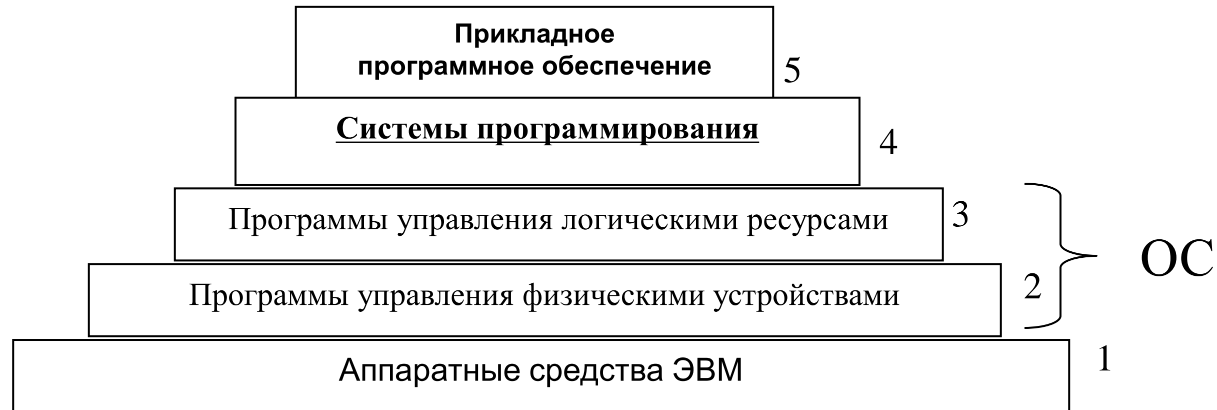
Системы программирования Интегрированные среды разработки Системы контроля версий

О. Г. Французов

Структура вычислительной системы



Система программирования (СП) — это комплекс программных инструментов и библиотек, который поддерживает **весь** технологический цикл создания программного продукта (ПП).

ПП — программа, оформленная, документированная и специфицированная таким образом, что ее можно использовать независимо, отчужденно от автора программы.

Этапы технологического цикла создания ПП

I. **Создание ПП.**

II. **Сопровождение:**

- попытка приспособить ПП к измененным целям,
- исправление ошибок, не выявленных на этапе I.

III. **Эксплуатация ПП.**

Создание ПП (1)

1. Анализ требований

Уточняются, формализуются и документируются требования заказчика к ПП, в результате создаются ***внешняя спецификация ПП***, т.е. характеристика программы с точки зрения заказчика.

Часть требований к ПП можно формально записать с помощью языков спецификаций (SDL,…), таблиц решений, функциональных диаграмм.

2. Проектирование

Выделяются отдельные модули, определяется их иерархия и сопряжение между ними. В результате создается общая ***схема иерархии*** и ***внешняя спецификация отдельных модулей***.

По внешним спецификациям разрабатывается внутренняя структура каждого модуля - выбирается алгоритм работы модуля и способ внутреннего представления данных.

Создание ПП (2)

3. Кодирование.

4. Компоновка и интеграция

5. Тестирование и отладка

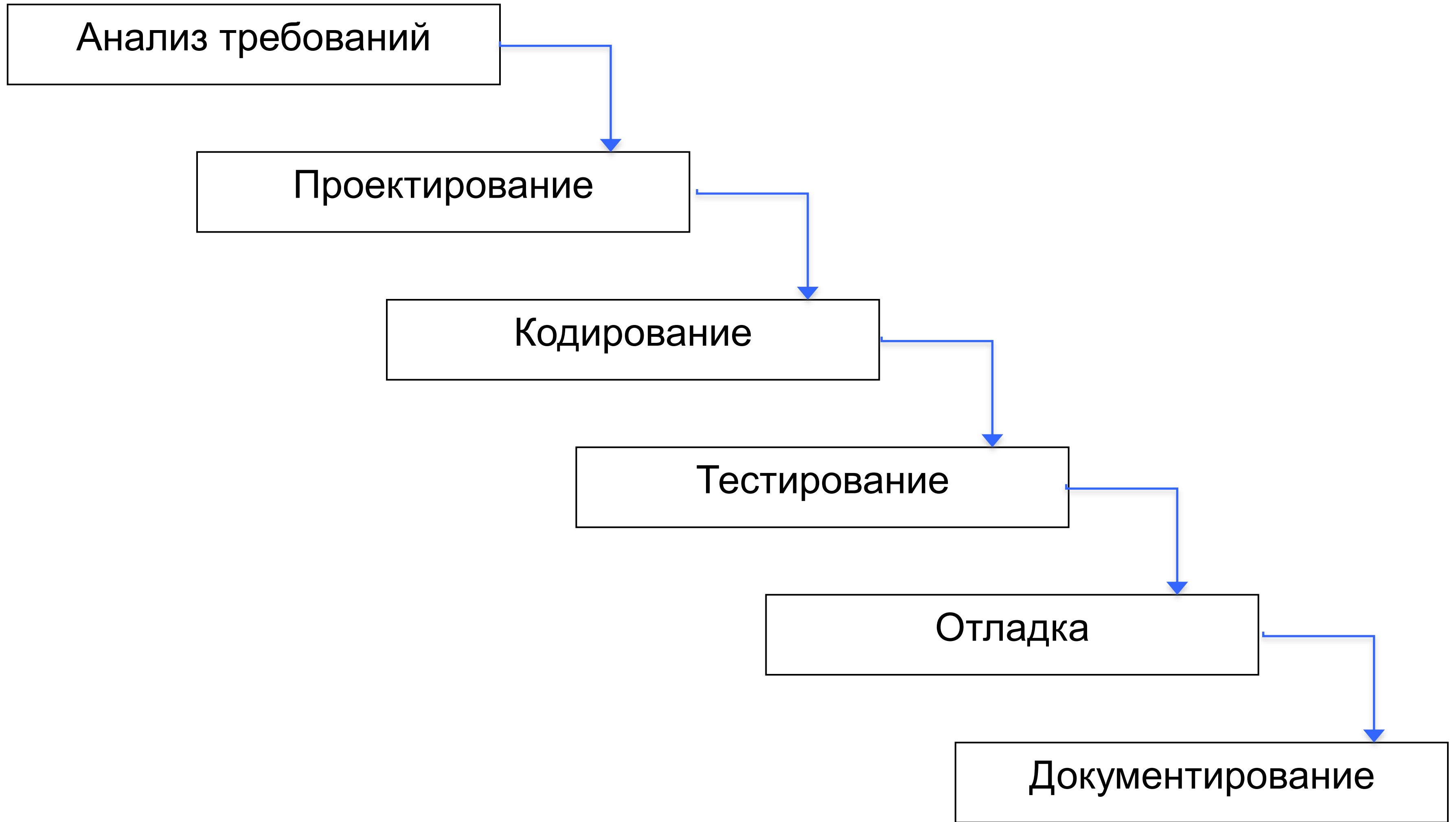
- Верификация (ПП работает согласно спецификации)
- Валидация (ПП пригоден для использования)
- Тестирование: ручное, автоматизированное; функциональное, интеграционное, модульное; регрессионное
- Формальное доказательство корректности работы

6. Документирование

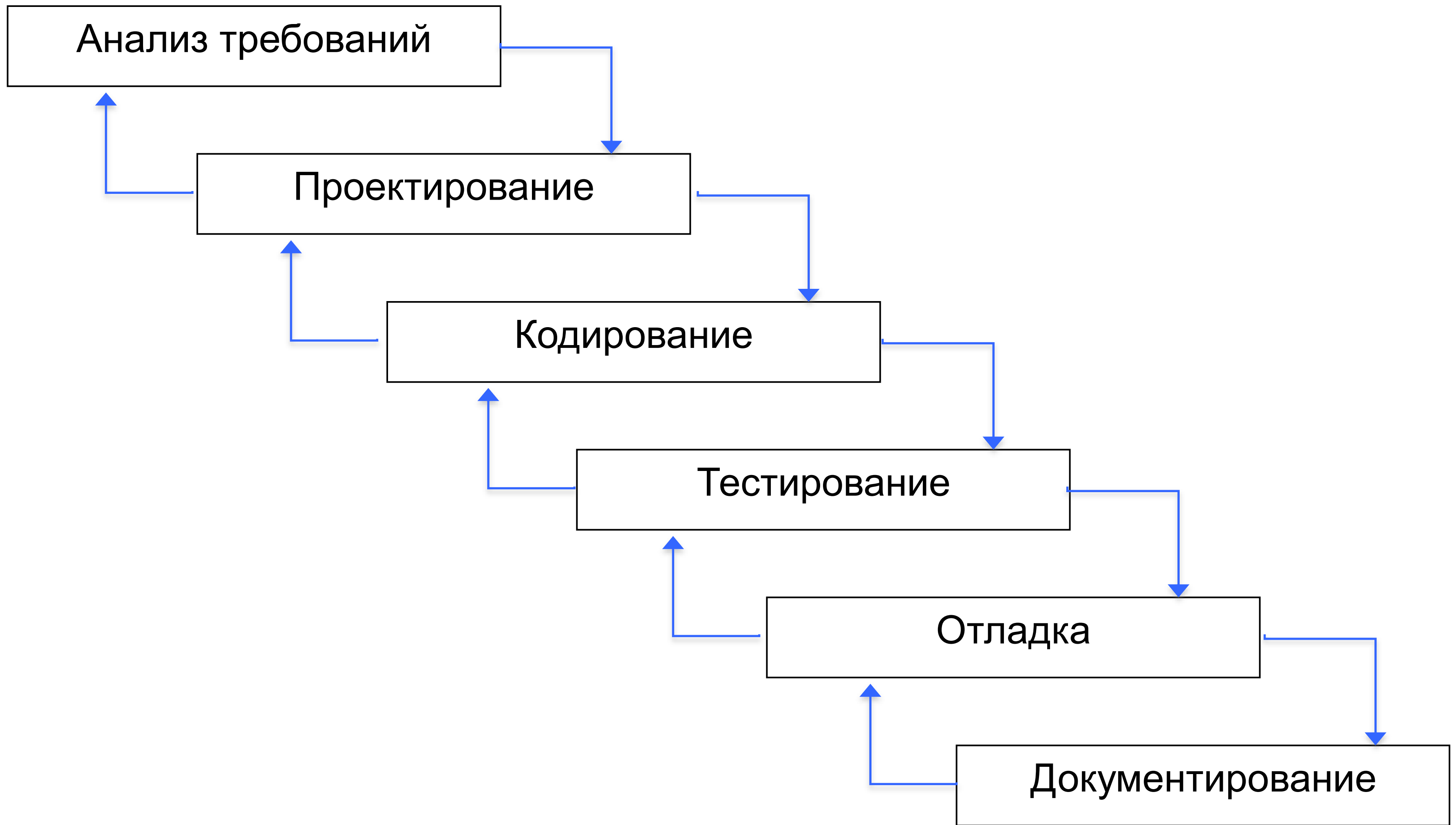
7. Внедрение

8. Сопровождение

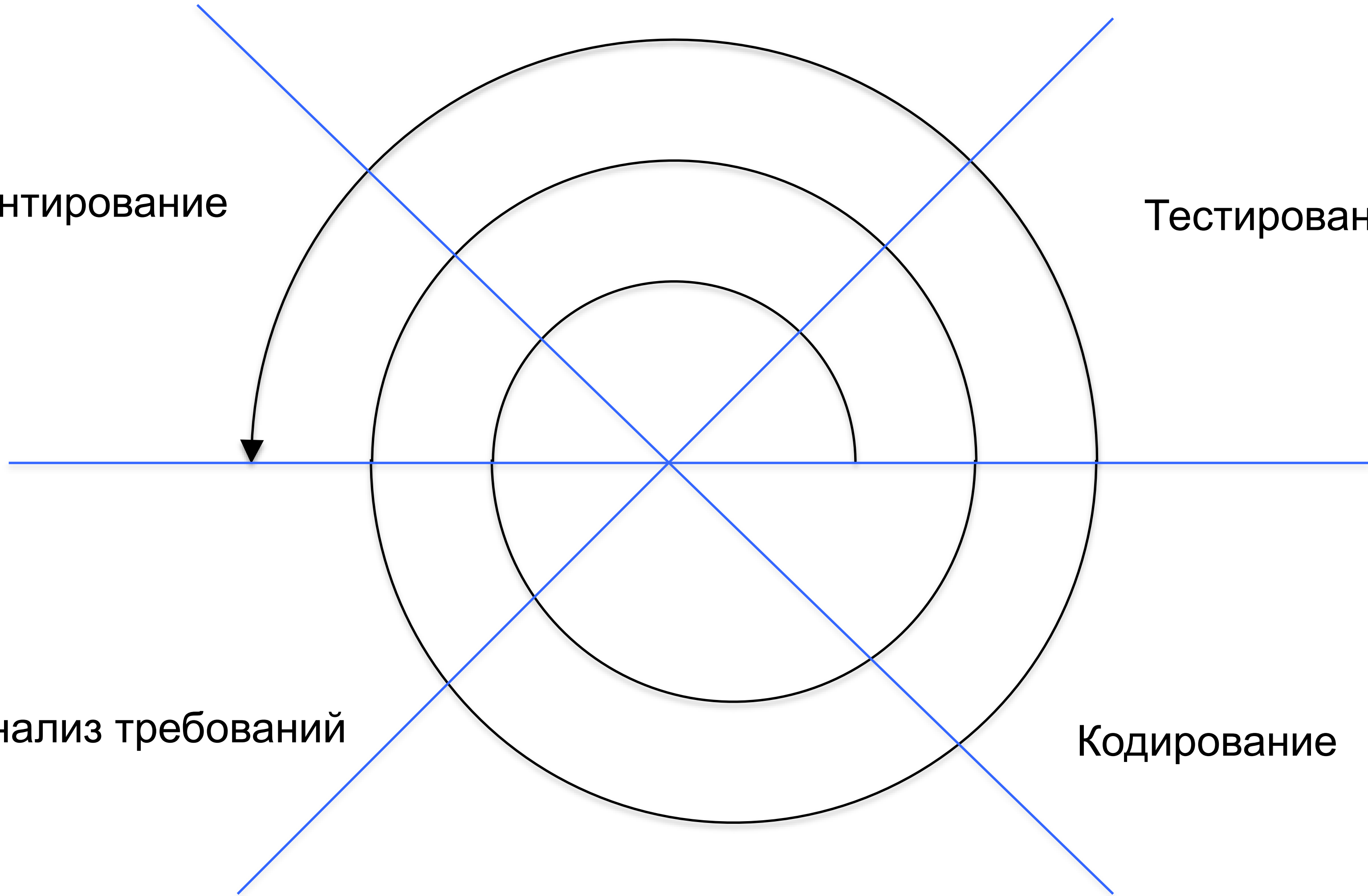
Каскадная модель



Каскадно-возвратная модель



Итерационная модель



Основные компоненты системы программирования.

1. **Транслятор** (переводит программы с языка программирования на машинный язык, что и позволяет выполнить их на ЭВМ).

2. **Макрогенератор** или макропроцессор (работает непосредственно перед транслятором, используется для получения макрорасширения исходной программы).

3. **Редактор текстов** (используется для составления программ на языке программирования).

4. **Редактор связей** или компоновщик (предназначен для связывания между собой (по внешним данным) объектных файлов, порождаемых компилятором, а также файлов библиотек, входящих в состав СП).

5. **Отладчик** (используется для проверочных запусков программ и исправления ошибок).

6. **Библиотеки стандартных программ** (облегчают работу программиста, используются на этапе трансляции и исполнения).

Дополнительные компоненты систем программирования

a) **Система контроля версий** для версионирования исходного текста ПП.

b) **Средства конфигурирования**

- помогают создавать **различные конфигурации** ПП в зависимости от конкретных параметров системного окружения, в котором ПП будет функционировать и от возможных различий отдельных версий ПП;
- поддерживают информацию обо всех предполагаемых и выполненных **изменениях ПП;**
- обеспечивают координированное **управление** развитием функциональности и улучшением характеристик системы.

c) **Средства тестирования** (помогают при составлении набора тестов).

d) **Профилировщик.** Профилирование — определение (в процентах) времени, затрачиваемого на выполнение отдельных фрагментов программы, как правило, для линейных участков кода (фрагментов программы, где нет передачи управления). Профилировщик часто используется для более эффективной оптимизации программы.

e) **Справочная система** (содержит справочные материалы по языку программирования и компонентам СП).

Дополнительные компоненты систем программирования

e) **Инструменты для статического анализа кода**

- Производят анализ логики работы программы без её исполнения (работают с исходным текстом программы).
- Основное применение — поиск мест, где может содержаться логическая ошибка (lint и аналоги).
- Также используются для организации навигации по коду (генерация т. н. тэгов), полуавтоматического рефакторинга и проч.

f) **Средства навигации по коду**. В простейшем варианте (ctags) — анализ исходного текста, поиск в нем символов (определений функций, классов) и формирование указателя найденных символов для использования в текстовом редакторе. В более сложных случаях отыскиваются также отношения наследования, места использования символа в коде и проч.

g) **Инструменты подготовки документации.** Используются для автоматической генерации списков классов, функций и т. п. по исходному коду. При этом автоматически извлекаются комментарии к коду, и на выходе генерируется документация к коду, которая может компоноваться с концептуальной документацией на ПП и его подсистемы.

h) **Управление разработкой**. Планирование, отслеживание замечаний.

Виды систем программирования

(*По стратегии интеграции*)

1. **Наборы независимых инструментов**

2. **Интегрированные системы программирования**

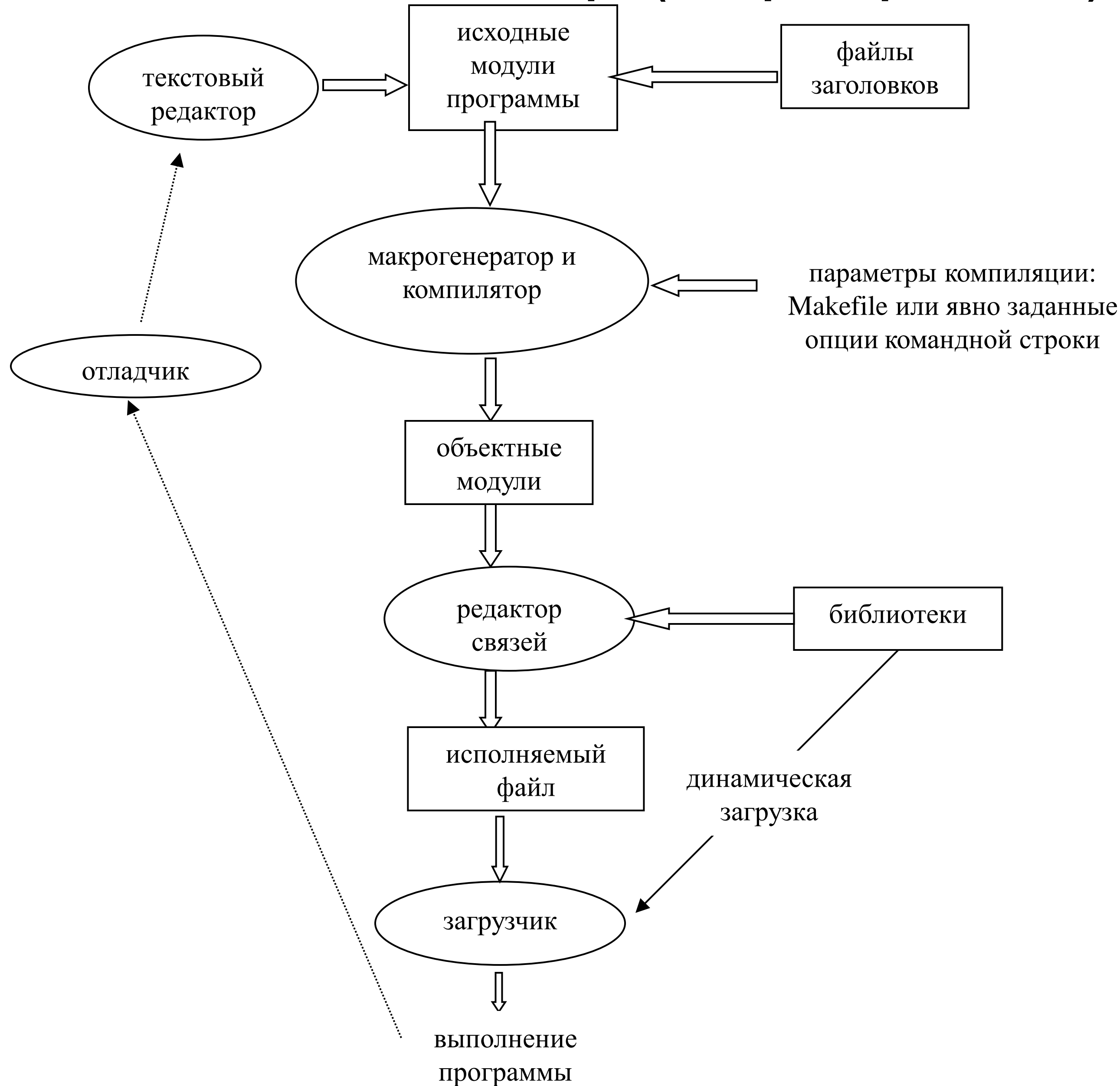
Стратегии трансляции

1. Компиляторы и ассемблеры

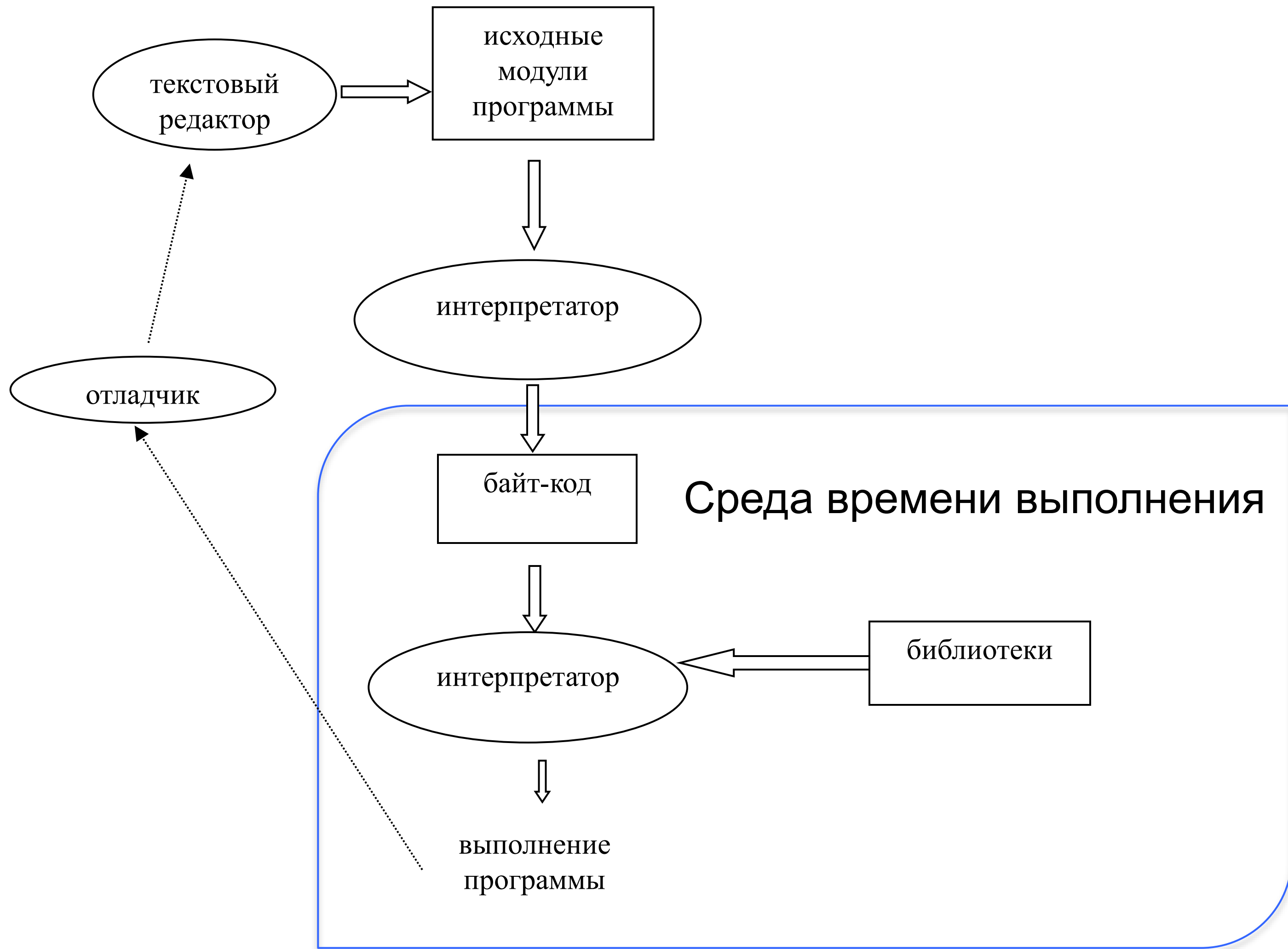
2. Интерпретаторы

3. Смешанная стратегия (байт-код, JIT-компиляция)

Общая схема функционирования основных компонентов СП на базе компилятора (на примере СП Си):



Общая схема функционирования основных компонентов СП на базе интерпретатора:



Интегрированная среда разработки

ИСР (IDE, integrated development environment) — комплекс программных средств, поддерживающих полный жизненный цикл ПП.

Простая ИСР содержит минимальный набор компонентов:

- текстовый редактор,
- компилятор,
- редактор связей,
- отладчик.

Состав продвинутой ИСР

- **модуль системы контроля версий** (все объекты, с которыми идет работа в рамках ИСР, хранятся в репозитории системы контроля версий);
- **графические средства** анализа и проектирования, обеспечивающие создание и редактирование иерархически связанных диаграмм, образующих модели ПП, а также графический пользовательский интерфейс, обеспечивающий взаимодействие с функциями API;
- **средства разработки приложений**, включая инструменты кодогенерации;
- **компилятор;**
- **текстовый редактор**;
- **средства статического анализа кода** с поддержкой навигации по коду и полуавтоматического рефакторинга;
- **средства документирования**;
- **средства тестирования и отладки**;
- **средства управления проектной деятельностью**, в т. ч. интеграция с системами отслеживания замечаний;
- **средства обратного конструирования** (позволяют восстановить логику и структуру программы по ее исходным текстам, с целью модификации или перенесения на другую платформу).

Текстовые редакторы

1. **Пакетные**
   + Макросредства

2. **Диалоговые**
   1. Строчные
   2. Экранные

Возможности текстового редактора (в ИСР)

1. Подготовка текста программы (обычные действия по созданию, редактированию, сохранению файла с текстом программы).

2. Многооконный интерфейс, управление окнами и вкладками.

3. Закладки, настраиваемые сочетания клавиш, шаблоны фрагментов текста, программное управление самим редактором

4. Интеграция с компилятором и средствами статического анализа кода.

5. Интеграция с отладчиком.

Возможности текстового редактора (в ИСР)

Интеграция с компилятором и/или средствами статического анализа кода:

- визуализация текста с выделением лексем (синтаксическая подсветка элементов языка),
- дополнение кода, интерактивная подсказка,

```
import string
print string.s
                rsplit
#---- Abbrevi    rstrip
#       - Snipp  split
#         can b  splitfields
#         'Ctrl  strip
#         Toolb
#         start
#
```

```
import string
print string.split(
                   split(s [,sep [,maxsplit]]) -> list of strings
#---- Abbreviation  Return a list of the words in the string s, using sep as the
#       - Snippets f delimiter string.
#         can be inserted by typing the snippet name followed by
```

- всплывающие подсказки об атрибутах идентификаторов, если на них установить курсор, отображение ошибок, обнаруженных на этапе компиляции, в тексте программы,
- навигация по коду (переход к определению имени, поиск мест использования имени, поиск имени, навигация по иерархии наследования),
- рефакторинг кода (от простейших случаев: переименование идентификатора, генерация заглушки — до сложных: выделение базового класса, перенос метода вверх или вниз по иерархии).

Возможности текстового редактора (в ИСР)

4. Интеграция с отладчиком:

- отображение контрольных точек останова при отладке,
- отображение текущего значения объекта, при наведении курсора на идентификатор.

```
NSLog(@"################ CALL OUT ACCESSORY TAPPED: control button type = %d", ((
if (((UIButton *)control).buttonType == 2) {
    NSLog(@" ############
    leftCallOutButton.avail
GDB: Stopped after step
ed)retainCount {
  NSUIntegerMax;
elease {
nothing
orelease {
  self;
```

```
UIButton *                  control                         0x176970
  UIControl                 UIControl                       {...}
  CFMutableDictionaryRef    _contentLookup                  0x176bb0
  UIEdgeInsets              _contentEdgeInsets              {...}
  UIEdgeInsets              _titleEdgeInsets                {...}
  UIEdgeInsets              _imageEdgeInsets                {...}
  UIImageView *             _backgroundView                 0x0
  UIImageView *             _imageView                      0x196aa0
  UILabel *                 _titleView                      0x0
  struct {...}              _buttonFlags                    {...}
    unsigned int   reversesTitleShadowWhenHighlighted   0
    unsigned int   adjustsImageWhenHighlighted          1
    unsigned int   adjustsImageWhenDisabled             1
    unsigned int   autosizeToFit                        0
    unsigned int   disabledDimsImage                    0
    unsigned int   showsTouchWhenHighlighted            0
    unsigned int   buttonType                           2
    unsigned int   shouldHandleScrollerMouseEvent       1
```

Задачи отладчика в рамках ИСР

1. пошаговое выполнение программы (шаг = строка; с трассировкой внутри вызываемой функции и без нее),

2. выполнение программы до строки, в которой в редакторе стоит курсор,

3. выделение выполняемой в данный момент строки,

4. приостановка выполнения программы, при этом:

- можно запросить значение переменной,
- можно заказать вычисление некоторого выражения,
- можно изменить значение переменной и продолжить выполнение программы (но не всякий отладчик позволяет изменять программный код, т.е. поддерживает частичную перекомпиляцию),

5. расстановка/снятие точек останова, которые визуализируются в текстовом редакторе,

6. выдача всей информации в терминах исходной программы.

Стратегии тестирования

Стратегия тестирования — это метод, используемый для отбора тестов, которые должны быть включены в тестовый комплект.

Стратегия является эффективной, если тесты, включенные в нее, с большой вероятностью обнаружат ошибки тестируемого объекта. Эффективность стратегии зависит от комбинации природы тестов и природы ошибок, на поиск которых эти тесты направлены.

•***Стратегия поведенческого теста*** основана на технических требованиях. Тестирование, выполняемое с помощью стратегии поведенческого теста, называется *поведенческим тестированием, функциональным тестированием* или *тестированием черного ящика*.

•***Стратегия структурного теста*** определяется структурой тестируемого объекта. Тестирование, выполненное с помощью стратегии структурного теста, называется также *тестированием белого ящика*. Стратегия структурного теста требует полного доступа к структуре объекта, то есть к исходной программе.

•***Стратегия гибридного теста*** является комбинацией поведенческой и структурной стратегий. Модули и низкоуровневые компоненты часто тестируются с помощью структурной стратегии. Большие компоненты и системы в основном тестируются с помощью поведенческой стратегии. Гибридная стратегия полезна на всех уровнях.

Способы тестирования

- Тестирование проводится не только на той стадии разработки программ, которая специально для этого предназначена, но и на предшествующих стадиях – при автономной отладке программ, еще до объединения их в единый программный комплекс. Такое тестирование называется ***модульным***. Его обычно проводят сами разработчики, которые проверяют точное соответствие программы выданной им спецификации.

- ***Интеграционное тестирование*** призвано проверить все аспекты работы программы от правильности взаимодействия внутренних программных компонентов до правильности взаимодействия программного комплекса с его пользователями.

- Во время ***пользовательского тестирования*** результаты работы программы проверяются с прикладной точки зрения.

- ***Нагрузочное тестирование*** дает возможность проверить безопасную и эффективную работу созданной программы в нормальном и пиковом режимах ее использования. Функциональность на этом этапе проверяется только в смысле ее влияния на важнейшие технические параметры программы, например, на время реакции системы на запрос пользователя.

- Важной в тестировании является возможность проведения ***регрессионного тестирования***. Регрессионные тесты, повторяемые после каждого исправления программы, позволяют убедиться, что функциональность программы, не связанная с внесенным исправлением, не затронута этим исправлением и не утрачена из-за него.

Редактор связей

Редактор связей (компоновщик) предназначен для связывания между собой (по внешним данным) объектных файлов, порождаемых компилятором, а также файлов библиотек, входящих в состав СП.

Редактор связей выполняет следующее:

- связывает между собой по внешним данным объектные модули, порождаемые компилятором и составляющие единую программу,
- связывает файлы статически подключаемых библиотек с целью получения единого исполняемого модуля,
- готовит таблицу трансляции относительных адресов для загрузчика,
- готовит таблицу точек вызова функций динамически подключаемых библиотек.

Типы библиотек

Библиотеки являются существенной частью систем программирования.

В настоящее время можно выделить 3 типа библиотек:

1. Библиотеки функций (или подпрограмм).
2. Библиотеки классов.
3. Библиотеки компонентов.

Библиотеки функций

Библиотеки функций во многом определяют возможности систем программирования в целом. Чем больше выбор библиотечных функций СП предоставляет пользователю, тем лучшие позиции она имеет на рынке средств разработки программного обеспечения.

Различают:

- *библиотеки для языков программирования* (например, функции ввода-вывода, работа со строками) и

- *библиотеки для решения задач в конкретной проблемной области* (например, функции, реализующие алгоритмы линейной алгебры).

Библиотеки функций представляют собой **откомпилированные объектные модули,** а необходимые фрагменты библиотеки функций включаются в исполняемый файл на этапе работы редактора связей**.**

Библиотеки классов

Библиотеки классов также являются важной частью современных систем программирования, базирующихся на ООЯП.

Недостаток библиотеки классов — все ее классы должны быть написаны на том же ЯП, на котором пишется программа, куда интегрируются библиотечные классы.

В библиотеке классов различают:

- *конкретные классы;*
- *абстрактные классы, иерархии классов;*
- *шаблоны классов,* иерархии шаблонов классов.

Библиотеки классов включаются в программу на этапе компиляции и компилируются со всей программой вместе.

Библиотеки компонентов

Библиотеки компонентов - это библиотеки готовых откомпилированных программных модулей, предназначенных для использования в качестве составных частей программ, и которыми можно манипулировать во время разработки программ.

Компоненты бывают **локальные** (находящиеся на той же ЭВМ, где создается ПП) и **распределенные** (расположенные на сервере и доступные по сети ЭВМ).

Примеры технологий, использующих библиотеки компонентов:

- Технология CORBA (Common Object Request Broker Architecture) от международной группы OMG позволяет использовать программные компоненты, размещённые как локально, так и дистанционно. Использование CORBA-компонент не зависит от языка, на котором они были написаны.

- Технология COM (Common Object Model) от компании Microsoft под ОС Windows позволяет использовать локально размещённые компоненты, независимо от языка их реализации. Её развитие привело к распределённой архитектуре DCOM (Distributed COM), а затем к ActiveX.

- Технология Java Beans от Sun Microsystems позволяет использовать компоненты, написанные на языке Java. Так как реализация Java-машины существует почти для всех ОС, отсутствует жёсткая привязка к конкретной ОС.

Динамически подключаемые библиотеки (ДБ)

ДБ в отличие от статических библиотек подключаются к программе не во время компиляции программы, а непосредственно в ходе её выполнения.

На этапе компоновки программы редактор связей, встречая вызовы функций ДБ, вместо процедуры связывания формирует таблицу точек вызова функций ДБ для последующей операции динамического связывания. Таким образом, процесс полной компоновки завершается уже на этапе выполнения целевой программы.

Преимущества ДБ:

- не требуется включать в программу объектный код часто используемых функций, что существенно сокращает объем кода;

- различные программы, выполняемые в некоторой ОС, могут пользоваться кодом одной и той же библиотеки, содержащейся в ОС;

- изменения и улучшения функций библиотек сводится к обновлению только содержимого ДБ, а уже существующие тексты программ не требуют перекомпиляции (этот же факт может оказаться недостатком, если при модификации функций меняется логика их работы, поэтому использование ДБ накладывает определенные обязательства как на разработчика программы, так и на создателя библиотеки).

Как правило, динамически подключаются системные функции ОС и общедоступные функции программного интерфейса (API).

Существует возможность создавать свои ДБ для использования при разработке прикладных программ.

Критерии проектирования стандартных библиотек. Требования по составу

Стандартная библиотека должна:

- обеспечивать поддержку свойств языка (например, управление памятью, предоставление информации об объектах во время выполнения программ);

- предоставлять информацию о зависящих от реализации аспектах языка, (например, о максимальных размерах целых значений);

- предоставлять функции, которые не могут быть написаны оптимально для всех вычислительных систем на данном языке программирования (например, *sqrt*() или *memmove*() — пересылка блоков памяти);

- предоставлять программисту нетривиальные средства, на которые он может рассчитывать, заботясь о переносимости программ (например, средства работы со списками, функции сортировки, потоки ввода/вывода);

- предоставлять основу для расширения собственных возможностей, в частности, соглашения и средства поддержки, позволяющие обеспечить операции для данных, имеющих определяемые пользователями типы, в том же стиле, в котором обеспечиваются операции для встроенных типов (например, ввод/вывод);

- служить основой и теоретическим базисом других библиотек.

Требования по свойствам компонентов стандартной библиотеки (1)

Компоненты стандартной библиотеки должны:

- иметь ***общезначимый*** характер (структуры данных и алгоритмы для работы с ними – стек, очередь, список, …, сортировка, поиск, копирование, …); быть важными и удобными для использования всеми программистами;

- быть настолько ***эффективными***, чтобы у пользователей библиотеки не возникало потребности заново программировать библиотечные средства;

- быть ***независимыми от*** конкретных ***алгоритмов*** или предоставлять возможность указывать алгоритм в качестве параметра;

- оставаться элементарными, чтобы не терять эффективности из-за излишних усложнений или попыток совместить различные функции в одной;

Требования по свойствам компонентов стандартной библиотеки (2)

- быть ***безопасными*** (устойчивыми к неправильному использованию, использование библиотеки не должно провоцировать ошибки, а наоборот, снижать их вероятность);

- обладать достаточной полнотой (***завершенностью***), чтобы ни у кого не возникало желания что-то заменить или доопределить;

- обладать удобной и безопасной системой умолчаний;

- поддерживать общепринятые стили программирования;

- обладать способностью к расширению, чтобы *работать с типами, определяемыми пользователем, было так же хорошо, как и со встроенными (базовыми) типами (**сочетаемость с базовыми типами данных и базовыми операциями**)*.

СП под UNIX. Координатор GNU Make.

Make существенно упрощает процесс сборки проектов.

Make отслеживает изменившиеся файлы и перекомпилирует при обращении к нему только их и файлы, связанные с ними по компиляции.

Информация о зависимостях по компиляции и необходимые команды по компиляции содержатся в файле Makefile (makefile или в файле с соответствующей структурой, имя которого задается при обращении к Make: Make -f <имя_файла>), который должен находиться в текущей директории.

Makefile состоит из последовательности записей вида:

```
цель: зависимости_по_компиляции
     команда ОС UNIX
     ...
     команда ОС UNIX
...
```

Цель - имя целевого файла или название действия.

Если обращение к Make происходит без параметра, то выполняются действия по достижению первой цели, если же параметр есть, то Make достигает цель, имя которой совпадает с именем параметра.

Если цель - имя файла, Make автоматически по дате модификации файлов, указанных среди файлов-зависимостей по компиляции, определяет, какие из них должны быть перекомпилированы и выполняет соответствующие команды.

В Makefile должны быть указаны зависимости и команды для получения как промежуточных объектных файлов, так и исполняемых файлов.

Пример 1. Makefile (для модельного SQL-интерпретатора):

```
client:  client.o
         cc -o client client.o

server:  server.o  parse.o  getlex.o  table.o
         cc -o server server.o  parse.o  getlex.o  table.o

table.o: table.c  table.h
         cc –c  table.c

parse.o: parse.c  parse.h  getlex.h  table.h
         cc –c  parse.c

getlex.o: getlex.c  parse.h  getlex.h
         cc –c  getlex.c

server.o: server.c  parse.h  getlex.h
         cc –c server.c

client.o: client.c
         cc –c  client.c

clean:
         rm *.o

all:  client   server
```

Некоторые дополнительные возможности создания Make-файлов

В начале файла можно вводить макросы для обозначения каких-либо часто повторяющихся фрагментов текста файла в виде:

имя = текст ,

а затем там, где нужно, использовать введенные имена таким образом:

$(имя) .

Некоторые предопределенные макроопределения:

`$@` - полное имя текущей цели,

`$*` - имя текущей цели без типа файла (суффикса),

`$?` - список зависимостей, которые обновились с момента предыдущего обновления цели,

`$<` - полное имя исходного файла, к которому применяется правило трансформации.

В язык для Makefile введены некоторые предопределенные правила суффиксов по умолчанию для стандартного получения результирующих файлов по исходным файлам.

Например, правило трансформации ***.c.o*** означает, что для того, чтобы получить файл с расширением *.o* (если есть файл с расширением *.c*), надо выполнить указанную команду.

Строка, начинающаяся символом '#' , является комментарием.

Пустые строки игнорируются.

Любая строка, оканчивающаяся символом '\', продолжается на следующую строку.

gcc -MM позволяет сгенерировать фрагмент make-файла с зависимостями модулей.

Пример 2. Makefile (для модельного SQL-интерпретатора):

```
cc = gcc
serv_o = server.o parse.o getlex.o table.o

client:  client.o
	$(cc) -o client client.o

server: $(serv_o)
	$(cc) -o server $(serv_o)

.c.o:
	$(cc) -c $*.c

table.c:        table.h
parse.c:        parse.h getlex.h table.h
getlex.c:       parse.h getlex.h
server.c:       parse.h getlex.h

clean:
	rm *.o

all:	client server
```

Системы контроля версий

Система контроля версий в самом общем понимании осуществляет отслеживание версий (ревизий) некоего набора объектов (например, файлового дерева).

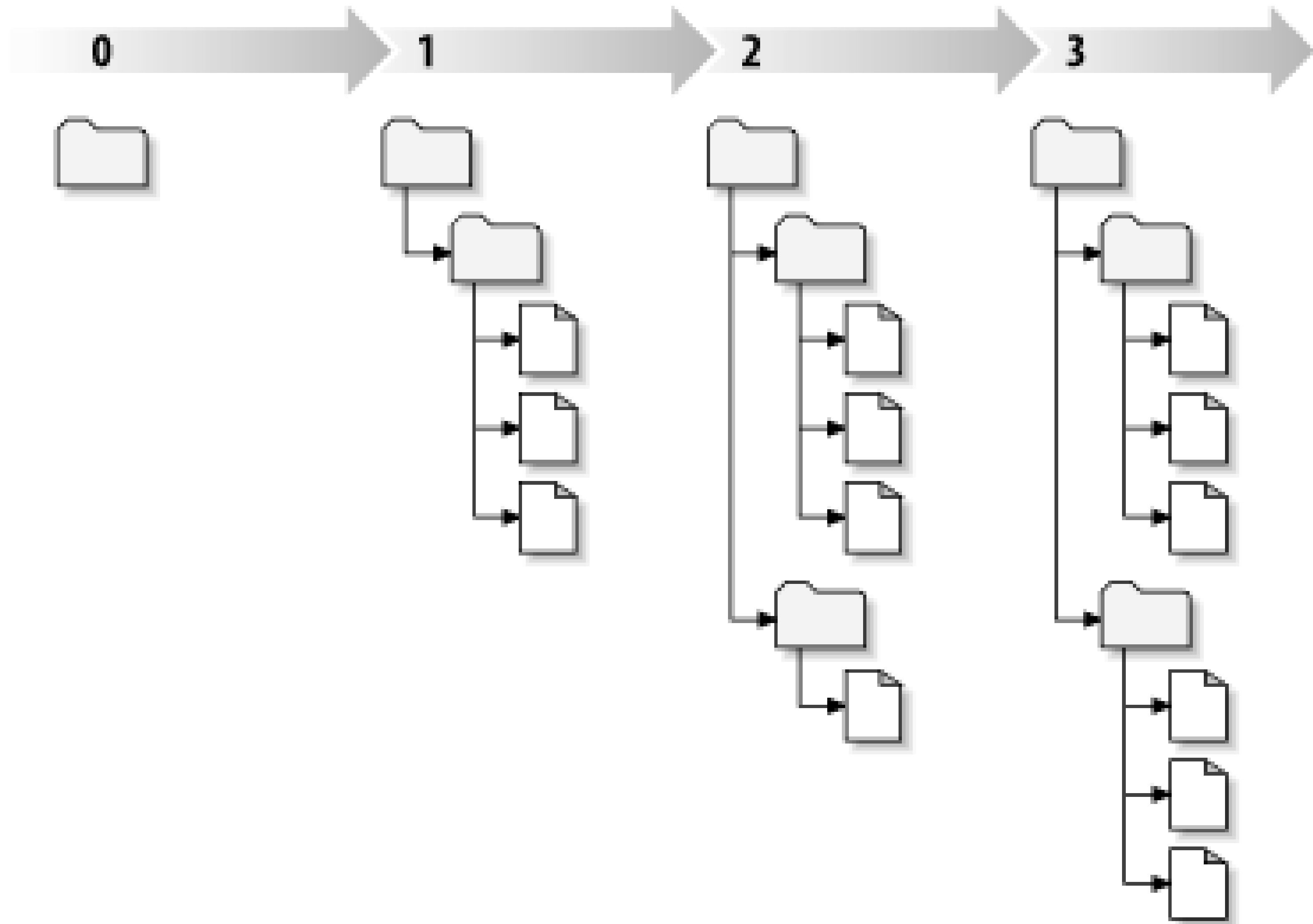
Применительно к процессу разработки ПП говорят

• об **управлении исходным кодом** (source control),
при этом отслеживаются ревизии исходного кода ПП и других ресурсов, необходимых для сборки ПП,

• или об **управлении конфигурациями** (configuration management),
при этом отслеживаются ревизии всего окружения, в том числе сопутствующих материалов, таких как файлы данных и документация.

Система контроля версий позволяет фиксировать ревизии исходного кода ПП, возвращаться к любой из них, отслеживать авторство изменений, производить анализ истории изменений и т. д.

Системы контроля версий



Развитие систем контроля версий

- Старейшие системы:
  SCCS (1972), RCS (1982)
  - отслеживается один файл на одной машине
  - рядом с файлом хранится история всех его ревизий
- Проектные клиент-серверные системы:
  CVS (1990), SVN (2000)
  - поддерживается отслеживание файлового дерева
  - фиксируется ревизия дерева в целом
  - история ревизий хранится в репозитории (на сервере)
  - работа с кодом ведется в рабочих копиях
- Распределенные системы:
  BitKeeper (1998), Darcs (2002), Git (2005), Mercurial (2005)
  - каждая рабочая копия может иметь собственный репозиторий
  - работа с репозиторием не требует доступа к серверу
  - возможна децентрализованная разработка

Контроль версий: основные понятия

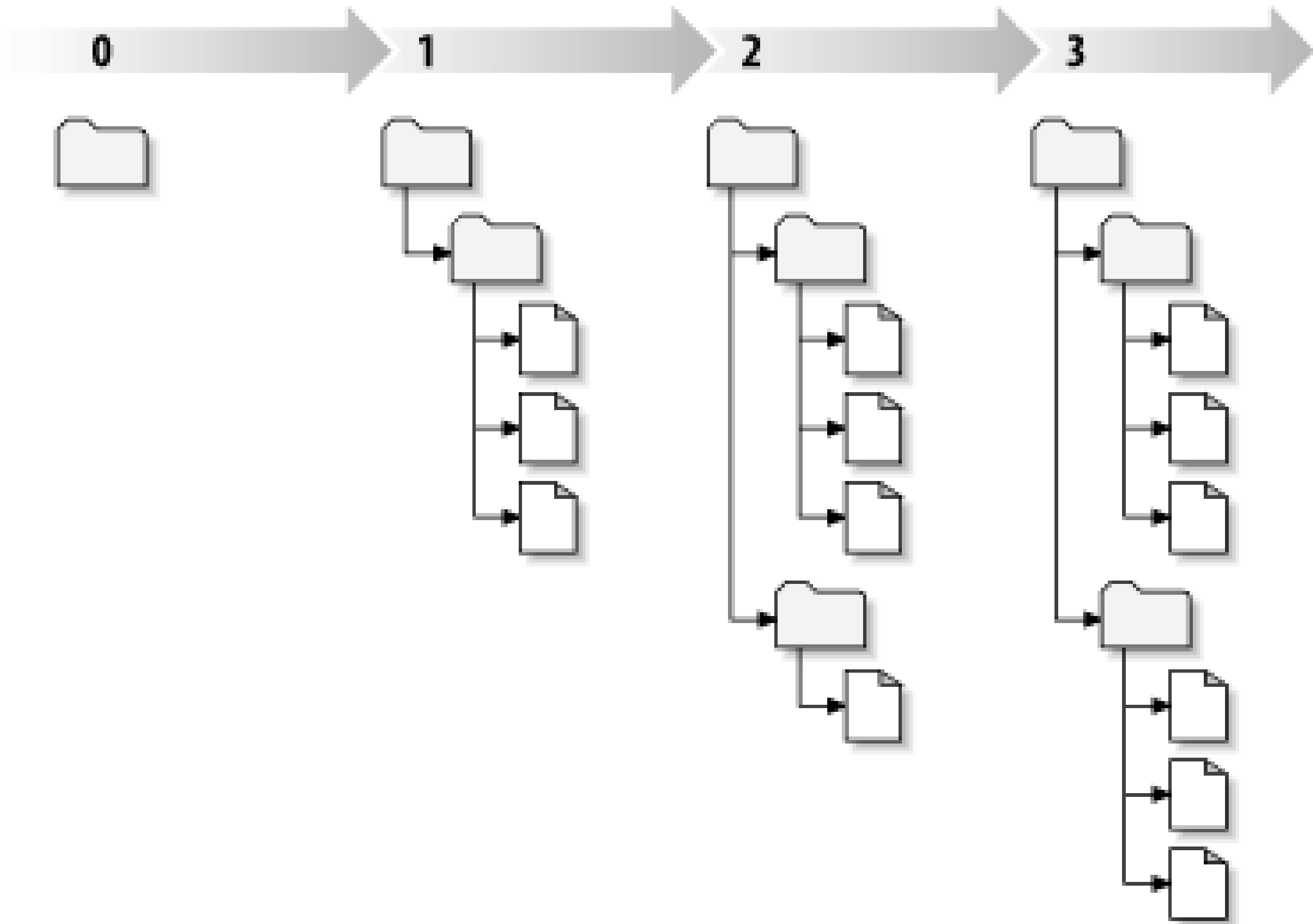
Сущности

- **Дерево**
- **Ревизия**
- **Набор изменений** (changeset)
- **Ветка**
- **Репозиторий**
- **Рабочая копия**

Операции

- **Фиксация** (commit)
- **Обновление** на ревизию
- **Ветвление**
- **Слияние** (merge)
- **Передача изменений** (pull/push)

История изменений дерева



История изменений дерева

Ветки

Контроль версий: классификация

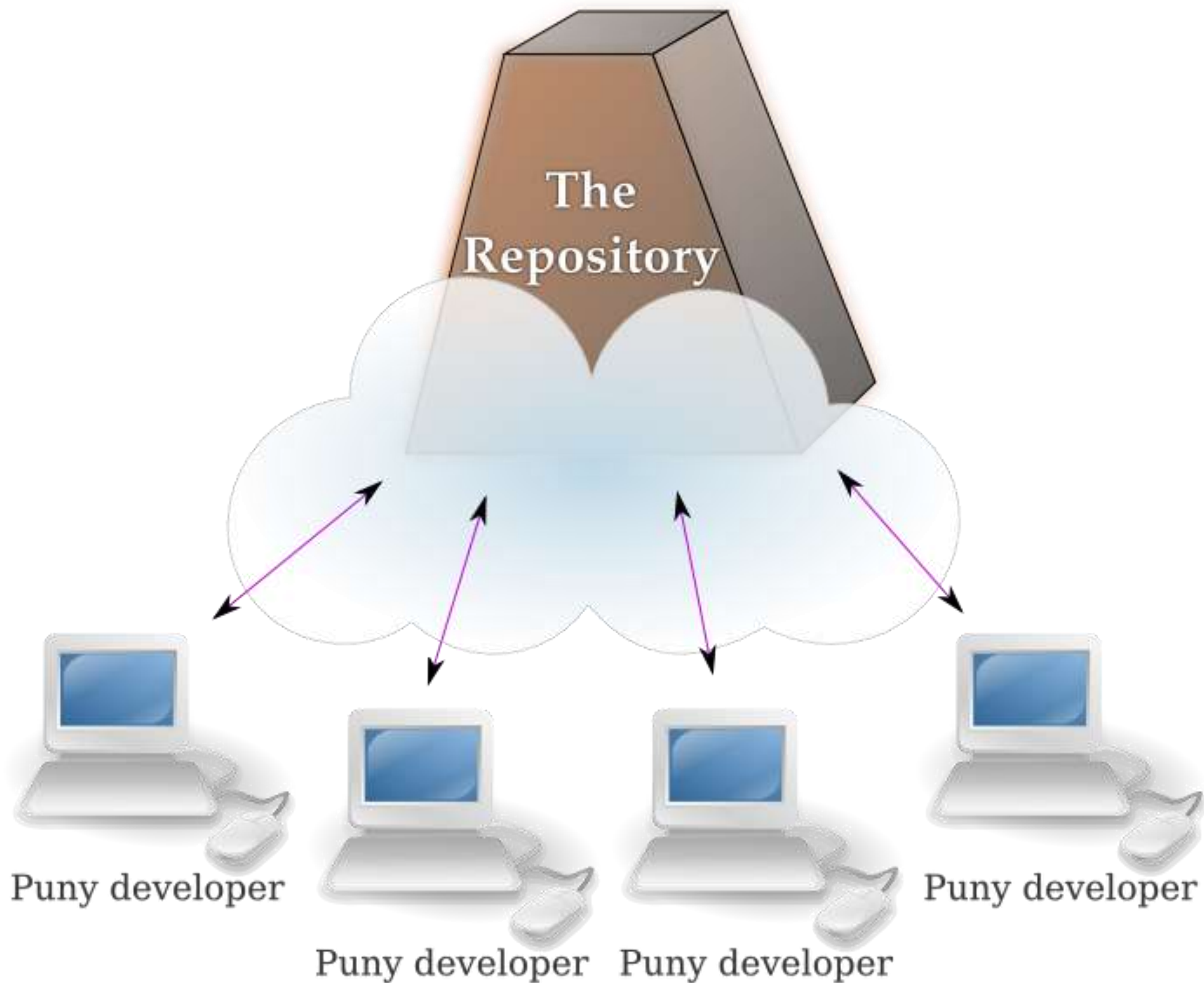
По расположению репозитория:

- централизованные (CVS, SVN),
- распределенные (Git, Mercurial, Darcs),
- комбинированные (Bazaar).

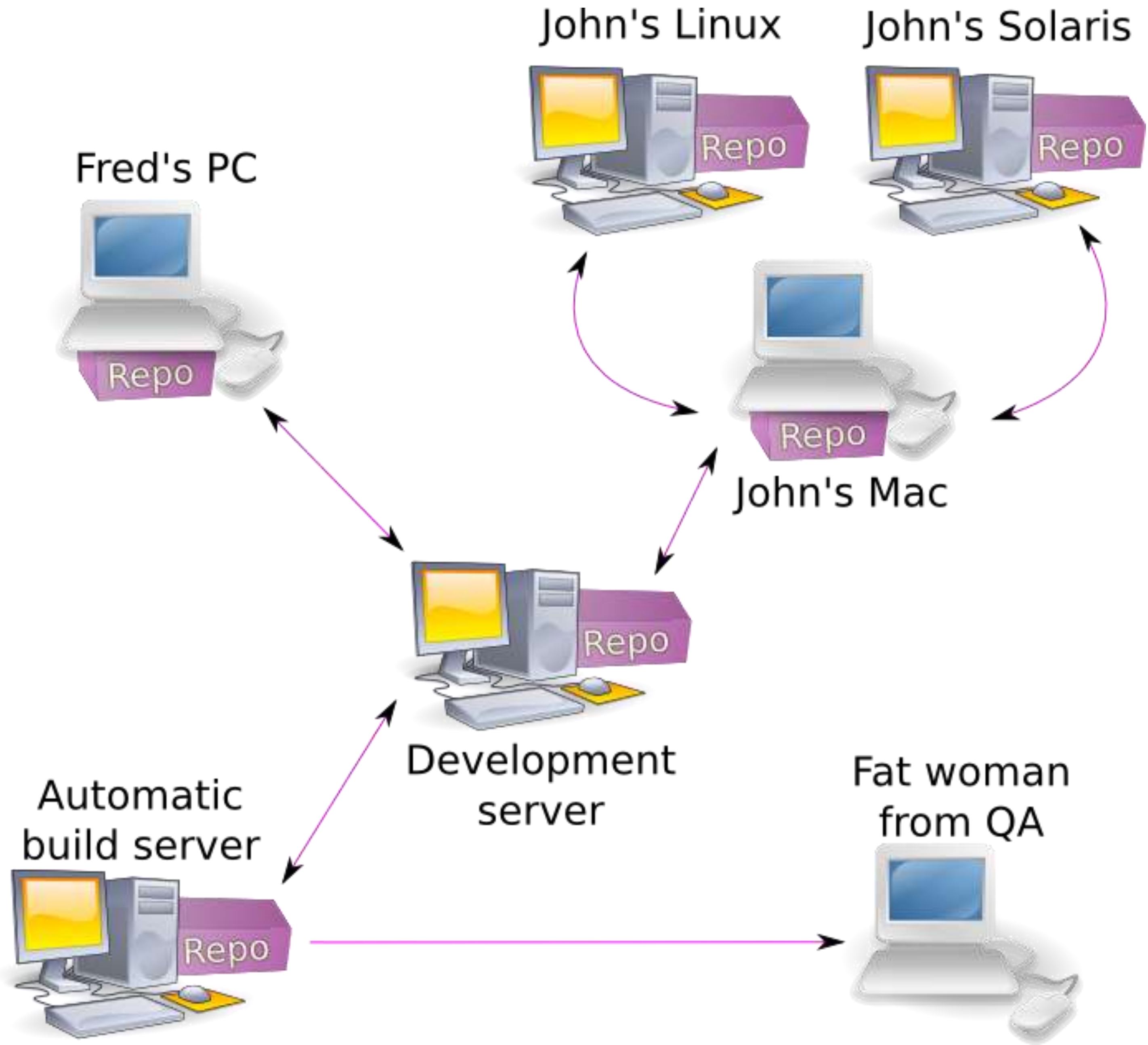
По объекту отслеживания:

- отслеживающие ревизии (CVS, SVN),
- отслеживающие наборы изменений:
  - наборы изменений организованы в ациклический орграф (Git, Mercurial)
  - наборы изменений организованы как набор патчей (Darcs)

Централизованные системы



Распределенные системы



Приемы работы с системами контроля версий

1. **Линейная работа**. Последовательная фиксация прогресса работы над ПП.
2. **Совместная линейная работа**. Совместная работа с фиксацией по принципу «кто успел» и последующим слиянием изменений.
3. **Ветки для разработки**, в которых новая функциональность дорабатывается до готовности к выпуску версии ПП.
4. **Ветки для тестирования**.
5. **Ветки для сопровождения старых версий.**
6. **Метки** (тэги) для фиксации значимых ревизий (например, версий ПП).
7. **Анализ истории** (в том числе аннотирование кода).

Режим аннотирования кода

```
[1586]   110  class ReportModule(Component):
[1]      111
[1860]   112      implements(INavigationContributor, IPermissionRequestor, IRequestHandler,
         113                 IWikiSyntaxProvider)
[1586]   114
[6901]   115      items_per_page = IntOption('report', 'items_per_page', 100,
         116          """Number of tickets displayed per page in ticket reports,
         117          by default (''since 0.11'')""")
         118
         119      items_per_page_rss = IntOption('report', 'items_per_page_rss', 0,
         120          """Number of tickets displayed in the rss feeds for reports
         121          (''since 0.11'')""")
[11096]  122
[1586]   123      # INavigationContributor methods
         124
         125      def get_active_navigation_item(self, req):
         126          return 'tickets'
         127
         128      def get_navigation_items(self, req):
[4143]   129          if 'REPORT_VIEW' in req.perm:
[5776]   130              yield ('mainnav', 'tickets', tag.a(_('View Tickets'),
[4787]   131                                                 href=req.href.report()))
[1586]   132
[11096]  133      # IPermissionRequestor methods
[1860]   134
[11096]  135      def get_permission_actions(self):
```

Популярные современные системы

1. Git
   - Высокая скорость
   - Github

2. Mercurial
   - Простота в использовании
   - Кроссплатформенность

3. Subversion (SVN)
   - Централизованная система

Когда следует применять системы контроля версий?

Всегда

CASE-средства

CASE-средства (Computer Aided Software Engineering) – программные средства, поддерживающие полуавтоматическую разработку комплексного ПП на всех стадиях его жизненного цикла.

Основные характерные особенности CASE-средств:

- наличие мощных **графических средств** для описания и документирования ПП, обеспечивающие удобный интерфейс с разработчиком и развивающие его творческие возможности;
- **интеграция** отдельных компонент CASE-средств, обеспечивающая управляемость процессом разработки программной системы;
- наличие средств автоматического или автоматизированного кодирования и документирования ПП

Современные CASE-средства

Примером наиболее известного CASE-средства является объектно-ориентированное CASE-средство

Rational Rose (компании Rational Software Corporation).

В основе работы Rational Rose лежит построение диаграмм и спецификаций унифицированного языка моделирования

UML (Unified Modeling Language),

определяющих архитектуру проекта, его статические и динамические аспекты.

UML — язык для определения, представления, проектирования и документирования программных систем различной природы.

Некоторые дополнительные возможности современных систем программирования

Существует множество других программных средств, помогающих в проектировании, модификации и кодировании программ. Например,

- системы, преобразующие программы на процедурном (императивном) ЯП в программы на ООЯП (поскольку ОО программы считаются более простыми в сопровождении, это бывает полезно),
- системы, анализирующие исходный код, с целью получения высокоуровневых абстракций (например, Java $\Rightarrow$ диаграммы UML),
- средства, предоставляющие среду разработки программ по диаграммам UML,
- средства сборочного программирования (из готовых программных модулей, официально распространены достаточно слабо в связи с различными правовыми проблемами копирования модулей, низким качеством модулей и их плохой документацией).

Статистика отмечает, что около 80% программного обеспечения создается по уже имеющемуся.

Следовательно, большой интерес представляют собой репозитории, поддерживающие архивы, документацию и интеллектуальный поиск нужных прототипов и фрагментов проектов программ для реализации эффективного сборочного программирования.

Q & A