Оглавление

[Intro to Agile 3](#_Toc314256481)

[Гибкие методики разработки 3](#_Toc314256482)

[Манифест гибких методик разработки 4](#_Toc314256483)

[Основные черты гибких методологий 5](#_Toc314256484)

[Scrum 5](#_Toc314256485)

[Жизненный цикл разработки и спринта в Scrum (2 уровня) 6](#_Toc314256486)

[Роли в Scrum 7](#_Toc314256487)

[Артефакты в Scrum 8](#_Toc314256488)

[Практики Scrum 9](#_Toc314256489)

[XP 11](#_Toc314256490)

[XP 11](#_Toc314256491)

[Практики XP 11](#_Toc314256492)

[Парное программирование 13](#_Toc314256493)

[MVP 14](#_Toc314256494)

[Типичные проблемы с UI 14](#_Toc314256495)

[Passive View, диаграмма и назначение паттерна 15](#_Toc314256496)

[Ответственности слоев в шаблоне MVP 15](#_Toc314256497)

[Data Access 15](#_Toc314256498)

[Преимущества введения Data Access layer 15](#_Toc314256499)

[DDD 17](#_Toc314256500)

[DDD 17](#_Toc314256501)

[Назначение сборок DDD, их типичные классы 17](#_Toc314256502)

[SOLID 17](#_Toc314256503)

[SRP 17](#_Toc314256504)

[OCP 18](#_Toc314256505)

[LSP 18](#_Toc314256506)

[ISP 19](#_Toc314256507)

[DIP 19](#_Toc314256508)

[Закон Деметера 19](#_Toc314256509)

[TDD 20](#_Toc314256510)

[Определение и назначение TDD 20](#_Toc314256511)

[Цикл написания кода в технике TDD 21](#_Toc314256512)

[Цикл разработки через тестирование 21](#_Toc314256513)

[Атрибуты хорошего теста 22](#_Toc314256514)

[Collaboration Tools 22](#_Toc314256515)

[Непрерывная интеграция 22](#_Toc314256516)

[Clean Code 22](#_Toc314256517)

[Ключевые понятия Clean Code 22](#_Toc314256518)

[Именование 22](#_Toc314256519)

[Комментарии: достоинства и недостатки 23](#_Toc314256520)

[Refactoring 23](#_Toc314256521)

[Refactoring 23](#_Toc314256522)

# Intro to Agile

## Гибкие методики разработки

Определение гибких методик разработки, примеры, их место в программной инженерии

O) – это упрощенная методика организации производства для небольших и средних по размеру команд специалистов занимающиъся разработкой программного продукта в условиях неясных или быстро меняющихся требований.

Примеры:

1. **Scrum** — методология управления разработкой информационных систем для [гибкой разработки программного обеспечения](http://ru.wikipedia.org/wiki/%D0%93%D0%B8%D0%B1%D0%BA%D0%B0%D1%8F_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F). Scrum чётко делает акцент на качественном контроле процесса разработки.

Подход впервые описали Хиротака Такеути и Икудзиро Нонака[[1]](http://ru.wikipedia.org/wiki/Scrum#cite_note-0) в статье *The New Product Development Game* (*Гарвардский Деловой Обзор*[[2]](http://ru.wikipedia.org/wiki/Scrum#cite_note-1),*январь-февраль 1986*). Они отметили, что проекты, над которыми работают небольшие, кросс-функциональные команды, обычно систематически производят лучшие результаты, и объяснили это как «подход регби». В [1991 году](http://ru.wikipedia.org/wiki/1991_%D0%B3%D0%BE%D0%B4) ДеГрейс и Шталь в книге *«Злые проблемы, справедливые решения»*[[3]](http://ru.wikipedia.org/wiki/Scrum#cite_note-2) ссылались на этот подход, как на Scrum (*толкотня; схватка вокруг мяча (в регби)*), спортивный термин, приведённый в статье Такеути и Нонакой. [Кен Швабер](http://ru.wikipedia.org/w/index.php?title=%D0%9A%D0%B5%D0%BD_%D0%A8%D0%B2%D0%B0%D0%B1%D0%B5%D1%80&action=edit&redlink=1) в начале [1990-х](http://ru.wikipedia.org/wiki/1990-%D0%B5) использовал подход, который привел Scrum в его компанию. Впервые метод Scrum был представлен на общее обозрение задокументированным, чётко сформированным и описанным совместно Сазерлендом и Швабером на[OOPSLA’96](http://ru.wikipedia.org/w/index.php?title=OOPSLA&action=edit&redlink=1)[[4]](http://ru.wikipedia.org/wiki/Scrum#cite_note-3) ([англ.](http://en.wikipedia.org/wiki/OOPSLA)) в [Остине](http://ru.wikipedia.org/wiki/%D0%9E%D1%81%D1%82%D0%B8%D0%BD_(%D0%A2%D0%B5%D1%85%D0%B0%D1%81)). Швабер и Сазерленд на протяжении следующих лет работали вместе, чтобы обработать и описать весь их опыт и лучшие практические образцы для индустрии в одно целое, в ту методологию, что известна сегодня как Scrum. Швабер объединил усилия с Майком Бидлом[[5]](http://ru.wikipedia.org/wiki/Scrum#cite_note-4) в [2001 году](http://ru.wikipedia.org/wiki/2001_%D0%B3%D0%BE%D0%B4), чтобы детально описать метод в книге «Agile Software Development with SCRUM». Не обращая внимания на то, что для Scrum предрекли судьбу управления проектами по разработке ПО, он может также использоваться в работе команд обслуживания программного обеспечения (*software maintenance teams*), или как подход управления разработкой и сопровождением программ: *Scrum of Scrums*.

1. **Экстрема́льное программи́рование** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Extreme Programming*, *XP*) — одна из [гибких методологий разработки](http://ru.wikipedia.org/wiki/%D0%93%D0%B8%D0%B1%D0%BA%D0%B0%D1%8F_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8) [программного обеспечения](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5). Авторы методологии — [Кент Бек](http://ru.wikipedia.org/w/index.php?title=%D0%91%D0%B5%D0%BA,_%D0%9A%D0%B5%D0%BD%D1%82&action=edit&redlink=1), [Уорд Каннингем](http://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D0%BD%D0%BD%D0%B8%D0%BD%D0%B3%D0%B5%D0%BC,_%D0%A3%D0%BE%D1%80%D0%B4), [Мартин Фаулер](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%80%D1%82%D0%B8%D0%BD_%D0%A4%D0%B0%D1%83%D0%BB%D0%B5%D1%80) и другие.
2. **Бережливая разработка программного обеспечения** — [методология разработки программного обеспечения](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F), использующая методы концепции [бережливого производства](http://ru.wikipedia.org/wiki/%D0%91%D0%B5%D1%80%D0%B5%D0%B6%D0%BB%D0%B8%D0%B2%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%B4%D1%81%D1%82%D0%B2%D0%BE). Возникла из среды сторонников концепции [гибкой методологии разработки](http://ru.wikipedia.org/wiki/%D0%93%D0%B8%D0%B1%D0%BA%D0%B0%D1%8F_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8).

Впервые освещена в одноимённой книге ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Lean Software Development*) Мэри Поппендик и Toма Поппендика. В книге представлены традиционные принципы бережливого производства применительно к разработке программного обеспечения, также набор из 22 инструментов (практик) и их сравнение с гибкой методологией разработки. Мэри и Том участвовали в ряде различных конференций, посвящённых методике Agile, что объясняет известность концепции бережливого производства среди сообщества гибкой методология разработки.

Основополагающие принципы Agile-манифеста

Наивысшим приоритетом для нас является удовлетворение потребностей   
заказчика, благодаря регулярной и ранней поставке ценного программного   
обеспечения.

Изменение требований приветствуется, даже на поздних стадиях разработки.   
Agile-процессы позволяют использовать изменения для обеспечения заказчику  
конкурентного преимущества.

Работающий продукт следует выпускать как можно чаще, с периодичностью   
от пары недель до пары месяцев.

На протяжении всего проекта разработчики и представители бизнеса должны   
ежедневно работать вместе.

Над проектом должны работать мотивированные профессионалы. Чтобы   
работа была сделана, создайте условия, обеспечьте поддержку и полностью   
доверьтесь им.

Непосредственное общение является наиболее практичным и эффективным   
способом обмена информацией как с самой командой, так и внутри команды.

Работающий продукт — основной показатель прогресса.

Инвесторы, разработчики и пользователи должны иметь возможность   
поддерживать постоянный ритм бесконечно. Agile помогает наладить такой   
устойчивый процесс разработки.

Постоянное внимание к техническому совершенству и качеству   
проектирования повышает гибкость проекта.

Простота — искусство минимизации лишней работы — крайне необходима.

Самые лучшие требования, архитектурные и технические решения рождаются   
у самоорганизующихся команд.

Команда должна систематически анализировать возможные способы   
улучшения эффективности и соответственно корректировать   
стиль своей работы.

## Манифест гибких методик разработки

история? сам манифест

История

В феврале 2001 в штате Юта США был выпущен [Манифест гибкой методологии разработки ПО](http://agilemanifesto.org/). Он являлся альтернативой управляемым документацией, «тяжеловесным» практикам разработки программного обеспечения, таким как «метод водопада», являвшимся золотым стандартом разработки в то время. Данный манифест был одобрен и подписан представителями методологий Extreme programming (XP), Crystal Clear, DSDM, Feature-Driven Development, Scrum, Adaptive Software Development, Pragmatic Programming. Гибкая методология разработки использовалась многими компаниями и до принятия манифеста, однако именно после этого события произошло вхождение Agile-разработки в массы.

Agile-манифест разработки программного обеспечения

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 |

## Основные черты гибких методологий

(планирование, разработка, команда)

1. Команда разработчиков самостоятельно организует свою работу
2. Располагаются в общем открытом офисе
3. Ежедневные собрания
4. Тесная связь с заказчиком
5. Личное общение предпочтительней документации.

# Scrum

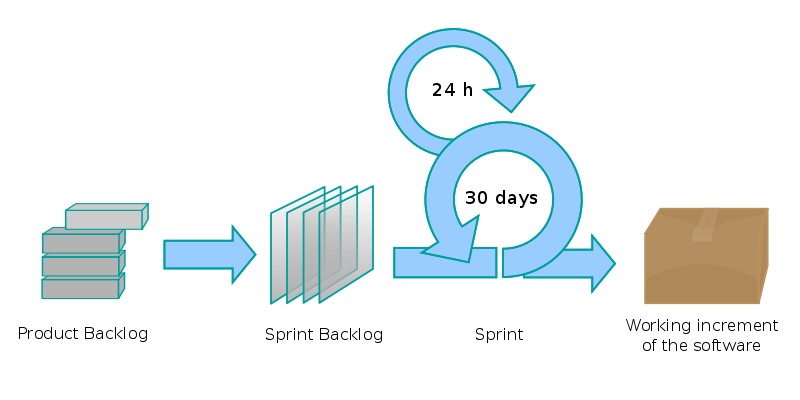
Scrum — это набор принципов, на которых строится процесс разработки, позволяющий в жёстко фиксированные небольшие промежутки времени (спринты[[6]](http://ru.wikipedia.org/wiki/Scrum#cite_note-5) от 2 до 4 недель) предоставлять конечному пользователю работающее ПО с новыми возможностями, для которых определён наибольший приоритет. Возможности ПО к реализации в очередном спринте определяются в начале спринта на этапе планирования и не могут изменяться на всём его протяжении. При этом строго-фиксированная небольшая длительность спринта придаёт процессу разработки предсказуемость и гибкость.

Главные действующие роли в Scrum: ScrumMaster — тот, кто ведёт Scrum митинги и следит, чтобы при этом соблюдались все принципы Scrum (роль не предполагает ничего кроме корректного ведения самого Scrum-а, руководитель проекта скорее относится к Product Owner и не должен являться ScrumMaster); Владелец Продукта(Product Owner) — человек, который представляет интересы конечных пользователей и других заинтересованных в продукте сторон; и кросс-функциональная Команда (Scrum Team), состоящая как из разработчиков, так и из тестировщиков, архитекторов, аналитиков и т. д. (при этом размер команды в идеале составляет 7±2 человека). Команда является единственным полностью вовлечённым участником разработки, и отвечает за результат как единое целое. Никто кроме команды не может вмешиваться в процесс разработки на протяжении спринта.

На протяжении каждого спринта создаётся функциональный рост программного обеспечения. Набор возможностей, которые реализуются в каждом спринте, происходят из этапа, называемого product backlog (документация запросов на выполнение работ), обладающего наивысшим приоритетом по уровню требований к работе, который должен быть выполнен. Запросы на выполнение работ (backlog items), определенных на протяжении совета по планированию спринта (sprint planning meeting), перемещаются в этап спринта. На протяжении этого собрания Владелец Продукта информирует о заданиях, которые должны быть выполнены. Тогда Команда определяет, сколько из желаемого они могут выполнить, чтобы завершить необходимые части на протяжении следующего спринта[[7]](http://ru.wikipedia.org/wiki/Scrum#cite_note-schwaber-6). Во время спринта команда выполняет определенный фиксированный список заданий (т. н. sprint backlog). На протяжении этого периода никто не имеет права менять список требований к работе, что следует понимать как заморозку требований (requirements) во время спринта.

## Жизненный цикл разработки и спринта в Scrum (2 уровня)

Pre-game, product backlog, sprint backlog, deployment; shippable increment; scrum meetings



Product backlog — это документ, содержащий список требований к функциональности, которые упорядочены по степени важности. Product backlog представляет собой список того, что должно быть реализовано. Элементы этого списка называются «историями» (user story) или элементами backlog’a (backlog items). Product backlog открыт для редактирования для всех участников Scrum-процесса.

Sprint Backlog — содержит функциональность, выбранную Product Owner из Product Backlog. Все функции разбиты по задачам, каждая из которых оценивается командой. Каждый день команда оценивает объем работы, который нужно проделать для завершения задач.

Burndown chart — показывает, сколько уже исполнено и сколько ещё остаётся сделать.

Daily Scrum Meeting

Этот митинг проходит каждое утро в начале дня. Он предназначен для того, чтобы все члены команды знали, кто и чем занимается в проекте. Длительность этого митинга строго ограничена и не должна превышать 15 минут. Цель митинга - поделиться информацией. Он не предназначен для решения проблем в проекте. Все требующие специального обсуждения вопросы должны быть вынесены за пределы митинга.

Скрам митинг проводит Скрам Мастер. Он по кругу задает вопросы каждому члену команды

1. Что сделано вчера?
2. Что будет сделано сегодня?
3. С какими проблемами столкнулся?

Скрам Мастер собирает все открытые для обсуждения вопросы в виде Action Items, например в формате что/кто/когда, например

Демо и ревью спринта

Рекомендованная длительность: 4 часа

Команда демонстрирует инкремент продукта, созданный за последний спринт. Product Owner, менеджмент, заказчики, пользователи, в свою очередь, его оценивают. Команда рассказывает о поставленных задачах, о том как они были решены, какие препятствия были у них на пути, какие были приняты решения, какие проблемы остались нерешенными. На основании ревью принимающая сторона может сделать выводы о том, как должна дальше развиваться система. Участники миитинга делают выводы о том, как шел процесс в команде и предлагает решения по его улучшению.

Скрам Мастер отвечает за организацию и проведение этого митинга. Команда помогает ему составить адженду и распланировать кто и в какой последовательности что представляет.

Подготовка к митингу не должна занимать у команды много времени (правило - не более двух часов). В частности, именно поэтому запрещается использовать презентации в Power Point. Подготовка к митингу не должна занимать у команды более 2-х часов.

## Роли в Scrum

Scrum Master, Product Owner, Team. Кто, зачем?

В методологии Scrum имеется всего три роли:

1. Scrum Master — самая важная роль в методологии знает Scrum. В обязанности Scrum Master'а входит обеспечение максимальной работоспособности и продуктивности команды, четкого взаимодействия между всеми участниками проекта, своевременное решение всех проблем, тормозящих или останавливающих работу, ограждение команды от всех воздействий извне во время итерации и обеспечение следования процессу всех участников проекта.
2. Product Owner — человек, поставляющий требования программистам. Обычно Product Owner является представителем заказчика или представляет рынок, на котором реализуется продукт. Product Owner должен составить бизнес-план и план развития с требованиями. Исходя из имеющейся информации, Product Owner подготавливает список требований, отсортированный по значимости. В обязанности этого сотрудника входит своевременное предоставление требований к продукту, определение дат и содержания релизов, эффективное управление приоритетами и корректировка требований.
3. Team — группа, состоящая из пяти–девяти человек. В нее входят люди с различными навыками — разработчики, аналитики, тестировщики. Нет заранее определенных и поделенных ролей в команде, ограничивающих область действий членов команды. Первая задача этой команды — поставить реально достижимую, прогнозируемую, значимую цель для итерации. Вторая задача — достигнуть эту цель в отведенные сроки и с заявленным качеством. Цель итерации считается достигнутой, если все поставленные задачи реализованы, весь код написан по стандартам кодирования, программа протестирована и все найденные дефекты устранены. В обязанности всех членов Scrum-Team входит участие в выборе цели итерации и определение результата работы. Они должны эффективно взаимодействовать со всеми участниками команды, самостоятельно организовывать свою работу, предоставлять владельцу рабочий продукт в конце каждого цикла.

## Артефакты в Scrum

Backlog (виды, сколько колонок, кто что заполняет), Task board, пример реализации, расчет простого burndown-графика

1. Product Backlog — это список имеющихся на данный момент бизнес-требований и технических требований к системе с указанием приоритетов. Product Backlog включает в себя задачи, ошибки, технологии, проблемы, запросы и т.д.

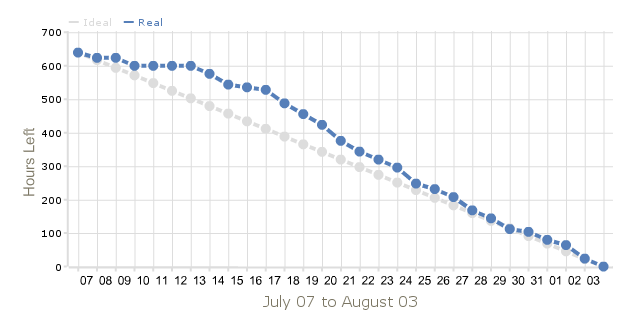
Обязательные поля

* ID — уникальный идентификатор, порядковый номер, применяемый для идентификации историй в случае их переименования.
* Название (Name) — краткое описание истории. Оно должно быть однозначным, чтобы и разработчики, и product owner могли понять, о чем идёт речь и отличить одну историю от другой.
* Важность (Importance) — степень важности данной истории, по мнению product owner’a. Обычно представляет собой натуральное число, иногда для этой цели используются [числа Фибоначчи](http://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%A4%D0%B8%D0%B1%D0%BE%D0%BD%D0%B0%D1%87%D1%87%D0%B8). Чем больше значение, тем выше приоритет.
* Предварительная оценка (initial estimate) — начальная оценка объёма работ, необходимого для реализации истории по сравнению с другими историями. Измеряется в story point’ах. Приблизительно соответствует числу «идеальных человеко-часов».
* Как продемонстрировать (how to demo) — краткое пояснение того, как завершённая задача будет продемонстрирована в конце спринта. Данное поле может представлять собой код автоматизированного теста для приёмо-сдаточного испытания.

Дополнительные поля

* Иногда, также, используются дополнительные поля в product backlog, в основном для того, чтобы помочь product owner’у определиться с его приоритетами.
* Категория (track). Например, «панель управления» или «оптимизация». При помощи этого поля product owner может легко выбрать все пункты категории «оптимизация» и установить им низкий приоритет.
* Компоненты (components) — указывает, какие компоненты (например, база данных, сервер, клиент) будут затронуты при реализации истории. Данное поле состоит из группы checkbox’ов, которые отмечаются, если соответствующие компоненты требуют изменений.
* Инициатор запроса (requestor). Product owner может захотеть хранить информацию о всех заказчиках, заинтересованных в данной задаче. Это нужно для того, чтобы держать их в курсе дела о ходе выполнения работ.
* ID в системе учёта дефектов (bug tracking ID) — если вы используете отдельную систему для учёта дефектов (например. [Jira](http://ru.wikipedia.org/wiki/Atlassian_JIRA)), тогда в описании истории полезно хранить ссылки на все дефекты, которые к ней относятся.

1. Sprint Backlog содержит функциональность, выбранную Product Owner из Product Backlog. Все функции разбиты по задачам, каждая из которых оценивается командой.
2. Результатом Sprint является готовый продукт, который можно передавать заказчику. Каждый sprint представляет собой «водопад». В течение спринта делаются все работы по сбору требований, дизайну, кодированию и тестированию продукта.



## Практики Scrum

Ретроспектива, демонстрация, Scrum-митинг, планирование

Планирование спринта (Planning Meeting)

Происходит в начале итерации.

Выбирается объём работ, обязательства по выполнению которой за спринт принимает на себя команда

Обсуждается и определяется, каким образом будет реализован этот объём работ

Каждая запись в Product Backlog, принятая к реализации, разбивается на подзадачи, которые оцениваются в идеальных человеко-часах

Ограничено сверху 4-8 часами в зависимости от продолжительности итерации, опыта команды и т. п.

**Митинг (Daily Scrum)**

Burndown-диаграмма, отображает текущее состояние спринта, степень отставания/опережения плана.

Происходит каждый день в течение спринта. Является «пульсом» хода спринта. Митингу присущи следующие ограничения:

начинается точно вовремя;

все могут наблюдать, но только «свиньи» говорят;

длится не более 15 минут;

проводится в одном и том же месте в течение спринта.

В течение митинга каждый член команды отвечает на 3 вопроса.

Что сделано с момента предыдущего митинга до текущего?

Что будет сделано с момента текущего митинга до следующего?

Какие проблемы мешают достижению целей спринта? (Над решением этих проблем работает ScrumMaster. Обычно это решение проходит за рамками митинга и в составе лиц, непосредственно затронутых данным препятствием.)

**Демонстрация (Demo Meeting)**

Происходит в конце итерации (спринта).

Команда демонстрирует инкремент функциональности продукта всем заинтересованным лицам.

Привлекается максимальное количество зрителей.

Все члены команды участвуют в демонстрации (один человек на демонстрацию или каждый показывает, что сделал за спринт).

Ограничена 4-мя часами в зависимости от продолжительности итерации и инкремента продукта.

**Ретроспектива (Retrospective Meeting)**

Все члены команды рассказывают своё отношение к ходу прошедшего спринта.

Отвечают на два основных вопроса (Что было сделано хорошо в прошедшем спринте? Что надо улучшить или не допускать в следующем?).

Выполняют улучшение процесса разработки (решают вопросы и фиксируют удачные решения).

Ограничена 1—3-мя часами.

# XP

## XP

(определение, история, ценности и круг решаемых проблем)

Extreme Programming – Экстремальное программирование

Экстремальное программирование (XP) – это упрощенная методология организации разработки программ для небольших и средних по размеру команд разработчиков, занимающихся созданием программного продукта в условиях неясных или быстро меняющихся требований.

Цели XP

Основными целями XP являются повышение доверия заказчика к программному продукту путем предоставления реальных доказательств успешности развития процесса разработки и резкое сокращение сроков разработки продукта. При этом XP сосредоточено на минимизации ошибок на ранних стадиях разработки. Это позволяет добиться максимальной скорости выпуска готового продукта и даёт возможность говорить о прогнозируемости работы. Практически все приемы XP направлены на повышение качества программного продукта.

Решаемые проблемы

1. Смещение графиков( изменение условий).
2. Текучка кадров
3. Кол-во дефектов
4. Недостаток взаимодействий
5. Потеря полезности\ актуальности

Ценности

1. Общение
2. Простота
3. Обратная связь
4. Смелость
5. Уважениие.

## Практики XP

Приемы XP (практики)

Обычно XP характеризуют набором из 12 правил (практик), которые необходимо выполнять для достижения хорошего результата. Ни одна из практик не является принципиально новой, но в XP они собраны вместе.

1. Планирование процесса. Вся команда разработчиков собирается вместе, принимается коллективное решение о том, какие свойства системы будут реализованы в ближайшей итерации. Трудоемкость реализации каждого свойства определяется самими программистами.
2. Тесное взаимодействие с заказчиком. Представитель заказчика должен быть членом XP-команды. Он пишет ПИ, выбирает истории, которые будут реализованы в конкретной итерации, и отвечает на вопросы, касающиеся бизнеса. Представитель заказчика должен бытьэкспертом в автоматизируемой предметной области. Необходимо наличие постоянное обратной связи с представителем заказчика.
3. Общесистемные правила именования. Хорошие системные правила именования предполагают простоту именования классов и переменных. Команда разработчиков должна иметь единые правила именования.
4. Простая архитектура. Любое свойство системы должно быть реализовано как можно проще. Программисты в XP-команде работают под девизом: «Ничего лишнего!». Принимается первое простейшее работающее решение, реализуется необходимый уровень функциональности на данный момент. Тем самым экономится время программиста.
5. Рефакторинг. Это оптимизация существующего кода с целью его упрощения, Такая работа должна вестись постоянно. Сохраняя код прозрачным и определяя его элементы всего один раз, программисты сокращают число ошибок, которые впоследствии придется устранять. При реализации каждого нового свойства системы программист должен подумать над тем, можно ли упростить существующий код и как это поможет реализовать новое свойство. Кроме того, нельзя совмещать рефакторинг с дизайном: если создается новый код, рефакторинг следует отложить.
6. Парное программирование. Все программисты должны работать в парах: один пишет код, другой смотрит. Таким образом, необходимо размещать группу программистов в одном месте. XP наиболее успешно работает в нераспределенных коллективах программистов и пользователей.
7. 40-часовая рабочая неделя. Программист не должен работать более 8 часов в день. Необходимость сверхурочной работы – это четкий индикатор проблемы на данном конкретном направлении разработки. Поиск причин сверхурочной работы и их скорейшее устранение – одно из основных правил.
8. Коллективное владение кодом. Каждый программист в коллективе должен иметь доступ к коду любой части системы и право вносить изменения в любой код. Обязательное правило: если программист внес изменения и система после этого работает некорректно, то именно этот программист должен исправить ошибки.
9. Единые стандарты кодирования. Стандарты кодирования нужны для обеспечения других практик: коллективного владения кодом, парного программирования и рефакторинга. Без единого стандарта выполнять эти практики как минимум сложнее, а в реальности вообще невозможно: группа будет работать в режиме постоянной нехватки времени. Команда работает над проектом продолжительное время. Люди приходят и уходят. Никто не кодирует в одиночку и код принадлежит всем. Всегда будут моменты, когда необходимо будет понять и скорректировать чужой код. Разработчики будут удалять дублирующий код, анализировать и улучшать чужие классы и т. п. Со временем нельзя будет сказать, кто автор конкретного класса. Следовательно, все должны подчиняться общим стандартам кодирования – форматирование кода, именование классов, переменных, констант, стиль комментариев. Вышесказанное означает, что все члены команды должны договориться об общих стандартах кодирования. Неважно каких, но все обязаны им подчиняются.
10. Небольшие релизы. Минимальная итерация – один день, максимальная – месяц; чем чаще осуществляются релизы, тем больше недостатков системы будет выявлено. Первые релизы помогают выявить недостатки на самых ранних стадиях, далее функциональность системы расширяется на основании ПИ. Поскольку пользователь включается в процесс разработки начиная с первого релиза, то он оценивает систему и выдает пользовательскую историю и замечания. На основании этого определяется следующая итерация, то есть, каким будет новый релиз. В XP все направлено на обеспечение непрерывной обратной связи с пользователями.
11. Непрерывная интеграция. Интеграция новых частей системы должна происходить как можно чаще, как минимум раз в несколько часов. Основное правило интеграции следующее: интеграцию можно производить, если все тесты проходят успешно. Если тесты не проходят, то программист должен либо внести исправления и тогда интегрировать составные части системы, либо вообще не интегрировать их. Правило это – жесткое и однозначное. Если в созданной части системы имеется хотя бы одна ошибка, то интеграцию производить нельзя. Частая интеграция позволяет быстрее получить готовую систему, вместо того чтобы тратить на сборку неделю.
12. Тестирование. В отличие от большинства остальных методологий тестирование в XP – одно из важнейших составляющих. Экстремальный подход предполагает, что тесты пишутся до написания кода. Каждый модуль обязан иметь unit test – тест данного модуля. Таким образом, в XP осуществляется регрессионное тестирование, «неухудшение качества» при добавлении функциональности. Большинство ошибок исправляются на стадии кодирования. Тесты пишут сами программисты, любой из них имеет право написать тест для любого модуля. Еще один важный принцип: тест определяет код, а не наоборот (test-driven development), то есть кусок кода кладется в хранилище тогда и только тогда, когда все тесты прошли успешно, в противном случае данное изменение кода отвергается.

## Парное программирование

(роли, обязанности, преимущества, проблемы)

Па́рное программи́рование — техника [программирования](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), при которой весь [исходный код](http://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) создаётся парами людей, программирующих одну задачу, сидя за одним рабочим местом. Один [программист](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%81%D1%82) управляет компьютером и, в основном, думает над кодированием в деталях. Другой программист сосредоточен на картине в целом и непрерывно просматривает код, производимый первым программистом. Время от времени они меняются ролями, обычно, каждые полчаса.

Преимущества

Повышение дисциплины

Программисты в паре чаще «делают то, что нужно» и реже устраивают длинные перерывы.

Лучший код

Партнёры в паре менее склонны к неудачным решениям и производят более качественный код.

Гибкий поток работы

Высокий боевой дух

Коллективное владение кодом

Коллективное владение (пары меняются) означает, что каждый несёт ответственность за весь код. Таким образом, каждый вправе вносить изменения в любой участок кода. Парное программирование поддерживает эту практику: работая в парах, все программисты получают доступ ко всем частям кода. Важное преимущество коллективного владения кодом заключается в том, что оно ускоряет процесс разработки, поскольку, при появлении ошибки, её может устранить любой программист.

Наставничество

Каждый, даже начинающий программист, знает что-то, чего не знают другие. Парное программирование — безболезненный способ распространить эти знания.

Командный дух

Меньше прерываний

Экономическая обоснованность

Высокое качество дизайна

Обратная связь

Непрерывность проверки кода

Ошибки обнаруживаются сразу же, как только появились, это позволяет экономить и время, и деньги. Команда разработчиков сплочается, что позволяет закончить работу раньше и в лучшем качестве.

Обучение

Программисты постоянно обмениваются знаниями.

# MVP

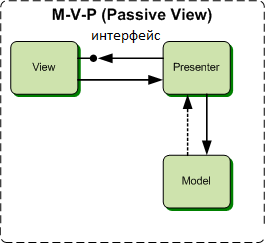
## Типичные проблемы с UI

примеры смешений, последствия

1. слишком много логики
2. представление данных
3. события
4. поведения
5. ввод данных
6. представление данных
7. получение данных

## Passive View, диаграмма и назначение паттерна

почему пассивная? в чем проявляется и зачем так сделано



Пассивная потому что в view нет ни какой логики.

Сделано для возможности легкого переноа на другие виды визуализации.

## Ответственности слоев в шаблоне MVP

Концепция MVC позволяет разделить данные, представление и обработку действий пользователя на три отдельных компонента:

1. Модель ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Model). Модель предоставляет знания: данные и методы работы с этими данными, реагирует на запросы, изменяя своё состояние. Не содержит информации, как эти знания можно визуализировать.
2. Представление, вид ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) View). Отвечает за отображение информации (визуализация). Часто в качестве представления выступает [форма (окно)](http://ru.wikipedia.org/wiki/%D0%9E%D0%BA%D0%BD%D0%BE_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) с графическими элементами.
3. Контроллер ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Controller). Обеспечивает связь между пользователем и системой: контролирует ввод данных пользователем и использует модель и представление для реализации необходимой реакции.

# Data Access

## Преимущества введения Data Access layer

пример на сокрытие деталей подсистемы работы с данными

* Бизнес-логика зависит от платформы
  + Смена технологии – проблема
* Структура данных диктует дизайн бизнес-объектов
  + Плохой дизайн
* Смешение бизнес-логики и логики хранения

Сложно тестировать и сопровождать

# DDD

## DDD

Определение, назначение, стандартная и усовершенствованная архитектура

UI

Application

Domain

Infrastructure

UI

Application

Domain

Infrastructure

Presentation

зависимость

создание

наследование

## Назначение сборок DDD, их типичные классы

При рассказе о типичных классах иллюстрировать знание примерами

# SOLID

(определение, назначение, пример нарушения и последствия)

## SRP

В [объектно-ориентированном программировании](http://ru.wikipedia.org/wiki/%D0%9E%D0%9E%D0%9F), принцип единственной обязанности обозначает, что каждый [объект](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) должен иметь одну обязанность и эта обязанность должна быть полностью [инкапсулирована](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности.

Термин был введён [Робертом С. Мартином](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%80%D1%82%D0%B8%D0%BD,_%D0%A0%D0%BE%D0%B1%D0%B5%D1%80%D1%82) в одноименной статье как часть его Принципов объектно-ориентированного проектирования, ставших популярными благодаря его книге Быстрая разработка программ. Принципы, примеры, практика [[1]](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%B5%D0%B4%D0%B8%D0%BD%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9_%D0%BE%D0%B1%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8#cite_note-0). Мартин описал её как основанную на принципе [связности](http://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), сформулированном [Томом ДеМарко](http://en.wikipedia.org/wiki/Tom_DeMarco) в его книге Structured Analysis and Systems Specification[[2]](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%B5%D0%B4%D0%B8%D0%BD%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9_%D0%BE%D0%B1%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8#cite_note-1).

Мартин определяет обязанность как причину изменения и заключает, что [класс](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) или [модуль](http://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) должны иметь одну и только одну причину измениться. Например, представьте себе модуль, который составляет и печатает отчёт. Такой модуль может измениться по двум причинам. Во первых, может измениться само содержимое отчёта. Во вторых, может измениться формат отчёта. Оба этих фактора изменяют модуль по разным причинам: в одном случае изменение содержательное, а во втором — косметическое. Принцип единственной обязанности говорит, что оба аспекта этой проблемы на самом деле являются двумя разными обязанностями, и в таком случае должны находиться в разных классах или модулях. Объединение двух сущностей, изменяющихся по разным причинам и в разное время, считается плохим проектным решением.

Причина, почему нужно сохранять направленность класса на единственную цель в том, что это делает класс более здоровым. Что касается приведённого выше примера, если произошло изменение в процессе составления отчёта — есть большая вероятность, что в негодность придёт код, отвечающий за печать, если он является частью того же класса.

Принцип единственной обязанности используется в методологиях проектирования «от обязанности», таких как [RDD](http://en.wikipedia.org/wiki/RDD) и [URDAD](http://en.wikipedia.org/wiki/URDAD)

## OCP

В [объектно-ориентированном программировании](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) принцип открытости/закрытости устанавливает следующее положение: «программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения»;[[1]](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%BE%D1%82%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D1%81%D1%82%D0%B8/%D0%B7%D0%B0%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D1%81%D1%82%D0%B8#cite_note-0) это означает, что такие сущности могут позволять менять свое поведение без изменения их [исходного кода](http://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4). Это особенно значимо в производственной среде, когда изменения в исходном коде потребуют проведение [пересмотра кода](http://ru.wikipedia.org/w/index.php?title=%D0%9F%D0%B5%D1%80%D0%B5%D1%81%D0%BC%D0%BE%D1%82%D1%80_%D0%BA%D0%BE%D0%B4%D0%B0&action=edit&redlink=1), [модульного тестирования](http://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) и других подобных процедур, чтобы получить право на использования его в программном продукте. Код, подчиняющийся данному принципу, не изменяется при расширении и поэтому не требует таких трудозатрат.

Термин «принцип открытости/закрытости» имеет два значения. Оба значения используют [наследование](http://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) для решения дилеммы, но цели, способы и результаты — различны.

## LSP

Принцип подстановки Барбары Лисков ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Liskov Substitution Principle, LSP) в [объектно-ориентированном программировании](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) является специфичным определением подтипапредложенным [Барбарой Лисков](http://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%81%D0%BA%D0%BE%D0%B2,_%D0%91%D0%B0%D1%80%D0%B1%D0%B0%D1%80%D0%B0) в 1987 году на конференции в основном докладе под названием Абстракция данных и иерархия [[1]](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%BF%D0%BE%D0%B4%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B8_%D0%91%D0%B0%D1%80%D0%B1%D0%B0%D1%80%D1%8B_%D0%9B%D0%B8%D1%81%D0%BA%D0%BE%D0%B2#cite_note-FamilyValues-0).

В последующей статье[[2]](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%BF%D0%BE%D0%B4%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B8_%D0%91%D0%B0%D1%80%D0%B1%D0%B0%D1%80%D1%8B_%D0%9B%D0%B8%D1%81%D0%BA%D0%BE%D0%B2#cite_note-subtyping-1) Лисков кратко сформулировала свой принцип следующим образом:

Пусть q(x) является свойством, верным относительно объектов x некоторого типа T. Тогда q(y) также должно быть верным для объектов y типа S, где S является подтипом типа T.

Роберт С. Мартин определил[[3]](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%BF%D0%BE%D0%B4%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B8_%D0%91%D0%B0%D1%80%D0%B1%D0%B0%D1%80%D1%8B_%D0%9B%D0%B8%D1%81%D0%BA%D0%BE%D0%B2#cite_note-robert_martin-2) этот принцип так:

Функции которые используют базовый тип должны иметь возможность использовать подтипы базового типа не зная об этом.

Таким образом, идея Лисков о «подтипе» даёт определение понятия замещения — если S является подтипом T, тогда объекты типа T в программе могут быть замещены объектами типа S без каких либо изменений желательных свойств этой программы (например, [корректность](http://en.wikipedia.org/wiki/Correctness)).

Этот принцип является важнейшим критерием для оценки качества принимаемых решений при построении иерархий наследования. Сформулировать его можно в виде простого правила: тип S будет подтипом Т тогда и только тогда, когда каждому объекту oS типа S соответствует некий объект oT типа T таким образом, что для всех программ P, реализованных в терминах T, поведение P не будет меняться, если oT заменить на oS.

Более простыми словами можно сказать, что поведение наследуемых классов не должно противоречить поведению, заданному базовым классом, то есть поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа.

Саттер и Александреску в своём руководстве по использованию Си++ для выражения этого принципа также используют фразу «подкласс не должен требовать от вызывающего кода больше, чем базовый класс, и не должен предоставлять вызывающему коду меньше, чем базовый класс». По мнению данных авторов, публичное наследование в Си++ можно употреблять только тогда, когда оно удовлетворяет принципу Лисков. Приватное наследование, по их же мнению, дозволено использовать для доступа к protected части базы и перекрытия виртуальных методов. В любом же ином случае, то есть для всего лишь повторного использования кода из базы, наследование применять нельзя.

Основания: использование публичного наследования для повторного использования кода приводит к тому, что внешний мир начинает считать класс Derived разновидностью класса Base, и возможно появление кода, явно использующего этот факт. Такое сильно сужает простор для манёвра архитектора в дальнейшем поддержании и рефакторинге класса Derived.

## ISP

[Роберт С. Мартин](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%80%D1%82%D0%B8%D0%BD,_%D0%A0%D0%BE%D0%B1%D0%B5%D1%80%D1%82) определил[[1]](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D1%80%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D0%B0#cite_note-robert_martin-0) этот принцип так:

Клиенты не должны зависеть от методов, которые они не используют.

Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

## DIP

Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

## Закон Деметера

(что можно, чего нельзя, назначение, допустимые нарушения)

Или закон наименьшего знания. Применительно к Java этот закон можно сформулировать как "используй только одну точку". Т.е. a.b.method() нарушает закон, а a.method() - нет.  
Допустим у нас есть объект O, у него есть метод M. Закон Деметра требует, чтобы в методе M мы пользовались услугами только следующих объектов:

самого O и его полей;

параметров, переданных в M;

любых объектов, созданных в M;

В часности, следует избегать вызова методов объектов, возвращаемых другими методами.

Аналогия из жизни. Если Вы хотите, чтобы собака побежала, глупо командовать ее ногами, лучше отдать команду собаке, а она уже разберется со своими ногами сама.

Достоинства: снижается связность, в результате получается код, который легче сопровождать, который содержит потенциально меньше ошибок.

Недостатки: зачастую приходится писать множество методов-оберток, которые только лишь делегируют работу кому-то еще. Объекты становятся более громоздкими.

# TDD

## Определение и назначение TDD

сопоставить с просто наличием unit-тестов

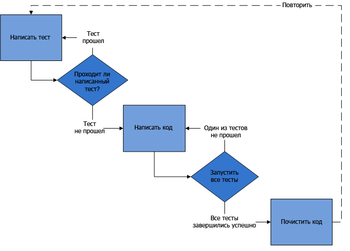
Разработка через тестирование (test-driven development, TDD) — техника [разработки программного обеспечения](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F), которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест и под конец проводится рефакторинг нового кода к соответствующим стандартам. [Кент Бек](http://ru.wikipedia.org/w/index.php?title=%D0%9A%D0%B5%D0%BD%D1%82_%D0%91%D0%B5%D0%BA&action=edit&redlink=1), считающийся изобретателем этой техники, утверждал в 2003 году, что разработка через тестирование поощряет простой дизайн и внушает уверенность (inspires confidence).[[1]](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%87%D0%B5%D1%80%D0%B5%D0%B7_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#cite_note-Beck-0)

В 1999 году при своем появлении разработка через тестирование была тесно связана с концепцией «сначала тест» (test-first), применяемой в [экстремальном программировании](http://ru.wikipedia.org/wiki/%D0%AD%D0%BA%D1%81%D1%82%D1%80%D0%B5%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)[[2]](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%87%D0%B5%D1%80%D0%B5%D0%B7_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#cite_note-Cworld92-1), однако позже выделилась как независимая методология.[[3]](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%87%D0%B5%D1%80%D0%B5%D0%B7_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#cite_note-Newkirk-2)

Тест — это процедура, которая позволяет либо подтвердить, либо опровергнуть работоспособность кода. Когда программист проверяет работоспособность разработанного им кода, он выполняет тестирование вручную. В данном контексте тест состоит из двух этапов: стимулирование кода и проверки результатов его работы. Автоматический тест выполняется иначе: вместо программиста стимулированием кода и проверкой результатов занимается компьютер, который отображает на экране результат выполнения теста: код работоспособен или код неработоспособен. Методика разработки через тестирование позволяет получить ответы на вопросы об организации автоматических тестов и выработке определенных навыков тестирования.

## Цикл написания кода в технике TDD

## Цикл разработки через тестирование

[](http://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Test-driven-development-ru.png)

[http://bits.wikimedia.org/skins-1.18/common/images/magnify-clip.png](http://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Test-driven-development-ru.png)

Графическое представление цикла разработки, в виде блок-схемы

Приведенная последовательность действий основана на книге Кента Бека «Разработка через тестирование: на примере» (Test Driven Development: By Example).[[1]](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%87%D0%B5%D1%80%D0%B5%D0%B7_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#cite_note-Beck-0)

**Добавление теста**

При разработке через тестирование, добавление каждой новой функциональности (feature) в программу, начинается с написания теста. Неизбежно этот тест не будет проходить, поскольку соответствующий код ещё не написан. (Если же написанный тест прошёл, это означает, что либо предложенная «новая» функциональность уже существует, либо тест имеет недостатки.) Чтобы написать тест, разработчик должен чётко понимать предъявляемые к новой возможности требования. Для этого рассматриваются возможные сценарии использования и пользовательские истории. Новые требования могут также повлечь изменение существующих тестов. Это отличает разработку через тестирование от техник, когда тесты пишутся после того, как код уже написан: она заставляет разработчика сфокусироваться на требованиях до написания кода — тонкое, но важное отличие.

**Запуск всех тестов: убедиться, что новые тесты не проходят**

На этом этапе проверяют, что только что написанные тесты не проходят. Этот этап так же проверяет сами тесты: написанный тест может проходить всегда и соответственно быть бесполезным. Новые тесты должны не проходить по объяснимым причинам. Это увеличит уверенность (хотя не будет гарантировать полностью), что тест действительно тестирует то, для чего он был разработан.

**Написать код**

На этом этапе пишется новый код так, что тест будет проходить. Этот код не обязательно должен быть идеален. Допустимо, чтобы он проходил тест каким-то неэлегантным способом. Это приемлемо, поскольку последующие этапы улучшат и отполируют его.

Важно писать код, предназначенный именно для прохождения теста. Не следует добавлять лишней и, соответственно, не тестируемой функциональности.

**Запуск всех тестов: убедиться, что все тесты проходят**

Если все тесты проходят, программист может быть уверен, что код удовлетворяет всем тестируемым требованиям. После этого можно приступить к заключительному этапу цикла.

**Рефакторинг**

Когда достигнута требуемая функциональность, на этом этапе код может быть почищен.  **Повторить цикл**

Описанный цикл повторяется, реализуя всё новую и новую функциональность. Шаги следует делать небольшими, от 1 до 10 изменений между запусками тестов. Если новый код не удовлетворяет новым тестам или старые тесты перестают проходить, программист должен вернуться к [отладке](http://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BB%D0%B0%D0%B4%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B). При использовании сторонних библиотек не следует делать настолько небольшие изменения, которые буквально тестируют саму стороннюю библиотеку[[3]](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%87%D0%B5%D1%80%D0%B5%D0%B7_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#cite_note-Newkirk-2), а не код, её использующий, если только нет подозрений, что библиотека содержит ошибки.

## Атрибуты хорошего теста

аббревиатура A TRIP <http://stackoverflow.com/questions/61400/what-makes-a-good-unit-test>

1. Фокусированный – одна функциональность
2. Быстрый
3. Независимый от друг друга, окружения, порядка запуска.
4. Такого же качества как и код приложения
5. Автоматический(легкий запуск)

# Collaboration Tools

## Непрерывная интеграция

(определение, назначение, реализация)

# Clean Code

## Ключевые понятия Clean Code

1. Дублирование – одинаковые строки похожи функции, классы, формы
2. Шум – любая информация которая не несет смысловой нагрузки, неиспользуемый код.
3. Принцип единой ответственности – у каждого класса одна зона ответственности.
4. Уровни абстракции – независимые уровни

## Именование

(классы, переменные/члены, функции/методы, файлы,...)

1. Классы с большой буквы без подчеркиваний, поля с маленькой, каждое новое слово с большой буквы.
2. Константы – большими буквами с подчеркиваниями.
3. Чем короче поле видимости тем хитрее.
4. Булевые с can, is, has.
5. Главное слово в имени класса существительное, для функции – глагол.
6. У наследника в имени ссылаться на существительное базового класса
7. Функции
8. Размер 50-75 строк
9. Полиморфизм вместо if else.
10. Лучше возвращать исключения вместо кода ошибки.

## Комментарии: достоинства и недостатки

Комментарии это ЗЛО!

Основная проблема – комментарий не компилируется

Плохие комментарии :

1. избыточный комментарий
2. закомментированный код
3. дезнформация

Допустимые комментарии:

1. юридические
2. поясняющие, поясняют зачем, а не как.
3. Не комментируйте плохой код, его нужно переписать.

# Refactoring

## Refactoring

определение, назначение, примеры, когда нужно использовать refactoring, а когда не нужно.

Рефакторинг ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) refactoring) — процесс изменения внутренней структуры [программы](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы[[1]](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3#cite_note-0). В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению её согласованности и четкости.

Цель рефакторинга — сделать код программы легче для понимания; без этого рефакторинг нельзя считать успешным.

Рефакторинг следует отличать от [оптимизации производительности](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%BA%D0%BE%D0%B4%D0%B0). Как и рефакторинг, оптимизация обычно не изменяет поведение программы, а только ускоряет ее работу. Но оптимизация часто затрудняет понимание кода, что противоположно рефакторингу[[2]](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3#cite_note-autogenerated1-1).

С другой стороны, нужно отличать рефакторинг от [реинжиниринга](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B8%D0%BD%D0%B6%D0%B8%D0%BD%D0%B8%D1%80%D0%B8%D0%BD%D0%B3_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F), который осуществляется для расширения функциональности программного обеспечения. Как правило, крупные рефакторинги предваряют [реинжиниринг](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B8%D0%BD%D0%B6%D0%B8%D0%BD%D0%B8%D1%80%D0%B8%D0%BD%D0%B3_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F).

**Причины применения рефакторинга**

Рефакторинг нужно применять постоянно при разработке кода. Основными стимулами его проведения являются следующие задачи:

1. необходимо добавить новую функцию, которая недостаточно укладывается в принятое архитектурное решение;
2. необходимо исправить ошибку, причины возникновения которой сразу не ясны;
3. преодоление трудности в командной разработке, которые обусловлены сложной логикой программы.

**Какой код должен подвергаться рефакторингу**

Во многом при рефакторинге лучше полагаться на интуицию, основанную на опыте. Но можно выделить наиболее очевидные причины, когда код нужно подвергнуть рефакторингу:

1. дублирование кода;
2. длинный метод;
3. большой класс;
4. длинный список параметров;
5. «завистливые» функции — это метод, который чрезмерно обращается к данным другого объекта;
6. избыточные временные переменные;
7. классы данных;
8. несгруппированные данные.

**Рефакторинг кода**

В программировании термин рефакторинг означает изменение исходного кода программы без изменения его внешнего поведения. В [экстремальном программировании](http://ru.wikipedia.org/wiki/%D0%AD%D0%BA%D1%81%D1%82%D1%80%D0%B5%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) и других гибких методологиях рефакторинг является неотъемлемой частью цикла разработки ПО: разработчики попеременно то создают новые тесты и функциональность, то выполняют рефакторинг кода для улучшения его логичности и прозрачности. Автоматическое [юнит-тестирование](http://ru.wikipedia.org/wiki/%D0%AE%D0%BD%D0%B8%D1%82-%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) позволяет убедиться, что рефакторинг не разрушил существующую функциональность.

Иногда под рефакторингом неправильно подразумевают коррекцию кода с заранее оговоренными правилами отступа, перевода строк, внесения комментариев и прочими визуально значимыми изменениями, которые никак не отражаются на процессе компиляции, с целью обеспечения лучшей читаемости кода (см. [code formatting](http://en.wikipedia.org/wiki/Prettyprint#Code_formatting_and_beautification)).

Рефакторинг изначально не предназначен для исправления [ошибок](http://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B3) и добавления новой функциональности, он вообще не меняет поведение программного обеспечения[[2]](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3#cite_note-autogenerated1-1) и это помогает избежать ошибок и облегчить добавление функциональности. Он выполняется для улучшения понятности кода или изменения его структуры, для удаления «мёртвого кода» — всё это для того, чтобы в будущем код было легче поддерживать и развивать. В частности, добавление в программу нового поведения может оказаться сложным с существующей структурой — в этом случае разработчик может выполнить необходимый рефакторинг, а уже затем добавить новую функциональность.

Это может быть перемещение поля из одного [класса](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) в другой, вынесение фрагмента [кода](http://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) из [метода](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_(%D1%8F%D0%B7%D1%8B%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) и превращение его в самостоятельный метод или даже перемещение кода по иерархии классов. Каждый отдельный шаг может показаться элементарным, но совокупный эффект таких малых изменений в состоянии радикально улучшить проект или даже предотвратить распад плохо спроектированной программы.

**Методы рефакторинга**

Наиболее употребимые методы рефакторинга:

1. Изменение [сигнатуры метода](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%B3%D0%BD%D0%B0%D1%82%D1%83%D1%80%D0%B0_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8) (Change Method Signature)
2. [Инкапсуляция](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) [поля](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) (Encapsulate Field)
3. Выделение [класса](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (Extract Class)
4. Выделение [интерфейса](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81) (Extract Interface)
5. Выделение локальной переменной (Extract Local Variable)
6. Выделение [метода](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_(%D1%8F%D0%B7%D1%8B%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) (Extract Method)
7. Генерализация типа (Generalize Type)
8. Встраивание (Inline)
9. Введение [фабрики](http://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B1%D1%80%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) (Introduce Factory)
10. Введение [параметра](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (Introduce Parameter)
11. Подъём поля/метода (Pull Up)
12. Спуск поля/метода (Push Down)
13. Замена условного оператора [полиморфизмом](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_%D0%B2_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0%D1%85_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) (Replace Conditional with Polymorphism)

**Изменение сигнатуры метода (Change Method Signature)**

Суть изменения сигнатуры метода заключается в добавлении, изменении или удалении параметра метода. Изменив сигнатуру метода, необходимо скорректировать обращения к нему в коде всех клиентов. Это изменение может затронуть внешний интерфейс программы, кроме того, не всегда разработчику, изменяющему интерфейс, доступны все клиенты этого интерфейса, поэтому может потребоваться та или иная форма регистрации изменений интерфейса для последующей передачи их вместе с новой версией программы.

**Инкапсуляция поля (Encapsulate field)**

В случае, если у класса имеется открытое поле, необходимо сделать его закрытым и обеспечить методы доступа. После «Инкапсуляции поля» часто применяется «[Перемещение метода](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3#.D0.9F.D0.B5.D1.80.D0.B5.D0.BC.D0.B5.D1.89.D0.B5.D0.BD.D0.B8.D1.8F_.D0.BC.D0.B5.D1.82.D0.BE.D0.B4.D0.B0)».

**Выделение метода (Extract Method)**

Выделение метода заключается в выделении из длинного и/или требующего комментариев кода отдельных фрагментов и преобразовании их в отдельные методы, с подстановкой подходящих вызовов в местах использования. В этом случае действует правило: если фрагмент кода требует комментария о том, что он делает, то он должен быть выделен в отдельный метод. Также правило: один метод не должен занимать более чем один экран (25-50 строк, в зависимости от условий редактирования), в противном случае некоторые его фрагменты имеют самостоятельную ценность и подлежат выделению. Из анализа связей выделяемого фрагмента с окружающим контекстом делается вывод о перечне параметров нового метода и его локальных переменных.

**Перемещение метода (Move Method)**

Перемещение метода применяется по отношению к методу, который чаще обращается к другому классу, чем к тому, в котором сам располагается.

**Замена условного оператора полиморфизмом (Replace Conditional with Polymorphism)**

[Условный оператор](http://ru.wikipedia.org/wiki/%D0%A3%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80) с несколькими ветвями заменяется вызовом полиморфного метода некоторого базового класса, имеющего подклассы для каждой ветви исходного оператора. Выбор ветви осуществляется неявно, в зависимости от того, экземпляру какого из подклассов оказался адресован вызов.

Основные принципы:

вначале следует создать базовый [класс](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и нужное число подклассов;

в некоторых случаях следует провести оптимизацию условного оператора путем «[Выделения метода](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3#.D0.92.D1.8B.D0.B4.D0.B5.D0.BB.D0.B5.D0.BD.D0.B8.D1.8F_.D0.BC.D0.B5.D1.82.D0.BE.D0.B4.D0.B0)»;

возможно использование «[Перемещения метода](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3#.D0.9F.D0.B5.D1.80.D0.B5.D0.BC.D0.B5.D1.89.D0.B5.D0.BD.D0.B8.D1.8F_.D0.BC.D0.B5.D1.82.D0.BE.D0.B4.D0.B0)», чтобы поместить условный оператор в вершину иерархии наследования;

выбрав один из подклассов, нужно конкретизировать в нём полиморфный метод базового класса и переместить в него тело соответствующей ветви условного оператора;

повторить предыдущее действие для каждой ветви условного оператора;

заменить весь условный оператор вызовом полиморфного метода базового класса.

**Проблемы, возникающие при проведении рефакторинга**

1. проблемы, связанные с базами данных;
2. проблемы изменения интерфейсов;
3. трудности при изменении дизайна.