КОНФИГУРАЦИОННОЕ УПРАВЛЕНИЕ

П.Н. Советов

Москва — 2021

Оглавление

1. Введение

1.1. О чем этот курс

“Программная инженерия — это то, что происходит с программированием при добавлении времени и других программистов” (Russ Cox).

Отслеживание и управление изменениями в ПО — важная задача, которая возникает даже при работе над небольшими индивидуальными проектами. В этом курсе предлагается практико-ориентированный взгляд на конфигурационное управление. Изучаемые темы:

- менеджеры пакетов,
- конфигурационные языки,
- системы автоматизации сборки,
- системы контроля версий,
- контейнеризация приложений,
- непрерывная интеграция,
- системы управления проектами и задачами,
- генераторы документации.

Изначально управление конфигурацией применялось не в программировании. Под конфигурацией понимался состав деталей конечного продукта и «взаимное расположение частей» физического изделия. Таким образом, конфигурацией можно управлять, контролируя документы, описывающие конечный продукт, требования к нему, всю его проектную и технологическую документацию.

В связи с высокой динамичностью сферы разработки ПО, в ней конфигурационное управление особенно полезно. К процедурам можно отнести:

- управление версиями проекта,
- управление сборками проекта,
- контроль требований проекта,
- контроль документации
- и т.д.

Степень формальности выполнения данных процедур зависит от размеров проекта, и при правильном подходе данная концепция может быть очень полезна.

1.2. Основные определения

Конфигурационное управление — дисциплина идентификации компонентов системы, определения функциональных и физических характеристик аппаратного и программного обеспечения для проведения контроля внесения изменений и отслеживания конфигурации на протяжении ЖЦ, см. рис. 1. Это управление соответствует одному из вспомогательных процессов ЖЦ (ISO/IEC 12207), выполняется техническим и административным руководством проекта и заключается в контроле указанных характеристик конфигурации системы и их изменении; составления отчета о внесенных изменениях в конфигурацию и статус их реализации; проверки соответствия внесенных изменений заданным требованиям.

Рис. 1. Цели и задачи конфигурационного управления

Конфигурация системы — состав функций, программных и физических характеристик программ или их комбинаций, аппаратного обеспечения, обозначенные в технической документации системы и реализованные в продукте.

Элемент программной конфигурации — фрагмент программного обеспечения, вовлеченный в процесс конфигурационного управления и рассматриваемый как одна (атомарная) сущность.

К элементам конфигурации относятся такие объекты, как:

- рабочие документы;
- файлы исходных кодов;
- файлы ресурсов;
- файлы, создаваемые в результате сборки (исполняемые файлы, библиотеки и так далее);
- инструменты, используемые для разработки (их мы тоже должны учитывать для стандартизации и упрощения взаимодействия в команде);

Версия — это состояние элемента программной конфигурации, которое может быть восстановлено в любой момент времени независимо от истории изменения.

Элементы системы, их версии и конфигурации показаны на рис. 2.

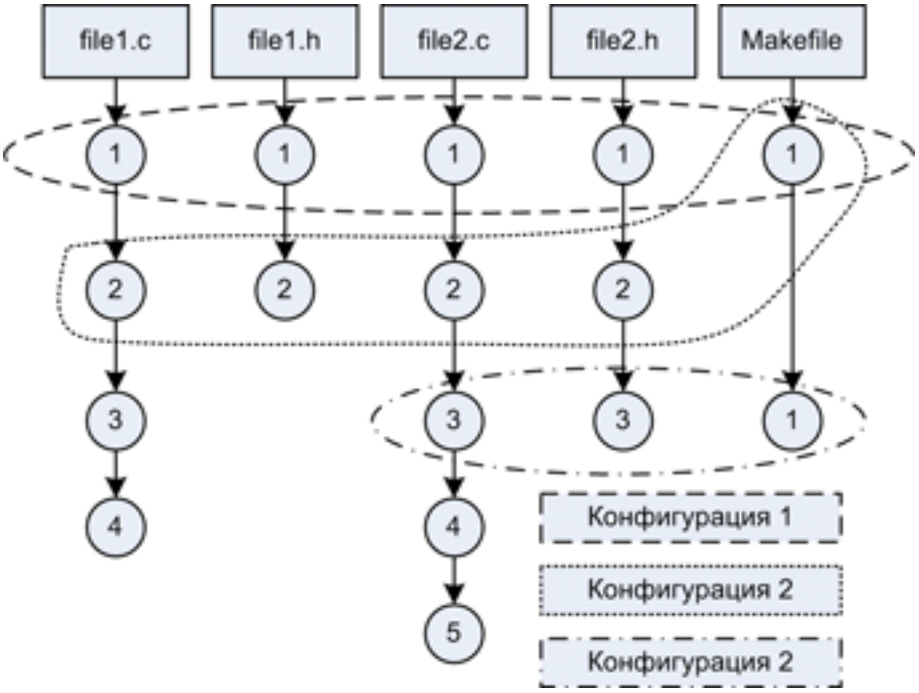

Рис. 2. Элементы, их версии и срезы-конфигурации

Конфигурация ПО включает набор функциональных и физических характеристик ПО, заданных в технической документации и достигнутых в готовом продукте. Т.е это сочетание разных элементов продукта вместе с заданными процедурами сборки и отвечающие определенному назначению. Элемент конфигурации — график разработки, проектная документация, исходный и исполняемый код, библиотека компонентов, инструкции по установке системы и др.

Область знаний «Управление конфигурацией ПО» состоит из следующих разделов:

- управление процессом конфигурацией (Management of SMC Process),
- идентификация конфигурации ПО (Software Configuration Identification),
- контроль конфигурации ПО (Software Configuration Control),
- учет статуса конфигурации ПО (Software Configuration Status Accounting),
- аудит конфигурации ПО (Software Configuration Auditing),
- управление релизами (версиями) ПО и доставкой (Software Release Management and Delivery).

Управление процессом конфигурации — это деятельность по контролю эволюции и целостности продукта при идентификации, контроле изменений и обеспечении отчетности информации, касающейся конфигурации. Включает:

- систематическое отслеживание вносимых изменений в отдельные составные части конфигурации и проведение аудита изменений и автоматизированного контроля за внесением изменений в конфигурацию системы или ПО;
- поддержка целостности конфигурации, ее аудит и обеспечение внесения изменений в один объект конфигурации, а также в связанный с ним другой объект;
- ревизия конфигурации на предмет проверки разработки необходимых программных или аппаратных элементов и согласованности версии конфигурации с требованиями;
- трассировка изменений в конфигурацию на этапах сопровождения и эксплуатации ПО.

Идентификация конфигурации ПО проводится путем выбора элемента конфигурации ПО и документирования его функциональных и физических характеристик, а также оформления технической документация на элементы конфигурации ПО.

Контроль конфигурации ПО состоит в проведении работ по координации, утверждению или отбрасыванию реализованных изменений в элементы конфигурации после формальной ее идентификации, а также оценке результатов.

Учет статуса конфигурации ПО проводится в виде комплекса мероприятий для определения уровня изменений в конфигурацию, аудита конфигурации в виде комплекса мероприятий по проверке правильности внесения изменений в конфигурацию ПО. Информация и количественные показатели накапливается в соответствующей БД и используются при управлении конфигурацией, составлении отчетности, оценке качества и выполнении других процессов ЖЦ.

Аудит конфигурации — это деятельность, которая выполняется для оценки продукта и процессов на соответствие стандартам, инструкциям, планам и процедурам. Аудит определяет степень удовлетворения элемента конфигурации заданным функциональным и физическим характеристикам системы. Различают функциональный и физический аудит конфигурации, который завершается фиксацией базовой линии (базиса) продукта. Сборка ПО — объединение корректных элементов ПО и конфигурационных данных в единую исполняемую программу.

Процедуру сборки проекта часто автоматизируют, то есть выполняют не из среды разработки, а из специального скрипта — **build-скрипта**. Этот скрипт используется тогда, когда разработчику требуется полная сборка всего проекта. А также он используется в

процедуре непрерывной интеграции (continuous integration) — то есть регулярной сборке всего проекта (как правило — каждую ночь). Во многих случаях процедура непрерывной интеграции включает в себя и регрессионное тестирование, и часто — создание инсталяционных пакетов. Общая схема автоматизированной сборки представлена на рис. 3.

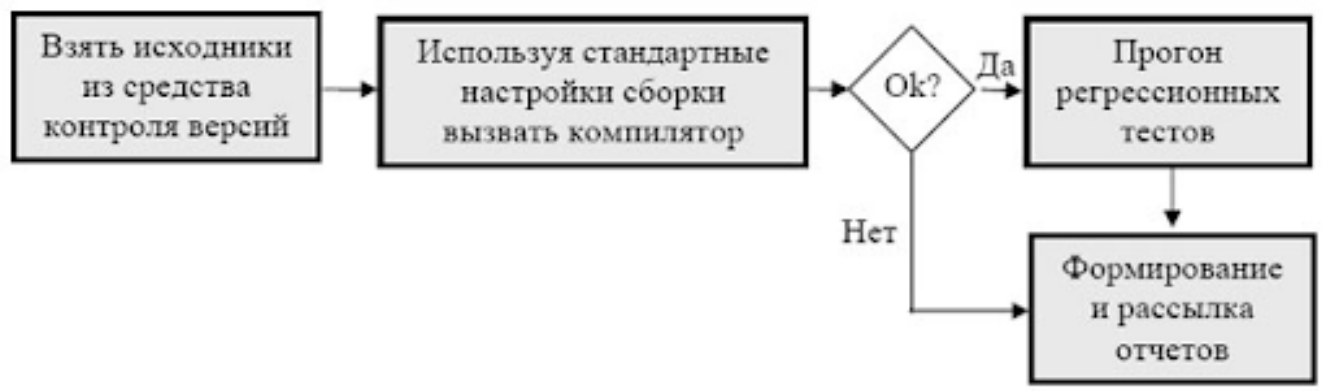

Рис. 3. Схема автоматизированной сборки ПО

Стабилизация конфигурации — это процесс получения новой конфигурации из имеющихся промежуточных конфигураций. Для этого процесса также используются также термины «выпуск» или «релиз». Результат стабилизации также может быть назван, в свою очередь, релизом или выпуском.

Управление релизами (версиями) ПО это: отслеживание имеющейся версии элемента конфигурации; сборка компонентов; создание новых версий системы на основе существующей путем внесения изменений в конфигурацию; согласование версии продукта с требованиями и проведенными изменениями на этапах ЖЦ; обеспечение оперативного доступа к информации относительно элементов конфигурации и системы, к которым они относятся. Управление выпуском охватывает идентификацию, упаковку и передачу элементов продукта и документации заказчику.

Базис (baseline) — формально обозначенный набор элементов ПО, зафиксированный на этапах ЖЦ ПО. Это конфигурация, выбранная и закрепленная на любом этапе жизненного цикла разработки как основа для дальнейшей работы.

На рис. 4 показан пример появления конфигураций во времени.

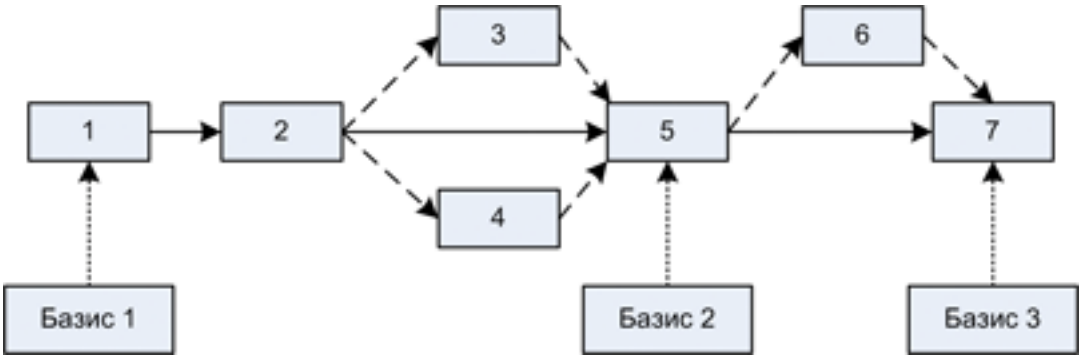

Рис. 4. Пример появления конфигураций во времени

Начальное состояние проекта — конфигурация 1. Она же является первым базисом, от которого будет идти дальнейшая разработка. Предположим, проект на начальной стадии. Через какое-то время появляется обновленная конфигурация 2. Разработка только началась и был выпущен релиз, чтобы дать команде какую-то основу для дальнейшей работы. В ходе проверки выяснилось, что базой для работы этот выпуск служить не может — есть непонятные и противоречивые места.

Для их устранения группы разработки делают доработки. В результате них появляются конфигурации 3 и 4 — обе они разработаны на основе 2, но друг с другом они не согласуются, поскольку не включают изменения друг от друга. СМ-инженер создает конфигурацию 5, сделанную на основе 2, 3 и 4. После проверки менеджмент объявляет конфигурацию базовой. По этому сигналу СМ-команда выпускает этот релиз как официальную базовую конфигурацию и разработчики берут уже ее за основу. Далее история повторяется, группа разработки вносит изменения — появляется конфигурация 5. Ее, в свою очередь, интегрирует СМ-инженер и она получает номер 7. Он также становится официальной базой для разработки.

2. Работа в командной строке

Текст.

3. Менеджеры пакетов

3.1. Нумерация версий ПО

Программная библиотека — контролируемая коллекция программных приложений и связанной с ними документации, предназначенная для использования в процессе разработки, эксплуатации и сопровождения программного обеспечения.

Библиотеки, модули, пакеты, расширения — элементы программных приложений, которые предназначены для модульной организации программы и повторного использования кода.

Существуют различные схемы нумерации версий:

- последовательные значения,
- по дате выпуска,
- различные экзотические схемы (TeX и проч.).

Указание стадии разработки в версии:

- начальная стадия разработки — альфа (например, 1.2.0-alpha.1),
- Стадия тестирования и отладки — бета (например, 1.2.0-beta),
- Выпуск-кандидат (например, 1.2.0-rc.3),
- Публичный выпуск (например, 1.2.0),
- Исправления после выпуска (1.2.5).

Ад зависимостей — проблема управления программными пакетами (библиотеками, модулями), которые зависят от определенных версий других программных пакетов.

В сложных случаях различные установленные программные продукты требуют наличия разных версий одной и той же библиотеки. В наиболее сложных случаях один продукт может косвенно требовать сразу две версии одной и той же библиотеки. Проблемы с зависимостями возникают у общих пакетов/библиотек, у которых некоторые другие пакеты имеют зависимости от несовместимых и различных версий общих пакетов.

Без формальной спецификации номера версий практически бесполезны для управления зависимостями. Ясно определив и сформулировав идею версионирования, становится легче сообщать о намерениях пользователям ПО.

В настоящее время популярность приобрела схема семантической нумерации версий рис. 5.

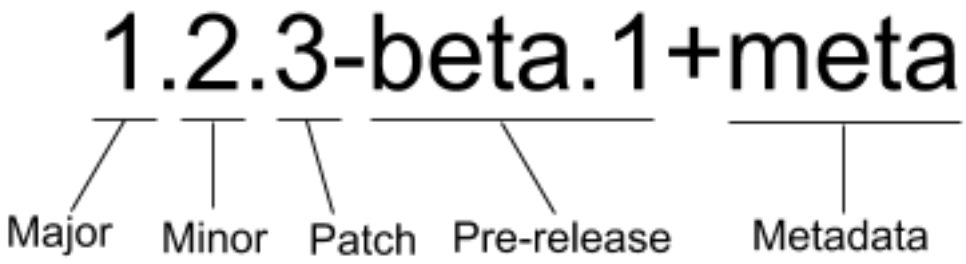

Рис. 5. Семантическая нумерация версий

Учитывая номер версии МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ, следует увеличивать:

1. МАЖОРНУЮ версию, когда сделаны обратно несовместимые изменения API.
2. МИНОРНУЮ версию, когда вы добавляете новую функциональность, не нарушая обратной совместимости.
3. ПАТЧ-версию, когда вы делаете обратно совместимые исправления.

Дополнительные обозначения для предрелизных и билд-метаданных возможны как дополнения к МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ формату.

Обычный номер версии ДОЛЖЕН иметь формат X.Y.Z, где X, Y и Z — неотрицательные целые числа и НЕ ДОЛЖНЫ начинаться с нуля. X — мажорная версия, Y — минорная версия и Z — патч-версия. Каждый элемент ДОЛЖЕН увеличиваться численно. Например: 1.9.0 -> 1.10.0 -> 1.11.0.

После релиза новой версии пакета содержание этой версии НЕ ДОЛЖНО быть модифицировано. Любые изменения ДОЛЖНЫ быть выпущены как новая версия.

Мажорная версия X (X.y.z | X > 0) ДОЛЖНА быть увеличена, если в публичном API представлены какие-либо обратно несовместимые изменения. Она МОЖЕТ включать в себя изменения, характерные для уровня минорных версий и патчей. Когда увеличивается мажорная версия, минорная и патч-версия ДОЛЖНЫ быть обнулены.

Мажорная версия ноль (0.y.z) предназначена для начальной разработки. Всё может измениться в любой момент. Публичный API не должен рассматриваться как стабильный.

Минорная версия (x.Y.z | x > 0) ДОЛЖНА быть увеличена, если в публичном API представлена новая обратно совместимая функциональность. Версия ДОЛЖНА быть увеличена, если какая-либо функциональность публичного API помечена как устаревшая (deprecated). Версия МОЖЕТ быть увеличена в случае реализации новой функциональности или существенного усовершенствования в приватном коде. Версия МОЖЕТ включать в себя изменения, характерные для патчей. Патч-версия ДОЛЖНА быть обнулена, когда увеличивается минорная версия.

Патч-версия Z (x.y.Z | x > 0) ДОЛЖНА быть увеличена только если содержит обратно совместимые баг-фиксы. Определение баг-фикс означает внутренние изменения, которые исправляют некорректное поведение.

Приоритет определяет, как версии соотносятся друг с другом, когда упорядочиваются. Приоритет версий ДОЛЖЕН рассчитываться путем разделения номеров версий на мажорную, минорную, патч и предрелизные идентификаторы. Приоритет определяется по первому отличию при сравнении каждого из этих идентификаторов слева направо: Мажорная, минорная и патч-версия всегда сравниваются численно. Пример: 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1. Когда мажорная, минорная и патч-версия равны, предрелизная версия имеет более низкий приоритет, чем нормальная версия.

3.2. Управление пакетами

Менеджер пакетов — ПО для управления процессом установки, удаления, настройки и обновления различных компонентов программного обеспечения. Менеджеры пакетов бывают следующих видов:

- уровня ОС (например, RPM в Linux),
- уровня языка программирования (например, npm для JavaScript),
- уровня приложения (например, Package Control для Sublime Text, управление плагинами для Eclipse).

Пакет содержит в некотором сжатом файловом формате программный код и метаданные. К метаданным относятся:

- указание авторства, версии и описание пакета,
- список зависимостей пакета,
- хэш-значение содержимого пакета.
- Пример метаданных из пакета для npm (JavaScript):

```
{
  "name": "my_package",
  "version": "1.0.0",
  "dependencies": {
    "my_dep": "^1.0.0",
    "another_dep": "~2.2.0"
  }
}
```

Пример метаданных из пакета для NuGet (.NET):

```xml
<?xml version="1.0" encoding="utf-8"?>
<package
 ↪  xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
    <metadata>
        <id>sample</id>
        <version>1.0.0</version>
        <authors>Microsoft</authors>
        <dependencies>
            <dependency id="another-package" version="3.0.0" />
            <dependency id="yet-another-package" version="1.0.0" />
        </dependencies>
    </metadata>
</package>
```

Пример метаданных из пакета для Maven (Java):

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <type>jar</type>
      <scope>test</scope>
      <optional>true</optional>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

Пример проблемы с зависимостями пакетов — ромбовидная зависимость, см. рис. 6.

Пакет A нуждается в пакетах B и C. B нужна версия 1 пакета D, а не версия 2. C требует D версии 2, а не версии 1. В этом случае, если предположить, что невозможно выбрать обе версии D, нет возможности построить A.

В общем случае задача разрешения зависимостей между пакетами является NP-полной. Для ее решения сегодня применяются SAT-решатели.

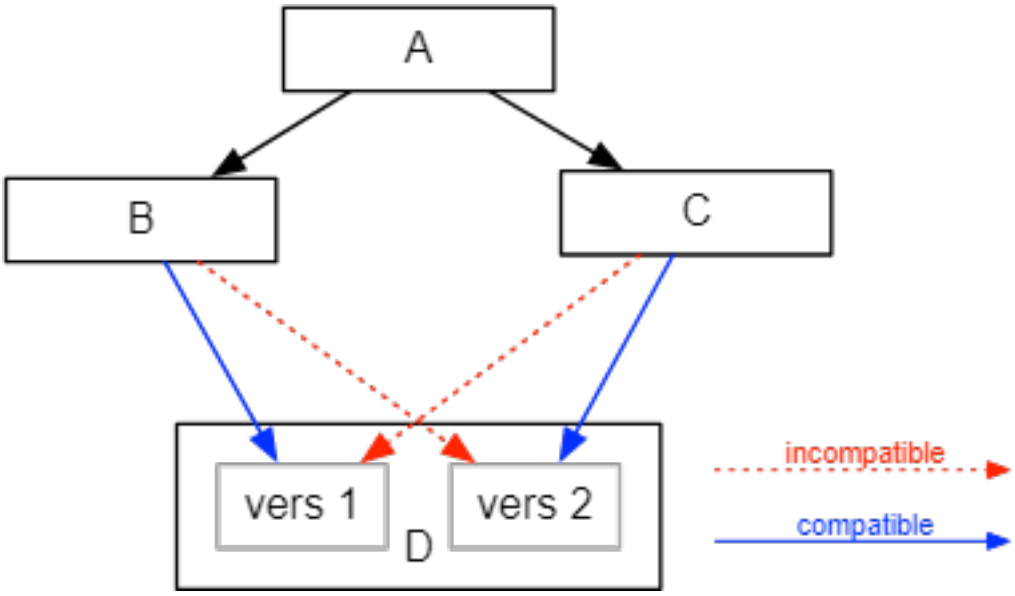

Рис. 6. Пример ромбовидной зависимости между пакетами

4. Конфигурационные языки

4.1. Виды конфигурационных языков

Параметры конфигурации ПО могут сохраняться в специальном виде, доступном для редактирования пользователями и другими программами. Для хранения может использоваться специальная базаа данных, но наиболее распространенным вариантом хранения настроек программы являются файлы конфигурации, содержимое которых написано на одном из конфигурационных языков.

В качестве конфигурационных языков могут использоваться:

- непрограммируемые форматы описания конфигурации,
- языки программирования общего назначения,
- программируемые DSL для задач конфигурирования.

Формальный язык — множество конечных слов над конечным алфавитом.

Языки программирования определяются с помощью синтаксиса и семантики:

- синтаксис — форма или структура выражений, предложений и программных единиц,
- семантика — значение выражений, предложений и программных единиц.

4.2. Представление синтаксиса языка с помощью БНФ

Форма Бэкуса-Наура (БНФ) — формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. В БНФ присутствуют:

- нетерминальные символы: абстракции BNF.

- терминальные символы: лексемы.
- грамматика: набор правил.

У правила есть левая и правая стороны; правило состоит из терминальных и нетерминальных символов. Пример БНФ-описания арифметического выражения:

```
<expr> ::= <term> "+" <expr>
         |  <term>

<term> ::= <factor> "*" <term>
         |  <factor>

<factor> ::= "(" <expr> ")"
         |  <const>

<const> ::= integer
```

4.3. Простые форматы описания конфигурации

Одним из старейших форматов являются *S-выражения* языка Lisp. Это нотация для древовидных структур, реализованных в виде вложенных списков. В Lisp S-выражения применяются и для данных, и для программ. S-выражение, заключенное в скобки, определяется индуктивно, как одна из альтернатив:

- атом,
- выражение вида (x . y), где x и y — S-выражения.

В БНФ:

```
<s-exp> ::= <atom>
          |  '(' <s-exp-list> ')'

  <s-exp-list> ::= <sexp> <s-exp-list>
         |

  <atom> ::= <symbol>
         |  <integer>
         |  #t  |  #f
```

Пример S-выражения:

```
(users
   ((uid 1)   (name root) (gid 1))
   ((uid 108) (name peter) (gid 108))
   ((uid 109) (name alex) (gid 109)))
```

К недостаткам S-выражений для описания конфигурации относятся:

- сложность чтения для человека синтаксиса с большим числом скобок,
- нет стандарта S-выражений для описания конфигурационных данных.

Классическим способом представления файлов конфигурации является формат INI из Windows, а также схожие с ним варианты conf из Unix. В основе формата — пара ключ-значение. Пары схожего назначения объединяются в секции:

```
[секция_1]
параметр1=значение1
параметр2=значение2

[секция_2]
параметр1=значение1
параметр2=значение2
```

В (расширенной) БНФ:

```
ini ::= {section}
section ::= "[" name "]" “\n” {entry}
entry ::= key "=" value “\n”
```

Здесь фигурные скобки обозначают повторение 0 или более раз.

Пример INI-файла (system.ini):

```
; for 16-bit app support
[386Enh]
woafont=dosapp.fon
EGA80WOA.FON=EGA80WOA.FON
EGA40WOA.FON=EGA40WOA.FON
CGA80WOA.FON=CGA80WOA.FON
CGA40WOA.FON=CGA40WOA.FON

[drivers]
wave=mmdrv.dll
timer=timer.drv

[mci]
[network]
Bios=1438818414
SSID=29519693
```

К недостаткам INI для описания конфигурации относятся: * отсутствие поддержки вложенных конструкций, * нет стандарта INI для описания конфигурационных данных.

Еще недавно наибольшую популярность среди форматов конфигурационных данных имел **XML** (eXtensible Markup Language).

XML разрабатывался как язык с простым формальным синтаксисом, удобный для создания и обработки документов программами и одновременно удобный для чтения и создания документов человеком, с подчеркиванием нацеленности на использование в Интернете.

В самом базовом виде XML напоминает S-выражения, в которых вместо скобок используются открывающийся и закрывающийся именованные теги.

Грамматика XML в БНФ здесь не приводится из-за ее большого объема.

Пример XML-данных:

```
<?xml version="1.0" encoding="UTF-8"?>
<shiporder orderid="889923"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="shiporder.xsd">
   <orderperson>John Smith</orderperson>
   <shipto>
      <name>Ola Nordmann</name>
      <address>Langgt 23</address>
      <city>4000 Stavanger</city>
      <country>Norway</country>
   </shipto>
   <item>
      <title>Hide your heart</title>
      <quantity>1</quantity>
      <price>9.90</price>
   </item>
</shiporder>
```

Важной особенностью XML является поддержка специальных схем для определения корректности XML-данных: DTD (Document Type Definition), XML Schema и другие. DTD определяет:

- состав элементов, которые могут использоваться в XML документе;
- описание моделей содержания, т.е. правил вхождения одних элементов в другие;
- состав атрибутов, с какими элементами XML документа они могут использоваться;
- каким образом атрибуты могут применяться в элементах;
- описание сущностей, включаемых в XML документ.

В XML Schema дополнительно введены типы данных для элементов. Фрагмент XML Schema описания элемента item из примера выше:

```
<xs:element name="item" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="note" type="xs:string" minOccurs="0"/>
      <xs:element name="quantity" type="xs:positiveInteger"/>
      <xs:element name="price" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

К недостаткам XML для описания конфигурации относится:

- высокая сложность формата, как для чтения человеком, так и для программного разбора.

Формат **JSON** (JavaScript Object Notation) основан на синтакисе JavaScript. Изначально он использовался в качестве текстового формата обмена данными, но со временем все чаще стал применяться и в качестве формата описания конфигурации приложения.

В JSON используются следующие типы данных:

- число в формате double,
- строка,
- логическое значение true или false,
- массив значений любого поддерживаемого типа,
- объект — словарь пар ключ-значение, где ключ — строка, а значение — любой поддерживаемый тип.
- специальное значение null.

Пример JSON-данных:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

Для проверки корректности JSON-данных в формат позже была добавлена схема — JSON Schema, которая во многом схожа с XML Schema. К недостаткам JSON для описания конфигурации относится:

- отсутствие поддержки комментариев.

Формат **YAML** (YAML Ain't Markup Language) предназначен для сериализации данных, но также часто используется в качестве конфигурационного языка. YAML с точки зрения синтаксиса имеет сходство с языком Python — для определения вложенности конструкций используются отступы. Особенностью YAML является возможность создания (с помощью &) и использования (с помощью *) ссылок на элементы документа. Это позволяет не приводить полностью повторно встречающиеся данные.

Пример YAML-файла:

```
---
receipt:     Oz-Ware Purchase Invoice
date:        2012-08-06
customer:
    first_name:   Dorothy
    family_name:  Gale

items:
```

```
    - part_no:   A4786
      descrip:   Water Bucket (Filled)
      price:     1.47
      quantity:  4

    - part_no:   E1628
      descrip:   High Heeled "Ruby" Slippers
      size:      8
      price:     133.7
      quantity:  1

bill-to:  &id001
    street: |
            123 Tornado Alley
            Suite 16
    city:   East Centerville
    state:  KS

ship-to:  *id001
...
```

К недостаткам YAML для описания конфигурации относятся:

- сложность редактирования сильно вложенных элементов, проблемы с отступом,
- сложность стандарта YAML (размер файла спецификации больше, чем у XML).

Формат **TOML** (Tom's Obvious, Minimal Language) специально предназначен для использования в конфигурационных файлах. Его синтаксис основан на INI. Спецификация TOML не имеет БНФ-описания.

Пример TOML-данных:

```
# This is a TOML document

title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
dob = 1979-05-27T07:32:00-08:00

[database]
enabled = true
ports = [ 8001, 8001, 8002 ]
data = [ ["delta", "phi"], [3.14] ]
temp_targets = { cpu = 79.5, case = 72.0 }

[servers]

  [servers.alpha]
```

```
ip = "10.0.0.1"
role = "frontend"

[servers.beta]
ip = "10.0.0.2"
role = "backend"
```

К недостаткам TOML для описания конфигурации относится:

- сложность спецификации языка.

4.4. Языки общего назначения

Использование языка программирования общего назначения для описания конфигурации ПО обладает рядом преимуществ по сравнению с простыми непрограммируемыми конфигурационными форматами:

- поддержка функций,
- поддержка циклов,
- поддержка типов данных,
- поддержка импорта файлов.

Благодаря поддержке функций, циклов и разбиению файлов на модули достигается принцип DRY (Don’t repeat yourself) — не допускать повторения одной и той же информации в программе. В результате конфигурационные файлы имеют компактную, удобную для редактирования форму.

Поддержка типов данных, в том числе пользовательских, позволяет осуществить на уровне типов проверку корректности программы — то, для чего в таких форматах, как XML, используются схемы.

Некоторые примеры использования языков общего назначения в качестве конфигурационных языков:

- В текстовом редакторе Emacs используется встроенный язык Elisp для описания конфигурации.
- В БД Tarantool используется Lua.
- Конфигурации на языке Python используются в setuptools и Jupyter.
- В Gradle конфигурация сборки может быть описана на Groovy и Kotlin.

К недостаткам языков общего назначения для задач конфигурирования относятся: * не безопасное исполнение стороннего кода, * сложность, чрезмерная выразительность языка (Тьюринг-полнота), мешающая задачам анализа и порождения конфигурационных данных.

4.5. Программируемые DSL для задач конфигурирования

Одним из наиболее интересных программируемых конфигурационных языков является Dhall. Этот язык со статической типизацией, основанный на принципах функционального программирования. Авторы определяют Dhall, как JSON + функции + типы + импорты.

Важной особенностью Dhall является тот факт, что это не Тьюринг-полный язык. Хотя в нем и есть циклы, их бесконечное исполнение не поддерживается. В Dhall имеется режим нормализация представления, при котором все языковые абстракции раскрываются и результат представляет собой уровень представления JSON или YAML.

Пример описания данных на Dhall:

```
let makeUser = \(user : Text) ->
      let home       = "/home/${user}"
      let privateKey = "${home}/.ssh/id_ed25519"
      let publicKey  = "${privateKey}.pub"
      in  { home = home
          , privateKey = privateKey
          , publicKey = publicKey
          }
    {- Add another user to this list -}
in  [ makeUser "bill"
    , makeUser "jane"
    ]
```

Результат преобразования в JSON:

```
[
  {
    "home": "/home/bill",
    "privateKey": "/home/bill/.ssh/id_ed25519",
    "publicKey": "/home/bill/.ssh/id_ed25519.pub"
  },
  {
    "home": "/home/jane",
    "privateKey": "/home/jane/.ssh/id_ed25519",
    "publicKey": "/home/jane/.ssh/id_ed25519.pub"
  }
]
```

Другими примерами программируемых конфигурационных языков являются: Nix, CUE (типы с ограничениями) и Jsonnet.

5. Системы автоматизации сборки

Автоматизация сборки — автоматизация создания программного обеспечения и связанные с этим процессы, включая компиляцию исходного кода, упаковку кода в дистрибутив и выполнение автоматических тестов.

Системы сборки автоматизируют выполнение повторяемых задач как на уровне отдельных пользователей, так и для крупных организаций.

Система сборки является **минимальной**, если она выполняет каждую задачу не более одного раза в процессе сборки, при этом задачи выполняются только в том случае, ес-

ли они прямо или косвенно зависят от входных данных, которые изменились с момента предыдущей сборки.

Важнейшим элементом системы сборки является **граф зависимостей задач**, см. рис. 7.

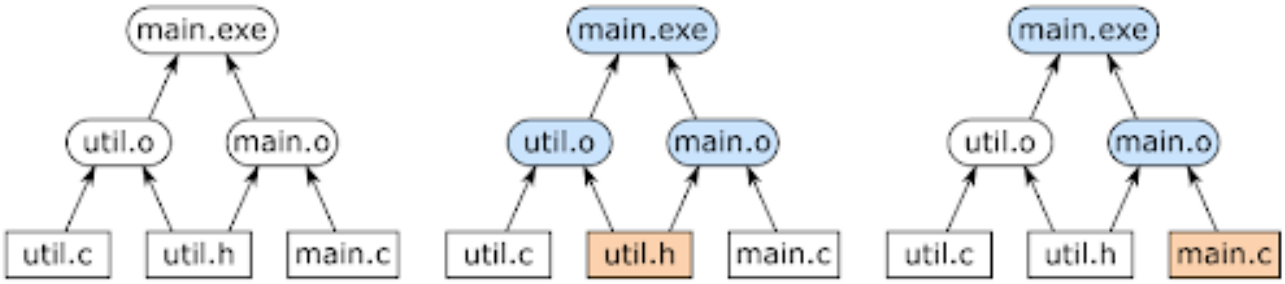

Рис. 7. Граф зависимостей задач и два варианта сборки: а) полное перестроение, б) частичное перестроение.

Система сборки включает в себя следующие элементы:

- Хранилище пар ключ-значение. Цель системы сборки привести хранилище в актуальное состояние. Во многих случаях хранилищем является файловая система, где ключами являются имена файлов, а значениями — содержимое файлов. Многие системы сборки используют хеш-суммы от значений для компактного описания данных и быстрой проверки данных на равенство.
- Входные, выходные и промежуточные значения. Входные значения могут предоставляться пользователем или зависеть от выполнения других задач. Выходные значения формируются в результате выполнения задачи. Промежуточные значения формируются в процессе выполнения задачи.
- Постоянная информация о сборке. Эта информация сохраняется от между сборками и представляет собой “память” системы сборки.
- Описание задачи. Пользователь предоставляет описание того, каким образом выходное значение по заданному ключу получается на основе входных зависимостей.

Система сборки принимает описание задачи, целевой ключ и хранилище и возвращает измененное хранилище, в котором целевой ключ и всего его зависимости принимают актуальные значения.

Постоянная информация о сборке может храниться в одном из следующих вариантов:

- Время модификации файлов в файловой системе. Если время модификации одного из файлов-зависимостей задачи более новое, чем время модификации файла-результата самой задачи — необходимо перестроить задачу.
- Специальный флаг для каждого ключа хранилища, сигнализирующий о том, что данные были обновлены. После сборки все флаги сбрасываются. Когда начинается новая сборка для всех ключей, значения которых изменились после прошлой сборки, устанавливается флаг. Если для ключа и всех его прямых и косвенных зависимостей флаг сброшен, то ключ перестраивать не требуется.
- Граф зависимостей, в котором хранятся хеш-значения файлов-зависимостей.
- Кэш-хранилище с адресацией по хеш-значению файла, а также история всех предыдущих сборок с указанием хеш-значений файлов для каждой задачи. Этот вариант

используется для облачного (распределенной) режима работы с системой сборки силами коллектива разработчиков.

Системы сборки различаются по типу используемого **алгоритма планировщика**:

- Топологическая сортировка,
- Выполнение с рестартами задач.
- Выполнение с приостановкой задач.

Также системы сборки различаются по типу зависимостей:

- Статические зависимости.
- Динамические зависимости.

Динамические зависимости образуются в процессе вычислений и не могут быть определены заранее, до начала выполнения сборки.

На рис. 8 показан пример динамической зависимости. В файле release.txt находятся все файлы выпуска. Этот файл образуется на основе слияния файлов bins.txt и docs.txt (документация). Файл release.tar является архивом выпуска. Зависимости release.tar не заданы статически. Они определяются содержимым файла release.txt. По этой причине, если информация о файле README добавляется в docs.txt, то необходима поддержка динамических зависимостей для правильной сборки release.tar.

Кроме того, в системе сборка может применяться **техника раннего среза** (early cutoff) — если задача выполнена, но ее результат не изменился с предыдущей сборки, то нет необходимости исполнять зависимые задачи, то есть процесс сборки можно завершить ранее.

На рис. 9 показан пример раннего среза. Если в main.c был добавлен новый комментарий, тогда сборка может быть остановлена после определения отсутствия изменений в main.o.

При использовании облачной системы сборки скорость сборки может быть существенно увеличена, благодаря разделению результатов сборки между участниками команды. При этом облачная сборка может поддержать вариант сборки, при котором локально образуются только конечные результаты сборки, а все промежуточные файлы остаются в облаке.

На рис. 10 приведен пример сценария работы с облачной системой сборки. Пользователь совершает следующие действия:

- Загружает исходные тексты, их хеш-значения 1, 2 и 3. Затем пользователь запрашивает сборку main.exe. Система сборки определяет с помощью изучения истории предыдущих сборок, что кто-то уже скомпилировал ранее именно эти исходные тексты. Результаты их сборки хранятся в облаке с хешами 4 (util.o) и 5 (main.o). Система сборки далее определяет, что для зависимостями с такими хешами есть main.exe с хешем 6. По ключу 6 из облачного хранилища извлекается конечный результат.
- Далее пользователь изменяет util.c, и его хеш становится равен 7. В облаке комбинации хешей (7, 2) не существует, то есть ранее никто еще не компилировал такой вариант исходного кода. Процесс продолжается до получения нового main.exe, после чего новые варианты файлов и их хеш-значения сохраняются в облаке.

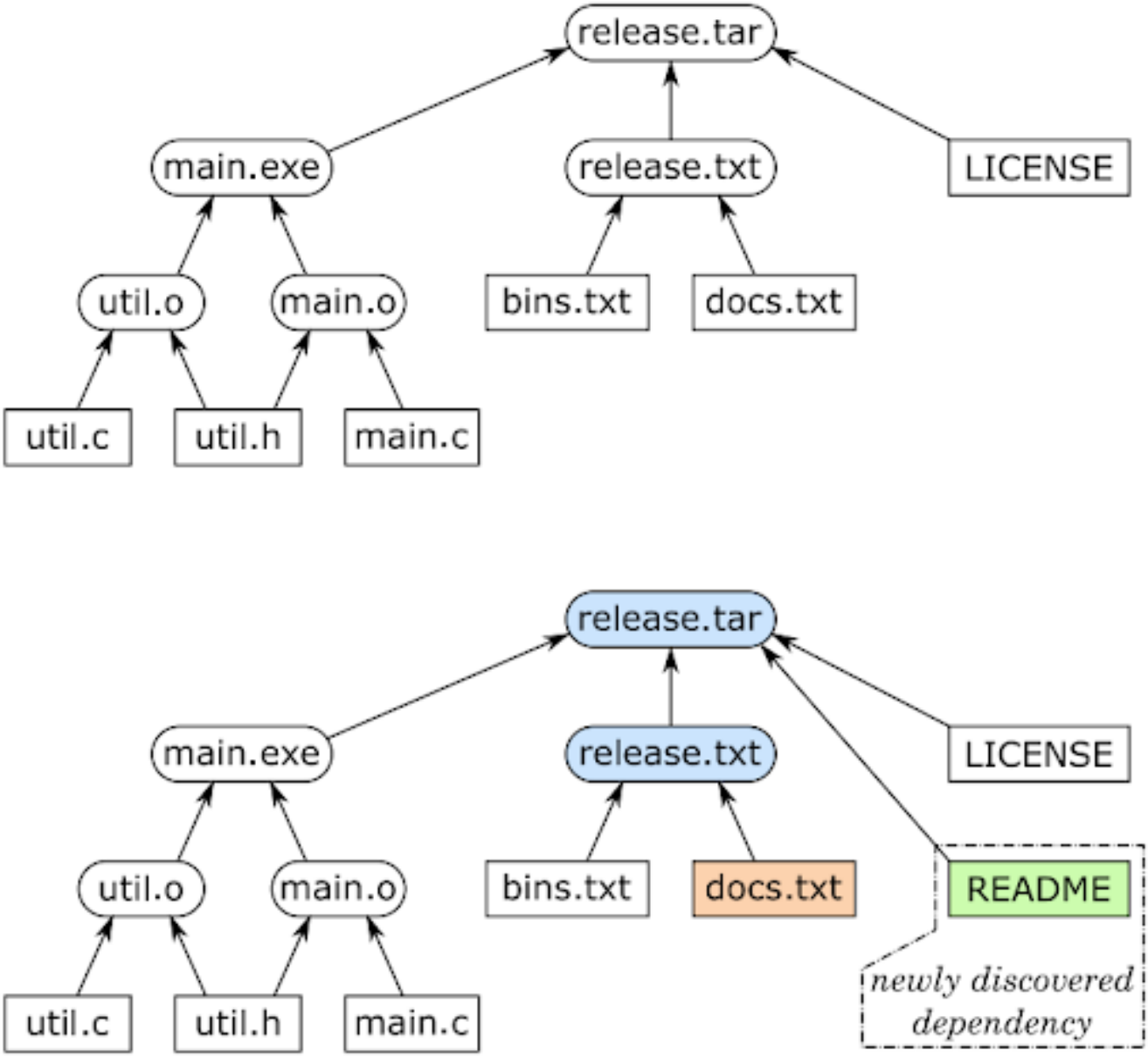

Рис. 8. Пример использования динамической зависимости

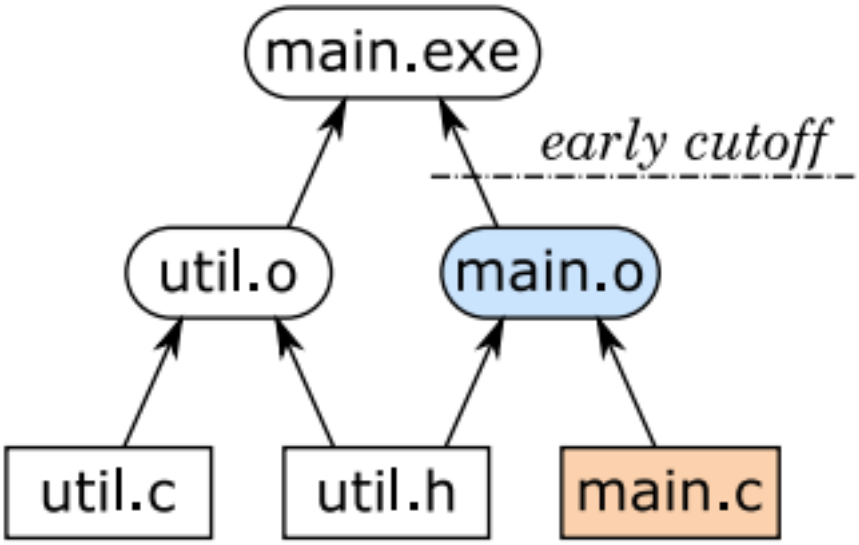

Рис. 9. Пример оптимизации раннего среза

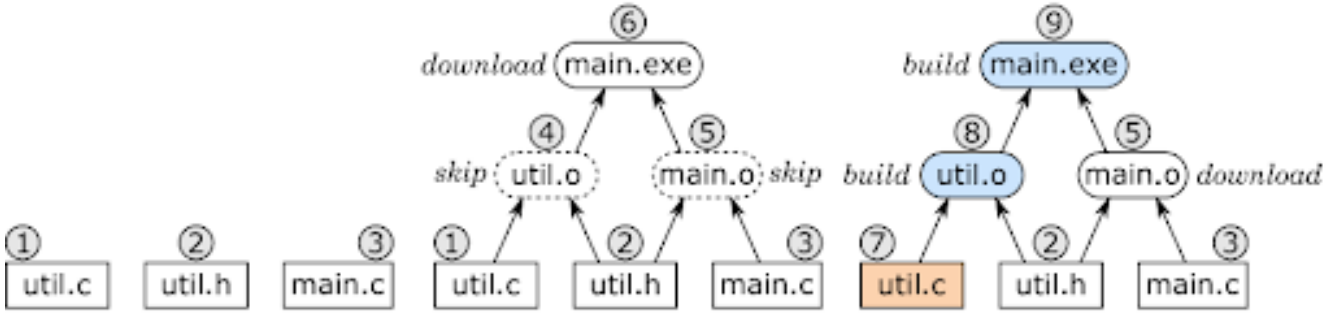

Рис. 10. Пример сценария работы с облачной системой сборки

5.1. Топологическая сортировка графа зависимостей

Топологической сортировкой называют порядок нумерации вершин ориентированного графа, при котором любое ребро идет из вершины с меньшим номером в вершину с большим. Очевидно, что не любой граф можно отсортировать топологически. Можно доказать, что топологическая сортировка существует для ациклических графов и не существует для циклических.

Пример топологической сортировки показан на рис. 11.

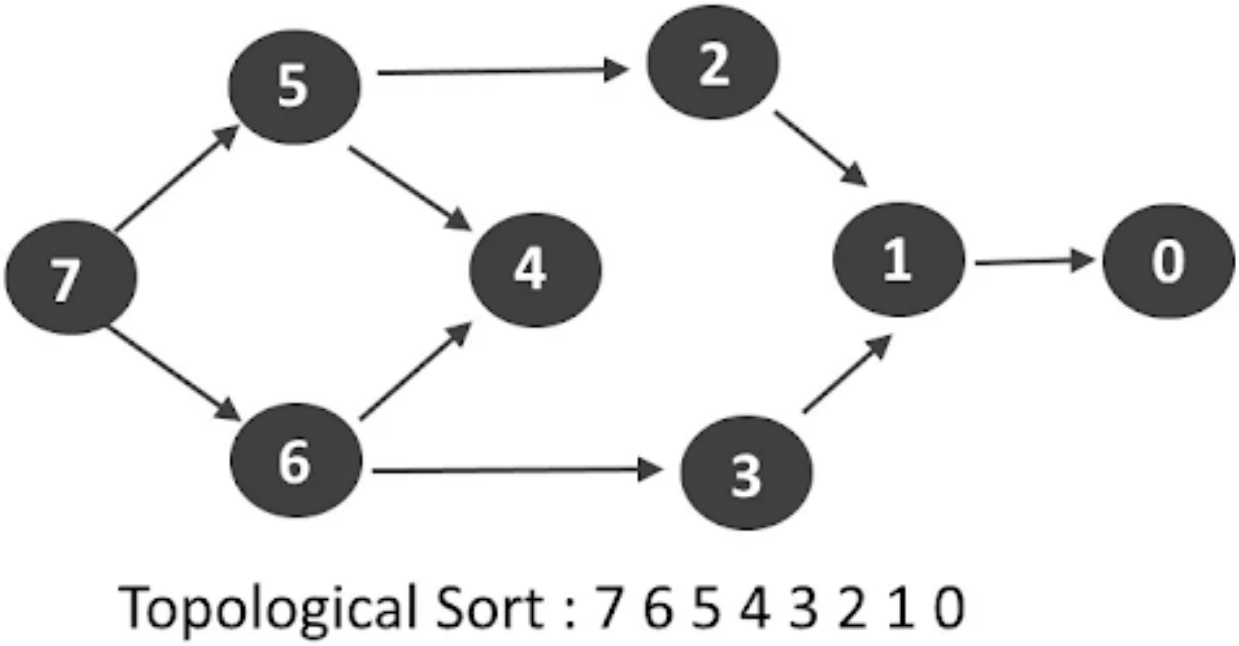

Рис. 11. Граф и результат его топологической сортировки

Алгоритм топологической сортировки на основе обхода графа в глубину:

```
// G – исходный граф
function topologicalSort():
    // проверить граф G на ацикличность
    fill(visited,false)
    for v in V(G)
        if not visited[v]
             dfs(v)
    ans.reverse()

function dfs(u):
    visited[u]=true
    for (u,v) in E(G)
        if not visited[v]
            dfs(v)
    ans.pushBack(u)
```

5.2. Система сборки make

Утилита make считывает из специального файла с именем Makefile или makefile в текущей директории инструкции о том, как (при помощи каких команд) компилировать и собирать программы, а также информацию, из каких файлов состоит программа, которую надо "сделать".

Одним из главных достоинств make (чрезвычайно полезным при создании больших программ) является то, что он сравнивает времена модификации файлов, и если, к примеру, файл file1.c новее, чем получаемый из него file1.o, то make поймет, что перекомпилировать надо только его, а остальные — не нужно (если они не изменились).

Makefile содержат три основных компонента:

- Правила, как из одних файлов создавать другие (например, .o из .c).
- Так называемые "зависимости", которые указывают, что, например, исполняемый файл proggie собирается из файлов prg_main.o и prg_funcs.o, а те, в свою очередь, получаются из файлов prg_main.c и prg_funcs.c.
- Определения переменных, позволяющие делать Makefile более гибкими.

Ниже приведен пример простейшего Makefile:

```
CC =              gcc
CFLAGS =          -c -W -Wall

%.o : %.c
                $(CC) $(CFLAGS) -o $@ $<

all:            program

program:        prg_main.o prg_funcs.o
                $(CC) -o proggie prg_main.o prg_funcs.o
```

Определения переменных. Строки вида "ИМЯ=значение" – это определения переменных. Для получения значения переменной используется запись "$(ИМЯ)" (знак доллара, а за ним имя переменной в скобках). Переменная может определяться через значения других переменных, например:

```
CFLAGS= -c $(WARNINGOPTIONS)
```

Правила. Запись "%.o : %.c" с последующей командой означает: "для любого файла .c, чтобы из него получить одноименный файл .o, надо выполнить такую-то команду", в данном случае:

```
gcc -c -W -Wall -o $@ $<
```

В правилах всегда используются специальные переменные $@ и $<. Переменная $@ обозначает "тот файл, который надо получить" (в данном случае .o), а $< — "исходный файл" (в данном случае .c). Такие переменные называются автоматическими. В примере:

```
all: library.cpp main.cpp
```

- $@ обозначает all,

- $< обозначает library.cpp,
- $^ обозначает library.cpp main.cpp.

Команда располагается на следующей строке, причем эта строка обязательно должна начинаться с символа табуляции.

Это довольно странное ограничение является основным источником ошибок и запутанности Makefile'ов — ведь визуально отличить символ табуляции от цепочки пробелов невозможно.

Зависимости. Запись вида

```
program: prg_main.o prg_funcs.o
```

означает, что файл program зависит от файлов prg_main.o и prg_funcs.o. Файл program называется целью (target), а файлы .o — зависимостями (dependencies). В системе make используется планировщик на основе алгоритма топологической сортировки, а постоянная информация о сборке извлекается из времени модификации файлов.

Эволюция систем автоматизации сборки показана на рис. 12.

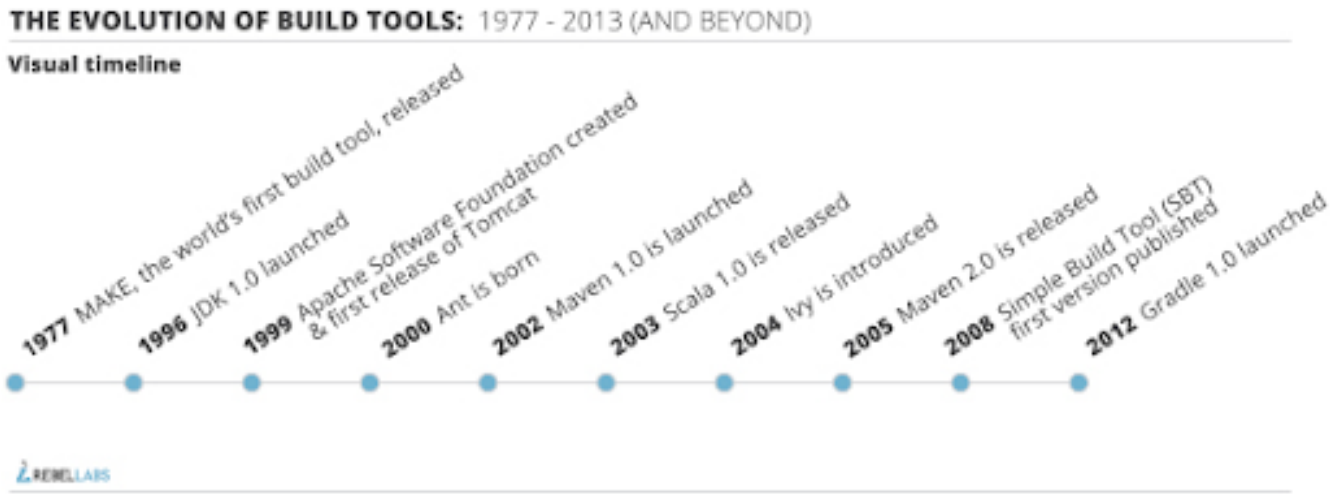

Рис. 12. Эволюция систем сборки

6. Системы контроля версий

Системы контроля версий — это класс программных систем, отвечающих за управление изменениями в компьютерных программах, документах, веб-сайтах и других наборах данных. Система контроля версий отслеживает изменения в файл или набор файлов в течение времени и позволяет вернуться позже к определённой версии файлов.

Коммит — фиксация факта изменений в системе контроля версий (СКВ).

В системе контроля версий обычно выделяются следующие функции:

- Хранение данных.
- Отслеживание изменений в данных (история, включая метаданные слияния).

- Распространение данных и истории изменений среди соавторов проекта.

Два наиболее распространенных варианта учета изменений данных в СКВ:

- набор изменений на основе различий (дельты),
- представление данных в виде направленного ациклического графа.

Системы CVS, Subversion, Perforce, Bazaar и т.д. представляют хранимую информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени (обычно это называют контролем версий, основанным на различиях), см. рис. 13.

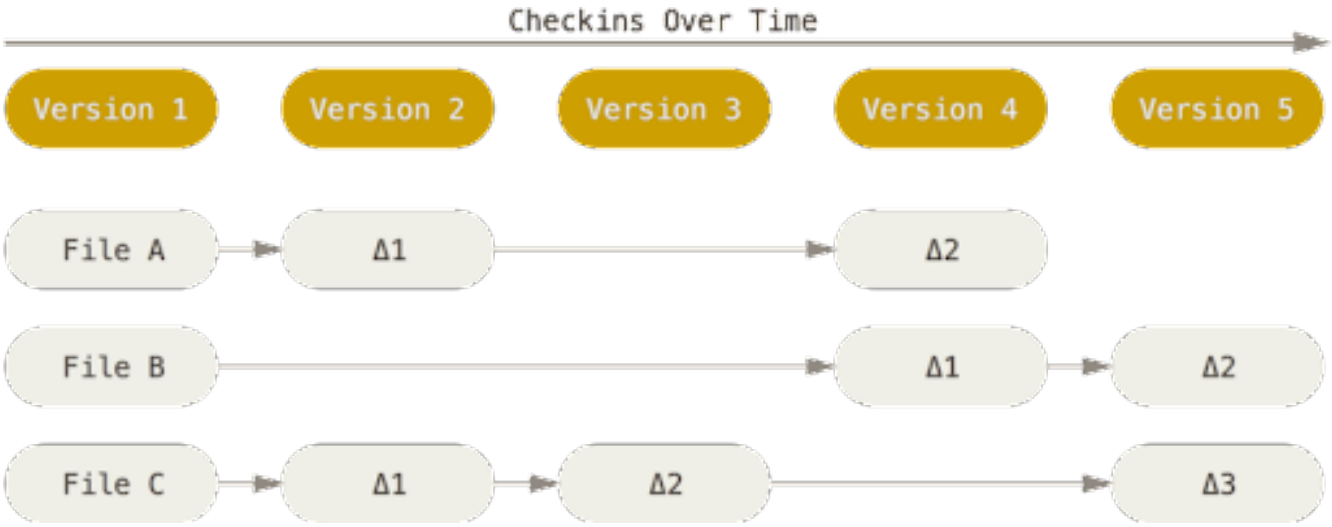

Рис. 13. Хранение данных как набора изменений относительно первоначальной версии каждого из файлов

Другой подход к хранению данных больше похож на набор снимков миниатюрной файловой системы. Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта, СКВ запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, СКВ не запоминает эти файлы вновь, а только создает ссылку на предыдущую версию идентичного файла, который уже сохранен. СКВ представляет свои данные как граф снимков, см. рис. 14.

С точки зрения типа используемой истории изменений системы контроля версий различаются:

- использованием линейной истории,
- использованием графового представления для истории изменений.

Системы контроля версий бывают различаются с точки зрения хранения и доставки своего содержимого:

- Локальные системы.
- Централизованные системы.
- Распределенные системы.

Локальные системы контроля версий. Многие люди в качестве метода контроля версий применяют копирование файлов в отдельную директорию (возможно даже, дирек-

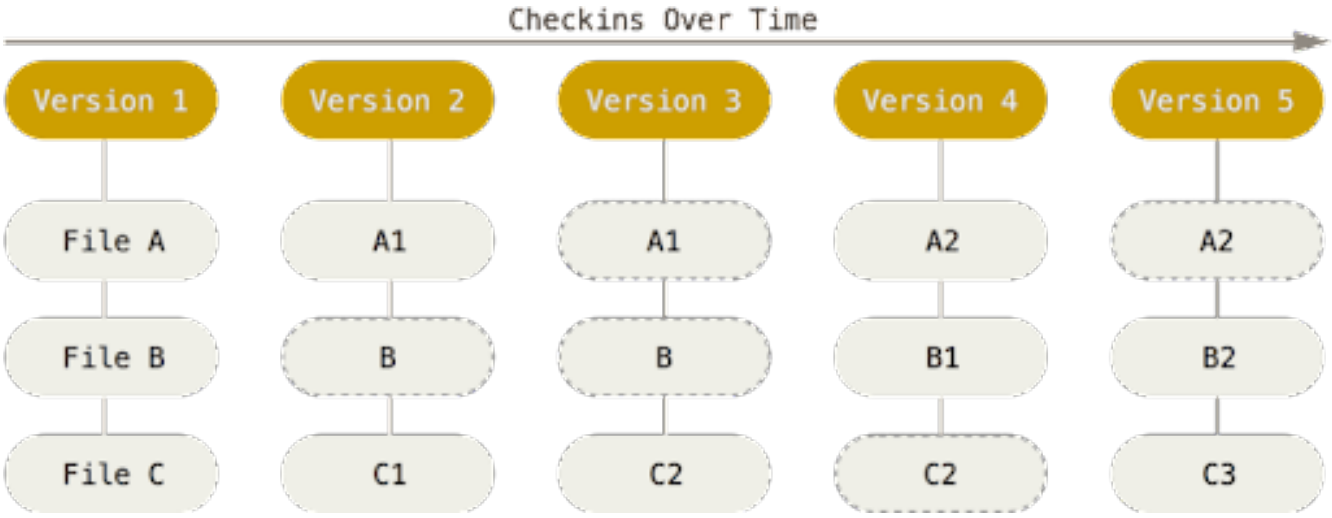

Рис. 14. Хранение данных как снимков проекта во времени

торию с отметкой по времени). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть, в какой директории вы находитесь, и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели. Для того, чтобы решить эту проблему, были разработаны локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий, см. рис. 15.

Одной из популярных СКВ была система RCS, которая и сегодня распространяется со многими компьютерами. RCS хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

Локальные СКВ страдают от следующей проблемы: когда вся история проекта хранится в одном месте, есть риск потерять все.

Централизованные системы контроля версий. Следующая серьезная проблема, с которой сталкиваются разработчики, — это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как CVS, Subversion и Perforce, используют единый сервер, содержащий все версии файлов, и некоторое количество клиентов получают файлы из этого централизованного хранилища, см. рис. 16. Применение ЦСКВ являлось стандартом на протяжении многих лет.

Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определенной степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, данный подход тоже имеет серьезные минусы. Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто

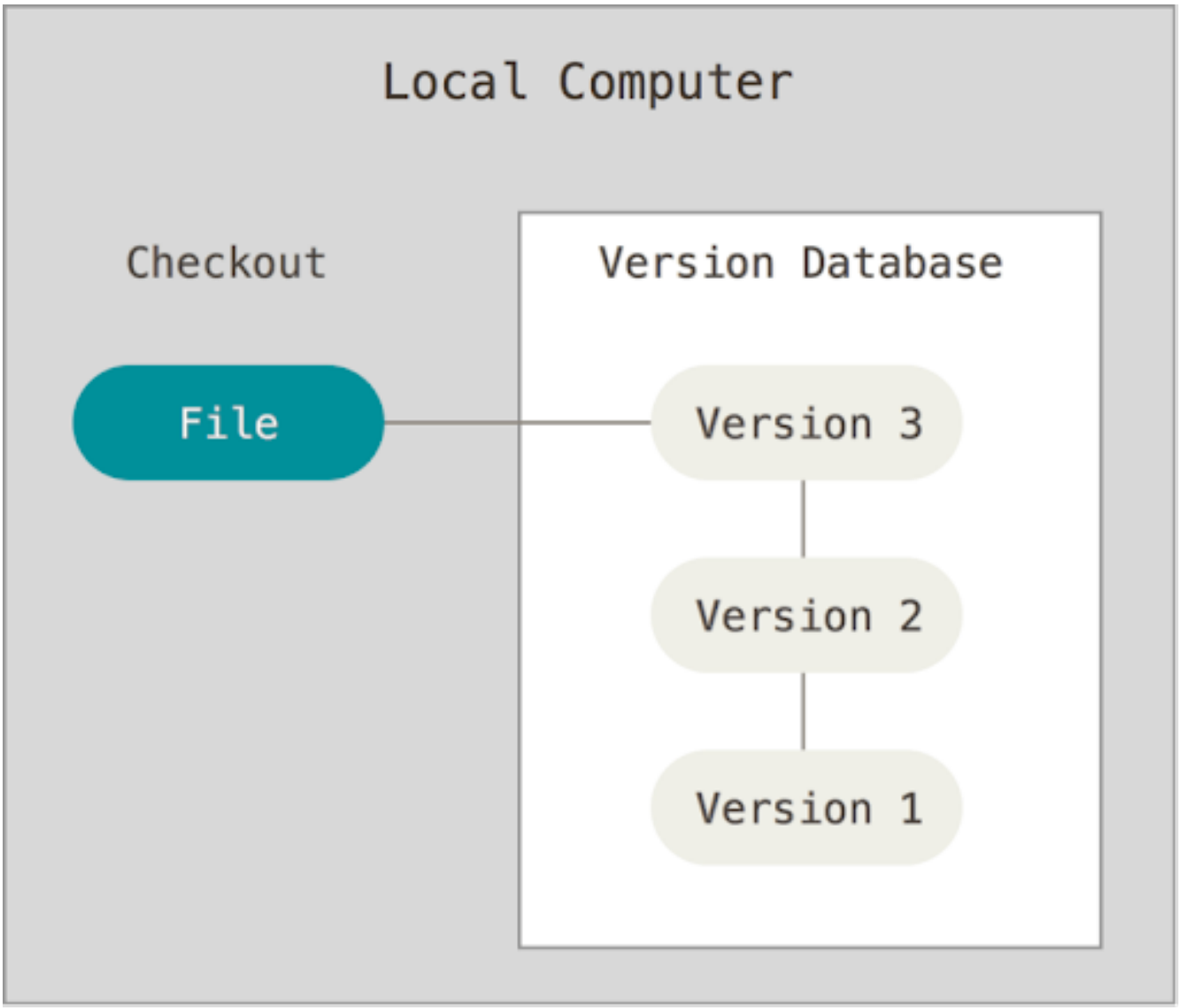

Рис. 15. Локальный контроль версий

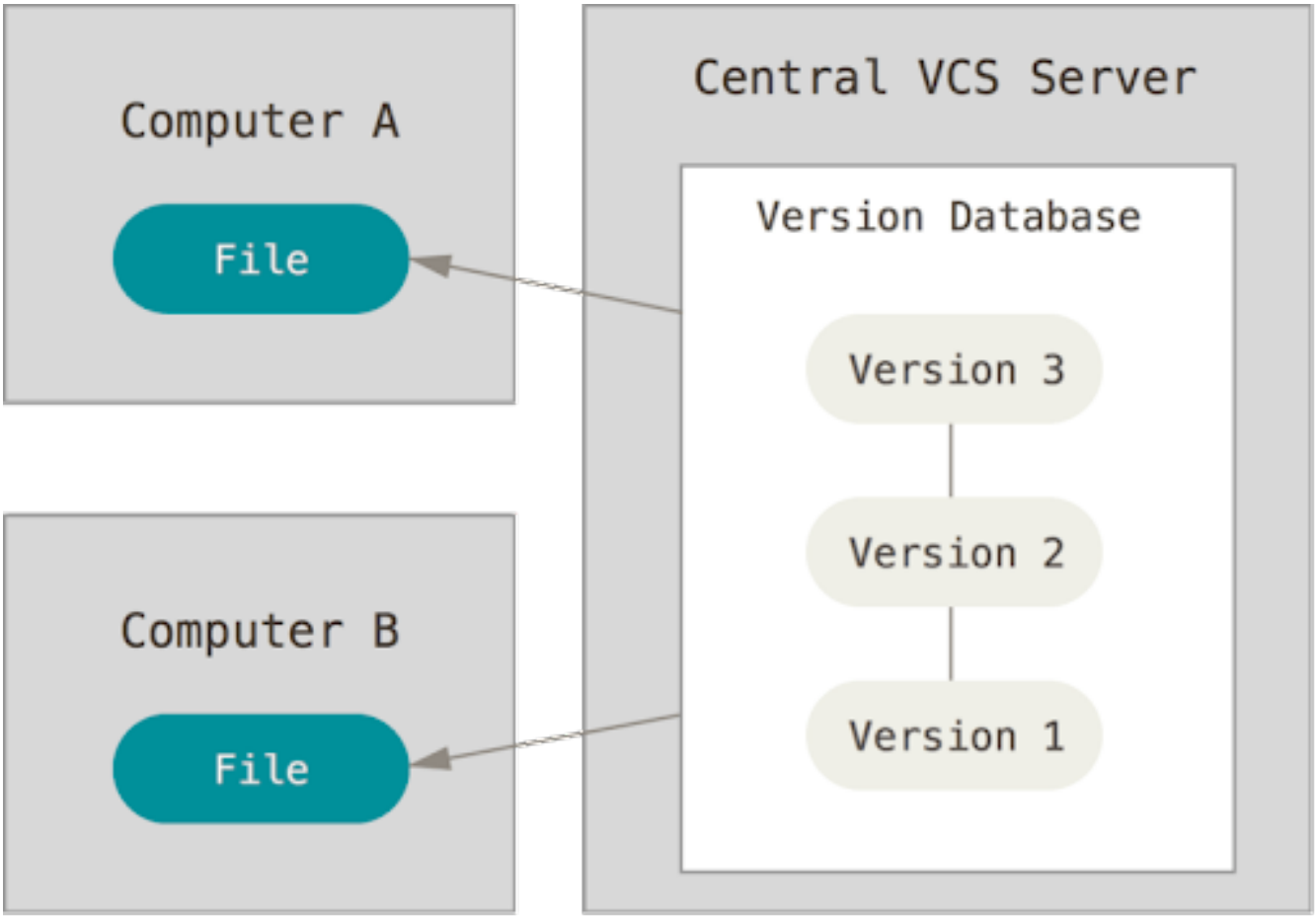

Рис. 16. Централизованный контроль версий

не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные точки сохранения отсутствуют, о будет потеряно все — вся история проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков.

Распределённые системы контроля версий. В РСКВ (таких как Git, Mercurial, Bazaar или Darcs) клиенты не просто скачивают снимок всех файлов (состояние файлов на определенный момент времени) — они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, будет недоступен, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным вариантом всех данных, см. рис. 17.

Многие РСКВ могут одновременно взаимодействовать с несколькими удаленными репозиториями, благодаря этому можно работать с различными группами людей, применяя различные подходы единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

Системы контроля версий также различаются по способу разрешения конфликтов при изменений одного и того же файла со стороны нескольких разработчиков:

- механизм блокировки,
- сценарий слияния изменений перед коммитом,
- сценарий слияния изменений после коммита.

При **блокировке** рабочие файлы обычно доступны только для чтения, поэтому их нельзя просто так изменить. Необходимо дать запрос СКВ сделать рабочий файл доступным для записи, заблокировав его; только один пользователь может делать это в любой момент времени. Когда изменения будут зарегистрированы, то данные разблокируются, и рабочий файл снова становится доступным только для чтения. Это позволяет другим пользователям заблокировать файл для внесения дальнейших изменений.

В **сценарии слияния перед коммитом** СКВ отслеживает, когда осуществляется попытка выполнить коммит файла или файлов, которые уже изменились с момента начала редактирования, и в результате система требует разрешения конфликта, прежде чем коммит может быть завершен.

Существуют СКВ, которые никогда не блокируют коммиты — это **сценарий слияния после коммита**. Если копия репозитория изменилась с момента извлечения файлов, коммит можно просто перенаправить в новую ветку. Впоследствии ветви могут оставаться отдельными; или любой разработчик может выполнить слияние, которое снова объединит их.

Модель «фиксация перед слиянием» приводит к очень гибкому стилю разработки, в котором разработчики создают и повторно объединяют множество небольших веток. Это упрощает экспериментирование и записывает все, что пробуют разработчики, таким образом, чтобы это было полезно для проверки кода.

В наиболее общем случае репозиторий под управлением СКВ, работающей по сценарию слияния после коммита, может иметь форму произвольно сложного ориентированного

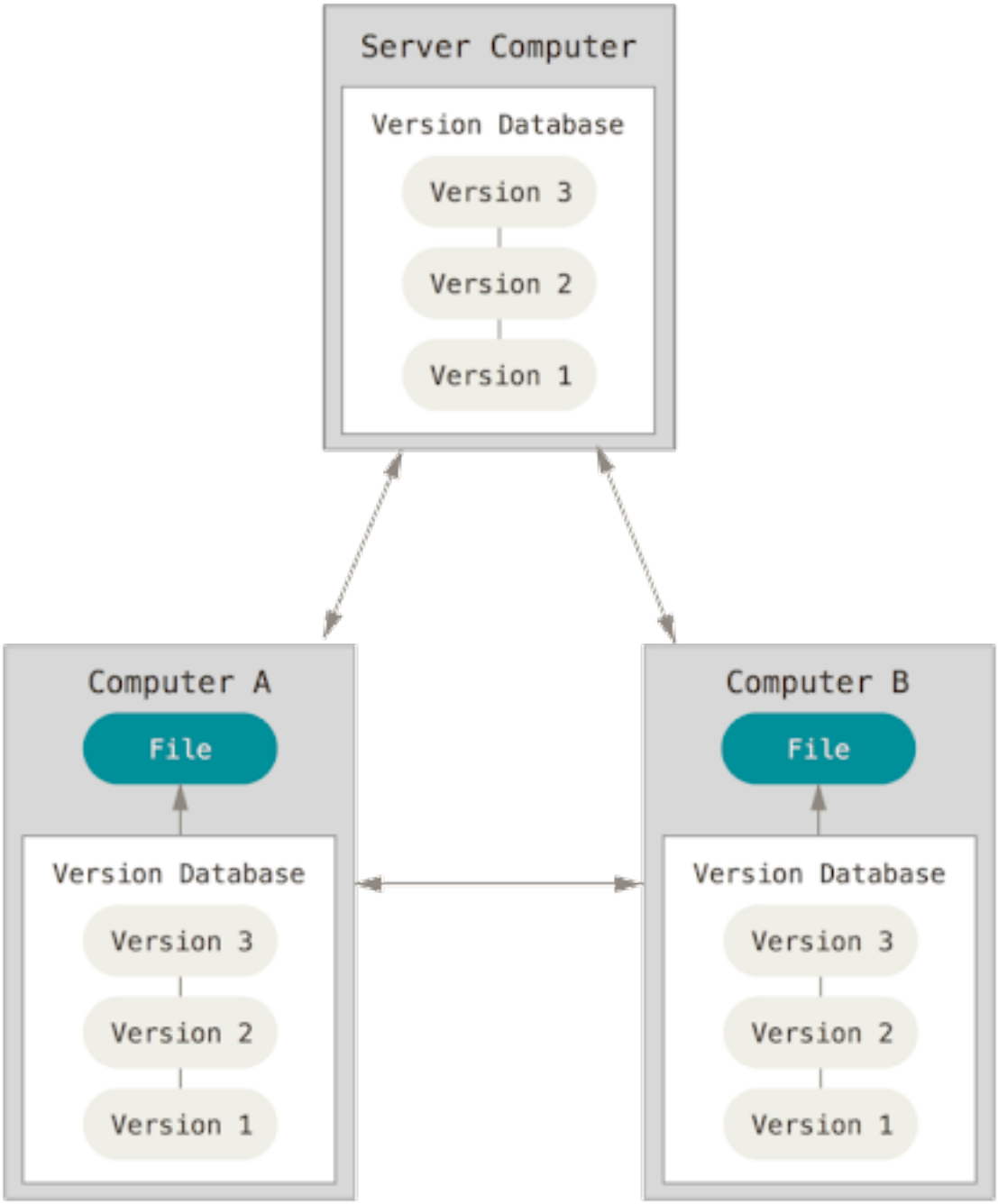

Рис. 17. Распределенный контроль версий

ациклического графа. Такие СКВ часто выполняют слияние с учетом истории, используя алгоритмы, которые пытаются учитывать не только содержимое сливаемых версий, но и содержимое их общих предков в графе репозитория.

6.1. История развития систем контроля версий

На рис. 18 показана временная шкала эволюции систем контроля версий.

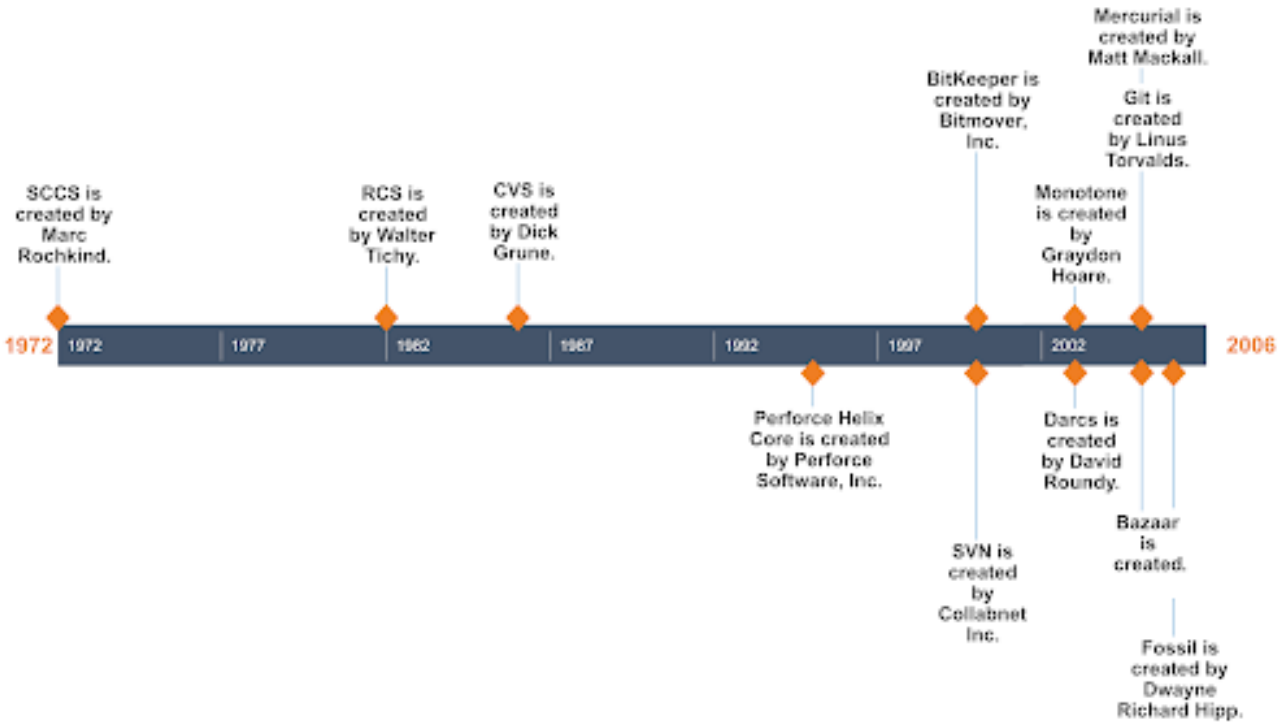

Рис. 18. Эволюция систем контроля версий

В таб. 1 представлены три поколения СКВ.

Таблица 1: Поколения систем контроля версий

Поколение	Модель взаимодействия	Единица операции	Разрешение конфликтов	Примеры
Первое	Локальный доступ	Файл	Блокировка	RCS, SCCS
Второе	Централизованное	Файл/Множество файлов	Слияние до коммита	CVS, SourceSafe, Subversion, Team Foundation Server
Третье	Распределенное	Множество файлов	Слияние после коммита	Bazaar, Git, Mercurial, Fossil

6.2. Система Git

Git напоминает миниатюрную файловую систему с утилитами, надстроенными над ней, нежели просто на СКВ.

Для работы большинства операций в Git достаточно локальных файлов и ресурсов — в основном, системе не нужна никакая информация с других компьютеров в вашей сети. Так как вся история проекта хранится прямо на локальном диске, большинство операций кажутся чуть ли не мгновенными.

В Git для всего вычисляется хеш-сумма, и только потом происходит сохранение. В дальнейшем обращение к сохраненным объектам происходит по этой хеш-сумме. Это значит, что невозможно изменить содержимое файла или директории так, чтобы Git не узнал об этом. Данная функциональность встроена в Git на низком уровне и является неотъемлемой частью его философии. Вы не потеряете информацию во время её передачи и не получите повреждённый файл без ведома Git.

Механизм, которым пользуется Git при вычислении хеш-сумм, называется SHA-1 хеш. Это строка длинной в 40 шестнадцатеричных символов (0-9 и a-f), она вычисляется на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Вы будете постоянно встречать хеши в Git, потому что он использует их повсеместно. На самом деле, Git сохраняет все объекты в свою базу данных не по имени, а по хеш-сумме содержимого объекта.

Когда вы производите какие-либо действия в Git, практически все из них только добавляют новые данные в базу Git. Очень сложно заставить систему удалить данные либо сделать что-то, что нельзя впоследствии отменить. Как и в любой другой СКВ, вы можете потерять или испортить свои изменения, пока они не зафиксированы, но после того, как вы зафиксируете снимок в Git, будет очень сложно что-либо потерять, особенно, если вы регулярно синхронизируете свою базу с другим репозиторием.

У Git есть три основных состояния, в которых могут находиться ваши файлы: зафиксированное (committed), изменённое (modified) и подготовленное (staged).

- Зафиксированный значит, что файл уже сохранён в вашей локальной базе.
- К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы.
- Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

Существует три основных элемента проекта Git:

- Git-директория (Git directory),
- рабочая директория (working directory),
- область подготовленных файлов (staging area).

Git-директория — это то место, где Git хранит метаданные и базу объектов вашего проекта. Это самая важная часть Git, и это та часть, которая копируется при клонировании репозитория с другого компьютера.

Рабочая директория является снимком версии проекта. Файлы распаковываются из сжатой базы данных в Git-директории и располагаются на диске, для того чтобы их можно было изменять и использовать.

Область подготовленных файлов — это файл, обычно располагающийся в вашей Git-директории, в нём содержится информация о том, какие изменения попадут в следующий коммит. Эту область ещё называют “индекс”, однако называть её stage-область также общепринято.

Базовый подход в работе с Git выглядит так:

- Вы изменяете файлы в вашей рабочей директории.
- Вы выборочно добавляете в индекс только те изменения, которые должны попасть в следующий коммит, добавляя тем самым снимки только этих изменений в область подготовленных файлов.
- Когда вы делаете коммит, используются файлы из индекса как есть, и этот снимок сохраняется в вашу Git-директорию.

Если определенная версия файла есть в Git-директории, эта версия считается зафиксированной. Если версия файла изменена и добавлена в индекс, значит, она подготовлена. И если файл был изменен с момента последнего распаковывания из репозитория, но не был добавлен в индекс, он считается измененным.

6.3. Модель данных Git

Снимки. Git моделирует историю файлов и каталогов в некотором каталоге верхнего уровня как серию снимков. В терминологии Git файл называется «blob» (Binary Large Object), это просто набор байтов. Каталог называется «деревом», и он сопоставляет имена с blobs или деревьями (то есть каталоги могут содержать в себе другие каталоги).

Снимок - это отслеживаемое СКВ дерево.

Пример дерева:

```
<root> (tree)
|
+- foo (tree)
|  |
|  + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

Модель истории. В Git **история** - это ориентированный ациклический граф снимков. В псевдокоде:

```
// файл — последовательность байт
type blob = array<byte>

// директория содержит именованные файлы или директории
type tree = map<string, tree | blob>
```

```
// коммит содержит родителей, метаданные и дерево
type commit = struct {
    parent: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

Объекты. Объект это blob, дерево или коммит. Объекты адресуются по своим хеш-значениям SHA-1.

```
type object = blob | tree | commit

objects = map<string, object>

def store(object):
    id = sha1(object)
    objects[id] = object

def load(id):
    return objects[id]
```

Ссылки. Ссылки указывают на объекты.

```
references = map<string, string>

def update_reference(name, id):
    references[name] = id

def read_reference(name):
    return references[name]

def load_reference(name_or_id):
    if name_or_id in references:
        return load(references[name_or_id])
    else:
        return load(name_or_id)
```

Управление git удобнее всего осуществлять из командной строки. Ниже представлены некоторые команды git:

Базовые команды

```
git help <command>: помощь по git
git init: инициализация нового репозитория
git status: текущий статус репозитория
git add <filename>: добавить файл в подготовительную область
git commit: создать новый коммит
git log: показать историю изменений
git log --all --graph --decorate: история, как граф
```

```
git diff <filename>: изменения по сравнению с состоянием в подготовительной области
git diff <revision> <filename>: изменения в разных снимках для файла
git checkout <revision>: обновление HEAD и текущей ветки
```

Ветки и слияние

```
git branch: показать ветки
git branch <name>: создать ветку
git checkout -b <name>: создать ветку и переключиться на нее
то же, что и git branch <name>; git checkout <name>
git merge <revision>: слияние в текущую ветку
```

Удаленная работа

```
git remote: список удаленных репозиториев
git remote add <name> <url>: добавить удаленный репозиторий
git push <remote> <local branch>:<remote branch>: отправить данные для изменения в
git branch --set-upstream-to=<remote>/<remote branch>: установить взаимосвязь между
git fetch: получить данные из удаленного репозитория
git pull: то же, что и git fetch; git merge
git clone: загрузить удаленный репозиторий
```

Отмена действий

```
git commit --amend: отредактировать содержимое коммита
git reset HEAD <file>: отменить добавление файла
```

7. Генераторы документации

Текст.

8. Виртуализация и контейнеризация

Текст.