

Continuous Integration

CONTINUOUS INTEGRATION

Оригинал: *Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily — leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.*

Martin Fowler

Вольный перевод: *Continuous Integration (далее — CI) — это практика разработки ПО, при которой участники команды интегрируют изменения настолько часто, насколько это возможно (как минимум, ежедневно или несколько раз в день). Каждая интеграция верифицируется автоматической сборкой (включая тесты) для определения ошибок интеграции настолько быстро, насколько это возможно. Мартин Фаулер*

Для нас — это будет практика (подход к разработке), при которой для каждого изменения кода (git push) будет автоматически запускаться

конвейер сборки и тестирования.

Continuous Delivery (непрерывная поставка) подразумевает ещё один шаг

вперёд в вопросе автоматизации.

Мы **доверяем автотестам настолько**, что на базе их результатов автоматически **формируем и выкладываем релиз** (например, публикуем

библиотеку в Maven Central) или внутреннем репозитории.

Затем его (релиз) можно подвергать исследовательскому тестированию, тестированию безопасности и т.д., но ключевое — мы **формируем релиз на**

базе решения автотестов.

Как вы видите, самые популярные это:

- `junit-jupiter-engine` — ядро JUnit Jupiter;
- `junit-jupiter-api` — API для написания автотестов (готовый набор классов, аннотаций и т.д.);
- `junit-jupiter-params` — API для написания параметризованных автотестов.

Версия продукта `major.minor.patch` разбивается на три ключевых составляющих:

- `major` — мажорная версия, в которой есть нарушающие совместимость изменения API;
- `minor` — минорная версия, в которой нет нарушающих совместимость изменений API;
- `patch` — багфиксы, не нарушающие совместимость API.

API (Application Programming Interface) — набор классов и методов*, предоставляемых нам библиотекой для использования из нашего кода.

Нарушение совместимости означает, что предоставляемый нам интерфейс меняется внешне.

Workflow — набор задач, в случае GitHub Actions — файл формата YML*, находящийся в каталоге `.github/workflows`.

В этом файле описана вся необходимая конфигурация.

При нажатии на кнопку Start Commit этот файл будет добавлен в ваш репозиторий и GitHub включит интеграцию с GitHub Actions (не забудьте сделать `git pull`)

Maven предлагает нам концепцию **LifeCycles** — жизненных циклов (набора последовательных активностей), которые нужны для управления проектом:

- `default` — сборка, тестирование и развёртывание;
- `clean` — очистка проекта и удаление всех сгенерированных артефактов;
- `site` — создание документации на проект.

Важно: когда мы пишем `mvn test`, то запускается фаза `test`, а именно запускаются все цели плагинов, привязанные к этой фазе.

Но **фазы выполняются последовательно**: если мы запускаем фазу `test`, то срабатывают все фазы до неё (в рамках конкретного LifeCycle).

Поэтому когда мы запускали `mvn test` будет запускаться и обработка ресурсов и компиляция.

Default Lifecycle ←

Phase	Description
<code>validate</code>	validate the project is correct and all necessary information is available.
<code>initialize</code>	initialize build state, e.g. set properties or create directories.
<code>generate-sources</code>	generate any source code for inclusion in compilation.
<code>process-sources</code>	process the source code, for example to filter any values.
<code>generate-resources</code>	generate resources for inclusion in the package.
<code>process-resources</code>	copy and process the resources into the destination directory, ready for packaging.
<code>compile</code>	compile the source code of the project.
<code>...</code>	...

Жизненные циклы разделены на фазы (Phases), а внутри фазы можно привязать цели плагинов (Goals):

Default Lifecycle Bindings - Packaging `ejb` / `ejb3` / `jar` / `par` / `rar` / `war`

Последовательность фаз ↓	Phase	plugin:goal
	<code>process-resources</code>	<code>resources:resources</code>
	<code>compile</code>	<code>compiler:compile</code>
	<code>process-test-resources</code>	<code>resources:testResources</code>
	<code>test-compile</code>	<code>compiler:testCompile</code>
	<code>test</code>	<code>surefire:test</code>
	<code>package</code>	<code>ejb:ejb</code> or <code>ejb3:ejb3</code> or <code>jar:jar</code> or <code>par:par</code> or <code>rar:rar</code> or <code>war:war</code>
	<code>install</code>	<code>install:install</code>
	<code>deploy</code>	<code>deploy:deploy</code>

Цель из себя представляет конкретную задачу, которую может выполнить конкретный плагин (представляйте это как метод в классе).

Фазы Сборки приложения

У тестов могут быть цели в разных фазах сборки

как узнать, к какой фазе какие цели привязаны?

A: можно посмотреть Effective POM либо воспользоваться специальным плагином: `mvn help:describe -Dcmd=clean` (через `-D` передаются аргументы в формате `ключ=значение`).

Общий формат запуска цели любого плагина выглядит вот так: `mvn groupId:artifactId:version:goal`.

Важно: не внедряйте ничего в командную работу в одностороннем порядке, потому что вы так решили. Обязательно согласуйте это с коллегами!

Ни в коем случае не вмешивайтесь в процессы и инструменты программистов, пока заранее с ними это не обговорите и не «продадите им идею».

Jar (Java Archive) — это архив, в который упаковываются скомпилированные Java-приложения* и метаданные для дальнейшего распространения (развёртывания на сервере, подключения к другим проектам, публикации в репозиториях).

Затем этот архив можно будет с помощью фазы `install` опубликовать в локальном репозитории (каталог `.m2` в вашем домашнем каталоге), либо в

Maven Central или другом репозитории с помощью фазы `deploy`.

Но в большинстве случаев в рамках тестирования запускают фазу `verify`, в которой можно проводить доп.проверки после упаковки.

Maven запускается в формате: `mvn [options] [<goal(s)>] [<phase(s)>]`, справку можно получить с помощью `mvn -h`

И тестов не было, то можно просто «ронять» сборку.

Поиск на сайте Maven можно найти на описание [конфигурации Surefire](#):

<failIfNoTests>

Set this to "true" to cause a failure if there are no tests to run. Defaults to "false".

- **Type:** `java.lang.Boolean`
- **Since:** 2.4
- **Required:** No
- **User Property:** `failIfNoTests`

Примечание*: в GitHub можно настроить Protected Branches (бранчи, в которые нельзя "заливать" код кому попало).

Code Coverage — метрика, показывающая, насколько наш код покрыт автотестами (т.е. % запущенного в результате прогона автотестов).

Минимальная конфигурация Jacoco

<plugin>

<groupId>org.jacoco</groupId>

<artifactId>jacoco-maven-plugin</artifactId>

<version>0.8.5</version>

<executions>

<execution>

<!-- id (придумываем сами) -->

<id>prepare-agent</id>

<phase>initialize</phase>

<goals>

<!-- какую цель выполняем (берём из справки) -->

<goal>prepare-agent</goal>

</goals>

</execution>

<execution>

<!-- id (придумываем сами) -->

<id>report</id>

<goals>

<!-- какую цель выполняем (берём из справки) -->

<goal>report</goal>

</goals>

</execution>

</executions>



</plugin>

JACOCO

- `prepare-agent` подготавливает агента JVM для отслеживания вызовов кода
- `report` генерирует отчёт

После запуска `verify` отчёт будет в каталоге `target/site/jacoco/index.html`:

statistics

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ru.netology.statistic		93 %		75 %	1	4	1	7	0	2	0	1
Total	2 of 33	93 %	1 of 4	75 %	1	4	1	7	0	2	0	1

- **зелёный фон** — выполнено при прохождении тестов
- **жёлтый фон** — выполнено не до конца (одна из веток не отработала)
- **красный фон** — не выполнено

Таким образом, мы видим, что наши тесты не покрывают нескольких участков кода (это необходимо исправить).

СИНДРОМ 100%

Исчерпывающее тестирование, как вы знаете, невозможно.

Поэтому нужно, чтобы вы всегда помнили следующее: **Code Coverage** вам покажет **только участки кода, которые не исполнялись в результате прогона тестов**.

Code Coverage **не говорит о том, что своими тестами вы покрыли все возможные сценарии**.

Покрытие в 100% показывает только то, что все участки кода в результате прогона тестовы были исполнены.

Вы должны по-прежнему использовать комбинаторику, тест-анализ и тест-дизайн для того, чтобы покрывать **сценарии использования**, а не строки кода