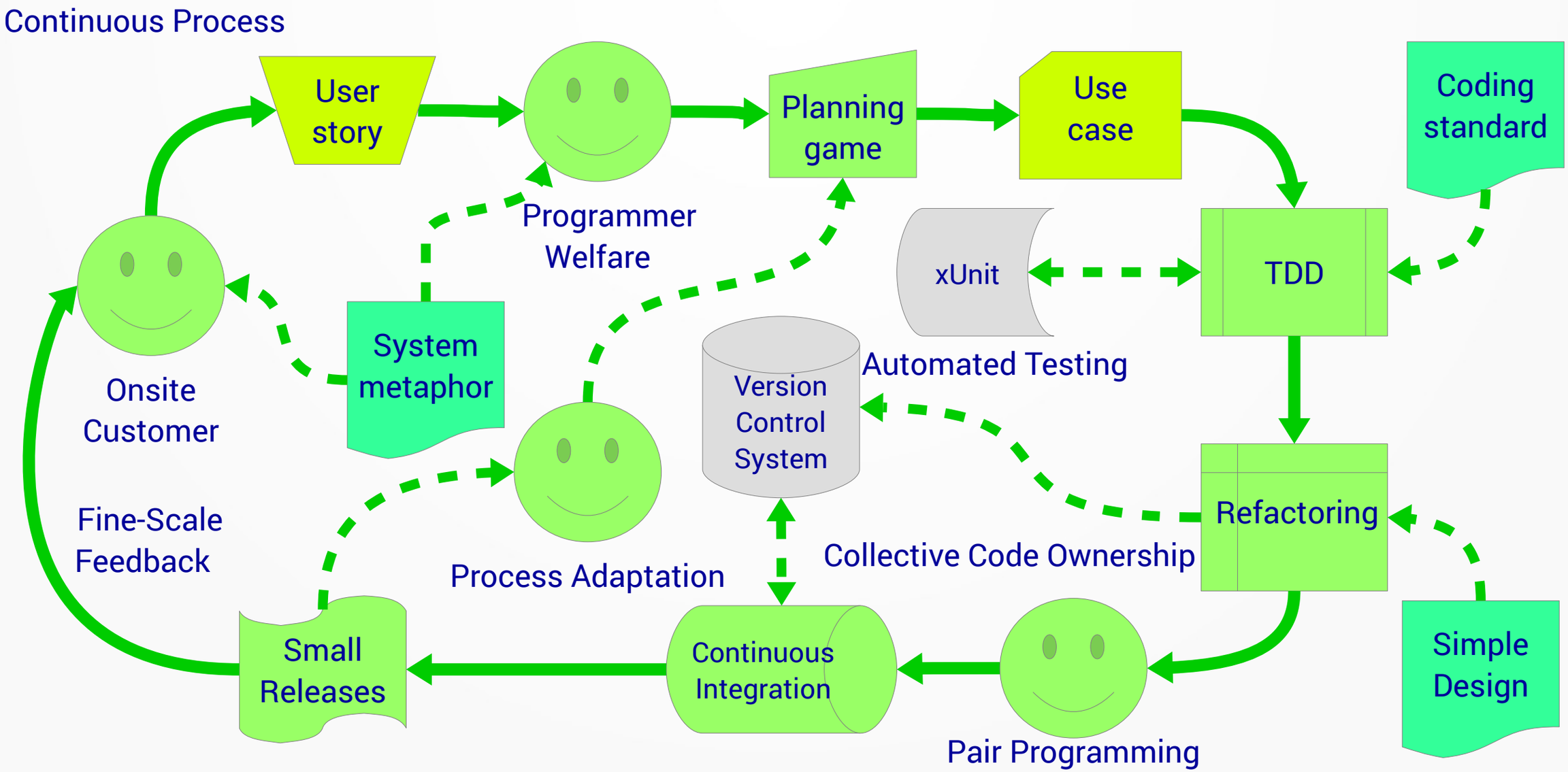
Therefore the program
will remain continually useful.

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

Наиболее употребимые методы рефакторинга:

- Изменение сигнатуры метода (change method signature)
- Инкапсуляция поля (encapsulate field)
- Выделение класса (extract class)
- Выделение интерфейса (extract interface)
- Выделение локальной переменной (extract local variable)
- Выделение метода (extract method)
- Генерализация типа (generalize type)
- Встраивание (inline)
- Введение фабрики (introduce factory)
- Введение параметра (introduce parameter)
- Подъём метода (pull up method)
- Спуск метода (push down method)
- Переименование метода (rename method)
- Перемещение метода (move method)
- Замена условного оператора полиморфизмом (replace conditional with polymorphism)
- Замена наследования делегированием (replace inheritance with delegation)
- Замена кода типа подклассами (replace type code with subclasses

Transformations

Refactorings have counterparts called Transformations. Refactorings are simple operations that change the structure of code without changing it's behavior. **Transformations** are simple operations that change the behavior of code:

1. ({} → nil) no code at all → code that employs nil
2. (nil → constant)
3. (constant → constant+) a simple constant to a more complex constant
4. (constant → scalar) replacing a constant with a variable or an argument
5. (statement → statements) adding more unconditional statements.
6. (unconditional → if) splitting the execution path
7. (scalar → array)
8. (array → container)
9. (statement → tail-recursion)
10. (if → while)
11. (statement → recursion)
12. (expression → function) replacing an expression with a function or algorithm
13. (variable → assignment) replacing the value of a variable.
14. (case) adding a case (or else) to an existing switch or if

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

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

?

Что бы вы ещё желали обсудить?