

[Архив] Версионирование

- 1.1. Версионность
 - 1.1.1. Формат SemVer
 - 1.1.2. Версионирование продукта
 - 1.1.3. Управление версиями и Git Flow
 - 1.1.4. Виды веток
 - 1.1.5. Ветви master и develop
 - 1.1.6. Ветвь feature
 - 1.1.7. Ветвь hotfix
 - 1.1.8. Ветвь release
 - 1.1.9. Номера версий и ветки GitFlow

2. Автоматизация версионирования

1.1. Версионность

При разработке даже относительно несложных продуктов нужно быть уверенным в том, кто релиз никогда не будет выпущен прежде, чем пройдет все стадии анализа и тестирования.

Инструментов, нужные для выполнения этих задач, давно существуют, но для автоматизации их применения требуются некоторые соглашения и шаблоны.

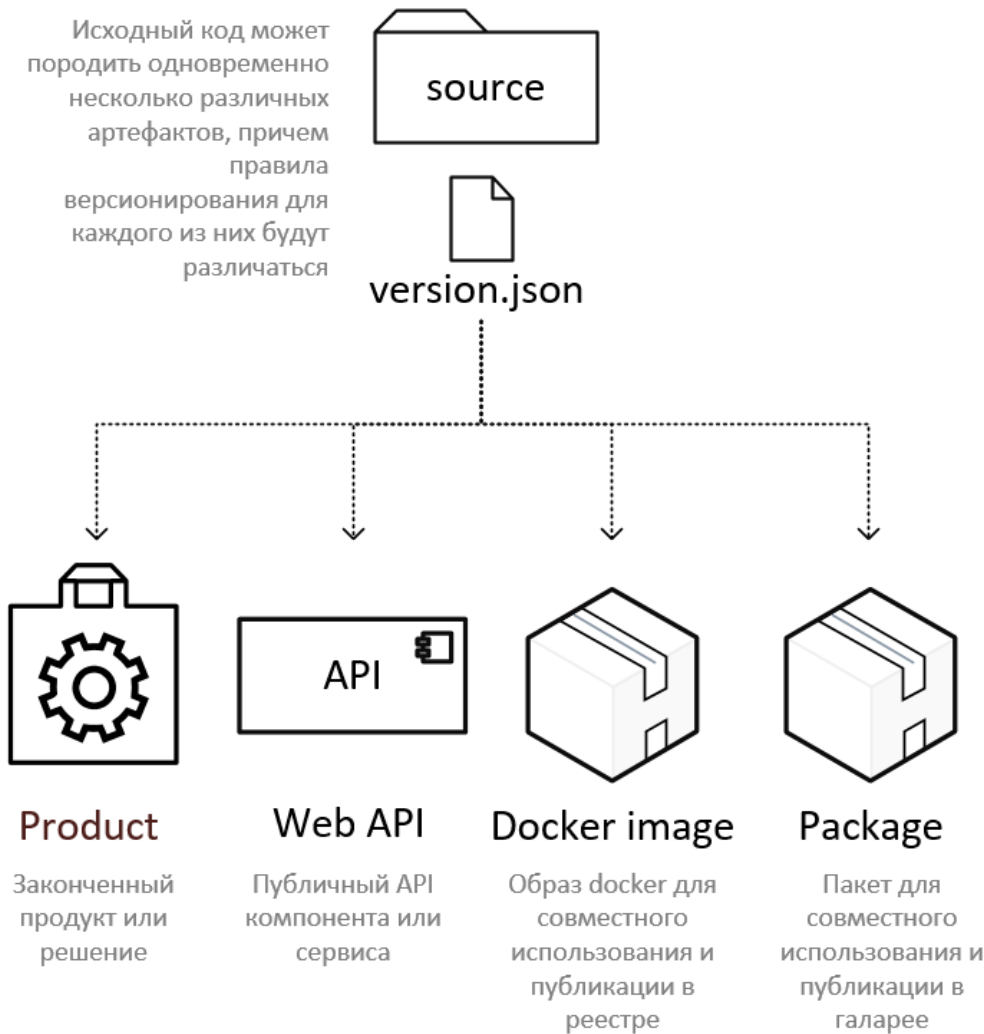
Как правило, в жизненном цикле продукта важную роль играет его версия: она помогает понять, что, когда, кем и в какой последовательности менялось. Например, необходимо убедиться, что конкретная версия продукта может быть развернута только в тех средах, где она способна работать. Для этого потребуется способ уникально идентифицировать версии продукта (и, разумеется, всех артефактов проекта).

Нужно отметить, что проблема версионирования довольно объемна и распространяется одновременно на разные части продукта (и проекта, порождающего продукт), причем разные части продукта (и разные артефакты проекта соответственно) имеют свои истории версий со своими правилами и особенностями. Однако в данной статье будет рассматриваться только версионирование продукта.

В качестве решения данной проблемы рекомендуется простой набор правил и требований, определяющих, как назначаются и увеличиваются номера версий.

В момент создания проекта ему присваивается его первая версия в формате, рассматриваемом ниже. Любые изменения в проекте (добавление-удаление файлов, изменение их атрибутов, исправление исходного кода) отражаются особым изменением номера версий или метаданных в версии продукта.

Исходный код может породить одновременно несколько различных артефактов, причем правила версионирования для каждого из них будут различаться



version.json

```
{
  // semVer.
  // .
  "version": "3.0.0-beta.1+meta",
  // ,
  "vcs": {
    "sha": "c4eb984a439f2b17c302166ed6b23048c207bffa",
    "branch": "master"
    "commitDate": "2020-04-20T11:10:34.164Z",
  },
  // ,
  "build": {
    "url": "https://tc/viewLog.html?buildId=6965196"
  }
}
```

1.1.1. Формат SemVer

Семантическое версионирование (Semantic Versioning) - схема версионирования, которая однозначно описывает смысл внесенных в продукт изменений и их историю.



На иллюстрации показаны семантические части строки версии. Наиболее важная из них первая часть **Major.Minor.Patch**, а полностью семантическая версия может состоять из следующих частей:

- **Major** - старший номер версии. Увеличивается, когда API библиотеки или приложения меняется обратно несовместимым образом.
- **Minor** - младший номер версии. Увеличивается, когда в API добавляются новые функции без нарушения обратной совместимости. При увеличении номера **Major** нумерация **Minor** сбрасывается в 0, то есть за версией 1.5 последует версия 2.0... а не 2.5
- **Patch** - номер патча (или номер сборки). Увеличивается при исправлении ошибок, рефакторинге и прочих изменениях, которые не меняют функционал. При увеличении **Major** или **Minor** в 0 сбрасывается **Patch**. В сценариях автоматической сборки **Patch** воспринимается как номер сборки (**Build**) и увеличивается при каждой сборке не 1, что в итоге позволяет легко отличать билды во время тестирования.
- **Pre-release** - необязательный, разделенный точками список, отделенный от трех номеров версии знаком **минус** (например: "1.2.3-beta.2"). Используется вместо тегов, чтобы пометить определенные вехи в разработке. Обычно это *alpha*, *beta*, *release candidate* и производные от них.
- **Metadata** - метаданные системы сборки. Разделен на составные части точками, но отделен от номеров версии или тегов плюсом (**+**). Данную информацию с точки продуктовой версии принято игнорировать, она интересна лишь разработчикам. В метаданные удобно включать имя ветки, из которой была собрана версия и хэш-код коммита. Например, 0.1.0-alpha.1+Branch.develop.Sha.cdb698d7521ef32.

Еще раз, простое правило версионирования:

Учитывая номер версии **Major.Minor.Patch**, следует увеличивать:

- **Major** - когда сделаны обратно несовместимые изменения API.
- **Minor** - когда добавлена новая функциональность, не нарушая обратной совместимости.
- **Patch** - когда делаются обратно совместимые исправления.
- **Pre-release** (опционально) - тэг в зависимости от фазы жизненного цикла продукта (alpha, beta), а номер билда - при каждом изменении.
- **Metadata** (опционально) - автоматически в процессе сборки.

Еще несколько правил:

- Начальная версия каждого нового продукта - 0.1.0
- В момент первого релиза ему присваивается версия 1.0.0
- Перед удалением из проекта части функционала необходимо выпустить промежуточный минорный релиз (например, 1.0.0 → 1.1.0), в котором функционал, запланированный к удалению в версии 2.0.0, будет помечен как устаревший.
- В строке версии может использоваться префикс **"v"** (например, v1.0.0), но его принято игнорировать.

Более подробно о правилах версионирования можно узнать по ссылке semver.org

1.1.2. Версионирование продукта

Для версионирования продукта используется следующее правило.

Мажорная версия:

Prototype → 1

MVP → 2

OFP → 3

OFP+ и т.д. → необходимо руководствоваться Semver, здравым смыслом и потребностями продукта.

Минорная версия и патч соответствуют правилам SemVer, но обычно используется правило:

Минорная версия:

Релиз спринта → +1

Патч:

После релиза версии и установки на прод для текущей версии увеличивается только патч.

1.1.3. Управление версиями и Git Flow

Рабочий процесс Git Flow представляет собой модель ветвления с жесткими границами между ветвями и особыми правилами переноса кода. Основные принципы довольно просты и интуитивно понятны.

Одним из преимуществ этой модели является то, что она очень хорошо работает с SemVer и моделью непрерывной интеграции и развертывания (CI/CD). От читателя ожидается, что он уже знаком с Git Flow, и документ описывает совместное использование Git Flow и SemVer.

1.1.4. Виды веток

Сборки для тестирования рекомендуется брать из веток **develop** или **release/\${version}**.

№	Ветка	Описание
1	master	<p><u>Описание:</u> основная ветка хранения продукта, содержащая последнюю по нумерации отгруженную версию; необходима для хранения линейной истории развития продукта, создания новых версий продукта с чистой историей коммитов путем перемещения в нее требуемой версии.</p> <p><u>Содержит merge</u> из веток: release/\${version}, develop.</p>
2	develop	<p><u>Описание:</u> ветка предназначена для разработки определённой версии/версий продукта.</p> <p>Вместо имени develop может быть использована версия продукта в формате SemVer, если есть необходимость поддерживать несколько мажорных версий.</p> <p><u>Примеры названий веток:</u></p> <ul style="list-style-type: none">• develop• 1.x.x• 1.5.x• 1.5.3 <p><u>Ответвляется от веток:</u> master, 1.x.x, release/\${version}.</p> <p><u>Содержит merge</u> из веток: feature/*, bugfix/*, hotfix/*.</p> <p><u>Содержит прямые commit'ы:</u> нет</p>
3	release/\${version}	<p><u>Описание:</u> ветка предназначена для подготовки к релизу версии/версий продукта. После релиза данная ветка замораживается для изменений.</p> <p><u>Примеры названий веток:</u></p> <ul style="list-style-type: none">• release/3.2.1 <p><u>Ответвляется от веток:</u> develop.</p> <p><u>Содержит прямые commit'ы:</u> нет.</p>

4	feature /* bugfix /* hotfix /* refactor /*	<p><u>Описание:</u> ветка разработки фрагмента функциональности, исправления ошибки и проведения рефакторинга в версии продукта.</p> <p><u>Правило именования ветки:</u> feature/\${version}/\${jira-ticket-key}/\${short-description}, где:</p> <ul style="list-style-type: none"> • \${version} – например, когда функциональность по одному и тому же тикету реализуется в двух разных версиях в двух разных ветках по-разному; • \${jira-ticket-key} – не указывается только в случае отсутствия тикета на работу или \${jira-ticket-key} указан в commit'ах, что позволяет организовать корректную связь с jira; • \${short-description} – краткое описание изменений <p><u>Примеры названий веток:</u></p> <ul style="list-style-type: none"> • feature/3.2.1/PROJECT-4511/add-new-code • bugfix/PROJECT-1766/my-module/memory-leak • refactor/rename-a-to-b • hotfix/2.2.0/PROJECT-123/PROJECT-12345/crash-app
---	---	--

1.1.5. Ветви master и develop

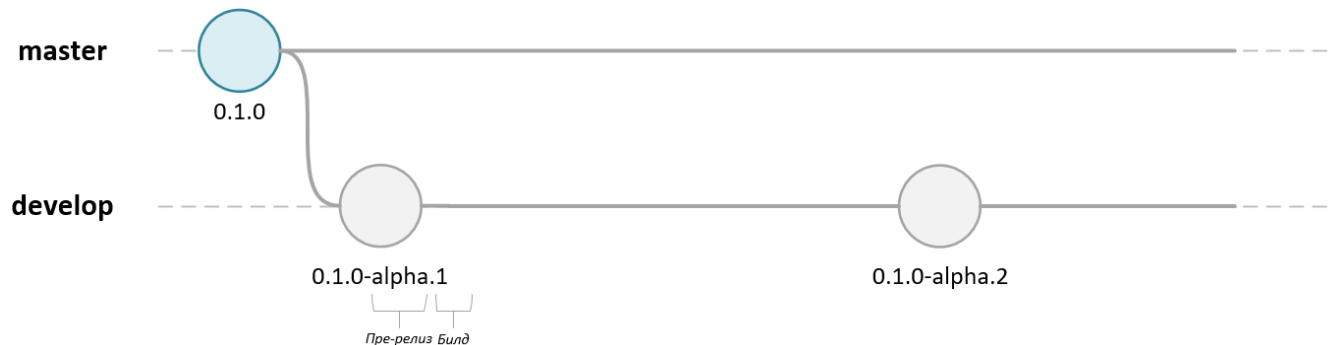
Эти две ветви долговечны и должны существовать всегда. Каждый из них представляет историю проекта в разных состояниях.

Ветка **master** представляет официальную историю релизов. Каждый новый релиз будет объединен с **master**, в этой ветке находится последнее официально опубликованное состояние проекта.

Ветка **develop** служит для интеграции функционала и содержит полную историю проекта. Новые ветки релиза должны быть созданы из ветки разработки и будут использоваться как временная ветка для подготовки новой версии.

Ветка **master** должна регулярно сливаться с **develop**.

В примере ниже и в соответствии с правилами SemVer, версия **develop** получается из ветки **release** с добавлением к ней префикса **pre-release** и номера билда.

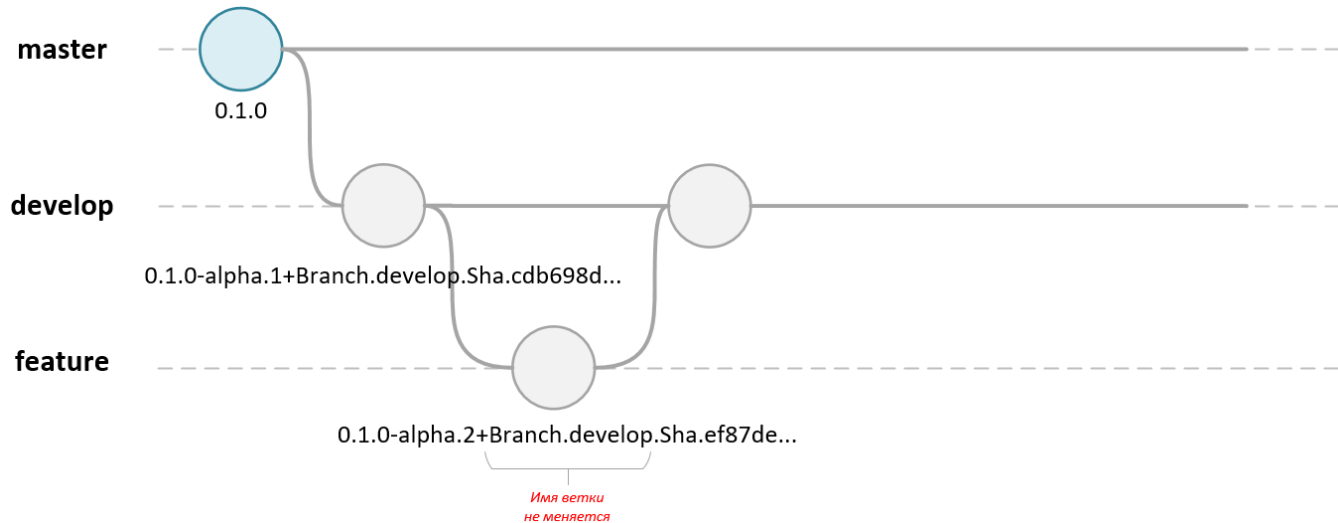


Удобно включить в версию имя ветки, на основе которой она создалась, и хэш коммита. Лучше автоматизировать этот процесс через конвейер CI или сборочным скриптом на стороне разработчика. На рисунке ниже можно заметить, как между версиями меняется номер билда и хэш (здесь и далее изменения отмечены синим цветом).



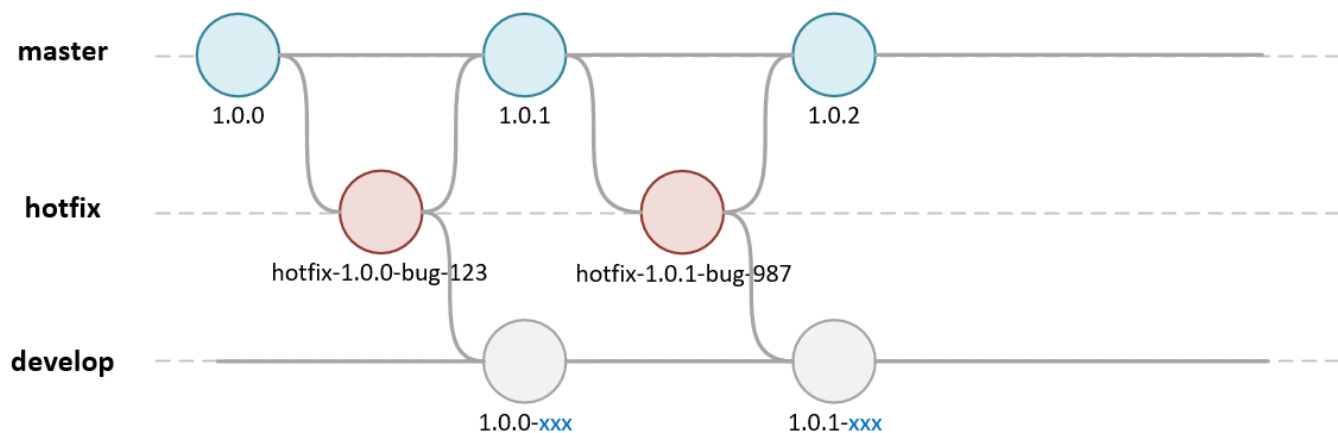
1.1.6. Ветвь feature

Каждая функция должна находиться в отдельной ветке, пока она не будет завершена, но так же стоит декомпозировать код таким образом, что бы функции были небольшими, а количество изменяемых файлов минимальным. Когда функция завершена, она объединяется с **develop** и готовой стать частью следующей релизной ветки. Номер версии **feature**-ветки может всегда имеет **pre-release**-префикс *alpha*, а версия ветки **develop** может иметь префикс *alpha* или *beta* (предпочтительно, но не обязательно). Политику выдачи префиксов можно автоматизировать.



1.1.7. Ветвь hotfix

Ветка **hotfix** используется для исправлений к релизам. Эти исправления должны быть сделаны с осторожностью, и ответственность за их негативное влияние на производственную среду несет разработчик и группа тестирования. Это позволяет команде разработчиков выполнять итерации выпусков, не прерывая остальную часть рабочего процесса и не ожидая следующего цикла выпуска. Исправление ответвляется от определенного релиза из **master**, а после завершения работы сливается с **master** и **develop**. Номер версии исправления формируется из версии релиза и включает идентификатор задачи в баг-трекере (например, Jira).
Только сейчас заметил, что я сначала сливал hotfix с master, а потом master с develop

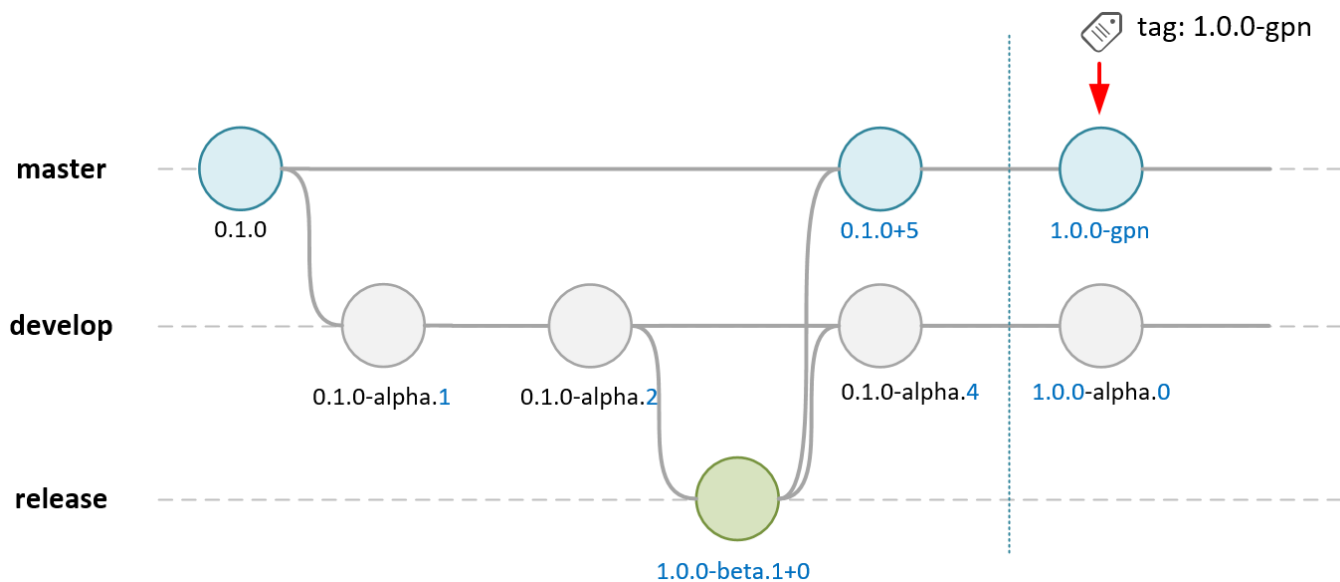


1.1.8. Ветвь release

В определенный момент (при приближении заранее определенной даты выпуска) или при других обстоятельствах, создается новая ветка **release /xxx** для подготовки и стабилизации новой версии продукта. После создания ветки релиза новые функции вводить не следует, и допускаются только изменения в форме исправлений ошибок, незначительных улучшений, правки документации и других задач, ориентированных на решение проблем выпуска. Команда разработчиков работает в тесном контакте с командой поддержки, чтобы перебрать проблемы в ветке релиза и обработать ее, пока она не будет считаться готовой к работе.

1. При создании ветки release на основе develop ее версия имеет pre-release-префикс beta.

- После внесения необходимых изменений в релизную ветку и слияния ее с master к последней применяется тэг с финальной версией 1.0.0-gpn.
- Теперь версия master будет иметь версию 1.0.0-gpn



1.1.9. Номера версий и ветки GitFlow

Примеры выше показали наличие некоторых правил, на основе которых версиям присваиваются **pre-release**-префиксы. В основном правило выглядит так: версия master не содержит префиксов (потому что она финальная), версия release и hotfix имеет префикс release candidate,

Правила присвоения версий в зависимости от ветки:

Ветка	Тип релиза	Формат версии	Пример ветки	Тэг	Пример версии
feature	alpha	Major.Minor.Patch-alpha.Build.Feature	feature/super-overkill-feature		1.1.0-alpha.1
develop	beta, [alpha]	Major.Minor.Patch-alpha[beta].Build	develop		1.1.0-beta.1
release	release candidate, [beta]	Major.Minor.Patch-beta[rc]1	release/1.1.0		1.1.0-rc.1
hotfix	release candidate	Major.Minor.Patch-rc1.Build	hotfix/1.1.0/bug-123		1.1.0-rc.2
master	stable	Major.Minor.Patch	master	v1.1.0	1.1.0

На рисунке ниже показано, как

1.1.0-alpha.1+Branch.develop.Sha.72ec3...



```
{
  "SemVer": "1.1.0-alpha.1",
  "Major": 1,
  "Minor": 1,
  "Patch": 0,
  "PreReleaseLabel": "alpha",
  "PreReleaseNumber": 1,
  "BranchName": "develop",
  "Sha": "72ec3bd8d1dfff3535f6df723350da9e13fda152b",
  "CommitDate": "2020-04-17"
}
```

2. Автоматизация версионирования

Для версионирования (то есть задания значения текущей версии в файле version.json) можно использовать два равнозначных подхода: ручной и автоматизированный.

- При ручном - разработчик (вероятнее всего тимлид) вручную задает значение поля SemVer согласно правилам, описанным в предыдущих частях документа.
- При автоматизированном версионировании заполнение файла version.json выполняется скриптом version.sh, который в свою очередь берет номер версии (и ее метаданные) из утилиты наподобие GitVersion, а утилита извлекает эти данные из локального репозитория git.

Но даже если выбран ручной режим, разработчику ничто не мешает запустить скрипт version.sh самостоятельно.

Ручное
версионирование

Автоматизированное
версионирование



Team
Lead



version.sh



GitVersion.exe



version.json



git



Artefact

Артефакт сборки:
продукт,
компонент,
библиотека