Структура проектов на C++ и управление ими

Е.А. Охотников

12 мая 2004 г.

Содержание

1 Введение

Данный документ описывает предложения по организации файловой структуры проектов на C++ и по основным способам управления C++ проектами. Основной внимание при разработке этих предложений было уделено возможности помещения исходного кода проектов под *version control*.

2 Основные понятия

2.1 Проект, подпроект, модуль

Проект — это базовое понятие. Под проектом подразумевается совокупность исходных файлов, распределенных каким-то образом по подкаталогам проекта. Проект может содержать один или более[1] "проектный файл" для постороения из исходных текстов чего-либо, что может считаться целью проекта. Например, для C++ это может быть статическая библиотека, динамическая библиотека, исполнимый файл.

Проект может нуждаться в *подпроектах*. Подпроект — это исходные (и не только) тексты другого проекта, импортированные в структуру каталогов проекта. Например, проект `so_4` нуждается в исходных текстах проектов `auto_ptr_3`, `cpp_util_2`, `memcheck_2`, `oess_1`, `threads_1` и др. Эти проекты объявляются подпроектами проекта `so_4`. При помощи специальной команды их исходные тексты копируются в каталог к проекту `so_4` и используются для компиляции `so_4`.

Важно, что исходные тексты проекта находятся под version control. Но импортированные исходные тексты подпроектов под version control самого проекта не помещаются.

Проект состоит из одного или нескольких *модулей*. Каждый модуль представляет из себя некую самостоятельную сущность, которая может выступать в качестве подпроекта в другом проекте. Например, проект `oess_1` состоит из модулей: `oess_1/defs`, `oess_1/io`, `oess_1/file`, `oess_1/stdsn`, `oess_1/db`, `oess_1/tlv` и др. Включение всех модулей подпроекта `oess_1`, например, в проект `so_4` может быть не выгодно, если проект `so_4` использует функциональность только модуля `oess_1/stdsn`. В этом случае в качестве подпроекта `so_4` использует модуль `oess_1/stdsn`.

Проект должен иметь хотя бы один *модуль по-умолчанию (default-module)*. Это модуль, который инсталлируется в качестве подпроекта, если какой-то проект не указывает, в каких конкретно модулях он нуждается. Например, если проект `so_4` указывает, что он нуждается в проекте `oess_1`, а модулем по-умолчанию в `oess_1` является `oess_1/defs`, то в проекте `so_4` в качестве подпроекта будет использоваться именно `oess_1/defs`.

2.2 Версия: поколение, ветвь, релиз

Версия — это некоторый идентификатор, который определяет функциональность проекта. Предлагается использовать идентификатор из трех составляющих: *поколение*

[1] Проект может вообще не содержать проектных файлов, если целью проекта является предоставление исходных файлов другим проектам. Для C++ примерами таких проектов являются `cpp_util_2`, `auto_ptr_3`, большая часть библиотек boost и т.д., в которых код сосредоточен в inline-функциях и не требует отдельной компиляции.

(generation), *ветвь (branch)*, *релиз (release)*. Например, запись `oess` версии 1.1.4 говорит о релизе 4 ветви "2" первого поколения проекта `oess`.

Поколение и *ветвь* определяют степень несовместимости различных версий проектов. Версии абсолютно не совместимы, если не совпадает значение поколения. Т.е., если проект `so` использовал `oess` первого поколения, то для перехода на второе поколение `oess` потребуется серьезная переделка проекта `so`. Значение поколения следует изменять, если в проекте происходят фундаментальные изменения. Например, меняется идеалогия работы с проектом.

Поколение проекта является настолько важным показателем, что номер поколения следует включать в имя проекта. Так, название `oess_1` сразу говорит о проекте "oess" первого поколения, название `so_4` — о проекте "so" четвертого поколения и т.д.

Разные версии проекта более-менее совместимы между собой, если в их версиях совпадает номер поколения, но не совпадают имена ветвей. Несовпадающие имена ветвей говорят о том, что переход с одной ветви на другую возможен без значительных изменений в использующем этот проект проекте. Т.е. гарантируется сохранение идеалогии работы с проектом, но в незначительной степени меняется синтаксис.

Если в качестве имен ветвей использовать числовые значения[2], например, 4.0.3, 4.1.5, 4.2.7, то увеличение номера ветви говорит о создании новой версии, не гарантирующей 100% совместимости "сверху-вниз".

Релиз задается числовым значением. Номер релиза увеличивается при внесении очередных "значимых"[3] изменений в проект. При этом, если версии проекта различаются только номером релиза, то должно гарантироваться, что версия с большим номером релиза обеспечивает 100% совместимость "сверху-вниз". Т.е. переход, например, с версии 4.1.5 на 4.1.18 должен пройти без проблем. Должно быть достаточно импортировать новую версию подпроекта и перекомпилировать проект.

Важно, что версия с большим номером релиза не гарантирует обратной совместимости. Т.е., в общем случае, нельзя после использования, например, версии 4.1.18 перейти на использование версии 4.1.5.

2.2.1 Имя проекта и номер версии

Под именем проекта можно понимать как имя некоторого глобального направления работ, в рамках которых будет создано несколько поколений и множество версий этого проекта. Например, имя `oess` может привести к существованию версий 1.0.*[4], 1.1.*, 1.2.*, 2.0.* и т.д. Поэтому, когда используется имя `oess` следует указывать, о какой конкретно версии или поколении идет речь. Например, `oess` v.1.1.4 или `oess` v.2.*.*.

Поскольку номер поколения нужно включать в имя проекта, то имя `oess_1` уже говорит о всех версиях проекта `oess` первого поколения. Для уточнения версии после имени проекта можно указывать имя ветви и номер ревизии. Например, `oess_1.1.4` — это то же самое, что `oess` v.1.1.4. Или `so_4.dombai.4` — то же самое, что `so` v.4.dombai.4.

[2]В качестве имени ветви может быть удобно использовать символьные идентификаторы. Например, собственное имя, данное этой ветви в процессе разработки. Так, при выпуске ветви проекта `SObjectizer` по названием "Домбай" имени ветви можно присвоить значение "domai".

[3]Значимость внесенных изменений — это субъективное понятие, определяемое самим разработчиком. Изменение номера релиза должно отражать тот факт, что в проект были внесены изменения, которые могут вызвать необходимость перехода на новую версию проекта. Например: добавлена новая функциональность, но при этом гарантируется совместимость. Или исправлены серьезные ошибки. Или выпущена новая версия документации.

[4]Звездочка в качестве номера релиза указывает, что номер релиза не имеет значения

2.3 Репозитории

Репозиторием назвается каталог, в котором собираются *опубликованные*[5] версии проектов. Данный каталог должен находиться под version control.

Разработчику должен быть доступен один или более репозиториев. Как минимум, ему нужен репозиторий, в который разработчик должен публиковать выпускаемые им версии проектов. Так же репозиторий необходим для получения необходимых версий используемых подпроектов.

Доступ разработчика к разным репозиториям может быть ограничен, если такое ограничение поддерживается системой контроля версий. Например, разработчик имеет read-write доступ к репозиторию, в который он публикует собственный проект. И read-only доступ к репозиториям, из которых он может только импортировать необходимые ему подпроекты.

Репозитории так же позволяют сгруппировать разрабатываемые проекты по тематическим направлениям. Например, репозиторий Objessty содержит проекты, относящиеся непосредственно к проекту `oess`: `oess`, `oess_sequence`, `oess_recovery_db` и т.д. Репозиторий SObjectizer содержит проекты, относящиеся непосредственно к проекту `so`: `so`, `so_log`, `so_log_rtl`, `gemont`, `omc` и т.д. Репозиторий Aggregate может содержать проекты: `smpp_pdu`, `smsg`, `aggregate`, `mbsms`. В этом случае разработчику проекта `aggregate_2` дается read-write доступ к репозиторию Aggregate и read-only доступ к репозиториям Objessty и SObjectizer.

2.4 Рабочий каталог

Проект должен находится под version control. Для внесения изменений в проект необходимо выполнить операцию *check out*, которая так или иначе поддерживается всеми системами контроля версий. Каталог, в который выполняется операция check out называется *рабочим каталогом (workspace)*.

Некоторые системы управления проектами, например, Subversion, позволяют делать check out одной и той же ревизии проекта в произвольное количество рабочих каталогов. Для описываемой ниже системы управления проектами не важно, в какое количество рабочих каталого проект был checked out. Важно только, что будет в результате изменения проекта опубликовано в репозитории.

3 Управление проектом

3.1 Файловая структура проекта

Для C++ проектов рекомендуется следующая файловая структура рабочего каталога:

```
workspace_name/
‘--dev/
   |  *.exe, *.dll, *.4xx (для компиляции и настройки всего проекта)
   |--lib/ *.lib
   |--project/
   |  |--module-1/ *.cpp, *.rc, *.4xx
   |  |  |--h/ *.hpp, *.h
```

[5] Подробнее о публикации проектов см.3.3.3

```
|  |  `--o/ *.obj, *.res
|  |--module-2/ *.cpp, *.rc, *.4xx
|  |  |--h/ *.hpp, *.h
|  |  `--o/ *.obj, *.res
|  |  ...
|  `--module-N/ *.cpp, *.rc, *.4xx
|     |--h/ *.hpp, *.h
|     `--o/ *.obj, *.res
|--sub_project_1/
|  |--module-1/
|  |  |--h/
|  |  `--o/
|  |  ...
|  `--module-N/
|     |--h/
|     `--o/
|  ...
`--sub_project_N/
   |--module-1/
   |  |--h/
   |  `--o/
   |  ...
   `--module-N/
      |--h/
      `--o/
```

Каталог `dev/`, в котором располагаются подкаталоги с исходными текстами проектов и подпроектов, называется *корневым каталогом проекта*. Корневой каталог используется чаще всего, т.к. именно в нем инициируется компиляция проекта, в него помещаются исполнимые файлы и DLL (на платформе Windows).

Каталог, в котором располагается каталог `dev/` называется *родительским каталогом* проекта. Родительский каталог проекта служит для хранения *манифеста проекта*. Из родительского каталога запускаются команды управления проектом (подробнее см.3.3). В родительский каталог проекта помещаются архивы, полученные в результате выполнения команды *snapshot*[6]

3.2 Манифест проекта

Манифест проекта — это специальный текстовый файл с именем `manifest`, расположенный в родительском каталоге проекта.

Манифест проекта используется для указания:

- имени проекта;
- имени ветви проекта и перечислению совместимости данной ветви с другими ветвями проекта;
- номера релиза проекта;

[6]Подробнее о команде snapshot см.3.3.4

- перечисления всех модулей проекта;
- перечисления всех необходимых подпроектов.

Пример манифеста проекта:

```
{project
  {name "cls_2"}
  {branch "6"
    {substitutes "4" {< 7} }
    {substitutes "5" {< 9} }
  }
  {release "4"}

  {module "default" {default}
    {paths {+ "dev/cls_2"}
           {+ "dev/howto/cls_2"}
           {+ "dev/guide/cls_2"}
    }

    {templates {+ cpp} {+ latex}}

    {requires {project "auto_ptr_3" {branch "1"} } }
  }

  {module "stl"
    {paths {+ "dev/cls_2/stl"}
           {+ "dev/howto/cls_2/stl"}
           {+ "dev/guide/cls_2/stl"}
    }

    {templates {+ cpp} {+ latex}}

    {requires {module "default"}}
  }
}
```

Данный манифест описывает 4-й релиз ветви “6” проекта `cls_2`. Данная ветвь может использоваться вместо ветвей “4” и “5” этого проекта (т.е. гарантируется 100% совместимость сверху-вниз)[7]

Проект состоит из двух модулей: `default` и `stl`. Модуль `default` будет включаться в качестве подпроекта, если в чьем-либо манифесте будет указано

[7]Могут быть случаи, когда новая ветвь гарантирует 100% совместимость сверху-вниз, но все-равно имеет собственное имя. Например:
- если новая ветвь получена в результате серьезного рефакторинга кода. Т.е. все внешние интерфейсы 100% совместимы, но внутреняя реализация полностью переработана;
- если разработка новой ветви и дальнейшее развитие проекта передано другой команде разработчиков. Тогда имена ветвей просто показывают какая команда отвечает за ветвь проекта;
- если код новой версии распространяется под другой лицензией. Например, вместо GPL применяется LGPL или BSD-лицензия.

`{requires {project "cls_2"}}`. Модуль `stl` будет включаться в качестве подпроекта только, если это явно указано в теге `{requires}`.

Модуль `default` включает в себя все исходные тексты из каталогов `dev/cls_2`, `dev/howto/cls_2`, `dev/guide/cls_2`. При этом в модуль `default` не включаются исходные тексты, которые входят в модуль `stl` (т.е. исходные тексты из каталогов `dev/cls_2/stl`, `dev/howto/cls_2/stl`, `dev/guide/cls_2/stl`).

Модуль `default` требует для себя подпроект `auto_ptr_3`. Причем ветвь "1" или любую на 100% с ней совместимую.

Модуль `stl` требует, чтобы вместе с ним использовался модуль `default` точно из той версии `cls_2`. Т.е., если какой-то проект указал в качестве подпроекта модуль `stl` из `cls_2.6.4`, то в этот же проект в качестве подпроекта будет включен модуль `default` из `cls_2.6.4`.

Подробнее манифест проекта рассматривается в 5.

3.3 Команды по управлению проектом

Предполагается, что существует некая утилита `prjman`, которая отвечает за выполнение команд по управлению проектами. Данная утилита поддерживает следующие команы:

- *populate*: повести наполнение проекта исходными текстами необходимых ему подпроектов. Подробнее в 3.3.1;
- *upgrade*: провести обновление исходных текстов необходимых ему подпроектов до новых версий. Подробнее в 3.3.2;
- *publish*: опубликовать текущую версию проекта в репозиторий, для того, чтобы ее можно было использовать в качестве подпроектов. Подробнее в 3.3.3;
- *snapshot*: зафиксировать текущую версию исходных текстов и исходных текстов всех его подпроектов в общий архив, который размещается в родительском каталоге проекта. Подробнее в 3.3.4.

3.3.1 Команда populate

Когда исходные тексты проекта в первый раз извлекаются из системы контроля версий в рабочий каталог, в рабочем каталоге нет исходных текстов необходимых ему подпроектов. Для того, чтобы начать реальную работу с проектом необходимо "населить" рабочий каталог подпроектами. Для этого предназначена команда *populate*.

Команда populate работает следующим образом:

1. считывает содержимое манифеста проекта и определяет все необходимые проекту подпроекты и их версии;
2. считывает из репозиториев манифесты для необходимых подпроектов и определяет все необходимые подпроектам подпроекты. Этот шаг повторяется до тех пор, пока не будут найдены **все** необходимые проекту подпроекты;
3. проверяет отсутствие конфликтов между версиями подпроектов;
4. при отсутствии конфликтов выбирает конкретные версии подпроектов, которые должны быть установлены;

5. разархивирует из архивов, которые находятся в репозитории, необходимые версии подпроектов в рабочий каталог проекта.

Конфликты. *Конфликт* возникает, когда несколько подпроектов нуждаются в разных версиях одного общего подпроекта. Например, есть проект `aggregate_2.2.0` который нуждается в подпроектах `smsg_2.7.0` и `mbapi_srv_3.1.4`. При этом, проект `smsg_2.7.0` нуждается в `oess_1.1.6`, а `mbapi_srv_3.1.4` в `oess_1.2.0`. По правилам формирования имен версий проекты `oess_1.1.6` и `oess_1.2.0` не 100% совместимы между собой. Поэтому использовать их в составе одного проекта (в данном случае `aggregate_2.2.0`) нельзя.

Выбор подпроекта из нескольких вариантов В некоторых случаях может потребоваться сделать выбор между несколькими совместимыми версиями одного проекта. Например, подпроект `smsg_2.7.0` требует версию `oess_1.1.2`, а подпроект `mbapi_srv_3.1.4` — `oess_1.1.4`. А в репозитории могут быть доступны еще более новые версии, например, `oess_1.1.6` и `oess_1.1.10`. В этом случае по-умолчанию будет выбрана версия `oess_1.1.10`, т.к. ожидается, что она более функциональна, и при этом, гарантирует совместимость с версиями 1.1.2 и 1.1.4. Но, если, например, сам проект `aggregate_2.2.0` жестко требует `oess_1.1.6`, то будет установлена именно версия `oess_1.1.6`[8].

3.3.2 Команда upgrade

Команда *upgrade* предназначена для обновления версий используемых подпроектов. Например, пусть при выполнении команды populate был установлен подпроект `oess_1.1.6`. По прошествии некоторого времени в репозиторий был помещен обновленный вариант проекта `oess_1.1.6`. Команда upgrade позволяет взять из репозитория обновленную версию проекта `oess` и установить ее в рабочий каталог.

Так же команда upgrade позволяет перейти на другую версию подпроекта. Например, сменить `oess_1.1.6` на `oess_1.2.0`. При этом `prjman` сначала удалит из рабочего каталога уже установленную версию `oess_1.1.6`.

Команда upgrade позволяет изъять из рабочего каталога ставшие уже не нужными подпроекты. Например, если проект `aggregate_2.2.0` больше не нуждается в проекте `gemont_1`, то имя `gemont_1` удаляется из манифеста проекта, после чего команда upgrade удаляет ранее установленые исходные тексты `gemont_1` из рабочего каталога.

3.3.3 Команда publish

Версия проекта проходит несколько стадий разработки: написание исходных текстов и предварительная отладка (alpha-тестирование), отладка (beta-тестирование), стабилизация, сопровождение. В основном, эти действия выполняются в рабочем каталоге проекта. Но, если сам проект может использоваться в качестве подпроекта, то наступает момент, когда код версии проекта нужно сделать доступным для использования в других подпроектах. Т.е. *опубликовать* версию проекта в репозиторий.

Публикация — это помещение в репозиторий специально сформированного архива с исходными текстами проекта и, возможно, некоторой дополнительной технической

[8]Подробнее механизмы разрешения конфликтов и выбора версий проектов расматриваются в 4

информации[9]. Осуществляет публикацию команда *publish*.

В репозиторий помещается архив, имя которого содержит:

- имя проекта;
- имя ветви и номер релиза проекта;
- *статус* версии проекта.

Статусом проекта называется стадия, на которой находится разработка данной версии. Различаются следующие статусы:

- *draft*: осуществляется черновой набор исходных текстов проектов. Не гарантируется, что проект вообще может быть успешно скомпилирован. Данный статус добавлен для "полноты" статусов, т.к. в большинстве случаев вряд ли потребуется публиковать проекты с данным статусом[10];
- *alpha*: исходный текст проекта, в основном, набран и успешно компилируется. Реализована базовая функциональность проекта. Проведены только первоначальные тесты. Поэтому не гарантируется корректная работа проекта. Как и в случае со статусом draft, публикация проекта с данным статусом может потребоваться при необходимости как можно более ранней интеграции проекта в другие проекты;
- *beta*: проект находится в стадии активной отладки. Набор исходного текста практически завершен. Вся требуемая от проекта функциональность реализована, но не полностью претестирована. Публикация проектов с даным статусом может оказаться необходимой в случае, когда наиболее полное тестирование проекта можно осуществить только в составе другого проекта. Например, разрабатывается новая версия проекта `aggregate_2`, которая использует новые версии разрабатываемых параллельно проектов `mbapi_srv_3` и `smsg_2`. Естественно, что проекты `mbapi_srv_3` и `smsg_2` проходят автономное тестирование, но для них тестирование в составе проекта `aggregate_2` имеет еще большее значение — ведь это, фактически, уже реальная эксплуатация. Так же может оказаться, что для полного тестирования `aggregate_2` может потребоваться две разных тестовых конфигурации. В этом случае возможен следующий подход:
  - в манифесте проекта `aggregate_2` указывается, что необходимы beta-версии `mbapi_srv_3` и `smsg_2`;
  - проекты `mbapi_srv_3`, `smsg_2`, `aggregate_2` публикуются со статусом beta;
  - создается проект `aggregate_2_testconf_1` для первой тестовой конфигурации. В манифесте этого проекта указывается необходимость использования beta-версии `aggregate_2`;

[9]Например, может оказаться удобным сохранение рядом с архивом проекта в репозитории копии манифеста проекта. Это позволит при выполнении команды populate легко получать из репозитория манифесты нужных подпроектов, не прибегая к разархивации опубликованных архивов проектов.

[10]Возможный случай испозования статуса draft: когда проект является подпроектом сразу для нескольких проектов и желательно сделать доступными, например, интерфейсные заголовочные файлы как можно раньше. В этом случае вполне достаточным будет использование версии проекта со статусом draft.

  - создается проект `aggregate_2_testconf_2` для первой тестовой конфигурации. В манифесте этого проекта указывается необходимость использования beta-версии `aggregate_2`;
  - тестирование происходит в рабочих каталогах проектов `aggregate_2_testconf_1` и `aggregate_2_testconf_2`;
  - при обнаружении ошибок в каком-либо из тестируемых подпроектов исправления осуществляются в рабочем каталоге этого подпроекта. После чего обновленная версия публикуется в репозиторий (команда publish) и извлекается из репозитория (команда upgrade) в рабочие каталоги тестовых проектов `aggregate_2_testconf_1` и `aggregate_2_testconf_2`.

- *stable*: стабильная версия, готовая для промышленной эксплуатации.

Отдельного статуса для случая сопровождения проекта не предусмотрено, т.к. сопровождение — это внесение изменений в стабильную версию для получения очередной стабильной версии. Т.е. результатом сопровождения должна стать публикация этой же версии опять со статусом stable. Если при сопровождении необходимо на некотрое время перевести проект в нестабильное состояние, то достичь это можно с использованием двух механизмов:

1. Средств системы контроля версий. Например, можно создать еще один рабочий каталог, в который выполнить операцию check out из системы контроля версий. Осуществить сопровождение проекта в этом каталоге, после чего опубликовать новую версию из этого каталога или из основного рабочего каталога, предварительно выполнив операцию обновления рабочего каталога из системы контроля версий. При этом разработчик, исходя из возможностей своей системы контроля версий, определяет, как должны соотносится между собой его рабочие каталоги. Например, использовать ли механизм branch-and-merge или check-out/check-in/update.

2. Если нужно для тчательной отладки внесенных при сопровождении изменений опубликовать промежуточную версию проекта в репозитории, то можно сделать это со статусом beta. Тем самым в репозиторий будет помещен новый архив с beta-версией, но последний архив со stable-версией не изменится.

Как сказано выше, публикуемый в репозиторий архив содержит в своем имени информацию о версии и статусе. Например, если публикуется проект `oess_1.2.0` со статусом beta, то в репозиторий помещается файл: `oess_1-[2.0]-(stable).tar.bz2`[11].

Если проект состоит из нескольких модулей, то может оказаться более удобным при публикации помещать в репозиторий отдельный архив на каждый модуль проекта. В случае с beta-версией проекта `oess_1.2.0` это привело бы к появлению в репозитории архивов:

```
oess_1-{db}-[2.0]-(beta).tar.bz2
oess_1-{defs}-[2.0]-(beta).tar.bz2
oess_1-{file}-[2.0]-(beta).tar.bz2
```

[11] Архивация средствами команд tar и bzip2, во-первых, показывает очень высокую степень сжатия (даже большую, чем solid-архивы rar 3.1) и, во-вторых, это open-source средства, существующие практически на всех платформах, включая массу Unix-ов, MacOS и Windows.

```
oess_1-{io}-[2.0]-(beta).tar.bz2
oess_1-{scheme}-[2.0]-(beta).tar.bz2
oess_1-{stdsn}-[2.0]-(beta).tar.bz2
oess_1-{tlv}-[2.0]-(beta).tar.bz2
oess_1-{util_cpp_serializer}-[2.0]-(beta).tar.bz2
```

3.3.4 Команда snapshot

Команда *snapshot* предназначена для получения архива с полными исходными текстами проекта и всех его подпроектов. Такой архив может использоваться для предоставления возможности компиляции проекта на компьютере, на котором нет репозиториев. Например, в следующих случаях:

- при выполнении контрольной компиляции и фиксации запускаемой в эксплуатацию версии проекта у заказчика;
- при передаче полных исходных текстов заказчику при продаже проекта;
- передача проекта третьим лицам для ознакомления, изучения, тестирования, сертификации и т.п.

После выполнения команды snapshot в родительском каталоге проекта создается архив, имя которого содержит имя проекта, имя ветви и номер релиза, статус, дату и время создания "снимка". Например, если 4 мая 2004 года в 13:54 выдать команду snapshot для стабильной версии `oess_1.1.4`, то будет создан архив `oess_1-[1.4]-(stable)-200405041354.tar.bz2`.

Команду snapshot можно использовать для получения и сохранения "снимка" переданной заказчику версии продукта с тем, чтобы при возникновении каких-либо сбоев можно было без проблем занятся поиском причин сбоев именно в той версии, которая была передана заказчику. Однако, для этих же целей могут применяться и средства системы контроля версий. Например, выделение (tag-инг) ветвей или возможность извлечения конкретных версий файлов. Может быть, оба подхода ("снимки" и возможности систем контроля версий) следует применять одновременно.

4 Разрешение конфликтов и выбор из нескольких версий проектов

Конфликты возникают, если несколько подпроектов требуют использования несовместимых версий общего подпроекта. Перед рассмотрением различных вариантов конфликтов нужно показать, как в манифесте проекта указываются необходимые подпроекты.

Для указания необходимого проекта используется подтег `{requires}` тега `{module}`. Тег `{requires}` имеет два основных формата.

4.1 Тег {requires {module}}

Первый формат:

```
{requires {module "имя модуля"}}
```

применяется, когда требудется указать, что модуль проекта нуждается в другом модуле того же самого проекта. Например, модуль `file` проекта `oess` нуждается в модуле `io`, а модуль `io`, в свою очередь, нуждается в `defs`. В манифесте проекта это выражается следующим образом:

```
{project {name "oess"} ...
  {module "defs" {default} ... }
  {module "io"
    {requires {module "defs"}} ... }
  {module "file"
    {requires {module "io"}} ... }
}
```

Первый формат тега `{requires}` является самым простым, т.к. он не приводит к возникновению конфликтов, поскольку модули берутся из одной версии проекта.

4.2 Тег {requires {project}}

Второй формат:

```
{requires
  {project "имя проекта"
    [{module "имя модуля"}]*

    {branch "имя ветви"
      [{preferred}]
      [{>= release_min}]
      [{< release_max}]
    }*

    [{status имя статуса}]*
  }
}
```

В теге `{project}` указывается имя необходимого подпроекта.

В теге `{module}` указывается имя модуля из подпроекта. Если из подпроекта нужны несколько модулей, то для каждого должен быть указан свой тег `{module}`. Например:

```
{reqires
  {project "oess"
    {module "stdsn"}
    {module "util_cpp_serializer"}
    {module "tlv"}
  ...
}
```

Тег `{branch}` предназначен для указания ветви подпроекта, возможности которой необходимы. Если проект может работать с несколькими ветвями подпроекта, то для каждой ветви нужно указывать свой тег `{branch}`, но одна из ветвей должна быть помечена как предпочтительная при помощи тега `{preferred}`.

Это может понадобиться, например, в следующем случае. Есть несколько ветвей проекта `oess_1`, которые не совместимы полностью между собой. Пусть это будут ветви “0”, “1”, “2”. Есть некий проект, скажем `oess_sequence_1` на который несовместимости перечисленных ветвей проекта `oess_1` не сказываются. Т.е. проект `oess_sequence_1` может работать как с `oess_1.0.*`, так и с `oess_1.1.*` и `oess_1.2.*`. Чтобы отразить этот факт в манифесте `oess_sequence_1` следует указать:

```
{requires
  {project "oess"
    {module "db"}

    {branch "0"}
    {branch "1" {preferred} }
    {branch "2"}
  }
}
```

Теги `{>=}` и `{<}` задают ограничения на номера релизов в конкретной версии проекта. Тег `{>=}` задает минимальный размер релиза, который удовлетворяет проект. Например, если класс `oess_1::io::ibinbuffer_t` появился только в `oess_1.1.2`, а проекту нужно использовать этот класс, то в теге `{>=}` нужно указать значение 2. Например:

```
{requires
  {project "oess"
    {module "io"}
    {branch "1" {>= 2}}
  }
}
```

Если тег `{>=}` не указан, то в качестве минимального номера релиза назначается 0.

Тег `{<}` задает номер релиза, начиная с которого подпроект не удовлетворяет использующий его проект. Необходимость этого тега менее очевидна, но на практике могут возникать потребности в нем. Например:

- при возникновении неожиданных проблем при использовании новых релизов подпроекта. Например, если выясняется, что проект `oess_sequence_1` перестает устойчиво работать с релизом 6 проекта `oess_1.1`, то до выяснения причины проблемы следует указать в манифесте `oess_sequence_1`, что он не должен работать с релизом 6 проекта `oess_1.1`;

- при возникновении не технических препятствий к использованию новых релизов. Например, если в новом релизе проекта `oess_1.1` использована стороняя библиотека, распространяемая по лицензии GPL[12]. Это означает, что и `oess_1.1`, и любой использующий его проект так же должен распространяться по лицензии GPL. Что может оказаться не приемлимым. В этом случае, если проекту не требуется функциональность нового релиза `oess_1.1` нужно запретить использование `oess_1`, начиная с нового релиза[13].

[12] www.gnu.org

[13] Гораздо меньше проблем эта ситуация вызвала бы, если бы для использования стороней GPL библиотеки была выпущена новая ветвь проекта `oess_1`. Но в жизни очевидные решения часто становятся таковыми слишком позно. Например, мне это решение пришло в голову уже после написания абзаца.

Если теги `{>=}` и `{<}`, то они формируют ограничения вида $[release_{min}, release_{max})$. Если задан только тег `{>=}`, то он формирует ограничение вида $[release_{min}, \infty)$. Если задан только тег `{<}`, то он формирует ограничение вида $[0, release_{max})$.

Тег `{status}` задает необходимый статус подпроекта. По-умолчанию, если тег `{status}` не задействован, в репозитории ищутся только проекты со статусом stable (см. 3.3.3). Если же задан тег `{status}` то в репозитории ищется версия подпроекта с указанным статусом. Возможные значения: `draft`, `alpha`, `beta` и `stable`.

4.3 Примеры конфликтов и выбора из версий

4.3.1 Автоматический выбор более нового релиза

Пусть в манифесте проекта `mbapi_srv_3.1.4` указано:

```
{requires {project "so_4" {branch "2" {>= 6}}}}
```

В манифесте проекта `smsg_2.7.0` указано:

```
{requires {project "so_4" {branch "2" {>= 7}}}}
```

В манифесте проекта `aggregate_2.2.0` указано:

```
{requires {project "mbapi_srv_3" {branch "1" {>= 4}}}}
{requires {project "smsg_2" {branch "7"}}}
```

При выполнении команды populate для проекта `aggregate_2.2.0` будет определено, что из-за требований подпроектов `mbapi_srv_3` и `smsg_2.7.0` необходимо использовать ветвь “2” подпроекта `so_4`. При этом, необходимо использовать стабильный релиз не ниже номера 7.

Если при выполнении populate обнаружится, что в репозитории подходящей версии `so_4` нет, то команда populate завершиться неудачно, а в проект `aggregate_2.2.0` никаких исходных текстов подпроектов помещено не будет.

4.3.2 Органичение выбора версии подпроекта

Пусть существует два релиза проекта `mbapi_srv_3`: `mbapi_srv_3.1.4` и `mbapi_srv_3.1.6`. Причем, `mbapi_srv_3.1.4` требует подпроект `so_4.2.7`, а `mbapi_srv_3.1.6` — `so_4.3.0`.

Проект `smsg_2.7.0` требует подпроект `so_4.2.7`.

Если же в манифесте `aggregate_2.2.0` указано:

```
{requires {project "mbapi_srv_3" {branch "1" {>= 4}}}}
{requires {project "smsg_2" {branch "7"}}}
```

то произойдет конфликт, который не имеет автоматического разрешения. Дело в том, что в репозитории будет найден более новый релиз `mbapi_srv_3.1.6` и этот релиз будет взят для импорта исходных текстов. Но необходим `so_4.3.0`, который несовместим с необходимым для `smsg_2.7.0` проектом `so_4.2.7`.

Причем нельзя разрешить этот конфликт попыткой указать версию `so_4` на уровне проекта `aggregate_2.2.0`:

- если попытаться указать версию `so_4.2.7`, то ее функциональных возможностей не будет хватать для `mbapi_srv_3.1.6` (не зря же в `mbapi_srv_3` было ограничение на версию `so_4`);
- если попытаться указать версию `so_4.3.0`, то неработоспособным может оказаться `smsg_2.7.0` — будь он адаптирован или протестирован под `so_4.3.0`, то в манифесте `smsg_2.7.0` для проекта `so_4` перечислялись бы ветви "2" и "3".

Решение состоит в том, чтобы в `aggregate_2.2.0` запретить использование релизов `mbapi_srv_3.1` начиная с номера 6:

```
{requires {project "mbapi_srv_3" {branch "1" {>= 4} {< 6}}}}
{requires {project "smsg_2" {branch "7"}}}
```

Почему нельзя было бы в манифесте `aggregate_2.2.0` сделать для `mbapi_srv_3` еще более точные ограничения (например, указав `{>= 5}`, чтобы использовалась только версия 3.1.5)? Потому, что это может вызвать проблемы, если `aggregate_2.2.0` войдет как подпроект в другой проект, где будет выставлено требование:

```
{requires {project "mbapi_srv_3" {branch "1" {>= 4} {< 5}}}}
```

Поэтому, в качестве значения для `{>=}` нужно указывать номер **минимально необходимого** релиза.

4.3.3 Разрешение конфликта с помощью тега {brunch}

Пусть проект `mbapi_srv_3.1.4` требует `oess_1.1.4`, а проект `smsg_2.7.0` — `oess_1.2.0`. При попытке совместного использования проектов `mbapi_srv_3.1.4` и `smsg_2.7.0` возникает не имеющий автоматического разрешения конфликт версий общего подпроекта `oess_1`.

Но, часто бывает так, что использованная в `mbapi_srv_3.1.4` функциональность `oess_1` совершенно не изменилась в `oess_1.2.0`. Поэтому проект `mbapi_srv_3.1.4` спокойно может работать и с `oess_1.2.0`. Если данный факт отразить в манифесте проекта `mbapi_srv_3.1.4`, то конфликт будет разрешен автоматически.

Для того, чтобы показать, что проект `mbapi_srv_3.1.4` способен работать с несколькими ветвями проекта `oess_1` в манифесте `mbapi_srv_3.1.4` следует указать:

```
{requires
  {project "oess_1"
    {brunch "1" {>= 4} {preferred}}
    {brunch "2"}
  }
}
```

Однако, к подобным перечислениям следует относиться с осторожностью. Как показано в манифесте, для `mbapi_srv_3.1.4` предпочтительной ветвью является ветвь "1" проекта `oess_1`. Этот должно означать, как минимум, что разработчики `mbapi_srv_3.1.4` отслеживают новые релизы `oess_1.1` и проверяют работоспособность своего проекта. Более того, разработчики ветви "1" проекта `oess_1` должны гарантировать, что все последующие релизы этой ветви будут на 100% совместимы с предыдущими.

Но разработчики ветви “2” проекта `oess_1` таким ограничением не связаны, т.к. они не обязаны гарантировать совместимость с ветвью “1”. Поэтому, при выпуске нового релиза ветви “2” проект `mbapi_srv_3.1.4` окажется не работоспособным. Для того, чтобы это не стало неприятным сюрпризом для пользователей проекта `mbapi_srv_3.1.4` в манифесте для “непредпочитаемых” ветвей следует устанавливать органичения “сверху”:

```
{requires
  {project "oess_1"
    {brunch "1" {>= 4} {preferred}}
    {brunch "2" {< 3} }
  }
}
```

Такое ограничение позволяет гарантировать, что проект `mbapi_srv_3.1.4` не будет задействован с непротестированными релизами “непредпочитаемых” ветвей подпроектов.

4.3.4 Разрешение конфликта с помощью тега {substitutes}

Пусть проект `mbapi_srv_3.1.4` требует `oess_1.1.4`, а проект `smsg_2.7.0` — `oess_1.2.0`. При попытке совместного использования проектов `mbapi_srv_3.1.4` и `smsg_2.7.0` возникает не имеющий автоматического разрешения конфликт версий общего подпроекта `oess_1`.

Если ветвь “2” проекта `oess_1`, не смотря на свое название, все же гарантирует 100% совместимость с ветвью “1”, то для разрешения подобных конфликтов в манифесте `oess_1.2` нужно указать факт совместимости с помощью тега `{substitutes}`:

```
{project
  {name "oess_1"}
  {branch "2"
    {substitutes "1" {< 5} } }
  {release 0}
  ...
}
```

В этом случае при совместном использовании `mbapi_srv_3.1.4` и `smsg_2.7.0` будет определено, что для `smsg_2.7.0` можно использовать `oess_1.2.0`. Поэтому из репозитория будет взят код `oess_1.2.0`.

Утверждение о том, что одна ветвь полностью заменяет другую ветвь, является относительным. Оно может быть корректным только для определенных релизов определенных ветвей. Для объяснения этого следует рассмотреть, каким образом происходит параллельное развитие ветвей проекта.

Пусть ветвь “2” была выделена из 4-го релиза ветви “1” для проведения рефакторинга кода и перехода на новые ветви подпроектов. Пусть на момент выхода релиза 0 ветви “2” не было новых релизов ветви “1”. Это означает, что на данный момент (ветвь “2”, релиз 0) действительно служит заменой (ветвь “1”, релиз 4). Но это не означает, что сразу все использующие данный проект проекты тут же перейдут на новую ветвь. Более того, это может оказаться физически не возможно. Поэтому, в течении какого-то времени обе ветви должны будут развиваться параллельно. А это приведет к необходимости синхронизации выпуска новых релизов обеих ветвей.

Если в новом релизе ветви “1” появляются новые функциональные возможности, то желательно, чтобы синхронно появился новый релиз ветви “2” с этими же возможностями. Но, на практике, такая синхронизация может быть не возможна. Особенно, если еще несколько ветвей декларируют свою совместимость с ветвью “1”. Поэтому, будет существовать некоторый интервал времени, в течении которого (ветвь “2”, релиз 0) **не является** заменой для ветви “2”, т.к. не поддерживает релиз 5. Если данный факт не будет отражен в манифесте проекта для ветви “2”, то это может привести к множеству проблем. Поэтому, в теге `{substitutes}` необходимо указывать, для каких конкретно релизов гарантируется совместимость.

Для этих целей в теге `{substitutes}` должен использоваться подчиненный тег `{<}`. В нем указывается минимальный номер релиза, с которым совместимость еще не обеспечена. Т.е., при выделении ветви “2” из ветви “1” в теге `{<}` нужно указать значение, на 1 большее номера релиза ветви “1” на момент выделения ветви “2”. Например, если ветвь “2” создается из (ветвь “1”, релиз 4), то в `{<}` нужно указать значение 5.

5 Манифест проекта

Файл `manifest` дожен располагаться в родительском каталоге проекта и иметь следующий формат:

```
{project
  {name "имя проекта"}
  {branch "имя ветви"
    [{substitutes "имя ветви" {< номер релиза}}]*
  }
  {release номер релиза}

  {module ...}*
}
```

Тег `{name}` задает имя проекта.

Тег `{branch}` задает имя ветви проекта и описывает совместимость с другими ветвями (см. 4.3.4).

Тег `{release}` задает номер релиза проекта.

Тег `{module}` описывает один из модулей проекта. Формат данного тега описан в 5.1.

5.1 Описание модуля проекта в манифесте

Описание каждого модуля проекта задается тегом `{module}` следующего формата:

```
{module "имя модуля" [{default}]
  {paths ...}
  [{files ...}]

  [{templates ...}]
  [{masks ...}]

  [{requires ...}]*
}
```

- тег `{default}` указывает, входит ли модуль в число модулей по умолчанию (см. 5.1.1);
- тег `{paths}` перечисляет имена каталогов, содержимое которых должно быть включено/исключено в/из состава модуля (см. 5.1.4);
- тег `{files}` перечисляет имена отдельных файлов, которые должны быть включены/исключены в/из состава модуля (см. 5.1.2);
- тег `{templates}` указывает, какие из предопределенных шаблонов масок файлов следует использовать (см. 5.1.5);
- тег `{masks}` перечисляет маски имен файлов, которые нужно использовать для поиска файлов в каталогах, перечисленных в теге `{paths}` (см. 5.1.3);
- тег `{requires}` перечисляет подпроекты, от которых зависит данный модуль (см. 4.1 и 4.2);

5.1.1 Тег {default}

Если в каком-то проекте, в теге `{requires}` указано только имя проекта, но не указаны имена модулей, например:

```
{requires {project "oess_1" {brunch "1"}}}
```

то при выполнении команды populate из проекта `oess_1` будут взяты только модули, помеченные как *default*.

5.1.2 Тег {files}

Тег `{files}` предназначен для включения в состав модуля файлов, которые не удовлетворяют маскам поиска файлов или находятся в каталогах, не указанных в теге `{paths}`. Так же тег `{files}` позволяет исключить из состава проекта отдельные файлы.

Для включения файла в состав модуля необходимо указать его имя в подтеге `{+}` тега `{files}`. Для исключения файла из состава модуля необходимо указать его имя в подтеге `{-}`. Например:

```
{files
  {+ "doxygen/Doxyfile"}
  {- "doxygen/Doxyfile.dox_only"}
}
```

5.1.3 Тег {masks}

Тег `{masks}` предназначен перечисления масок имен файлов, которые будут использоваться для поиска файлов в каталогах, которые перечисляются в теге `{paths}`.

С помощью подтега `{+}` задаются маски файлов, который должны быть включены в результат поиска. С помощью подтега `{-}` задаются маски имен файлов, которые должны быть исключены из результатов поиска. Задаются регулярными выражениями. Например, для указания масок файлов *.[cho]pp, *.c, *.h и исключения *.ddl.cpp:

```
{masks
  {+ ".+\\.[cho]pp"}
  {+ ".+\\.c" }
  {+ ".+\\.h" }
  {- ".+\\.ddl\\.cpp" }
}
```

5.1.4 Тег {paths}

Тег `{paths}` предназначен для перечисления имен каталогов, в которых нужно осуществлять поиск файлов (согласно масок из `{masks}`) для включения в состав модуля при выполнении операции publish.

С помощью подтега `{+}` задаются имена каталогов, в которых необходимо осуществлять поиск. С помощью подтега `{-}` задаются имена каталогов, которые нужно исключить из поиска. Например:

```
{paths
  {+ "dev/oess_1/defs"}
  {+ "dev/guide/oess_1" }
  {- "dev/guide/oess_1/old_version" }
  {- "dev/oess_1/defs/old_regression_tests" }
}
```

При поиске файлов из списка каталогов автоматически исключаются каталоги, перечисленные в теге `{paths}` других модулей. Например, если указано:

```
{module "defs"
  {paths
    {+ "dev/oess_1"}
    {+ "dev/guide/oess_1" }
    {- "dev/guide/oess_1/old_version" }
    {- "dev/oess_1/defs/old_regression_tests" }
  }
}
{module "io"
  {paths
    {+ "dev/oess_1/io"}
    {+ "dev/guide/oess_1/io"}
  }
}
```

то при публикации в состав модуля `defs` будут включены файлы из каталогов `dev/oess_1`, `dev/guide/oess_1`, но исключены файлы из каталогов: `dev/oess_1/io`, `dev/oess_1/defs/old_regression_tests`, `dev/guide/oess_1/io` и `dev/guide/oess_1/old_version`.

5.1.5 Тег {templates}

Перечислять одни и те же маски файлов в тегах `{masks}` для разных модулей и проектов непродуктивно и черевато ошибками. Гораздо удобнее заранее определить наборы

масок файлов для разных типов модулей и использовать их в описании модулей. Такие преопределенные наборы называются *шаблонами*. Каждый шаблон имеет собственное уникальное имя и доступен по этому имени утилите `prjman`.

Тег `{templates}` позволяет указывать, какие шаблоны должны использоваться для модуля. Имена шаблонов перечисляются в теге `{templates}` с помощью подтегов `{+}`:

```
{module "io"
  ...
  {templates {+ cpp} }
}
{module "stdsn"
  ...
  {templates {+ cpp} {+ objessty_ddl} }
}
{module "doxygen"
  {paths {+ "doxygen"}}
  {files {+ "doxygen/Doxyfile"}}
  {templates {+ doxygen}}
}
```

5.2 Пример файла манифеста для проекта smsg_2.7.0

```
{project
  {name "smsg_2"}
  {branch "7"}
  {release "0"}

  {module "_dirs_" {default}
    || Служит для организации правильной файловой структуры.
    {paths "dev/etc"
           "dev/lib"
           "dev/log"
           "dev/sop" }
    {templates {+ oppp} }
  }

  {module "smsg_2" {default}
    || Основная функциональность проекта.
    {paths "dev/smsg_2"
           "dev/howto/smsg_2"
           "dev/guide/smsg_2"
           "dev/etc/sample/smsg_2" }
    {templates {+ cpp}
               {+ objessty_ddl}
               {+ latex}
               {+ cfg} }
    {requires {module "_dirs_"} }
    {requires {project "cpp_util_2"}}
```

```
  {requires {project "auto_ptr_3"}}
  {requires {project "cls_2"}}
  {requires {project "smart_ref_3"}}
  {requires {project "memcheck_2"}}
  {requires {project "oess_1"
              {module "db"}}}
  {requires {project "so_4"}}
  {requires {project "so_log_1"}}
  {requires {project "so_sysconf_2"}}
  {requires {project "so_sysconf_log_1"}}
  {requires {project "libpcre++"}}
  {requires {project "cnv_util_1"
              {module "convert"}
              {module "translit"}}}
  {requires {project "gemont_1"
              {module "retranslator_sysconf"}}}
}

{module "smpp_smsc_2"
  || Модуль работы с SMPP SMS-центрами.
  {paths "dev/smsg_2/smpp_smsc_2"
         "dev/howto/smsg_2/smpp_smsc_2"
         "dev/guide/smsg_2/smpp_smsc_2"
         "dev/etc/sample/smsg_2/smpp_smsc_2" }
  {templates {+ cpp}
             {+ objessty_ddl}
             {+ latex}
             {+ cfg} }
  {requires {module "smsg_2"} }
  {requires
    {project "smpp_pdu_1"
      {branch "2"} } }
}

{module "subsys_smpp_smsc_2"
  || Модуль для создания кооперацийдля работы с
  || SMPP SMS-центрами средствами so_sysconf_2.
  {paths "dev/smsg_2/subsys/smpp_smsc_2" }
  {templates {+ cpp}
             {+ objessty_ddl}
             {+ latex}
             {+ cfg} }
  {requires {module "smpp_smsc_2"} }
  {requires {project "so_sysconf_2"} }
}

{module "dox"
  || Модуль для формирования doxygen-документации.
```

```
      {paths "doxygen"}
      {files "doxygen/Doxyfile"}
      {templates {+ dox}}
    }
  }
```