Предмет изучения: системы программирования

- Основные определения
- Программный продукт и его жизненный цикл
- Основные компоненты систем программирования
- Трансляторы: компиляторы и интерпретаторы
- Языки программирования и средства их формального описания
- Объектно-ориентированный подход к проектированию программных продуктов
- Язык программирования Си++

Иерархия программно-аппаратного обеспечения

Составляющие систем программирования

- Языки программирования – основные средства выражения потребностей пользователей – системных и прикладных программистов
- Трансляторы – основные компоненты систем программирования, обрабатывающие исходную информацию, выраженную пользователями на языках программирования
- Другие компоненты систем программирования, являющиеся их столь же неотъемлемыми частями, как и трансляторы

Основное определение

Системой программирования называется комплекс программных средств (инструментов, библиотек), предназначенных для поддержки программного продукта на протяжении всего жизненного цикла этого продукта

Программные продукты

- ***Программа*** предназначена для решения отдельной задачи автором программы и используется в конкретной операционной среде
- Программа неотделима от её автора. Только автор способен запустить программу в рамках некоторой вычислительной среды, снабдить её необходимыми для работы данными, понять результат её работы

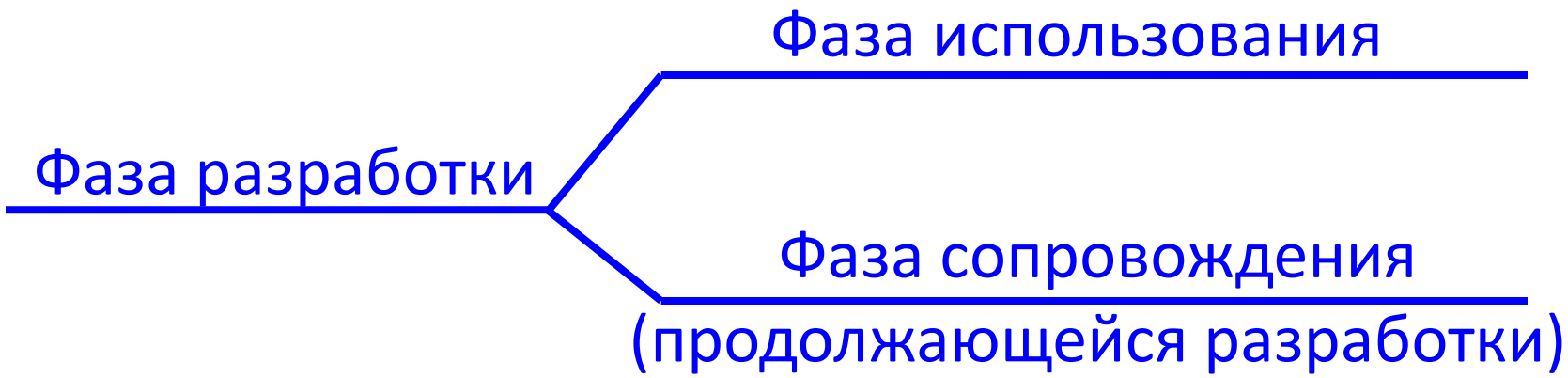
Программные продукты

- ***Программный продукт*** – это программа, которая работает без авторского надзора в рамках некоторого набора операционных сред. Программный продукт может исполняться, тестироваться и модифицироваться без участия автора (он отчуждён от автора)
- *"Дружественный"* интерфейс, наличие технической и пользовательской документации, наличие параметров настройки

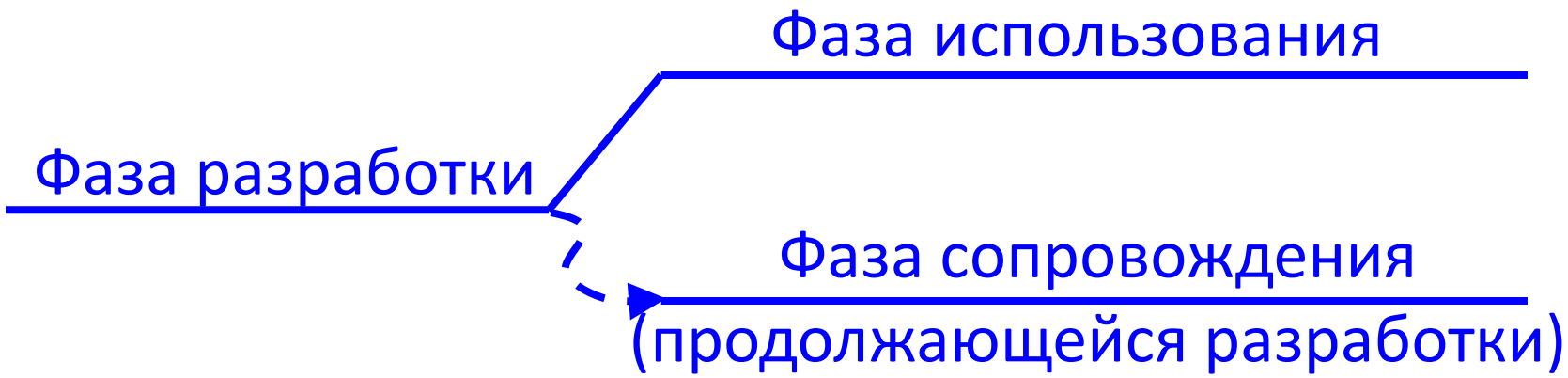
Программные продукты

- ***Интегрированный (системный) программный продукт*** есть комплекс программных продуктов (пакет)
- Согласованные интерфейсы программных продуктов, включённых в пакет. Одинаковые или похожие способы задания параметров, режимов работы и действий пользователя во всех компонентах

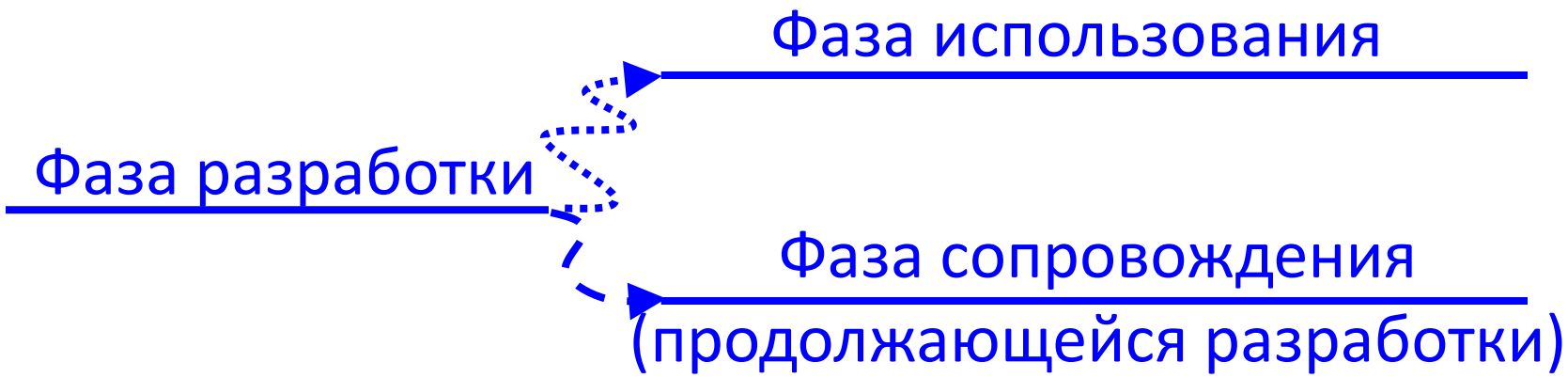
Фазы жизненного цикла программного продукта



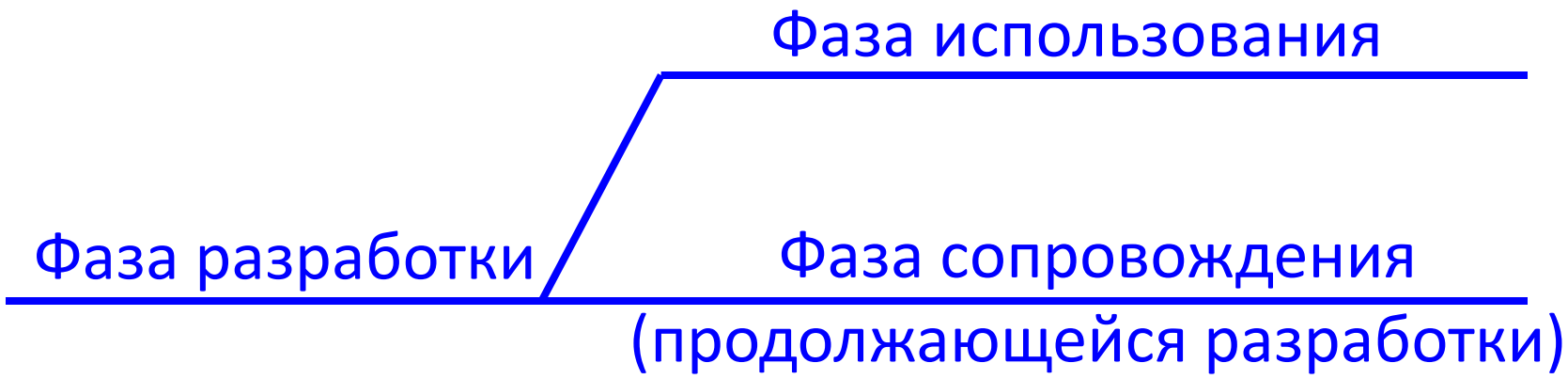
Искажение жизненного цикла программного продукта (срыв фазы сопровождения)



Искажение жизненного цикла программного продукта (срыв фазы использования)



Жизненный цикл программного продукта, сопровождаемого разработчиком



Этапы разработки программного продукта

- Анализ (определение) требований
- Проектирование
- Написание текста программ (собственно программирование, “кодирование”)
- Компоновка или интеграция программного комплекса
- Верификация, тестирование и отладка
- Документирование
- Внедрение
- Тиражирование
- Сопровождение, повторяющее все предыдущие этапы

Этапы разработки программного продукта

- Анализ (определение) требований:
  - Словарь терминов – система понятий для общения с пользователями
  - Создаваемые материалы: от текстов до формализованных описаний
  - Языки описания требований:
    - Таблицы решений
    - Функциональные диаграммы
    - Языки спецификаций (CLU, MSC, SDL, …)
  - Результат: внешняя спецификация, описание системы с точки зрения пользователя

Этапы разработки программного продукта

- Проектирование:
  - Проектирование структуры системы
  - Проектирование совокупности взаимосвязанных подсистем
  - Управление сложностью
  - Декомпозиция
    - Алгоритмическая декомпозиция
    - Объектно-ориентированная декомпозиция
  - Результат: схема иерархии подсистем, функциональность и интерфейсы каждой подсистемы
  - Результат: структуры данных и алгоритмы отдельных модулей

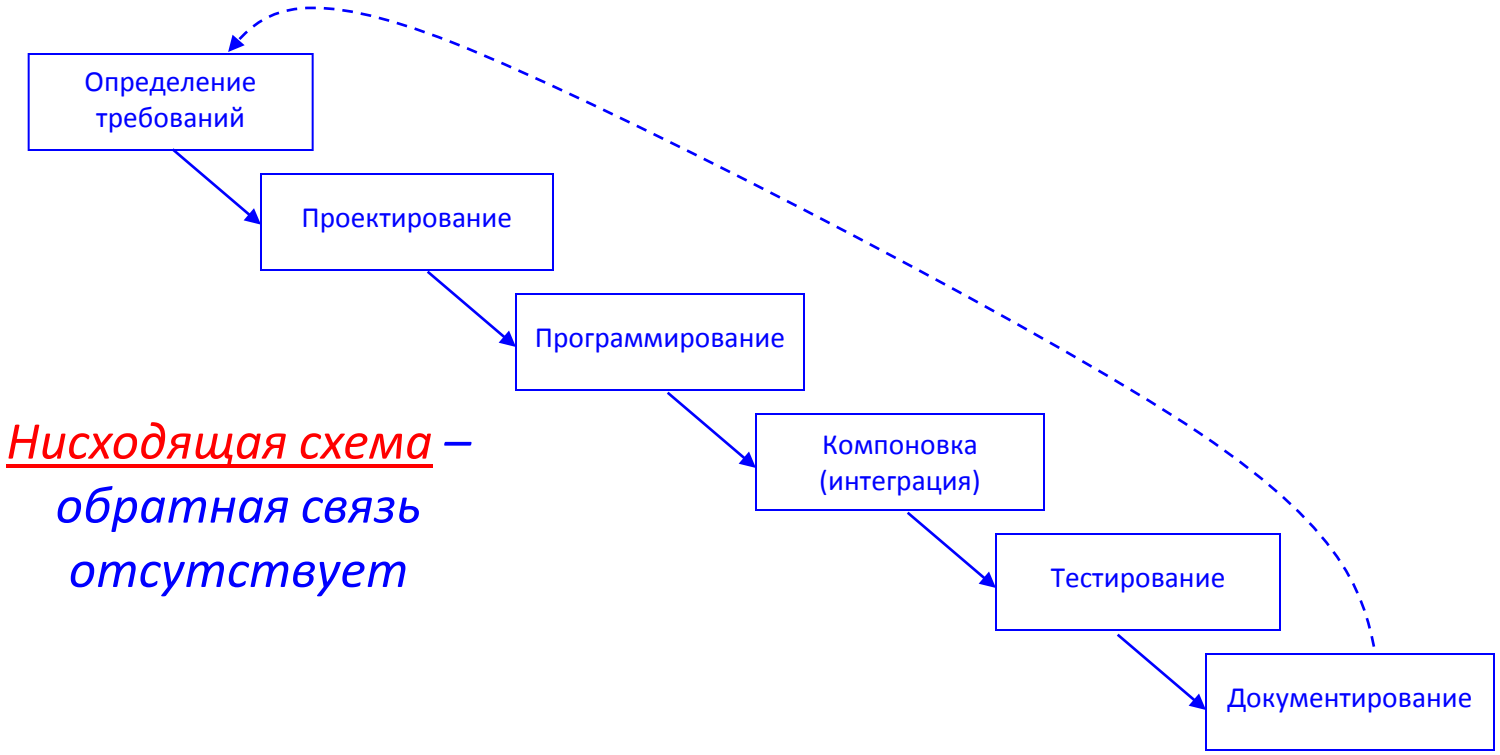
Этапы разработки программного продукта

- Написание текста программ
- Верификация, тестирование и отладка
  - Верификация – процесс проверки на правильность
    - Валидация – доказательство правильности программ с использованием логических методов
  - Тестирование – процесс обнаружения дефектов путем сравнения результатов работы программы с эталоном
    - Поведенческое, структурное, пользовательское, техническое, регрессивное тестирование и др.
  - Отладка – процесс выявления причин дефектов, а также их устранения

Этапы разработки программного продукта

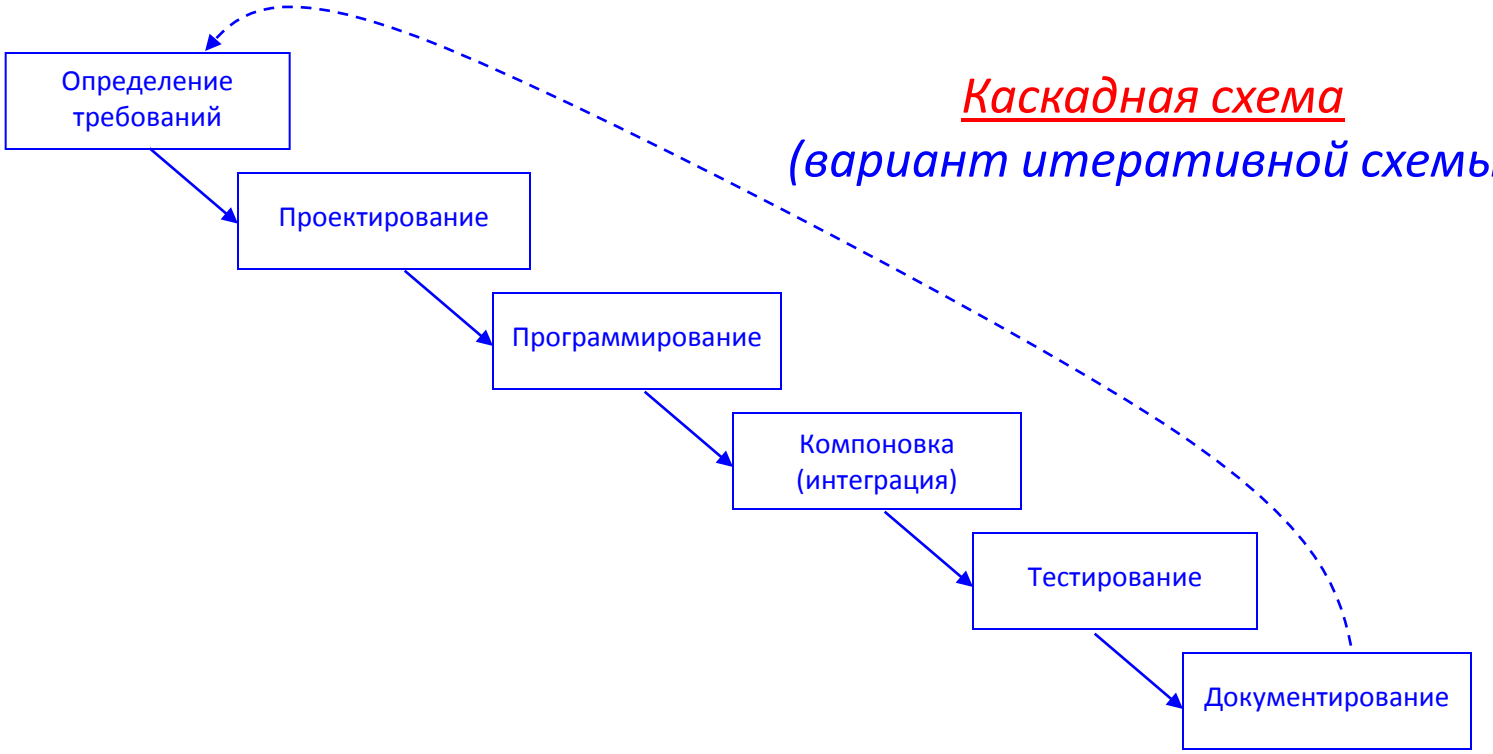
- Компоновка программного комплекса
  - Связывание отдельных частей программы в единую систему программного обеспечения
- Документирование
- Внедрение – процесс привлечения заказчика к использованию программного продукта
- Тиражирование
- Сопровождение

Идеальный случай разработки программного обеспечения



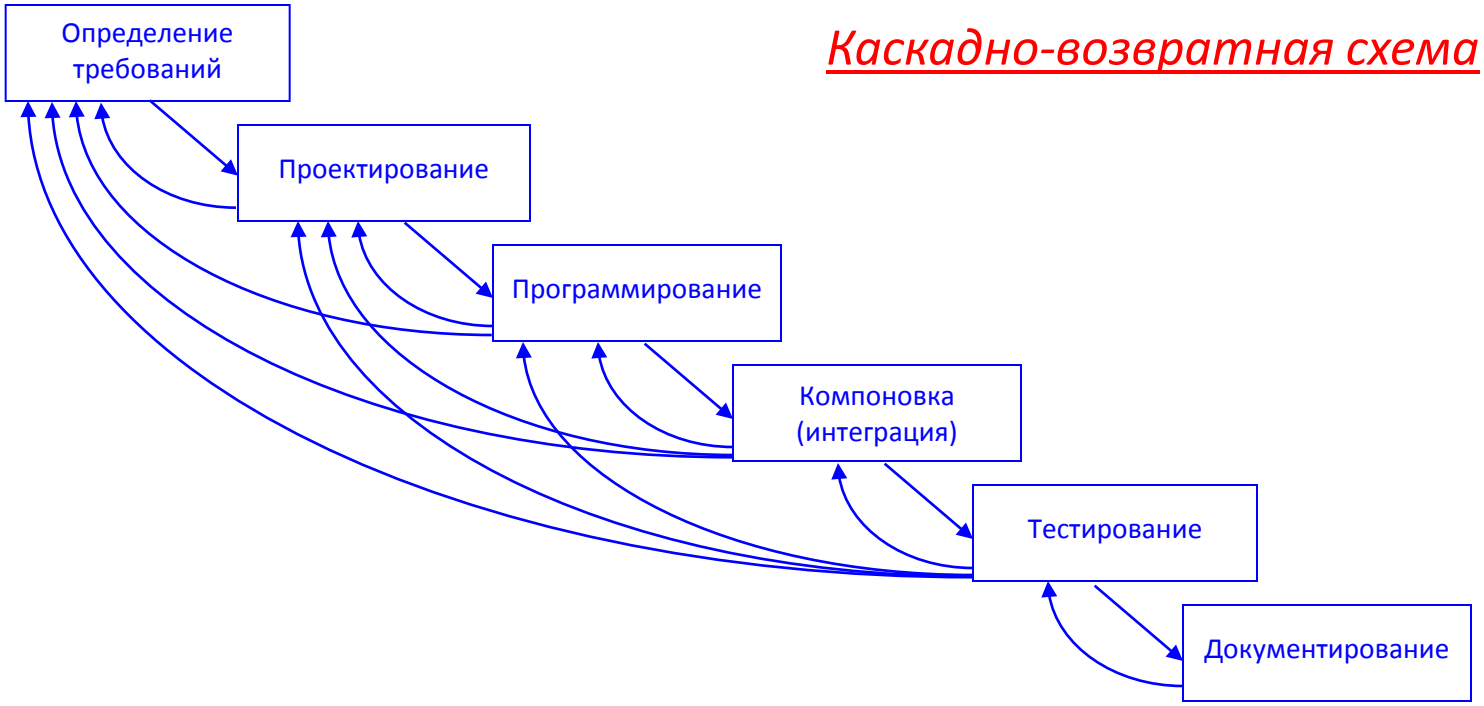
Нисходящая схема – обратная связь отсутствует

Итеративная разработка программного обеспечения



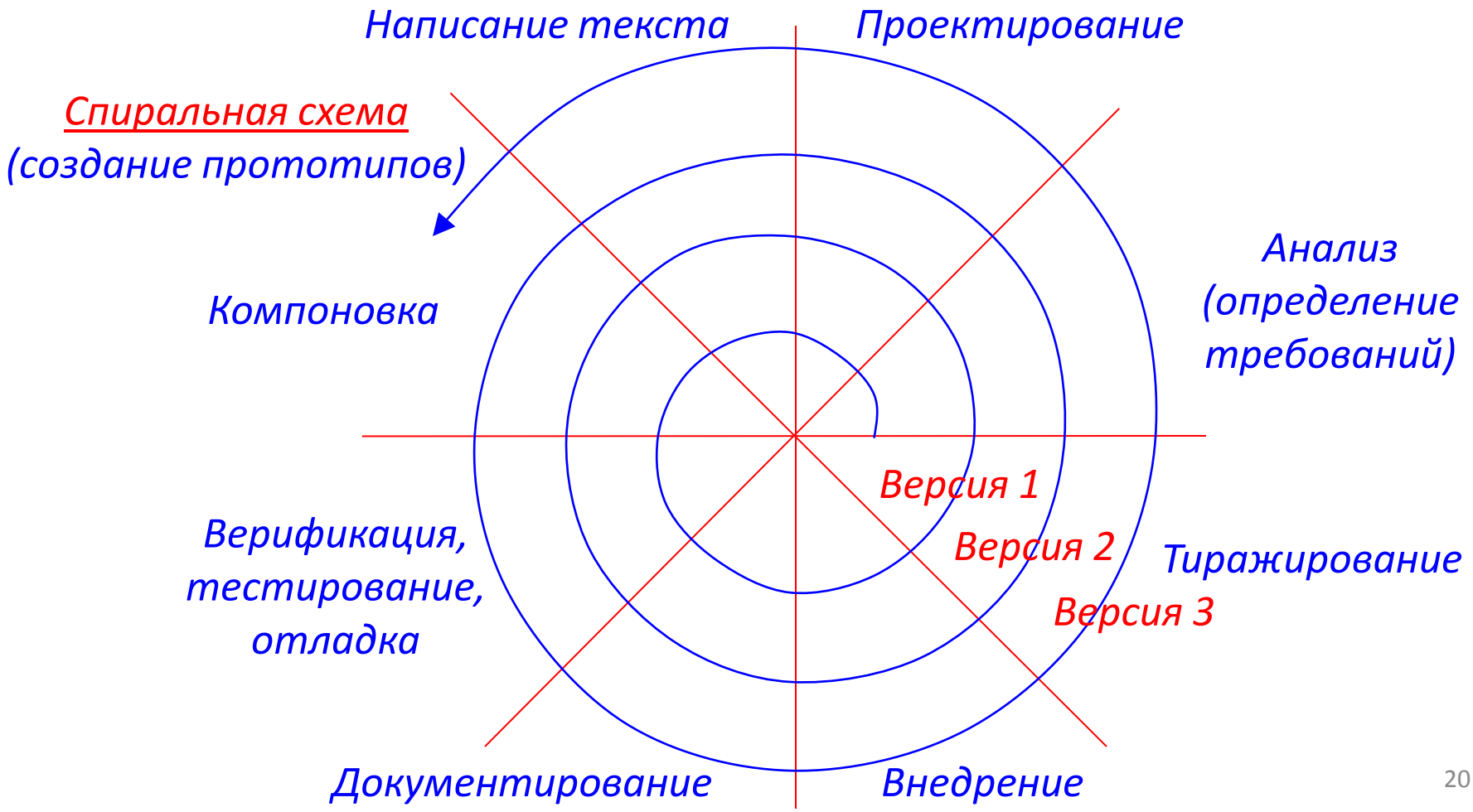
Каскадная схема
(вариант итеративной схемы)

Реальный ход разработки программного обеспечения



Каскадно-возвратная схема

Реальный ход разработки программного обеспечения



Основные требования к системам программирования

- Поддержка процесса разработки программного обеспечения на каждом этапе работы
- Согласованность интерфейсов
- Непротиворечивость результатов работы
- Полнота набора системных компонентов
- Удобство работы с системами программирования и их отдельными компонентами

Требования к системам программирования

- Процесс разработки программного продукта един, хотя он и состоит из отдельных этапов
- Содержание работ, производимых на всех этапах, базируется на результатах других этапов
- Все этапы должны быть согласованы между собой
- Для достижения этого согласования необходимо создавать и вести *базу данных проекта*

Требования к системам программирования База данных проекта

- В электронном виде поддерживается информация о проекте, истории его развития, контролируется его *внутренняя согласованность*
- Изменения требований должны указывать на места проекта, в которых эти требования использованы
- Постоянно контролируется *корректность*, *санкционированность изменений* и *непротиворечивость решений*

Требования к системам программирования Система управления проектом

- Заблаговременное планирование работ
- Предупреждение о возможных источниках затруднений (рисках)
- Управление устранимыми и неизбежными рисками
- Контроль и координация календарного плана работ
- Примеры систем: *Microsoft Project*, *TimeLine*, *SureTrack*, *Primavera Project Planner*, *OpenProj*

Требования к системам программирования Этап анализа требований

- Описание требований на формальных языках, анализ требований и их непротиворечивости
- Средства построения сетевых графиков, анализа занятости ресурсов и стоимости этапов работ
- *Основные компоненты*: текстовые и графические редакторы, средства контроля непротиворечивости таблиц решений, функциональных диаграмм, текстов на языках спецификаций

Требования к системам программирования Этап проектирования

- Обработка текстовых и графических материалов
- *Основные компоненты*: текстовые и графические редакторы, база данных проекта
- Средства автоматического построения визуальных описаний классов объектов, их просмотра и согласованного редактирования

Требования к системам программирования Этап программирования

- *Основные компоненты*: Средства автоматизации процесса написания программ и документации
- Средства автоматизации графического интерфейса пользователя
- Библиотеки
- Средства редактирования текстов программ
- Трансляторы и редакторы связей

Требования к системам программирования Этап компоновки

- Формирование программного комплекса из автономно запрограммированных, автономно отлаженных и протестированных компонентов, возможно объединённых в библиотеки
- *Основные компоненты*: редакторы связей
- Средства контроля версий программных компонентов

Требования к системам программирования Этапы отладки и тестирования

- *Основные компоненты*: Отладчики
- Генераторы тестов, позволяющие формировать входные данные для трансляторов
- Средства автоматизации прогонов тестов
- Средства автоматизации анализа результатов прогона тестов
- Средства анализа уровня тестового покрытия

Требования к системам программирования Этапы документирования, внедрения и тиражирования

- *Основные компоненты*: Средства подготовки и редактирования документации
- Средства управления проектами
- Средства управления версиями программных продуктов

Виды современных систем программирования

- *Наборы отдельных компонентов*
  - Текстовые редакторы, трансляторы, редакторы связей
- *Системы командных файлов*
  - Командные координаторы (*make*) и интерпретаторы (*shell*)
- *Интегрированные системы программирования*
  - Поддержка единой базы проектов (репозитория)
  - Поддержка визуальных методов проектирования
  - Использование унифицированного языка моделирования
  - Наличие и интеграция всех средств поддержки для всех этапов жизненного цикла программных продуктов


Классическая система программирования
Макрогенератор
Исходная программа
Система текстового редактирования
Библиотеки
Компилятор
Редактор связей (компоновщик)
Система графического редактирования
Ассемблер
Отладчик
Готовая программа
Объектный модуль
...
Загрузчик
(в составе ОС)
Выполнение

Типы трансляторов. Интерпретаторы и компиляторы

- Конечная цель создания программного продукта является достижение некоторого результата, способ получения которого закодирован в этой программе
- Этот результат может быть получен только при работе аппаратуры вычислительной системы, которой для работы передаётся программа, а также входные данные, требующиеся программе при её работе

Типы трансляторов. Интерпретаторы и компиляторы

- Варианты взаимодействия с аппаратурой в целях достижения требуемого результата:
  1. Не подразумевается никакой необходимости в системе программирования, кодирование программ ведётся непосредственно на машинном языке
  2. Программирование ведётся на языке, не совпадающем с машинным языком данной вычислительной системы, что требует наличия системы программирования, в которую должны быть включены компоненты, ответственные за преобразование исходной программы к виду, в котором она может быть понята вычислительной системой

Типы трансляторов. Интерпретаторы и компиляторы

- Преобразование исходных программ выполняется системами программирования с помощью компонентов, называемых ***трансляторами***, то есть программами, которые переводят исходную программу, написанную на некотором исходном (входном) языке, в другую программу, эквивалентную первой
- Получающаяся программа тоже формулируется на некотором языке, называемом *объектным языком*

Типы трансляторов. Интерпретаторы и компиляторы

- Процесс перевода с исходного языка на объектный язык охватывает сразу три программы и называется ***трансляцией***:
  1. При трансляции вычислительная система выполняет программу транслятора (***транслирующая программа***)
  2. Транслятор транслирует последовательность предложений входного языка, удовлетворяющую набору синтаксических и семантических правил (***транслируемая программа***)
  3. Результатом работы транслятора является программа, построенная по синтаксическим правилам выходного языка с учётом семантики выходного языка (***результирующая программа***)
- Результирующая программа полностью эквивалентна исходной программе

Ассемблеры

- Для каждой вычислительной машины имеется язык программирования, близкий к машинному языку, называемый автокодом или ***языком ассемблера***
- Программы, которые обрабатывают тексты на таких языках, называются ***ассемблерами***
- Для языков ассемблера разработан стандарт *“Standard for Microprocessor Assembly Language” IEEE 694-1985*, в котором указано, что они должны обрабатываться ассемблерами на основе принципа “один-в-один”

Компиляторы

- Термин ***компилятор*** обычно используется вместо термина транслятор в тех случаях, когда исходным языком трансляции является язык программирования высокого уровня (Паскаль или Си++), а объектным языком является автокод, язык ассемблера или машинный язык некоторой вычислительной машины
- Вычислительная система, для которой ведётся компиляция, называется ***целевой вычислительной системой***, в понятие о которой входят:
  - аппаратура ЭВМ
  - операционная система, работающая на этой аппаратуре
  - набор динамических библиотек , которые необходимы для выполнения объектной программы

Схема преобразования программ компиляторами

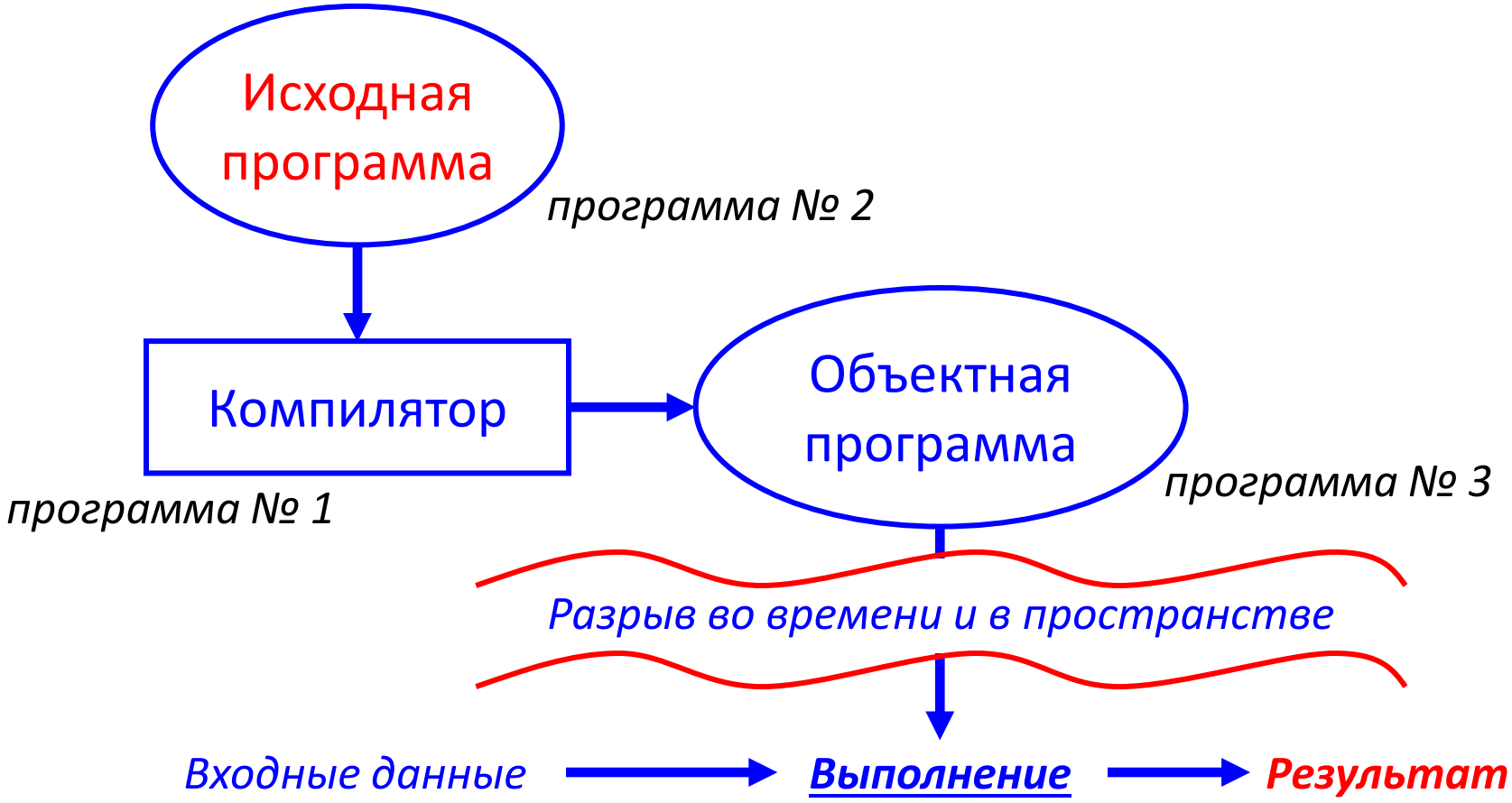
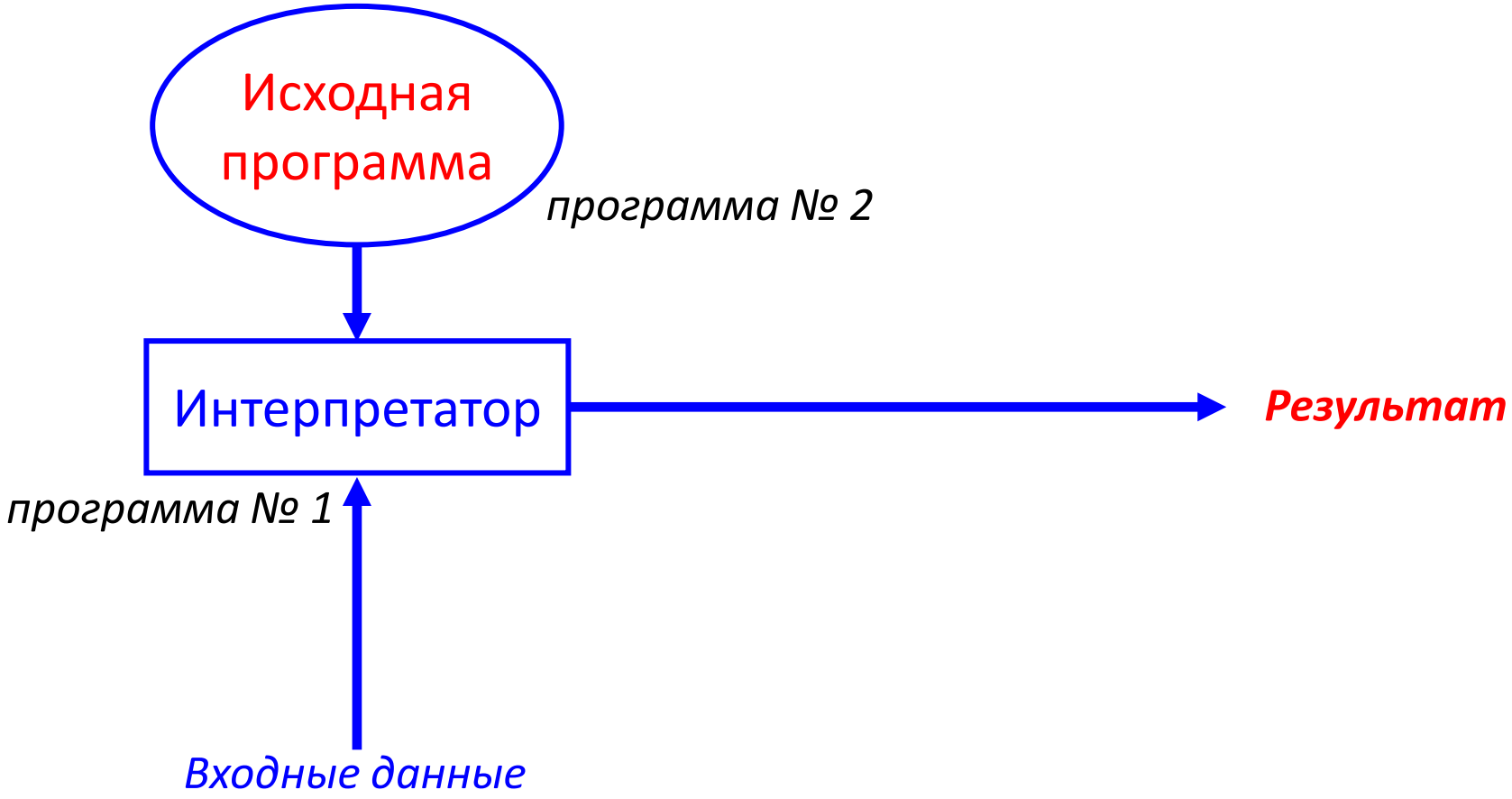


Схема получения результата программ интерпретаторами



Интерпретаторы

- *Интерпретация* выполняется программой, называемой *интерпретатором*
- При интерпретации программы она размещается в той области памяти вычислительной машины, которая предназначена для исходных данных выполняемых программ, интерпретатору также необходимы те же данные, что и при выполнении исходной программы
- В отличие от компилятора и ассемблера, *интерпретатор* исходного языка не просто обрабатывает текст исходной программы, а выполняет действия, которые этой программой предписываются

Принципиальное отличие интерпретатора от компилятора

- Интерпретатор ***не порождает объектную программу***, которая впоследствии должна выполняться, а выполняет её сам
- Итогом работы интерпретатора является результат, определяемый смыслом исходной программы, если исходная программа правильна синтаксически и семантически, либо сообщение об ошибке, в противном случае

Смешанная стратегия трансляции

- При работе интерпретаторов сначала производится преобразование исходной программы в некоторое внутреннее представление, которое затем программно интерпретируется
- Именно поэтому *интерпретаторы относят к трансляторам*
- Термин *транслятор* является самым общим и обозначает, как *компиляторы* и *ассемблеры*, так и *интерпретаторы*

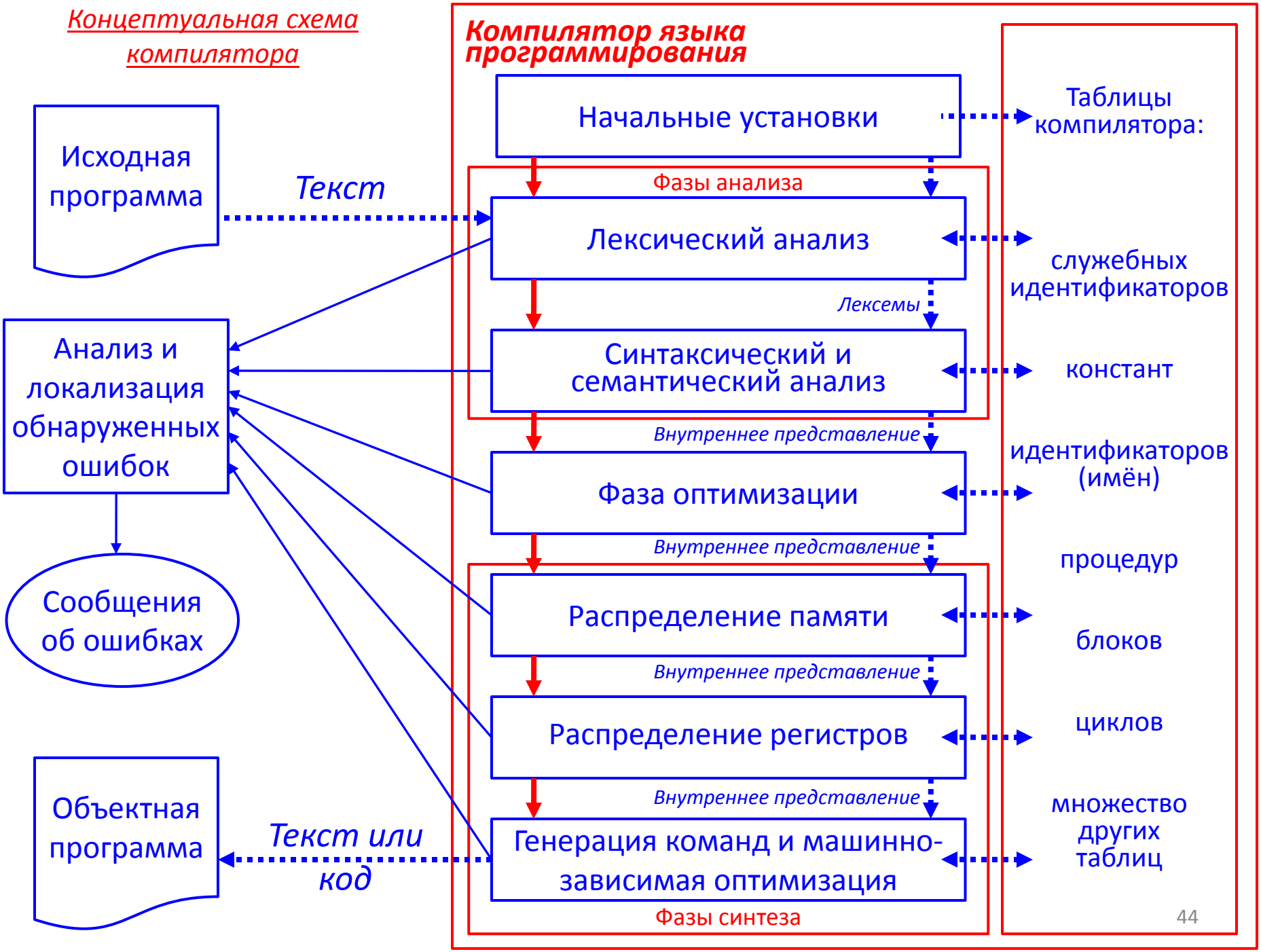

Концептуальная схема компилятора
Компилятор языка программирования
Начальные установки
Таблицы компилятора:
Исходная программа
Текст
Фазы анализа
Лексический анализ
служебных идентификаторов
Лексемы
Анализ и локализация обнаруженных ошибок
Синтаксический и семантический анализ
констант
Внутреннее представление
идентификаторов (имён)
Фаза оптимизации
Внутреннее представление
процедур
Сообщения об ошибках
Распределение памяти
блоков
Внутреннее представление
Распределение регистров
циклов
Внутреннее представление
Объектная программа
Текст или код
Генерация команд и машинно-зависимая оптимизация
множество других таблиц
Фазы синтеза

Схема работы компилятора

- Информационные таблицы
  - Таблицы служебных идентификаторов
  - Таблица констант
  - Таблица имён
  - Таблица процедур
  - Таблица блоков
  - Таблица циклов
  - Множество других таблиц

Схема работы компилятора

1. Начальные установки
2. Фазы анализа программ
    2.1. Лексический анализатор
        - Выделение и обработка лексем
        - Замена сложных конструкций
            - Диграфы (‘*:=*’, ‘****’, ‘*->*’, ‘*&&*’)
              и триграфы (‘*??<*’ ≡‘*(*’, ‘*??>*’ ≡‘*)*’, ‘*??=*’ ≡‘*#*’)
        - Замена знаков операций
            - (‘*AND*’, ‘*not*’, ‘*mod*’)
        - Макрорасширение
        - Исключение примечаний

Схема работы компилятора

2. Фазы анализа программ
    2.2. Синтаксический и семантический анализаторы
        - Проверка программ на синтаксическую и семантическую правильность
        - Формирование внутреннего представления для каждой составной части программы

Внутреннее представление программ

- Внутреннее представление программы в трансляторе зависит от обработки, которой должна подвергнуться программа
- В ассемблерах внутреннее представление близко к окончательному виду программы
- Внутреннее представления в компиляторах
  - двусвязные или древовидные списки
  - линейные последовательности операторов

Схема работы компилятора

3. Фазы оптимизации программ

Стратегии оптимизации

- Повышение скорости работы
- Уменьшение размеров программы

“Машинно-зависимая” оптимизация

“Машинно-независимая” оптимизация

Независимость рассматривается в терминах некоторого класса вычислительных архитектур

Схема работы компилятора

4. Фазы синтеза программ

    4.1. Распределение памяти и регистров

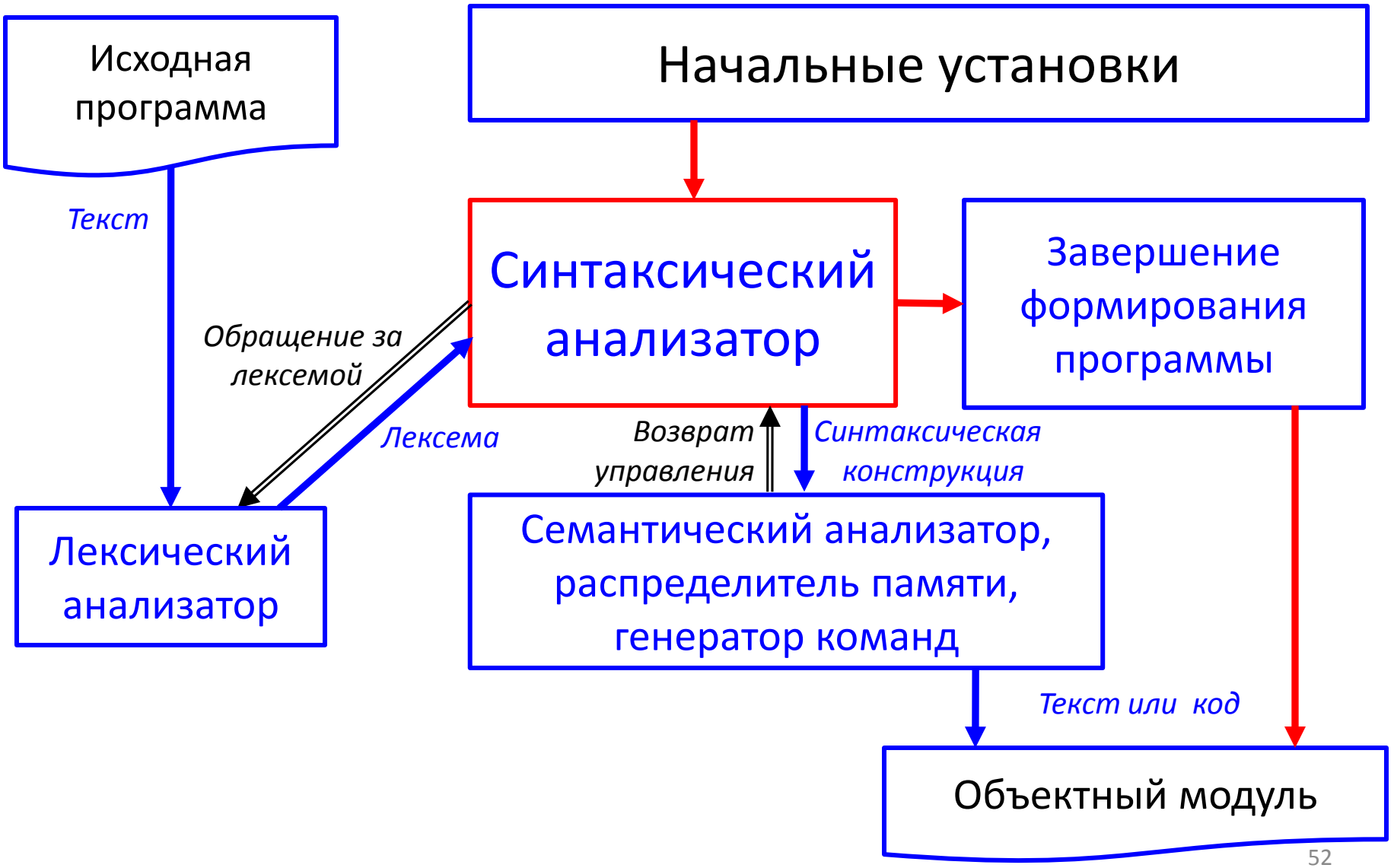
    4.2. Генерация команд и машинно-зависимая оптимизация

- В интерпретаторах фазы синтеза заменяются программой, которая фактически выполняет (интерпретирует) внутреннее представление исходной программы

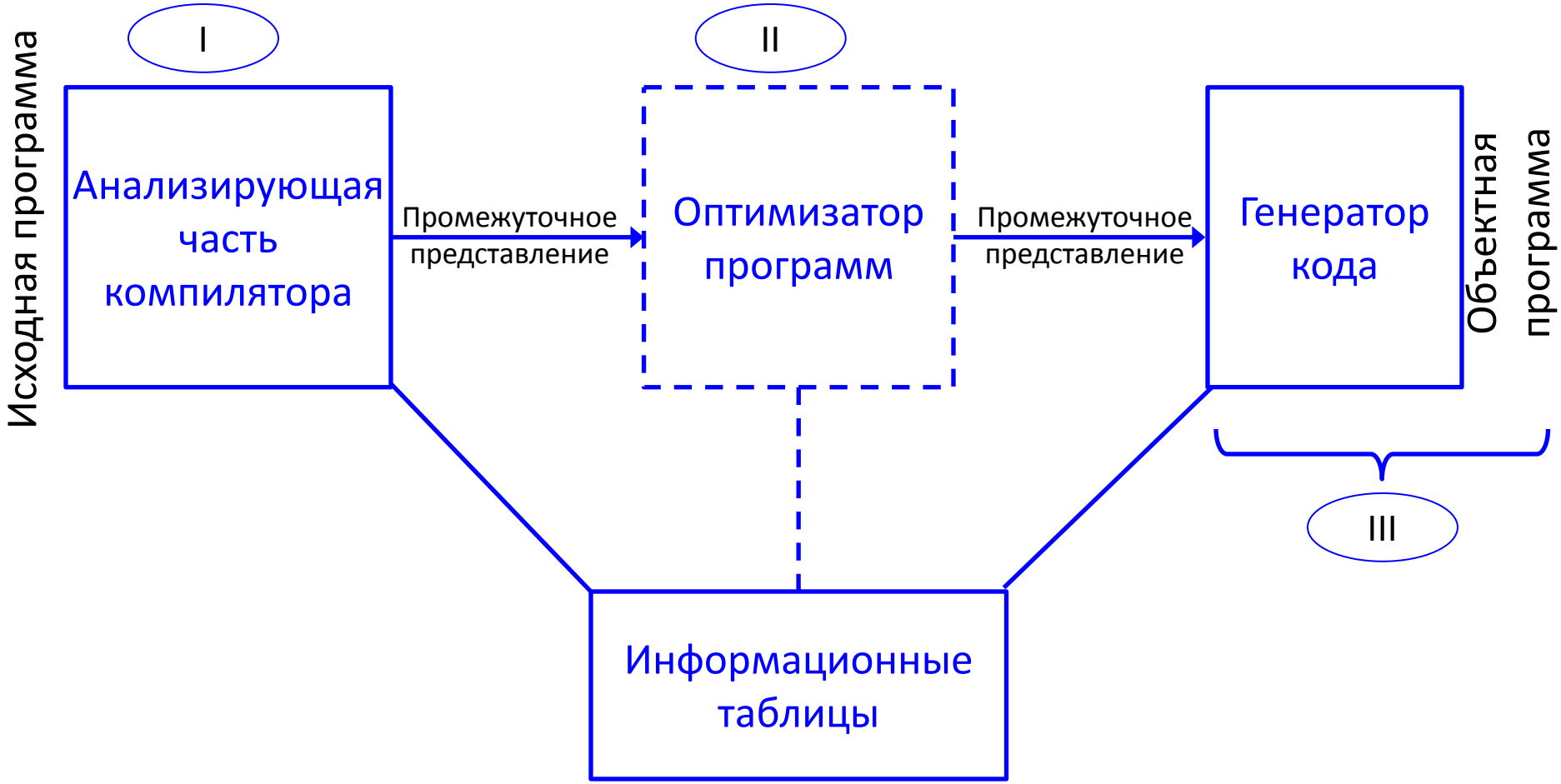
Однопроходный компилятор

- *Проход* – процесс последовательного чтения компилятором данных из внешней памяти, их обработки и записи результата во внешнюю память
- Количество проходов в разных компиляторах может исчисляться от одного до нескольких десятков
- Во время одного прохода может выполняться сразу несколько фаз компиляции, но некоторые фазы компиляции могут выполняться за несколько проходов

Однопроходный компилятор



Оптимизация в компиляторах



Выбор оптимизирующих преобразований

- Оптимизирующие преобразования должны быть *эквивалентными* (сохраняющими семантику)
- *“Стоимость” преобразования* должна быть сопоставима с “затратами” на них
- В результате преобразований характеристики *программы* в среднем *должны “улучшаться”*
- Основные виды оптимизации:
  - *Машинно-независимая оптимизация* выполняется на специально выделенной фазе компиляции (*II*)
  - *Машинно-зависимая оптимизация* проводится одновременно с генерацией объектной программы или уже после неё (*III*)

Выбор оптимизирующих преобразований

- Оптимизирующим преобразованиям подвергается внутреннее представление программы:
  1. Операции, необходимые для реализации высокоуровневых операций становятся на языках внутреннего представления явно выраженными:

     оператор исходной программы *s = s + a [i] * a [i]* скрывает, что вычисление адресов для элементов *a [i]* содержит подвыражения *sizeof (тип a [0]) * i*
  2. Внутреннее представление относительно независимо от объектной машины, что делает оптимизатор устойчивым к изменениям

Машинно-независимая оптимизация линейных участков

Линейные участки – выполняемые по порядку последовательности операций, имеющие один вход и один выход

Машинно-независимая оптимизация линейных участков

- вычисление выражений из констант на стадии компиляции
- арифметические преобразования
- устранение общих подвыражений (избыточных вычислений)
- удаление ненужных присваиваний и других операций
- распространение копий значений
- перестановка независимых смежных участков программ
- удаление недостижимых фрагментов программы
- оптимизация вычисления логических выражений

Машинно-независимая оптимизация линейных участков

- вычисление выражений из констант на стадии компиляции (свёртка операций):
  - непосредственное использование констант программистом: A = sin (2 * 3.14 * B)
  - возникновение констант-операндов после макрорасширений:
    #define Pi 3.1415926
    A = sin (2 * Pi * B)
  - компиляция сложных языковых конструкций:
    int a [10][10][10], b [10][10][10], c [10][10][10];
    a [3][4][i] = b [8][3][k] * c [3][2][j];
    a' [((3 * 10) + 4) * 10 + i] := b' [((8 * 10) + 3) * 10 + k] *
    c' [((3 * 10) + 2) * 10 + j];

Машинно-независимая оптимизация линейных участков

- арифметические преобразования
  - выражение A = B * C + B * D
    может заменяться выражением A = B * (C + D)

- замена операций на более “простые” и эффективные:

x := y ** 2 => x := y * y

x := y * 2 => x := y + y

x := y * 2 => x := y << 1

// это уже машинно-зависимое преобразование

Машинно-независимая оптимизация линейных участков

- устранение общих подвыражений (избыточных вычислений):
  - операция линейного участка может оказаться избыточной, если ранее на этом же линейном участке уже выполнялась идентичная операция, и никакой операнд данной операции не был изменён в промежутке между двумя идентичными операциями:

```
int a [p][q][r], b [p][q][r], c [p][q][r];
a [3][4][i] = b [8][3][k] * c [3][2][j];
a' [3 * q * r + 4 * r + i] :=
b' [8 * q * r + 3 * r + k] *
c' [3 * q * r + 2 * r + j];
```

Машинно-независимая оптимизация линейных участков

- удаление ненужных присваиваний и других операций
  - если между присваиваниями переменной значений (*f = v1* и *f = v2*) не было ни одного оператора, в котором использовалось бы значение *v1*, присваивание *f = v1* является бесполезным и удаляется из программы без изменения её смысла
- распространение копий значений
  - использование некоторых переменных заменяется использованием их копий (после присваивания *f = g* вместо *f* используется переменная *g*, а присваивание удаляется из программы)

Машинно-независимая оптимизация линейных участков

- перестановка независимых смежных участков программ
  - выражение A = 2 * B * 3 *C
    можно преобразовать в такое: A = (B * C) * (2 * 3)
  - непосредственное вычисление A = (B + C) + (D + E)
    требует сохранять промежуточный результат сложения B и C, после перестановки эта память не нужна: A = B + (C + (D + E)) или A = ((B + C) + D) + E
  - перестановка целочисленных операций в выражении *I/J*K* может привести к вычислению выражения *10*3/8* вместо вычисления *3/8*10*

Машинно-независимая оптимизация линейных участков

- удаление недостижимых фрагментов программы
- оптимизация вычисления логических выражений
  - пример: операцию логического *“ИЛИ”* можно не проводить, если известно, что один из её операндов имеет значение *“истина”*
  - правильный заголовок цикла языка Си

    while (! feof (F) && ((c = fgetc (F)) != ‘\0’)) { … }
  - ошибочный оператор языка Паскаль:

if (tree <> nil) and (tree^.next <> nil) then begin … end

Машинно-независимая оптимизация вызовов процедур

- Прямая подстановка функций (***inline***) может привести к увеличению скорости работы программы
  - особенно важна для установочных процедур
- Передача параметров через регистры процессора приводит к снижению времени работы программы
  - сильная зависимость от вычислительной машины
  - невозможность включать в общие библиотеки
  - трудности вычисления адресов параметров
- В некоторых языках можно указывать, какие переменные следует размещать на регистрах (например, с помощью ключевого слова ***register***)
- Девиртуализация вызовов

Машинно-независимая оптимизация циклов

Цикл – ***любая***

последовательность участков программы, которая может выполняться повторно

Машинно-независимая оптимизация циклов

- вынесение инвариантных вычислений из тела цикла
- замена операций с переменными цикла
- слияние, расщепление и развёртывание циклов

Машинно-независимая оптимизация циклов

- вынесение инвариантных вычислений из тела цикла
  - цикл *for (i = 0; i < limit – 2; i ++) A [i] = B * C * A [i];* при условии, что *B*, *C* и *limit* не изменяются в теле цикла, заменяется операциями

```
D = B * C; k = limit – 2;
for (i = 0; i < k; i ++) A [i] = D * A [i];
```

  - векторная архитектура делает оптимизацию циклов машинно-зависимой: снижение времени выполнения программы можно получить, не вынося вычисления из циклов, а внося их туда

Машинно-независимая оптимизация циклов

- замена операций с переменными цикла
  - последовательность операторов

    *S = 10; for (i = 0; i < N; i ++) A [i] = i * S;*

    может быть заменена на

    S = 10; T = 0; for (i = 0; i < N; i ++) A [i] = T, T = T + S;

  - значение *A [i]* требует индуктивной переменной: изменение значения переменной цикла на *1* приводит к изменению адреса элемента массива на *sizeof (A[0])*, значит, *& A [i] ≡ A+sizeof (A [0]) * i*

Машинно-независимая оптимизация циклов

- замена операций с переменными цикла
  - для цикла:

    S = 10; for (i = 0; i < N; i ++) R = R + F (S), S = S + 10;

    переменную цикла можно исключить:

    *S = 10; M = S + N * 10;*
    while (S < M) R = R + F (S), S = S + 10;

  - этим преобразованием за счёт введения дополнительной переменной *M* удаётся исключить *N* операций сложения для переменной *i*

Машинно-независимая оптимизация циклов

- слияние циклов

```
for (i = 0; i < n; i ++) { S1; }
for (i = 0; i < n; i ++) { S2; }
```

```
for (i = 0; i < n; i ++) { S1; S2; }
```

- расщепление циклов

```
for (i = 0; i < n; i ++) { if (x < y) { S1; } else { S2; } }
```

```
if (x < y)   for (i = 0; i < n; i ++) { S1; }
else         for (i = 0; i < n; i ++) { S2; }
```

Машинно-независимая оптимизация циклов

- развёртывание циклов

```
for (i = 0; i < n; i ++)    { A [i] = B [i] * C [i]; }
for (i = 0; i < n; i += 2) { A [i] = B [i] * C [i];
                             A [i+1] = B [i+1] * C [i+1];
                           }
```

- для целочисленных переменных *A* и *B* цикл

```
for (i = 1; i < 100; i ++) { … A = i * B; … }
```

может быть преобразован к виду:

```
for (i = 1; i < 100; i ++) { … A = A + B; … }
```

Машинно-зависимая оптимизация

1. Учёт регистровой структуры вычислительной аппаратуры
2. Удаление излишних команд
3. Оптимизация потока управления и удаление недостижимых участков программ
4. Снижение “стоимости” программы,
5. Использование машинных идиом, слияние, дробление и развёртывание циклов
6. Учёт векторных и конвейерных свойств архитектуры

Машинно-зависимая оптимизация

- Задача распределения регистров и оптимизации их использования есть NP-полная задача
- Методы распределения регистров:
  - “жёсткое” распределение
  - распределение на основе анализа графа потока управления
    - узлы – базовые блоки программы
    - дуги – переходы между базовыми блоками
  - распределение на основе раскраски графа взаимодействия регистров

Машинно-зависимая оптимизация

- Распределение на основе раскраски графа взаимодействия регистров:
  - число регистров полагается равным числу переменных
  - два узла (регистра) соединяются дугой, если два регистра должны хранить некоторые значения одновременно
  - никакие соседние узлы не имеют одинаковых цветов
  - число цветов соответствует числу реальных регистров
  - если цветов не хватает, узлы итеративно удаляются
  - максимально долго остаются на регистрах переменные, используемые во внутренних циклах программы

Машинно-зависимая оптимизация

- Удаление недостижимых участков программ

  ```
  #define DEBUG      0
  ...
  if (DEBUG) { ... }
  ```

- Снижение “стоимости” программы

  Деление на π = 3.1415 можно заменить умножением на величину $1/\pi$ = 0.3183

- Учёт векторных и конвейерных свойств архитектуры

  A+B+C+D+E+F => ((((A+B)+C)+D)+E)+F (один поток вычислений)

  A+B+C+D+E+F => ((A+B)+C)+((D+E)+F) (два потока вычислений), тогда операции A+B и D+E и сложения с использованием их результатов выполняются в параллельном режиме

Оптимизация программ

- Основное средство получения высокоэффективных программ – правильный выбор алгоритмов

  Замена алгоритма сортировки вставками алгоритмом быстрой сортировки приводит к изменению времени сортировки N элементов с $2.02N^2$ на $12log_2N$

  Для реальной программы сортировки 100 чисел это изменение уменьшает совокупное время работы в 2.5 раза, для 100000 чисел достигается снижение времени работы реальной программы в 1000 раз!

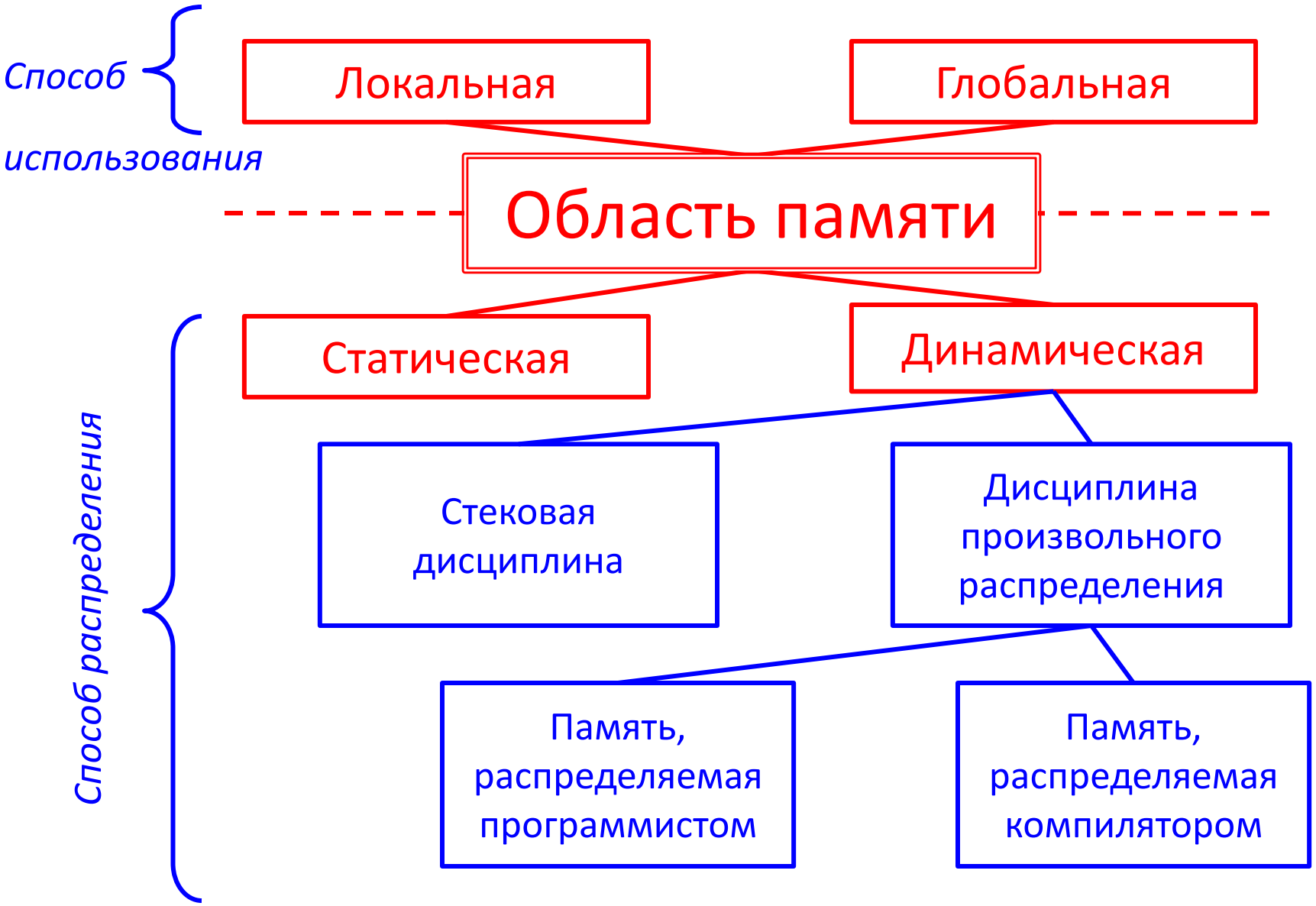
Распределение памяти

- Во время распределения памяти компилятор ставит в соответствие языковым конструкциям исходной программы *адрес*, определяет их *размер* и приписывает им *атрибуты* областей памяти, необходимых для этих языковых конструкций
- Выбор области памяти и распределение памяти в этой области проводятся
  - для объектов данных
  - для выполняемых фрагментов программ
    - операторов
    - блоков
    - функций и процедур

Распределение памяти

- Область памяти есть совокупность объединённых между собой элементов памяти, причём логика объединения задаётся семантикой входного языка
- Области памяти необходимы для хранения:
  - кодов пользовательских программ и данных, необходимых для работы этих программ
  - кодов системных программ, обеспечивающих поддержку пользовательских программ в период их выполнения
  - записей о текущем состоянии процесса выполнения программ (например, записей об активации процедур)

Характеристики областей памяти



Критерии выбора стратегии и дисциплины распределения памяти

- Эффективность начального распределения памяти
- Эффективность восстановления статуса “свободной памяти”
- Эффективность уплотнения свободных участков областей памяти (эффективность объединения свободных фрагментов во фрагменты суммарного размера)

Выбор стратегии и дисциплины распределения памяти

- Для кодов пользовательских программ, кодов системных программ, буферов ввода/вывода
  - *статические области памяти и стратегия статического распределения*

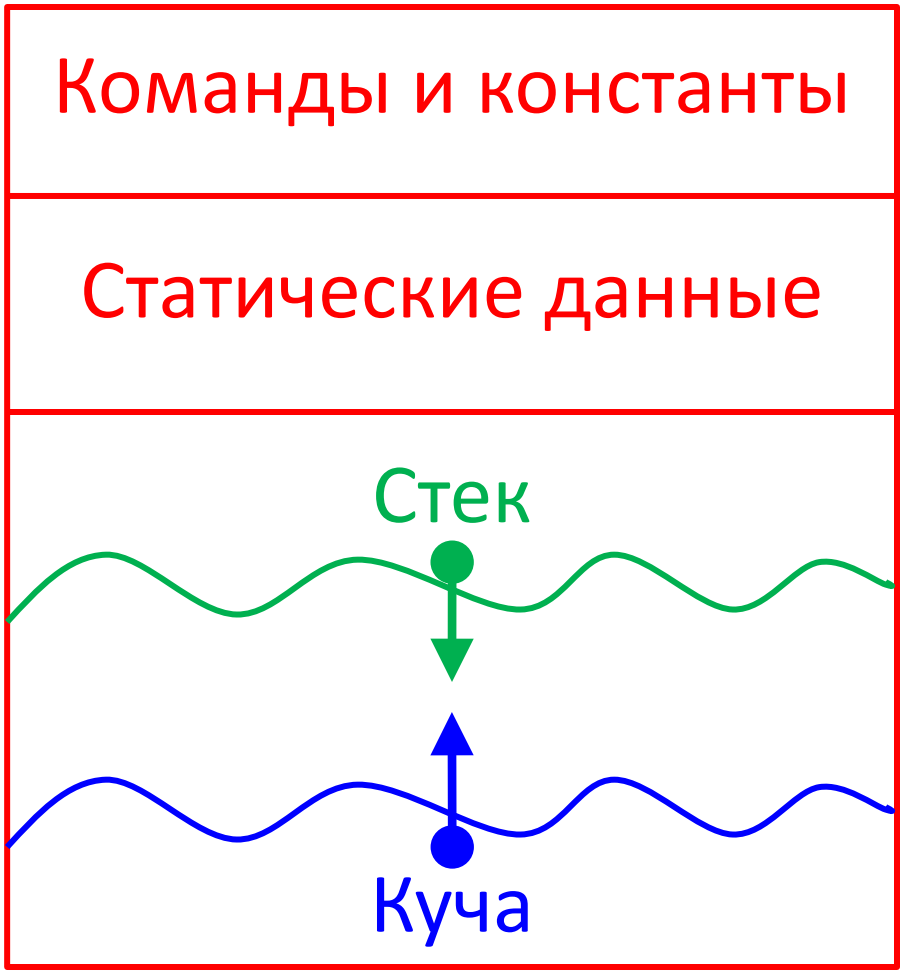
Выбор стратегии и дисциплины распределения памяти

- Для глобальных, статических переменных, констант, внутренних структур данных
  - *статические области памяти и стратегия статического распределения*
- Для локальных переменных
  - *динамическая стратегия со стековой дисциплиной*
- Для переменных, создаваемых по явному запросу
  - *динамическая стратегия с дисциплиной произвольного распределения* –

Выбор стратегии и дисциплины распределения памяти

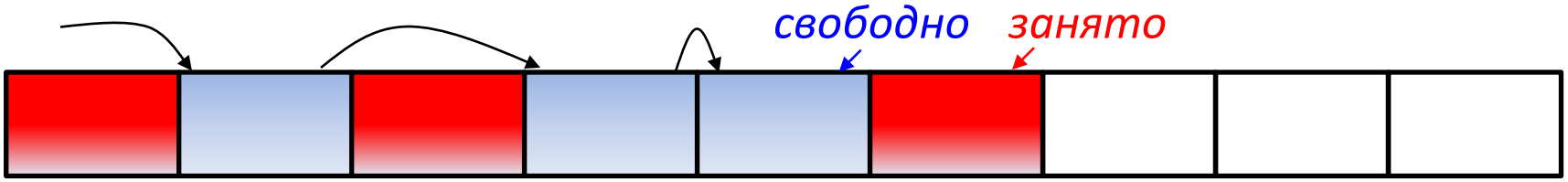
- Для записей о текущем состоянии процесса выполнения программ, а также для записей о входах в блоки операторов, которые по сути есть процедуры без параметров
  - *статическая стратегия с выделением фиксированных зон в памяти* (для каждой нерекурсивной процедуры и каждого блока нерекурсивных процедур)
  - *динамическая стратегия со стековой дисциплиной* (для языков программирования, поддерживающих рекурсивные процедуры)

Структура размещения отдельных областей в памяти

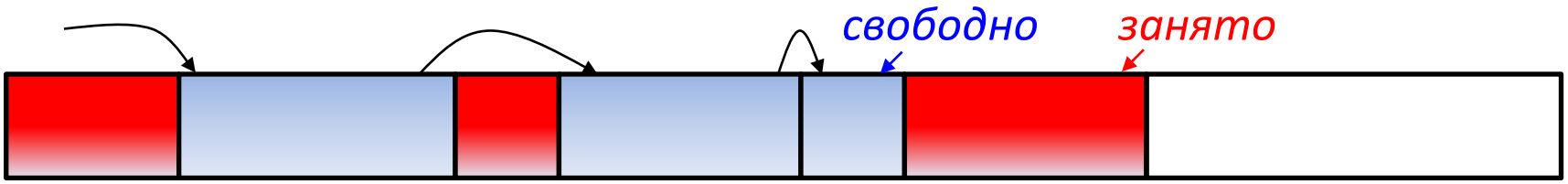


Технологии динамического распределения памяти

- Выделение блоков памяти стандартного размера



- Выделение блоков памяти, размер которых задан параметром запроса



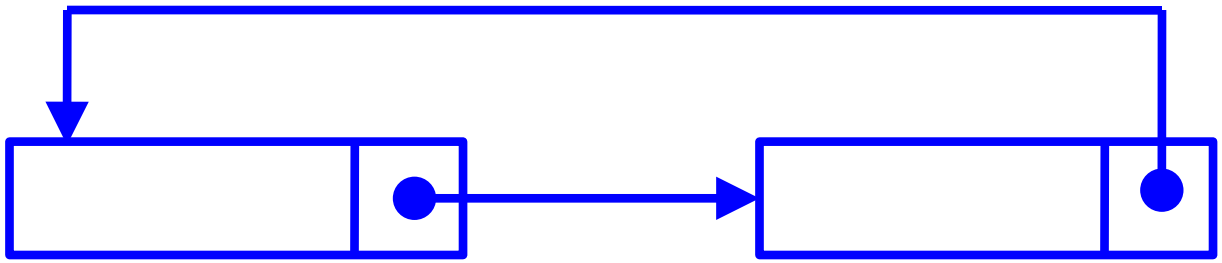
Типовая структура блока памяти

Технологии динамического распределения памяти

- Определение занятости блоков памяти с помощью счётчиков ссылок

p = q

- Проблема недостижимых циклических ссылок



Технологии динамического распределения памяти

- Определение занятости блоков памяти с помощью пометок
  - все ранее выделявшиеся блоки помечаются как свободные
  - анализируются все указатели и помечаются занятыми все блоки, на которые они ссылаются
  - итеративно анализируются все указатели, хранящиеся в виде значений полей объектов, размещённых в блоках, помеченных, как занятые; процесс останавливается, когда новые занятые блоки перестают возникать
  - все недостижимые по указателям блоки памяти остаются свободными, занятая ими память уплотняется, модифицируются значения действующих указателей

Основные компоненты системы программирования

- Средства интеграции компонентов системы программирования
- Редакторы текстов, включая универсальные макропроцессоры
- Трансляторы: интерпретаторы, компиляторы и ассемблеры вместе с препроцессорами
- Библиотеки и редакторы связей
- Загрузчики (в составе ОС)
- Средства конфигурирования
- Отладчики
- Средства тестирования
- Профилировщики
- Справочные системы

Управление вызовом системных программ

- Режим работы с командной строкой оператора или операционной системы:

```
gcc -c -S -da -dp -dA b.cpp
```

- Концепция командных файлов:

```
@echo off
@if exist t40.* del t40.*
@if exist kopu.?n del kopu.?n
@if exist kopu.?n0 del kopu.?n0
@..\..\exe\ac40 -d-l -itest.a40 -zt40.o40
@..\..\exe\rsv -it40.o40 -zt40.r40
@..\..\exe\nstr -k0x0 -p0x0 -it40.r40 -zt40.n40
@..\..\exe\pnk -ikopu.n40 -w0x1000 -zt40.m40
@del t40.?40
```

Интегрированная среда разработки

- Пользователь освобождается от необходимости устанавливать параметры запуска компонентов системы
- Не требуется знания языка управления заданиями операционной системы
- Работа ведётся в терминах описания ***программного проекта*** и его характеристик
- Пользователю предоставлен графический интерфейс
- Автоматически запускаются необходимые компоненты, от которых получаются результаты работы
- Основное преимущество интегрированной среды – в удобстве работы её пользователя
- Дополнительные удобства связаны с объединением в одном технологическом процессе нескольких компонентов системы программирования – редакторов, компиляторов, отладчиков

Редакторы текстов и макропроцессоры

- Текстовые редакторы: *пакетные* и *диалоговые*
- *Пакетные редакторы* не требуют непосредственного присутствия программиста для своей работы, получая на вход исправляемый текст и пакетное задание на редактирование:

  Заменить-18/19+
  текст замены
  +
  Исключить-23/27
  Переставить-34/38/2
  Исключить-45/К

- *Макроподстановка* (редактирование по шаблонам) состоит из ввода макроопределений и обработки макровызовов

Редакторы текстов и встроенные макропроцессоры

- В более традиционных системах (PL/1, Си, Си++, Java, различные макроассемблеры) макроопределения напоминают определения функций и их формальных параметров, а макровызовы напоминают операции вызова функций с фактическими параметрами:

```
   /* Препроцессор языка Си */
#define byte0(b) ((int)((b)>>24))
#define byte2(b) ((int)(((b)>>8)&0xff))
      i = byte0 (w) + byte2 (w);
   /* Макроассемблер NM6403 */
macro CALL_BUILTIN_2x1(Func, Arg1, Arg2, Res)
delayed call Func;
push Arg1;
push Arg2;
Res = gr7;
end CALL_BUILTIN_2x1;
      CALL_BUILTIN_2x1(IDiv32, IWord, IDivider, IResult);
```

Универсальные макропроцессоры

- Макровызовом может быть произвольная строка произвольного текста, при отождествлении входной строки с шаблоном проводится выделение фактических параметров:

```
.'$'0 (+-*/)        Управляющие символы
End program.        Шаблон № 1
4'F7$
This line repeats 4 times under count control'F1$
'F8 'F0$
$
EXPR ' TO CHECK.    Шаблон № 2
Transformation 5 yields '15$
'10'37,$
Iteration on '10. Next member '30$
'F8$
$$                  Далее следуют макровызовы
EXPR (PL1,PL2,PL3,(PL41,PL42,PL43),PL5,PL6,PL7) TO CHECK.
End program.
```

Диалоговые редакторы и текстовые процессоры

- Диалоговые редакторы: *строчные* и *экранные*
- Развитием экранных редакторов являются текстовые процессоры (*Microsoft Office WORD*)
- Текстовые процессоры концентрируются на атрибутах текста (шрифт, цвет и другие), эти программы чаще всего используются для подготовки программной документации, в особенности, технической документации
- Документы тестовых процессоров можно формировать и обрабатывать программным способом, для чего в системы программирования включены специальные библиотеки, обеспечивающие доступ как к самим документам, так и к отдельным их фрагментам – абзацам, таблицам, их элементам, отдельным словам, шрифтам и так далее

Функции редактора текста в рамках интегрированной среды

- Средство отображения хода процесса разработки программ
- Некоторые функции лексических и синтаксических анализаторов компиляторов: редактор выделяет ключевые слова языка особым шрифтом и цветом, “подсвечивает” соответствующие открывающие и закрывающие скобки
- Лексический анализ “на лету” – поиск и выделение лексем входного языка в тексте программы непосредственно в процессе её создания разработчиком
- Построение таблиц идентификаторов и констант для передачи их компиляторам
- Выдача подсказок, пояснений, вариантов текста и гиперссылок
- Выдача справочной информации по семантике и синтаксису используемого языка программирования, по операционной системе и системе программирования, по библиотеке

Редакторы связей (компоновщики)

- Компилятор не создаёт полную программу: работая с текстом одной части программы, он знает о других только то, что они *должны существовать*, и их *программные интерфейсы*
- *Основное назначение* редактора связей – осуществить привязку нескольких модулей, полученных в процессе нескольких сеансов компиляции, друг к другу
- *Основные задачи* редактора связей:
  - связывание между собой объектных модулей единой программы, порождаемых компилятором
  - подготовка таблицы трансляции относительных адресов
  - статическое подключение библиотек с целью получения единого исполняемого модуля
  - подготовка таблицы точек вызова функций динамических библиотек

Загрузчики (в составе ОС)

- Компиляторы и редакторы связей работают с *относительными* адресами объектов в зонах памяти
- Загрузчики чаще оказываются составными частями операционных систем, поскольку выполняемые ими функции зависят от архитектуры вычислительной системы и конкретной физической конфигурации этой системы
- *Основная задача* загрузчика:
  - преобразование условных относительных адресов разделов памяти в истинные (*абсолютные*)
- *Настраивающий загрузчик* выполняет трансляцию адресов в момент запуска программы
- *Динамический загрузчик* работает с динамически загружаемыми компонентами библиотек и объектными модулями

Библиотеки: статические и динамические

Свойство библиотеки	статической	динамической
• Накладные расходы во время работы программы	+	−
• Независимость готовых программ от использованных библиотек	±	±
• Эффективность использования памяти	−	+
• Трудоёмкость исправления программ при изменениях в библиотеке	−	+
• Размер готовых программ	−	+

Средства конфигурирования программных комплексов

- Этапы развития и виды средств управления конфигурацией
  - конфигурирование из командной строки
  - использование командных файлов
  - работа в интегрированных средах с проектами программных комплексов
  - использование систем управления версиями программных комплексов

Системы управления версиями программных комплексов

- Примеры действующих систем:
  - Visual SourceSafe (Microsoft Visual Studio)
  - Peforce SCM (Software Configuration Management)
  - Rational ClearCase (IBM)
  - Concurrent Versions System – CVS
  - Subversion
- Основные задачи СУВ – ответы на вопросы:
  - Кто совершил данное изменение?
  - Когда они его совершили?
  - Зачем они это сделали?
  - Какие ещё изменения произошли в то же самое время?

Средства отладки программ

- Важнейшие методы отладки программ:
  - расстановка операторов выдачи промежуточных результатов работы программы
  - исследование содержимого памяти, занятой командами или данными программы,
  - использование автоматизированных средств отладки (двоичных и символьных отладчиков)

Средства отладки программ

- Отладчик организует проверочные запуски программ, способствует локализации и исправлению ошибок
- Применение отладчика позволяет:
  - проводить пошаговое выполнение по командам, строкам текста или операторам входного языка
  - выполнять программу до достижения точки остановки
  - выполнять программу до истинности логического выражения над переменными и адресами программы
  - проводить трассировку и обратную трассировку
  - выдавать диагностику в терминах входного языка
  - просматривать (и изменять) значения переменных программы и содержимое областей памяти
  - изменять текст отлаживаемой программы и продолжать отладку без полной перекомпиляции

Средства отладки программ

- Интеграция отладчиков с другими компонентами систем программирования:
  - текстовые редакторы
    - помощь при расстановке точек останова
    - помощь при делении текста на отдельные строки при пошаговом исполнении
  - компиляторы и редакторы связей
    - доступ к таблицам имён и адресов, к описаниям областей видимости
  - редактирование текста непосредственно в процессе отладки

Средства тестирования программ

- *Тестирование* – процесс сравнения результатов работы программ с заранее рассчитанными результатами выполнения тестовых примеров
- *Стратегия (методы) тестирования* — систематические методы, используемые для отбора и/или создания тестов, которые должны быть включены в тестовый комплект
  - *Стратегия поведенческого теста* (поведенческое, функциональное тестирование или тестирование чёрного ящика) основана на технических требованиях
  - *Стратегия структурного теста* (тестирование прозрачного или белого ящика) определяется структурой тестируемого объекта и требует полного доступа к структуре объекта
  - *Стратегия гибридного теста* – комбинация поведенческой и структурной стратегий

Средства тестирования программ

- *Автономное тестирование* детально проверяет каждый программный компонент ещё до объединения в единый программный комплекс, не принимая во внимание аспекты взаимодействия с другими программами комплекса
- *Комплексное тестирование* проверяет правильность взаимодействия внутренних программных компонентов и правильность взаимодействия комплекса с пользователями
- *Пользовательское тестирование* проверяет результаты работы с прикладной точки зрения
- *Техническое тестирование* проверяет безопасную и эффективную работу программы в нормальном и пиковом режимах её использования, а также функциональность в смысле её влияния на технические параметры программы

Средства тестирования программ

- *Сценарии тестирования и тестовые примеры* должны охватывать все варианты возможного поведения и реакции программы, как в режиме нормальной работы, так и в случае возникновения необычных ситуаций (компилятор тестируется не только на правильных программах, но и на программах, содержащих все возможные ошибки)
- *Средства автоматизированного тестирования* программ обеспечивают управление тестированием, высокую скорость тестирования и повторяемость тестов
- *Регрессивное тестирование* – проверка сохранности ранее имевшейся функциональности, позволяя убедиться, что не только новая функциональность работает правильно, но и старая, имевшаяся в программном продукте до внесения в него изменений, не нарушена

Профилировщики

- *Профилирование программы* – один из способов повысить эффективность её работы, то есть определить время, затрачиваемое на выполнение отдельных её фрагментов
- Профилировщик выявляет проблемы, которые решаются:
  - отказом от лишних вычислений
  - корректировкой алгоритма во избежание вызова неэффективных функций
  - отказом от многократных повторных вычислений путём хранения результатов для последующего использования
- Профилировщики выявляют фрагменты программ, влияющие на производительность, с чем отладчики не справляются
- Профилировщик позволяет настраивать поведение системы в условиях реальной эксплуатации и визуализировать события для быстрого обнаружения проблемы
- Пример профилировщика – программа *prof* ОС UNIX

Справочные системы

- Справочные системы – обширные базы данных с включёнными в них сведениями по всем интересующим пользователей вопросам
- Метод удалённой работы с документацией: тексты документов не тиражируются и не передаются пользователям, но становятся доступными через Интернет
- Интеграция компонентов систем программирования позволяет совместить работу текстовых редакторов, компиляторов и справочных систем:
  - с помощью в текстового редактора можно выделять идентификаторы программы и получать информацию об объектах, имеющих такое имя, что значительно облегчает работу пользователям, желающим быстро вспомнить знакомую им информацию

Библиотеки систем программирования

- Причины широкого распространения библиотечных средств:
  - необходимость оказывать поддержку программам во время их исполнения на вычислительной машине
  - потребность накапливать полезные программы и передавать их другим пользователям, не раскрывая деталей реализации запрограммированных алгоритмов
- По функциональному наполнению все используемые в составе современных систем программирования библиотеки классифицируются следующим образом:
  - библиотеки функций, процедур и макроопределений
  - библиотеки классов
  - библиотеки компонентов

Библиотеки систем программирования

- Необходимость оказания системной поддержки программам, проходящим обработку в системах программирования, привело к созданию первых библиотек – *библиотек системных программ*
- *Пакеты прикладных программ* выполняют весь комплекс операций по обработке информации
- В Научно-исследовательском Вычислительном центре МГУ создана библиотека численного анализа для использования с трансляторами *pgf77* и *pgcc* с языков Фортран-77 и Си, разработанными Portland Group/STM [*http://www.srcc.msu.su/num_anal/lib_na/libnal.htm*]

Пакет прикладных программ NAG *(The Numerical Algorithms Group)*

- Языки программирования Си, варианты языка Фортран – Фортран-77/90/95
- Вычислительные системы
  - Intel x86-32, Intel x86-64, Compaq Alpha Tru64, IBM RS/6000
- Операционные системы
  - Microsoft Windows, Linux, Sun Solaris, Silicon Graphics IRIX
- Трансляторы Intel Linux *pgf77*, Intel Linux *g77*
- Набор из 76 статистических расширений для моделирования и мультивариативных методов непараметрической статистики для электронных таблиц Microsoft Excel для ОС Windows 95/98/NT/2000/XP/Vista/Win7

Библиотеки классов систем программирования

- *Библиотеки классов* могут представлять собой:
  - совокупности независимых классов
  - иерархии классов
  - иерархии шаблонов классов
- *Иерархические библиотеки*
  - с общим базовым классом (например, класс *TObject* библиотеки *VCL*)
  - с набором иерархических деревьев (“лес”, например, библиотека *STL*)

Библиотеки компонентов систем программирования

- В состав *библиотек компонентов* входят законченные программные модули для построения типичных приложений
- Библиотеки компонентов *включают в себя* генераторы отчётов, компоненты для построения сводных таблиц, компоненты для построения графиков и диаграмм, компоненты для создания графических интерфейсов и т. д.
- Библиотечные компоненты имеют *общее программное ядро* и проектируются на базе единых архитектурных принципов
- Компоненты, включаемые в библиотеки, *подчиняются правилам инкапсуляции*: они имеют открытые реализованные интерфейсы, детали реализации скрываются внутри библиотек и не видны пользователям

Библиотеки компонентов систем программирования

- Отдельные компоненты поставляются в виде *двоичных модулей* (в нетекстовом виде), что делает их независимыми от конкретных систем программирования и позволяет использовать в распределённом системном окружении
- Над компонентами можно проводить *операцию контейнеризации*, то есть помещения в контейнеры, допускающие внешнее визуальное представление, поддерживается развивающуюся технологию визуального программирования (в стиле “*drag & drop*”)
- Примеры библиотек компонентов:
  - COM (*Component Object Module*) и DCOM (*Distributed COM*)
  - CORBA (*Common Object Request Broker Architecture*)
  - библиотеки серверов приложений *.NET* и *J2EE*

Требования к библиотекам систем программирования

- Основное назначение библиотеки системы программирования – быть стандартной, то есть содержать средства, необходимые *для каждой реализации* данного языка

Требования к составу библиотек систем программирования

- *Поддержка свойств языка* (управление исключениями)
- *Предоставление информации* о зависящих от реализации аспектах языка (максимальный размер целых значений)
- *Предоставление функций*, которые не могут быть написаны оптимально для всех вычислительных систем на данном языке программирования (`sqrt()`, `memmove()`)
- *Предоставление средств переносимости* программ (работа со списками, функции сортировки, потоки ввода/вывода)
- *Предоставление основы для расширения* возможностей (соглашения и средства, позволяющие обеспечить операции для данных пользовательских типов, в стиле, в котором обеспечиваются операции для встроенных типов)
- *Стандартная библиотека должна быть основой и теоретическим базисом других библиотек*

Свойства компонентов библиотек систем программирования

- *Общезначимый* характер структур данных и алгоритмов (стек, список, очередь, …, сортировка, поиск, копирование, …)
- *Эффективность*
- *Независимость* от конкретных алгоритмов, предоставление возможности указывать алгоритм в качестве параметра
- *Элементарность*, отсутствие излишних усложнений или попыток совместить различные функции в одной
- *Безопасность* (устойчивость к неверному использованию) в большинстве типичных случаев использования
- *Завершённость* в своей функциональности
- *Сочетаемость* библиотечных средств друг с другом
- Удобная и безопасная *система умолчаний*
- *Общепринятый стиль* программирования

Разработка распределённых программ

Распределённая система

Набор независимых компьютеров, представляющихся их пользователям единой объединённой системой

(определение вольное, неверное, но пригодное)

Разработка распределённых программ

1. От пользователей *скрыты различия* между компьютерами и способы связи между ними
2. Пользователи и приложения *единообразно работают* в распределённых системах, независимо от того, где и когда происходит их взаимодействие

Основные требования к распределённым системам программного обеспечения:

- Прозрачность
- Открытость
- Масштабируемость

Разработка распределённых программ

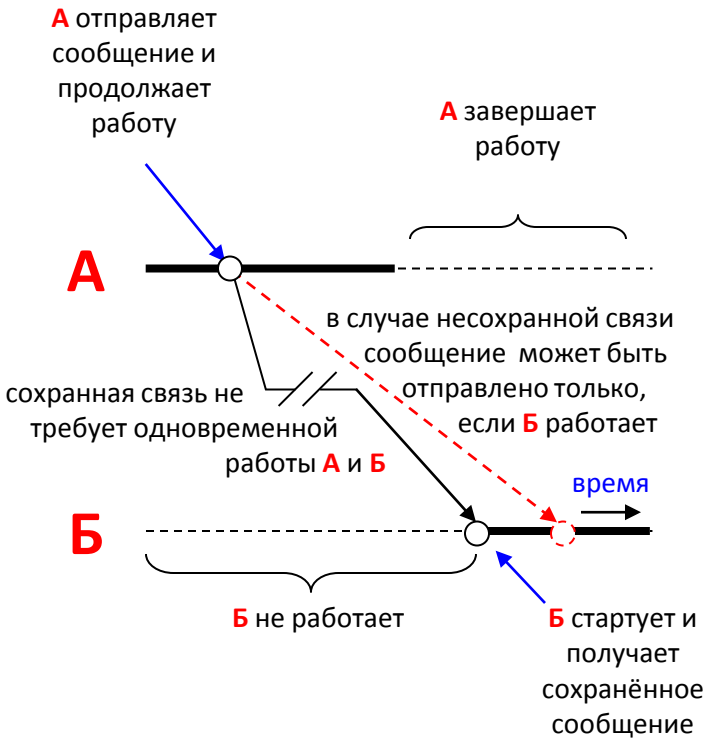
- *Прозрачностью* программной системы называется целый комплекс её свойств, благодаря которым обеспечивается сокрытие (упрятывание) особенностей реализации системы
- *Открытость* системы есть использование синтаксических и семантических правил, основанных на (открытых) стандартах
- С открытостью программных систем связаны их *переносимость* (способность приложений работать в составе разных распределённых систем) и *способность к взаимодействию*
- *Масштабируемость* может достигаться только с помощью *децентрализации* основных служб системы и управляющих алгоритмов

Разработка распределённых программ “клиент-сервер”

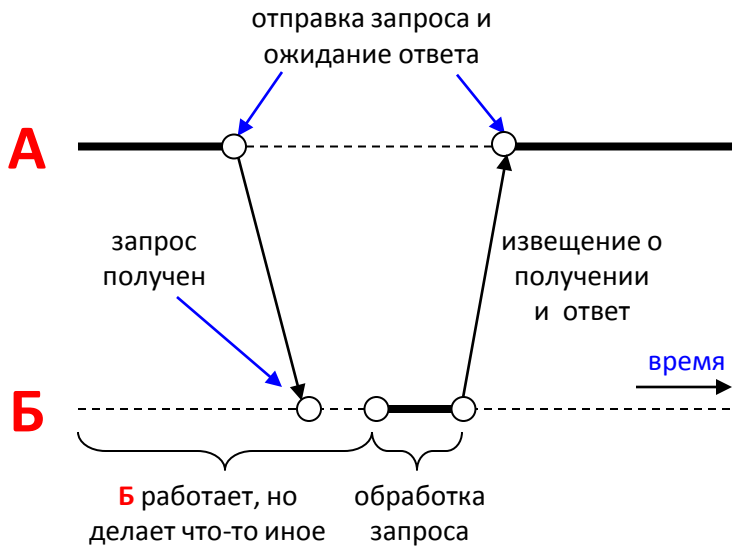
- *Клиентом называется программная система, посылающая запрос серверу на выполнение какой-либо услуги*
- *Сервером называется программная система, выполняющая задание, полученное по запросу от клиента*
- Взаимодействие между клиентом и сервером может быть *синхронным* и *асинхронным*
- *Синхронным* (*блокирующим*) называется такое взаимодействие, при котором клиент, отослав запрос, блокируется и может продолжать работу только после получения ответа от сервера
- В рамках *асинхронного* (*неблокирующего*) взаимодействия клиент после отправки запроса серверу может продолжать работу, даже если ответ на запрос ещё не пришёл

Виды взаимодействия в сети

http://sp.cmc.msu.ru/courses/sdpi/mdwrbook.pdf

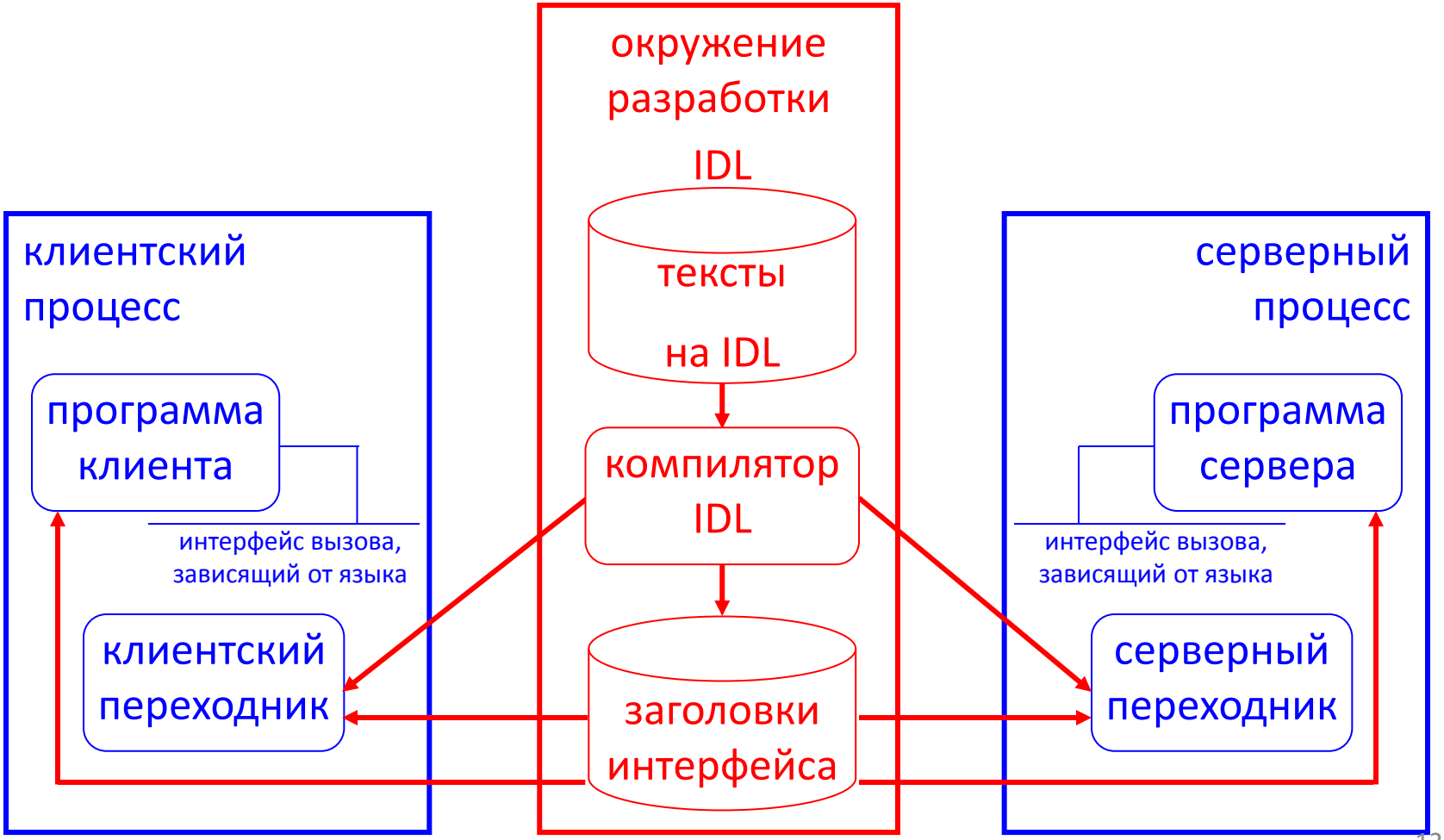


Асинхронная связь (сохранная и несохранная)

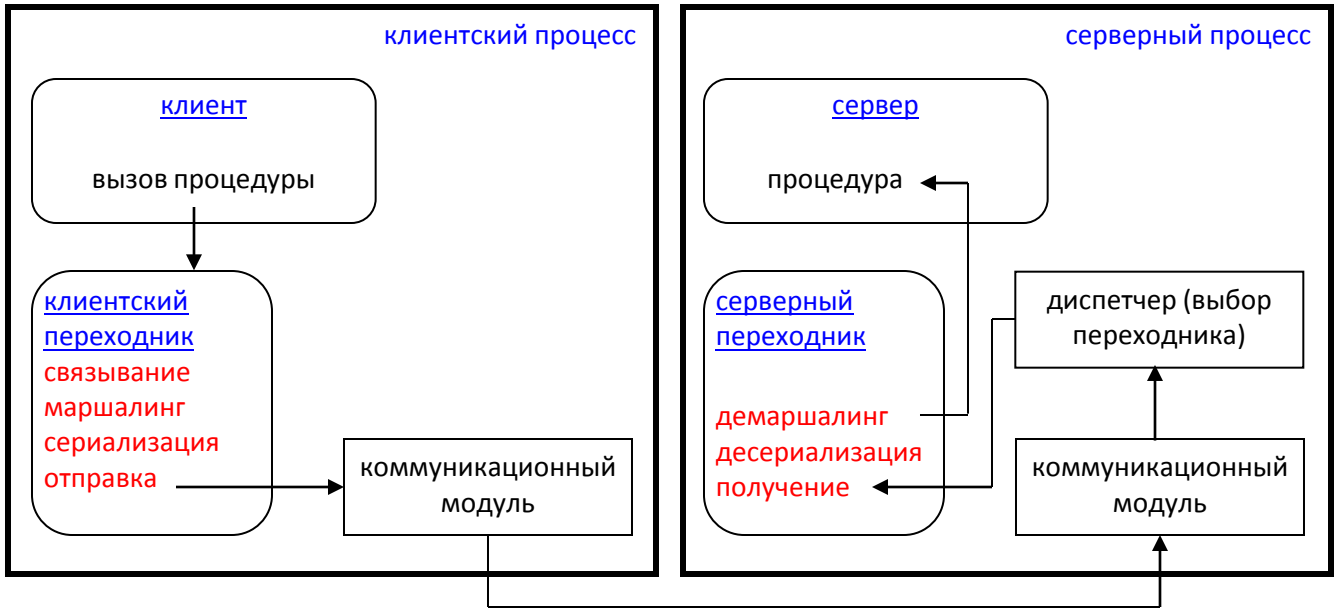


Несохранная синхронная связь, ориентированная на ответ

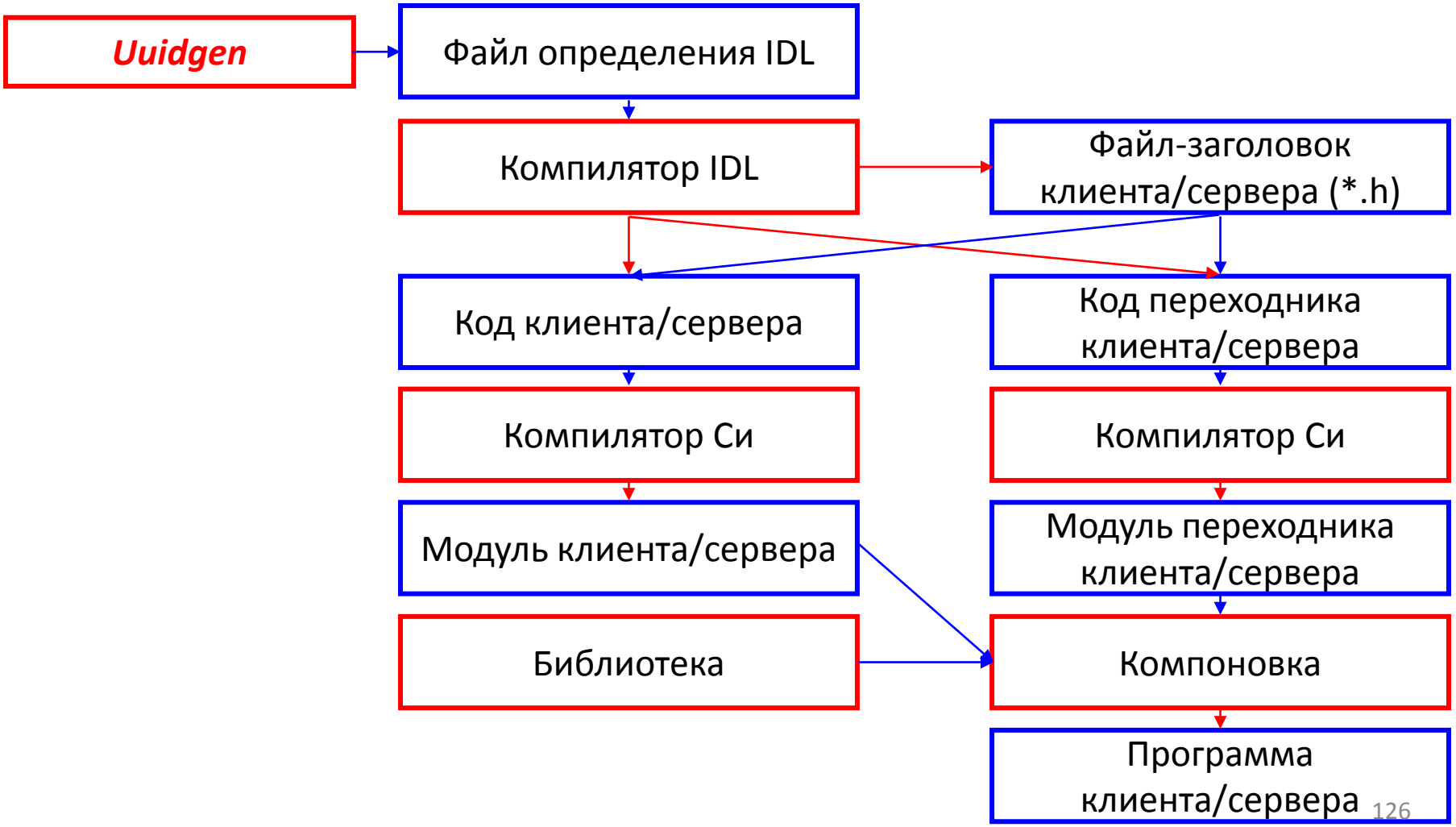
Разработка распределённых программ “клиент-сервер”



Принцип работы модели удалённого вызова процедуры



Разработка распределённых программ “клиент-сервер” в DCE



Разработка распределённых программ: COM/DCOM

- Технология COM (*Component Object Model*) и её вариант для распределённых систем DCOM (*Distributed COM*):
  - клиентская программа использует объекты своего программного сервера так, как если бы эти объекты являлись частью клиентской программы
- Основную роль играет интерфейс объектов, формируемый при помощи объектно-ориентированного языка IDL
- Клиенту известны только интерфейсы объектов
- Сервер предназначен для реализации объектов
- Переход к объектно-ориентированному взаимодействию привёл к пересмотру модели удалённого вызова процедуры и появлению модели ***удалённого обращения к методу***

Разработка распределённых программ: COM/DCOM

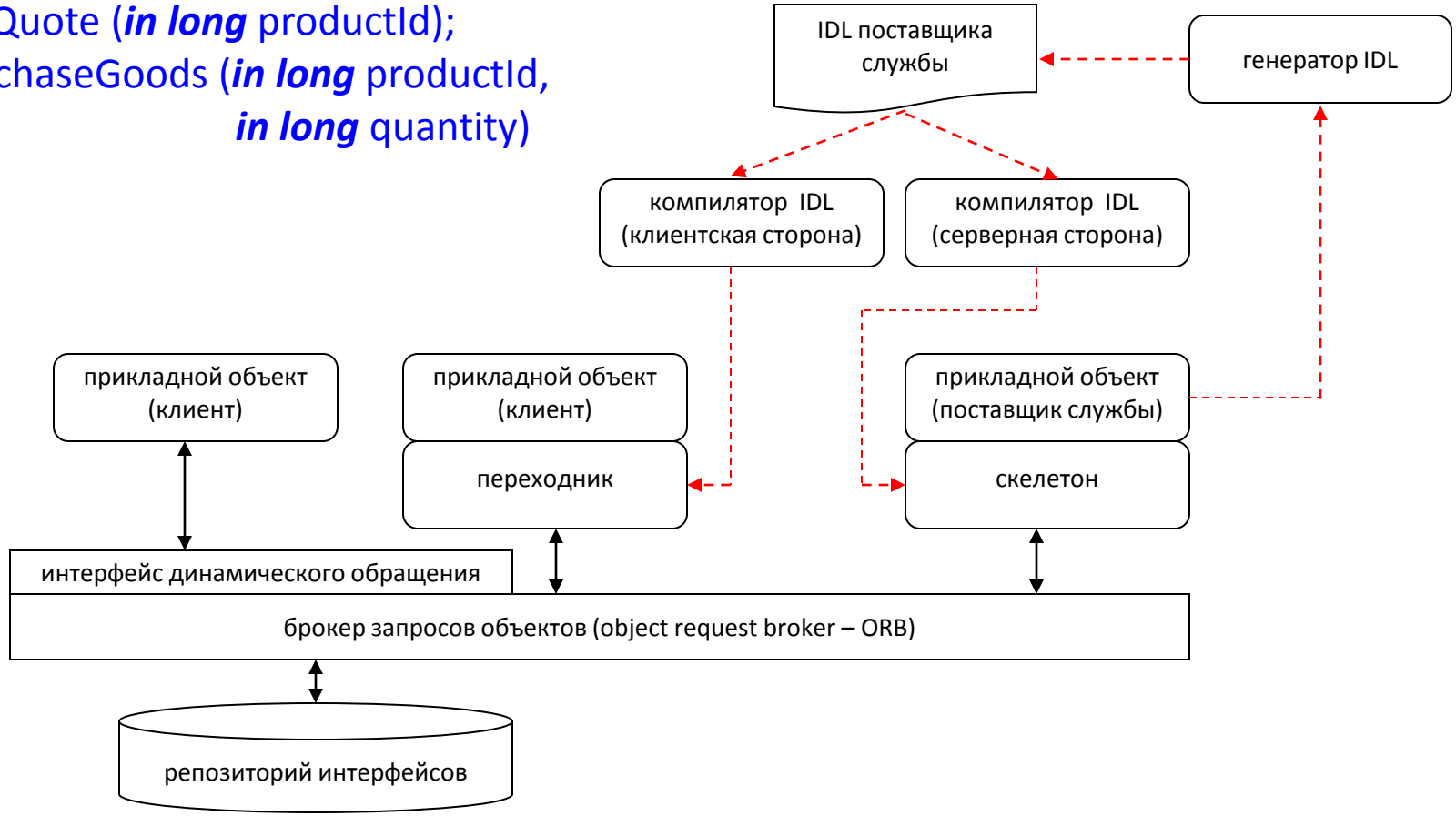
- Взаимодействие клиента и сервера проходит независимо от того, как клиент и сервер распределены по компьютерам:
  - на одном компьютере в рамках единого процесса (взаимодействие между клиентом и сервером происходит при помощи интерфейса объекта в едином адресном пространстве с использованием динамических библиотек)
  - на одной машине в рамках разных процессов по сокращённой схеме удалённого обращения к методу (с переходниками, маршалингом и демаршалингом, но без обращения к сети)
  - на различных компьютерах в рамках информационной сети (технология DCOM) с полным использованием модели удалённого обращения к методу

Разработка распределённых программ: CORBA

- *Брокерами объектов* называют распределённые системы программного обеспечения, основанные на понятии распределённого объекта и удалённом обращении к методам
- Стандарт *CORBA* (*Common Object Request Broker Architecture*) – спецификация для создания и управления распределёнными объектно-ориентированными приложениями
- Система, подчиняющаяся спецификации *CORBA*, состоит из:
  - брокера запросов объектов, содержащего базовые функции взаимодействия объектов
  - служб *CORBA*, доступных с помощью стандартизованного прикладного программного интерфейса
  - средств *CORBA*, предоставляющих службы верхнего уровня, необходимые приложениям

Разработка распределённых программ: CORBA

```
interface Purchasing {
   float getQuote (in long productId);
   float purchaseGoods (in long productId,
                        in long quantity)
}
```



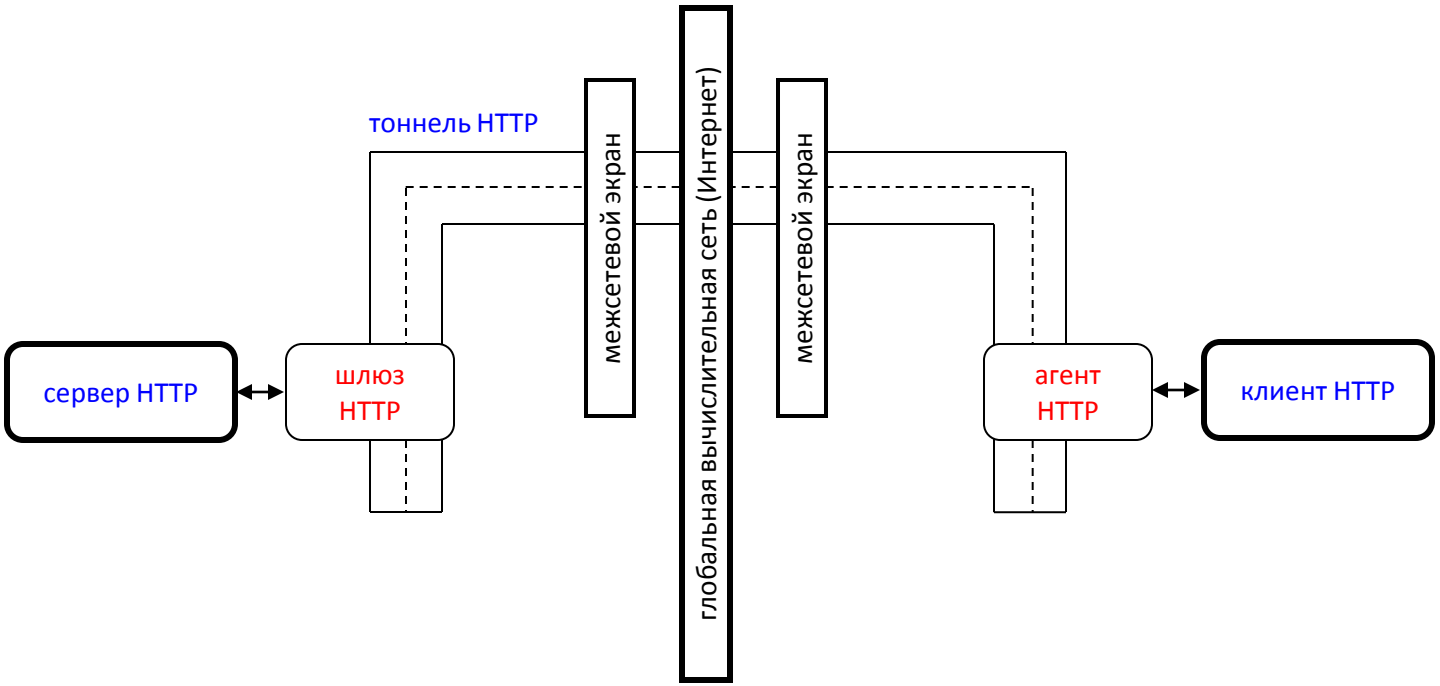
Разработка распределённых программ: CORBA

- Интерфейсы описываются на языке описания интерфейсов *IDL* системы *CORBA*
- Спецификации *IDL* транслируются в заместители объектов на стороне клиента и в скелетоны на стороне сервера
- Программа заместителя содержит в себе описание методов, предоставляемых реализацией объекта
- Скелетон защищает от проблем распределённости сервер
- Заместитель и скелетон могут быть написаны на любом из тех языков, которые поддерживаются компилятором с языка *IDL CORBA*: Си, Си++, Java, Smalltalk, Ада, Кобол, Лисп, PL/1, Python и IDLScript

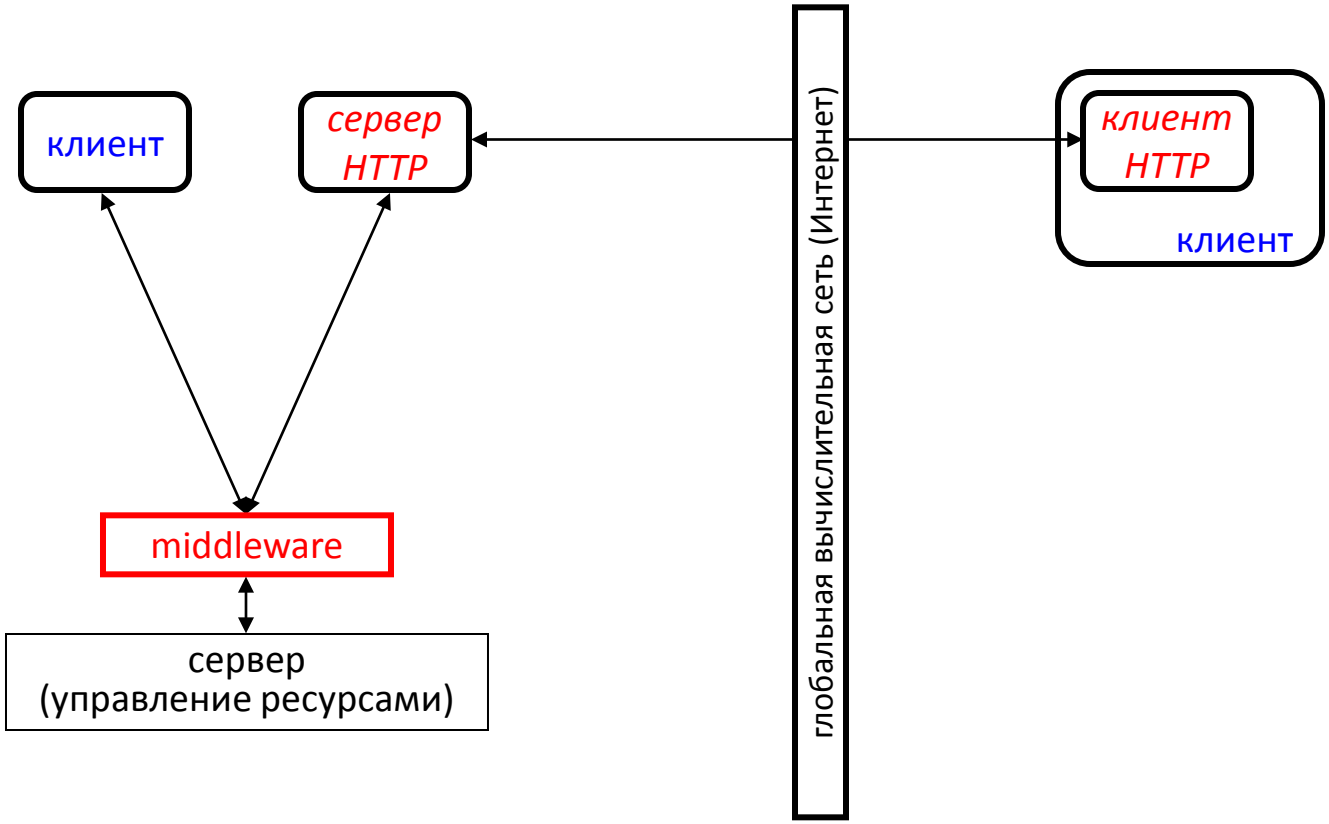
Разработка распределённых программ: CORBA

- Спецификация *CORBA* допускает обратные отображения: предусмотрено проведение отображения записи интерфейсов на языке Java в записи интерфейсов на *IDL*
- Обратное отображение позволяет создавать объекты, доступные из других приложений, написанных на других языках программирования

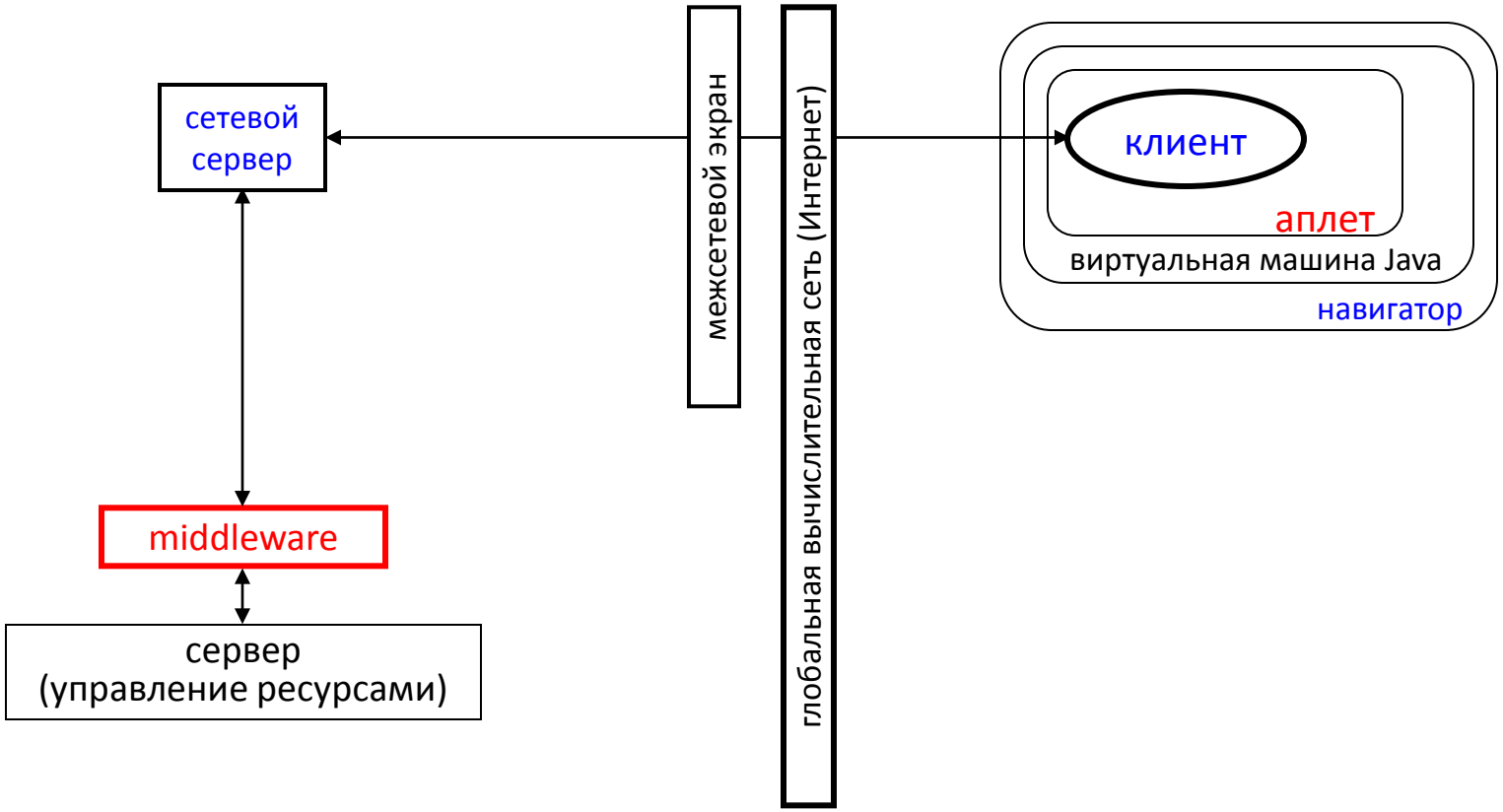
Глобальная информационная сеть: взаимодействие клиента и сервера HTTP



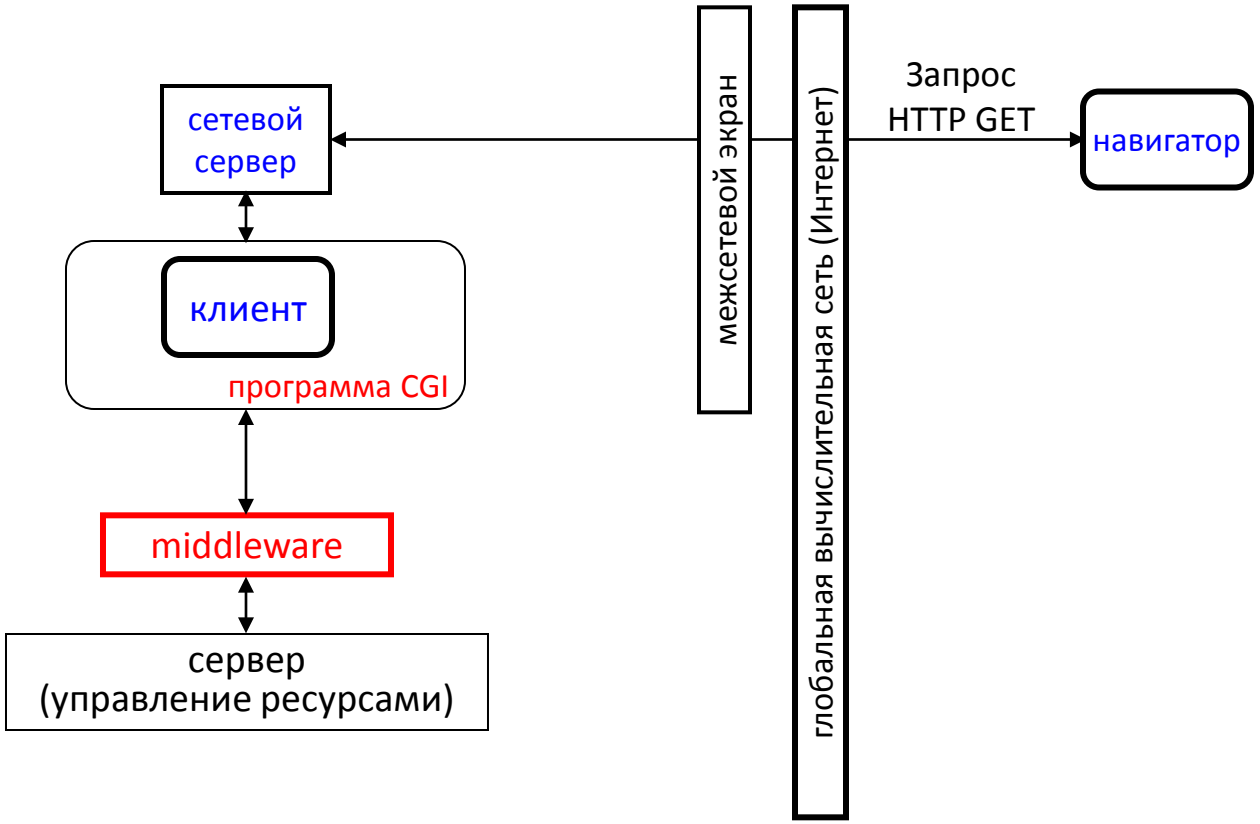
Расширение трёхъярусной архитектуры перемещением клиента на удалённый компьютер с доступом через Интернет (архитектура B2C – Business to Consumer)



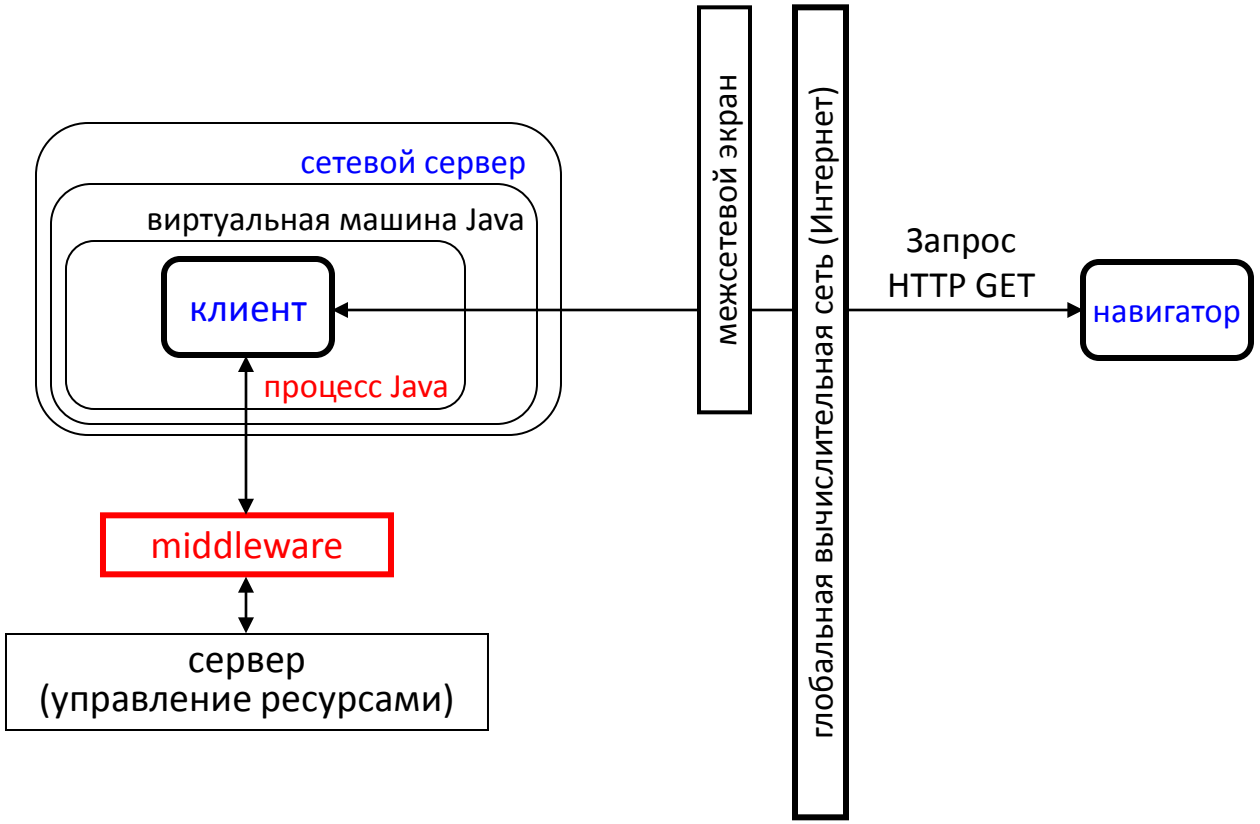
Аплеты как способ реализации удалённых клиентов



Программы *общего шлюзового интерфейса (CGI)* как способ взаимодействия с приложением, расположенным на серверной стороне



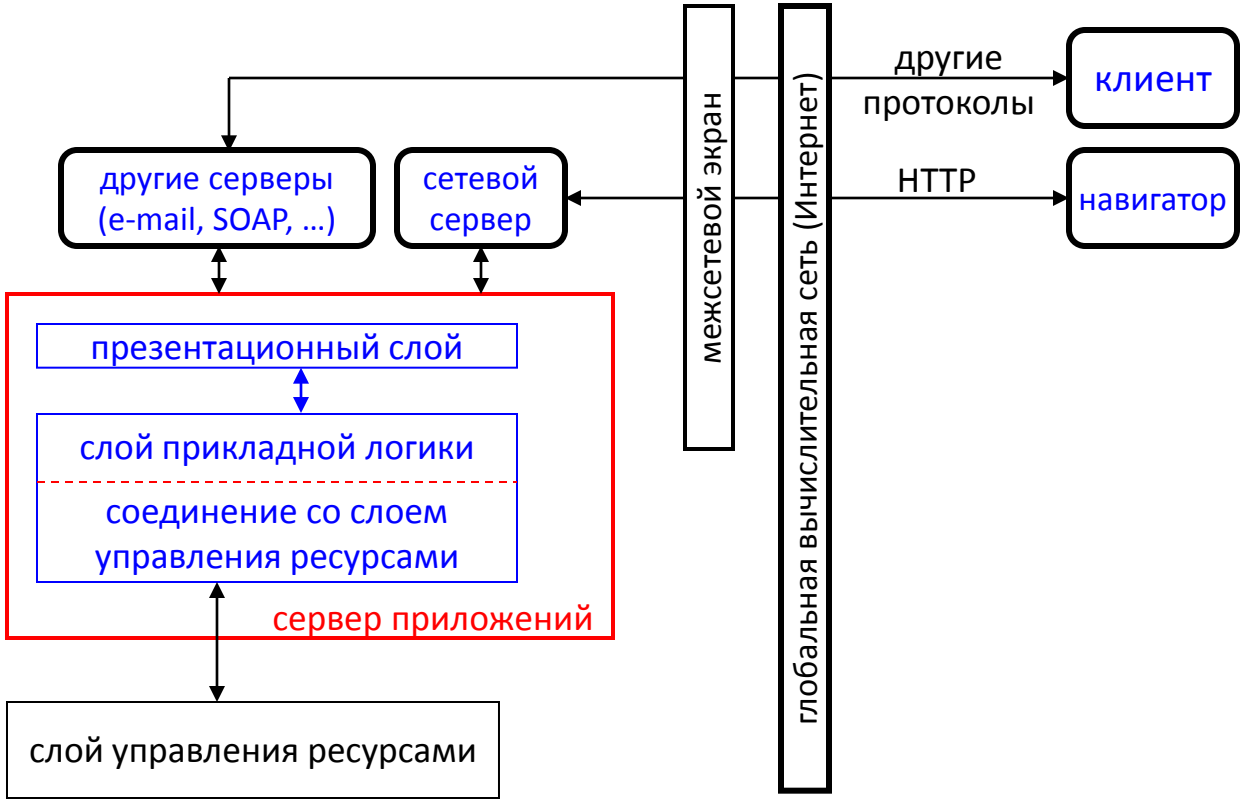
Сервлеты
как способ взаимодействия с приложением, расположенным на серверной стороне



Разработка распределённых программ: серверы приложений

- *Сервер приложений* – форма системной поддержки распределённых программ, взаимодействующих в Интернете
- Наиболее широко распространены серверы приложений
  - *J2EE* фирмы Sun Microsystems
  - *.NET* фирмы Microsoft
  - *OAS* фирмы Oracle Corporation
  - *WebLogic* фирмы BEA Systems (Oracle)
  - *WebSphere* компании IBM

Разработка распределённых программ: серверы приложений



Разработка распределённых программ: серверы приложений

- Поддержка взаимодействия и презентации в сервере J2EE:
  - *сервлеты*, язык тегов *JSP* (*Java Server Pages*)
  - прикладной интерфейс *JAXP* (*Java API for XML Parsing*)
  - система электронной почты (*Java Mail*)
  - служба аутентификации и авторизации *JAAS*
- Поддержка интеграции приложений в сервере J2EE:
  - компоненты *EJB* (*Enterprise Java Bean*)
  - интерфейс именования *JNDI* (*Java Naming and Directory Interface*)
  - служба сообщений *JMS* (*Java Message Service*)
  - транзакционный интерфейс *JTA* (*Java Transaction API*)
- Поддержка доступа к ресурсам в сервере J2EE:
  - связь с базами данных *JDBC* (*Java Data Base Connectivity*)
  - подключение архитектур *J2CA* (*J2EE Connector Architecture*)

Разработка распределённых программ: серверы приложений

- Серверы приложений позволяют работать с разнообразными клиентскими программами:
  - *сетевые навигаторы*
  - *приложения*, такие же, как и в традиционных системах, что позволяет интегрировать множество приложений в единые слаженно работающие комплексы, при этом в сами приложения никаких изменений вносить не требуется
  - *устройства*, например, мобильные телефоны
  - *программы электронной почты*
  - *клиенты сетевых служб*, то есть приложения, взаимодействующие с сервером приложений через стандартные протоколы сетевых служб

Системы программирования компании Microsoft – .NET и C#

- Ядро системы программирования *.NET Framework* – спецификация общеязыковой инфраструктуры *CLI* (*Common Language Infrastructure*)
- Спецификация определяет единый промежуточный язык *CIL* (*Common Intermediate Language*) и общую систему типов *CTS* (*Common Type System*), обеспечивающую совместимость типов
- По спецификации *CLI* программы не интерпретируются виртуальной машиной, а транслируются в процессе выполнения в машинный код с помощью *JIT*-компиляторов (*Just-In-Time compilers*)

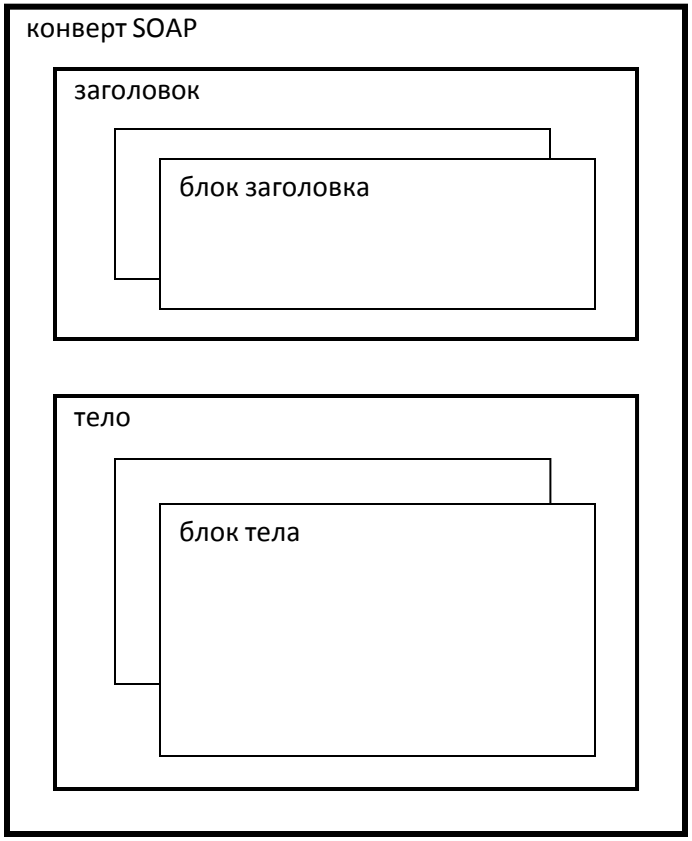
Системы программирования компании Microsoft – .NET и C#

- *Виртуальная исполняющая система VES* (*Virtual Execution System*) обеспечивает трансляцию и выполнение CIL-программ
- Общеязыковая исполняющая среда *CLR* (*Common Language Runtime*) – реализация *VES* на платформе MS Windows
- Реализации *CLI* включают в себя:
  - *.Net Compact Framework* (для мобильных устройств и игровых приставок *XBox*)
  - *Mono* (проект с открытым кодом для ОС Linux)
  - *Portable .NET* (часть проекта *dotGNU*)

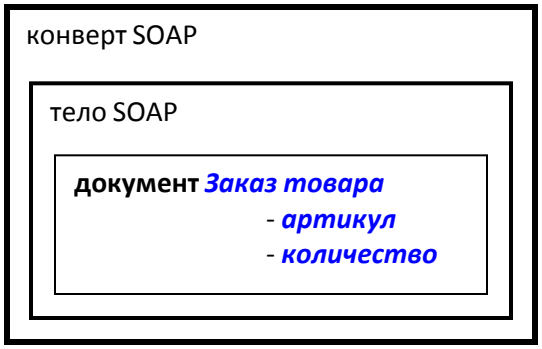
Требования сетевых служб к системам программирования

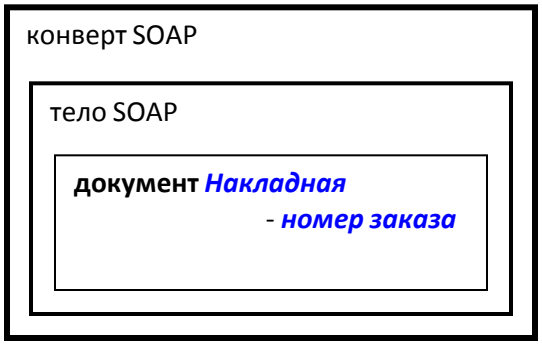
- формат описания сообщений, распространяемых в сети (*SOAP*)
- язык описания структур данных и интерфейсов сетевых служб и компилятор с него (*WSDL*)
- средства хранения описаний сетевых служб и поиска этих описаний (*UDDI*)
- средства динамического управления потоком сообщений (координация работы сетевых служб, контроллеры разговоров, ролевые протоколы, стандарт *WS-Coordination*)
- язык описания алгоритмов работы сетевых служб и компоненты для работы с ним (композиция служб, *BPEL4WS*)
- другие компоненты для поддержки сетевых служб

Формат сообщений SOAP

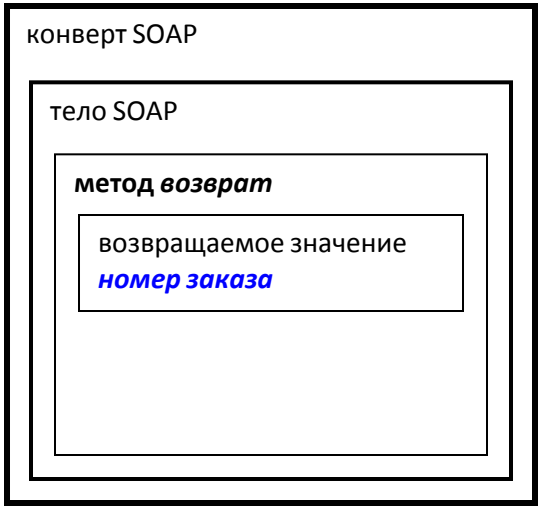


Взаимодействие в стиле документа

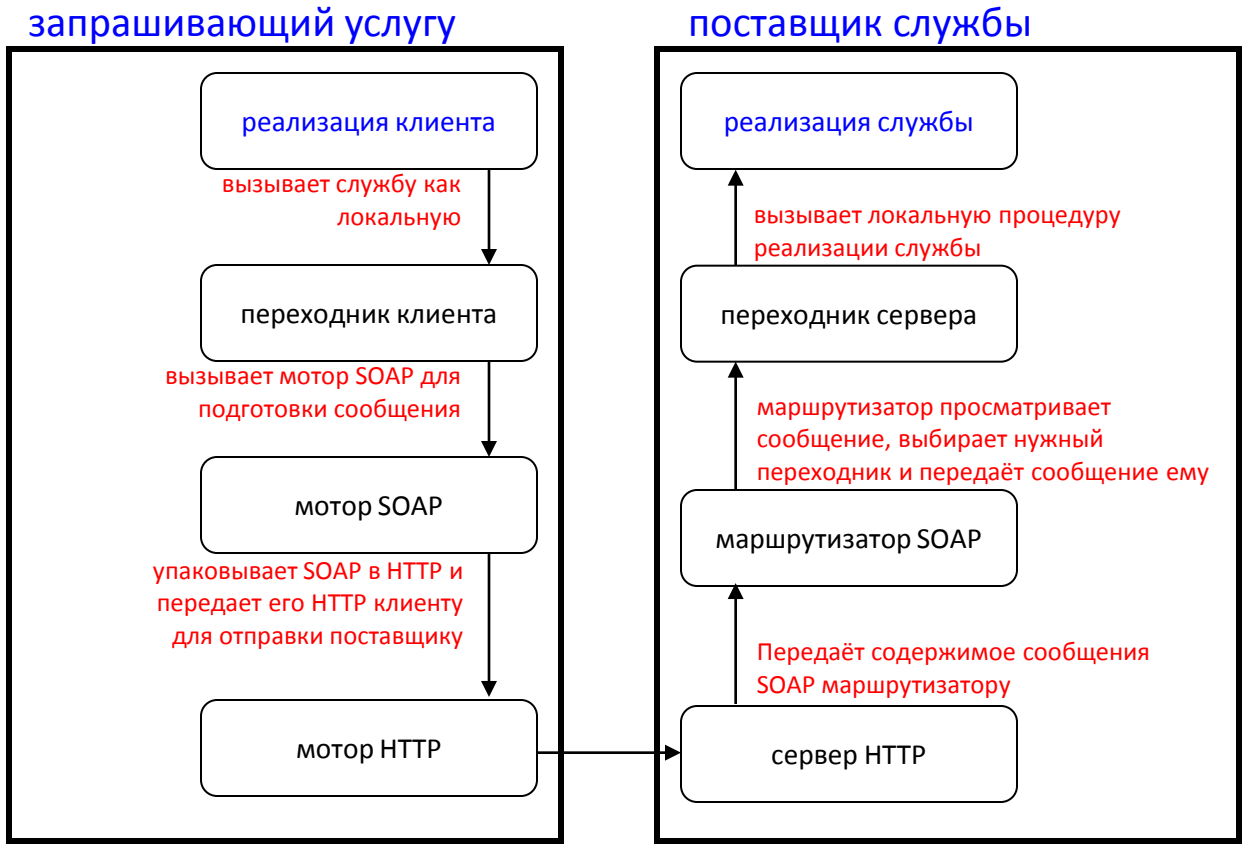




Взаимодействие в стиле вызова удалённой процедуры

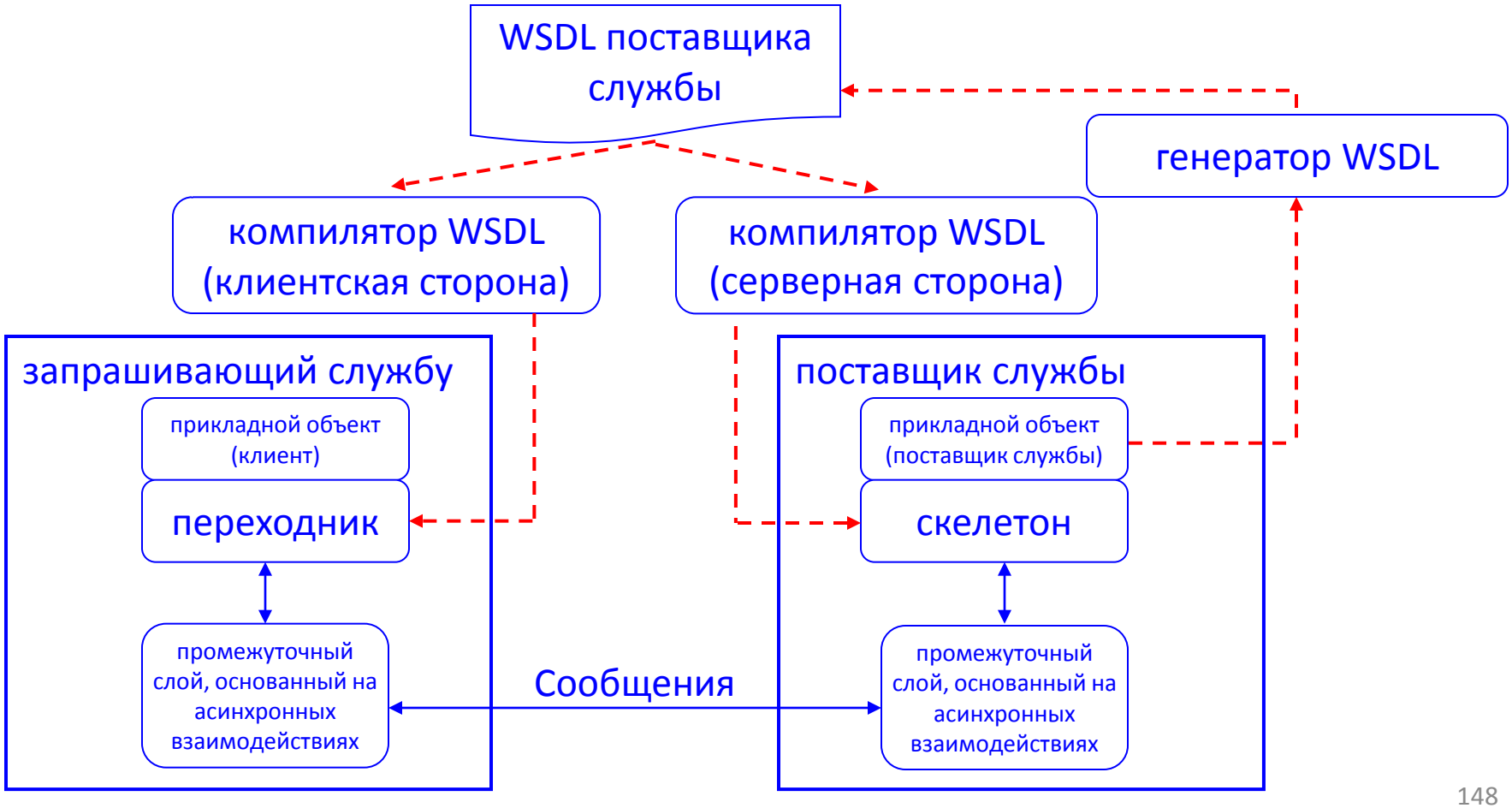


Простейшая реализация протокола SOAP



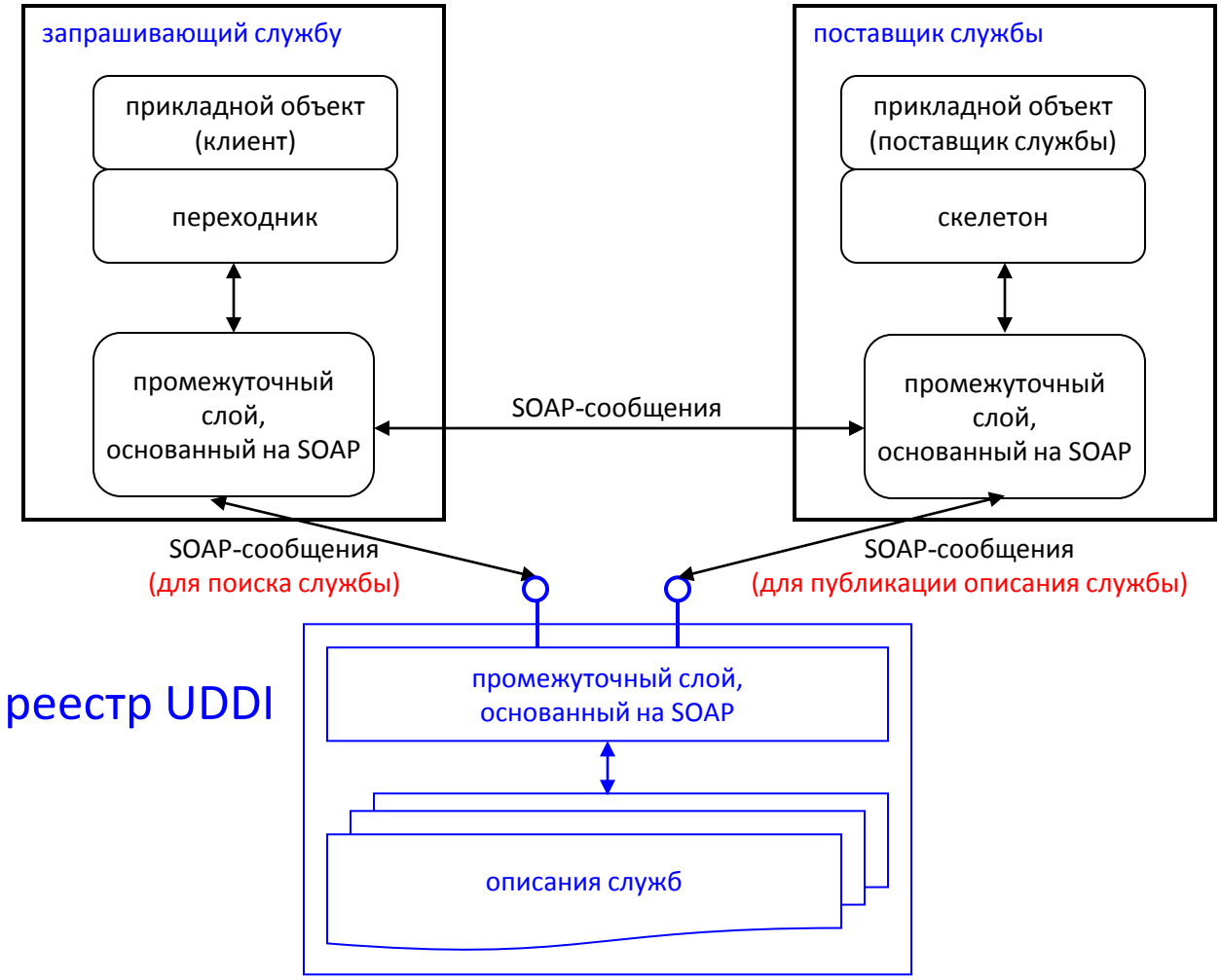
Разработка распределённых программ: сетевые службы

Пунктиром показаны действия, выполняемые на стадии разработки

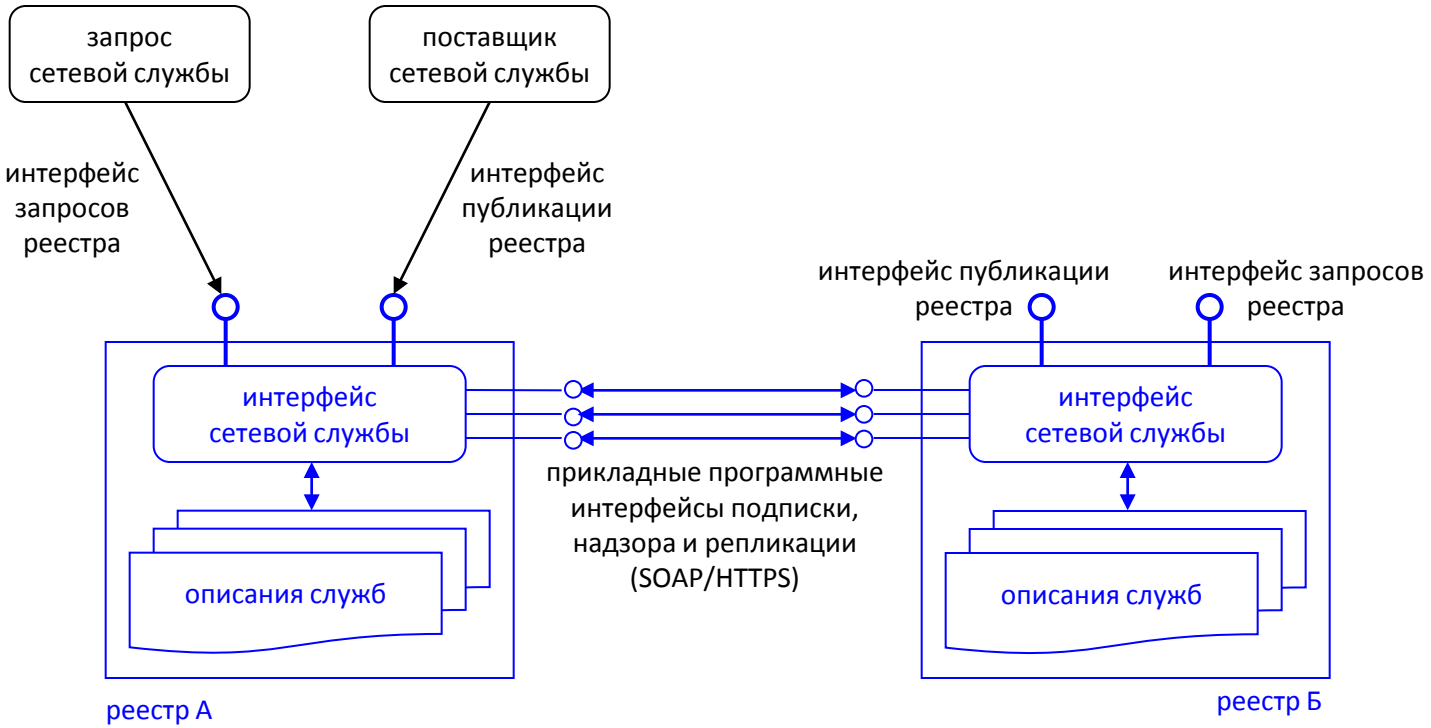


Сетевые службы и WSDL

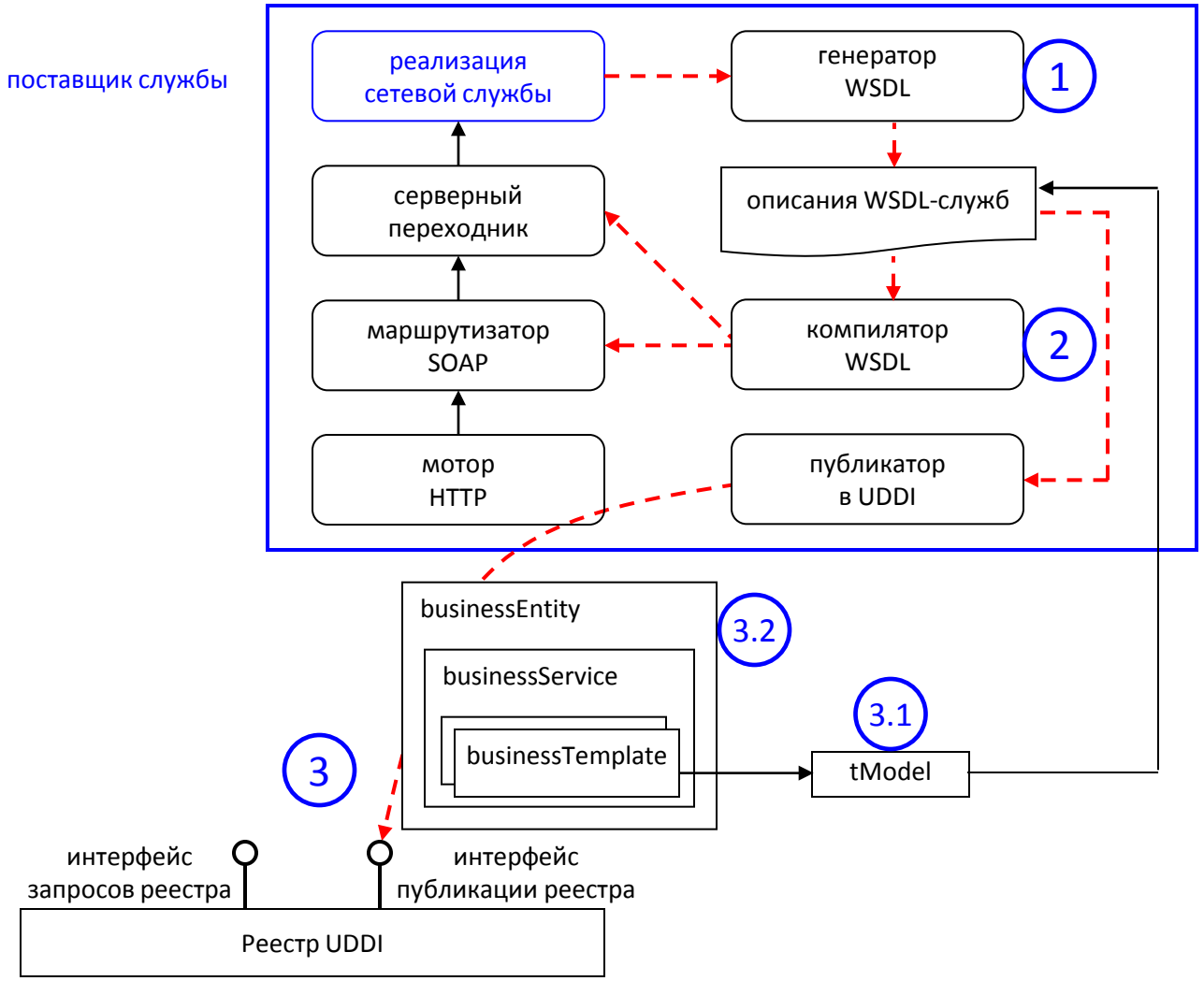
Передача информации о службах в реестр UDDI для поиска служб (при разработке и выполнении) и последующего обращения к ним



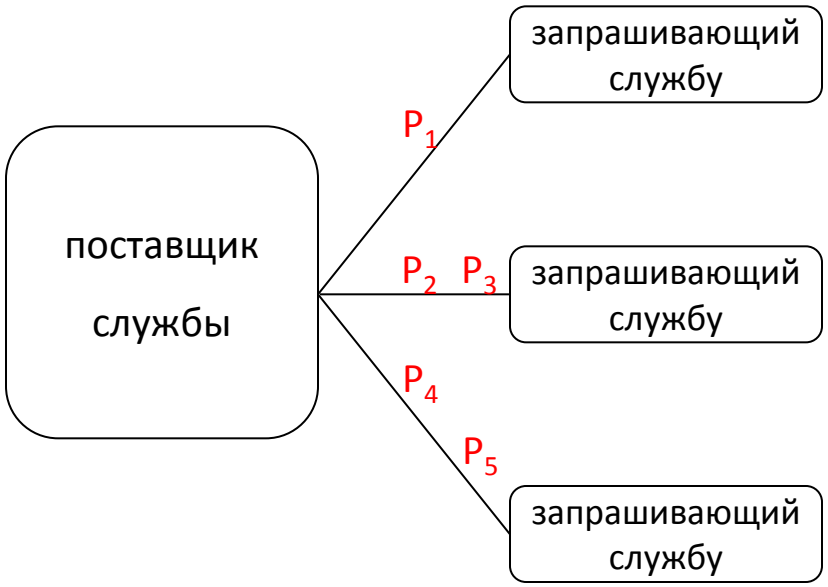
Взаимодействие с реестрами UDDI и между ними



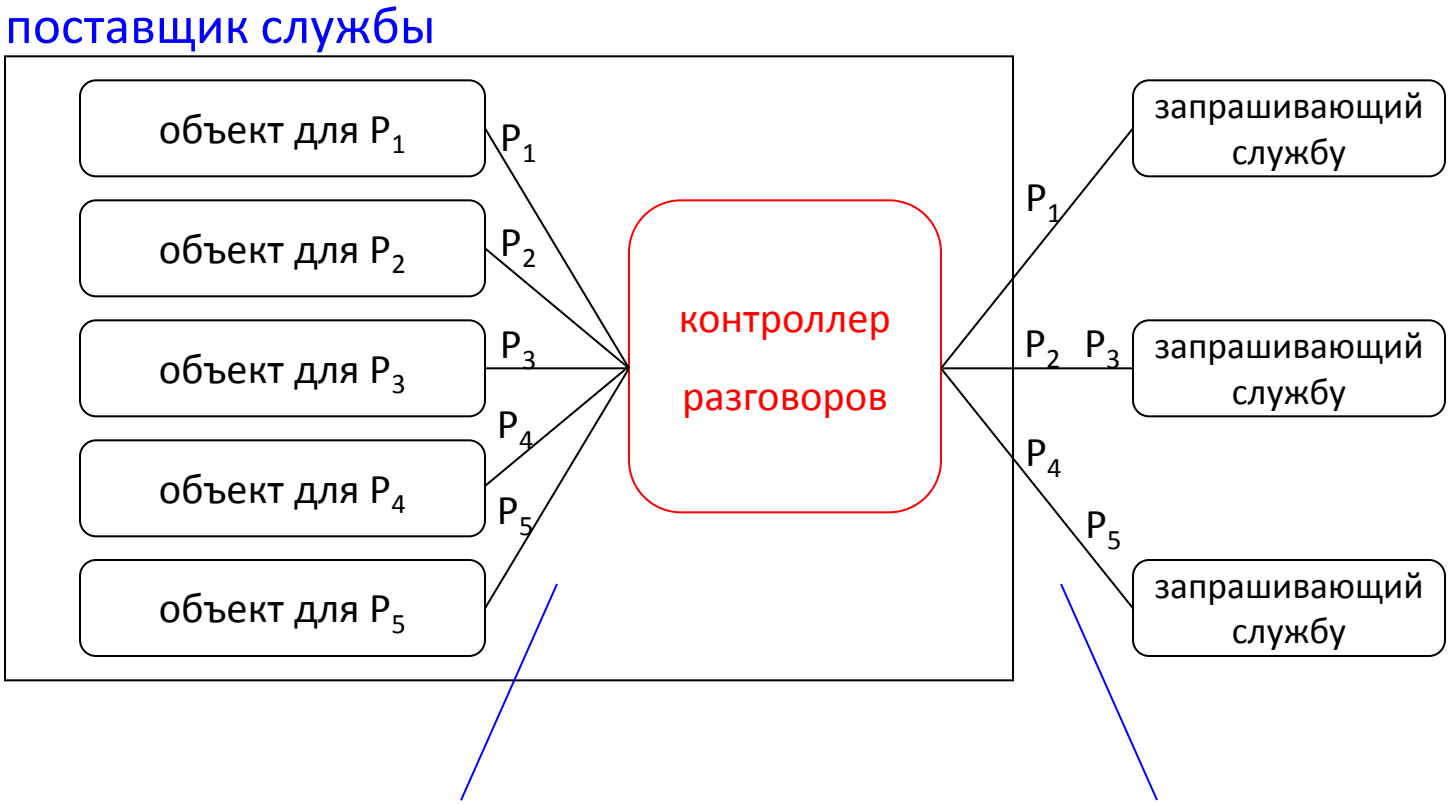
Представление внутренней системной службы в качестве сетевой



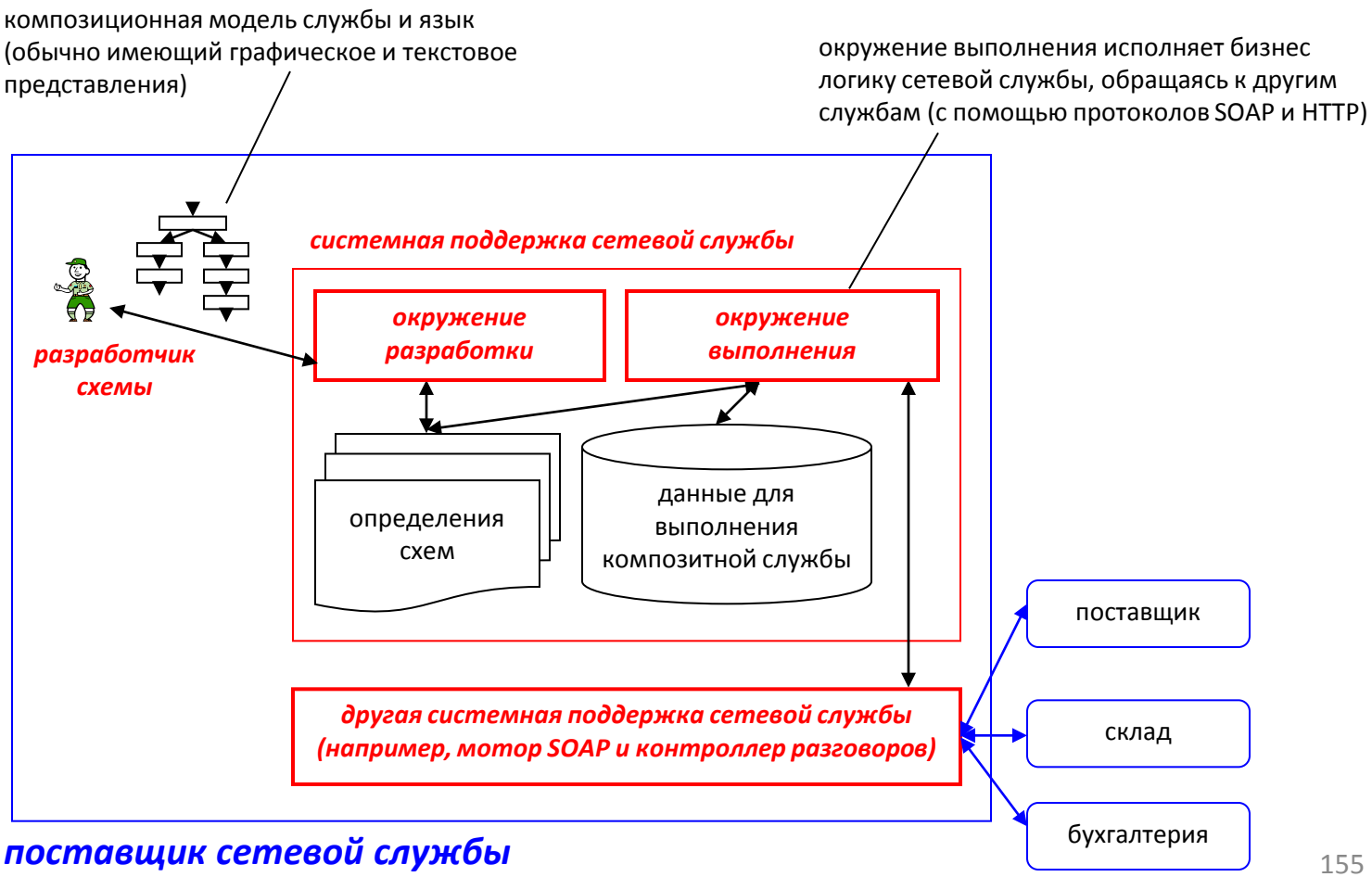
Взаимодействие нескольких клиентов с одной сетевой службой выполнением разговоров, совместимых с протоколом *P*



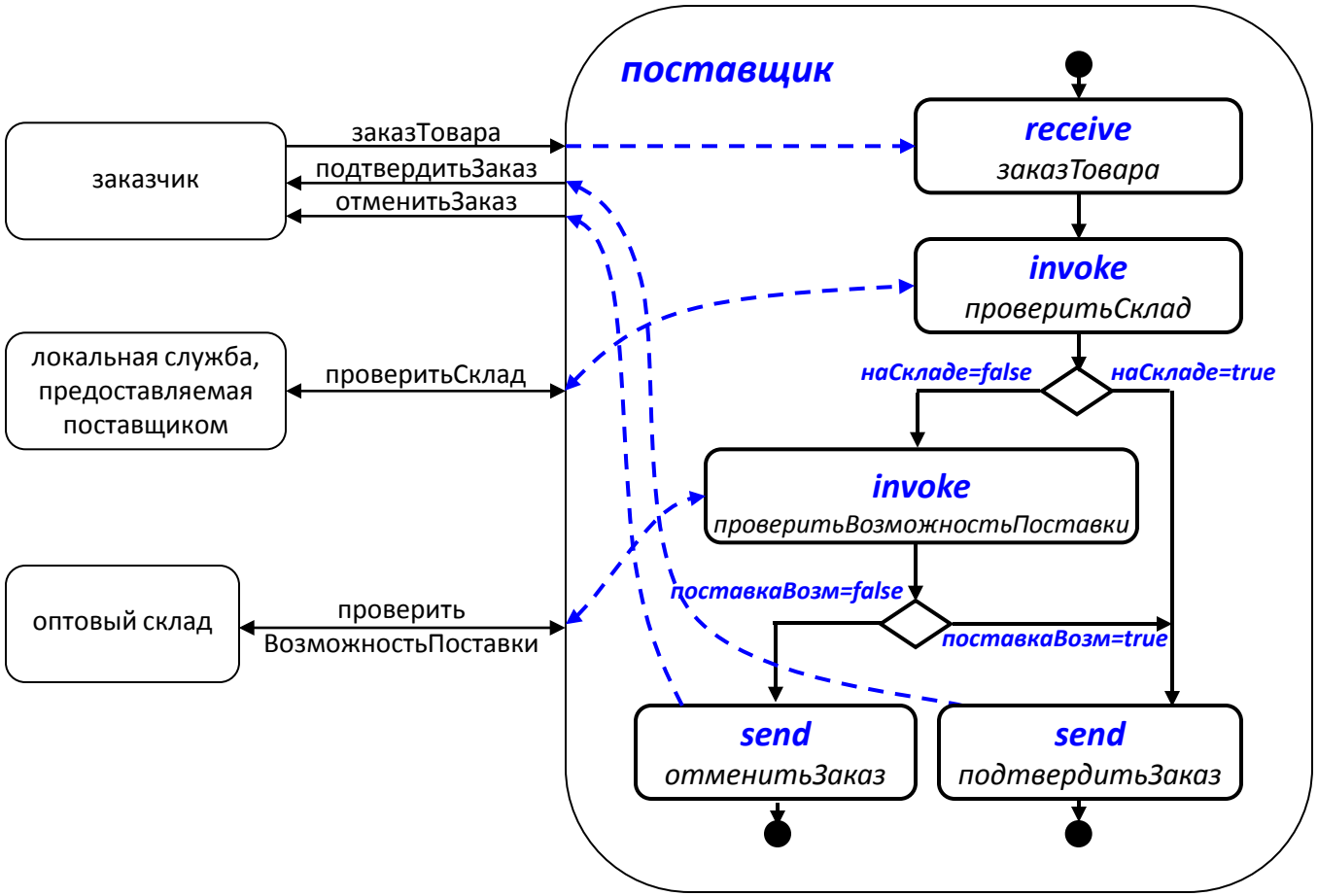
Контроллер разговоров



Сетевые службы, децентрализованные протоколы и стандартизация

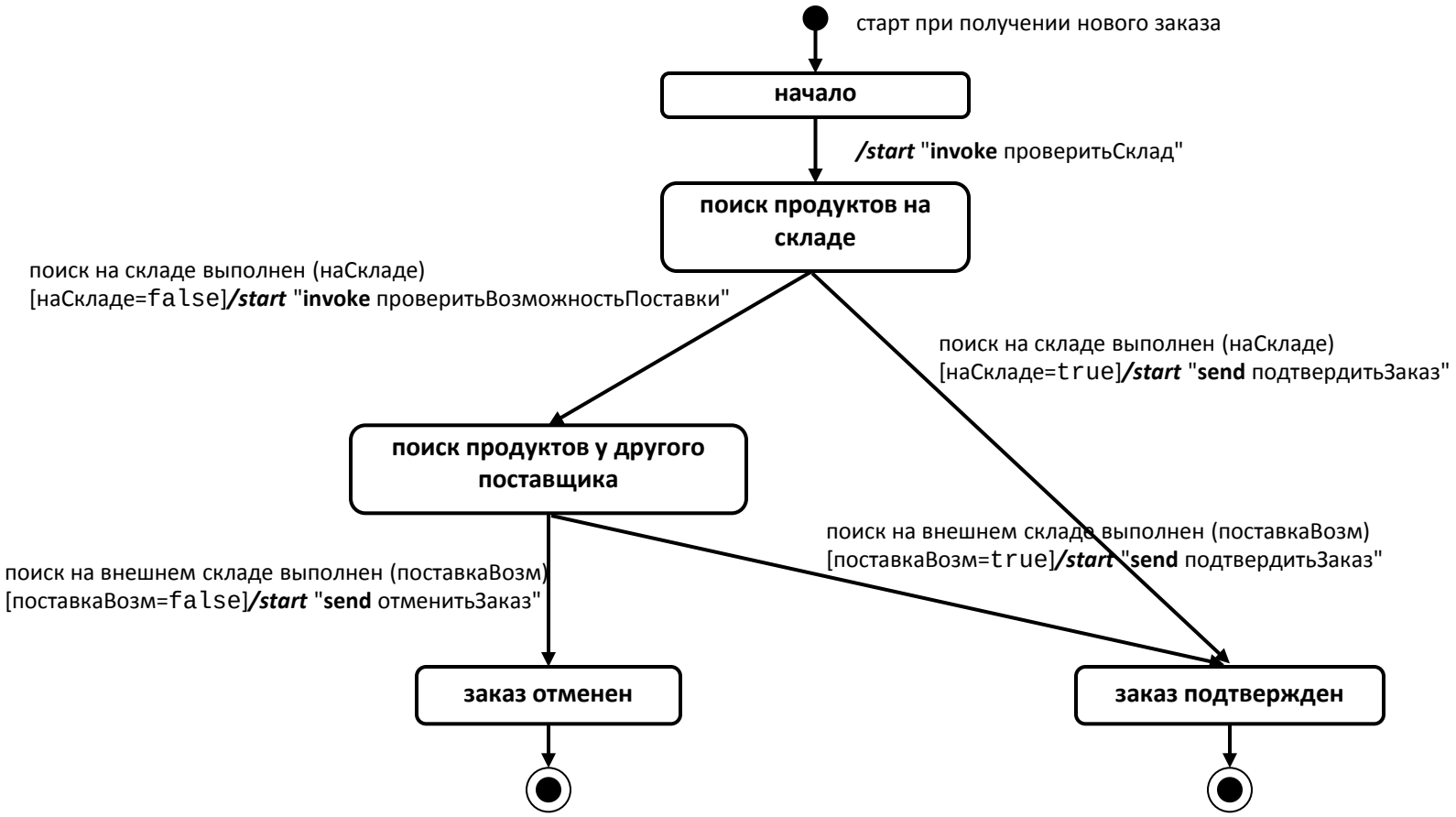


Модель сетевой службы поставщика товара в виде диаграммы активности

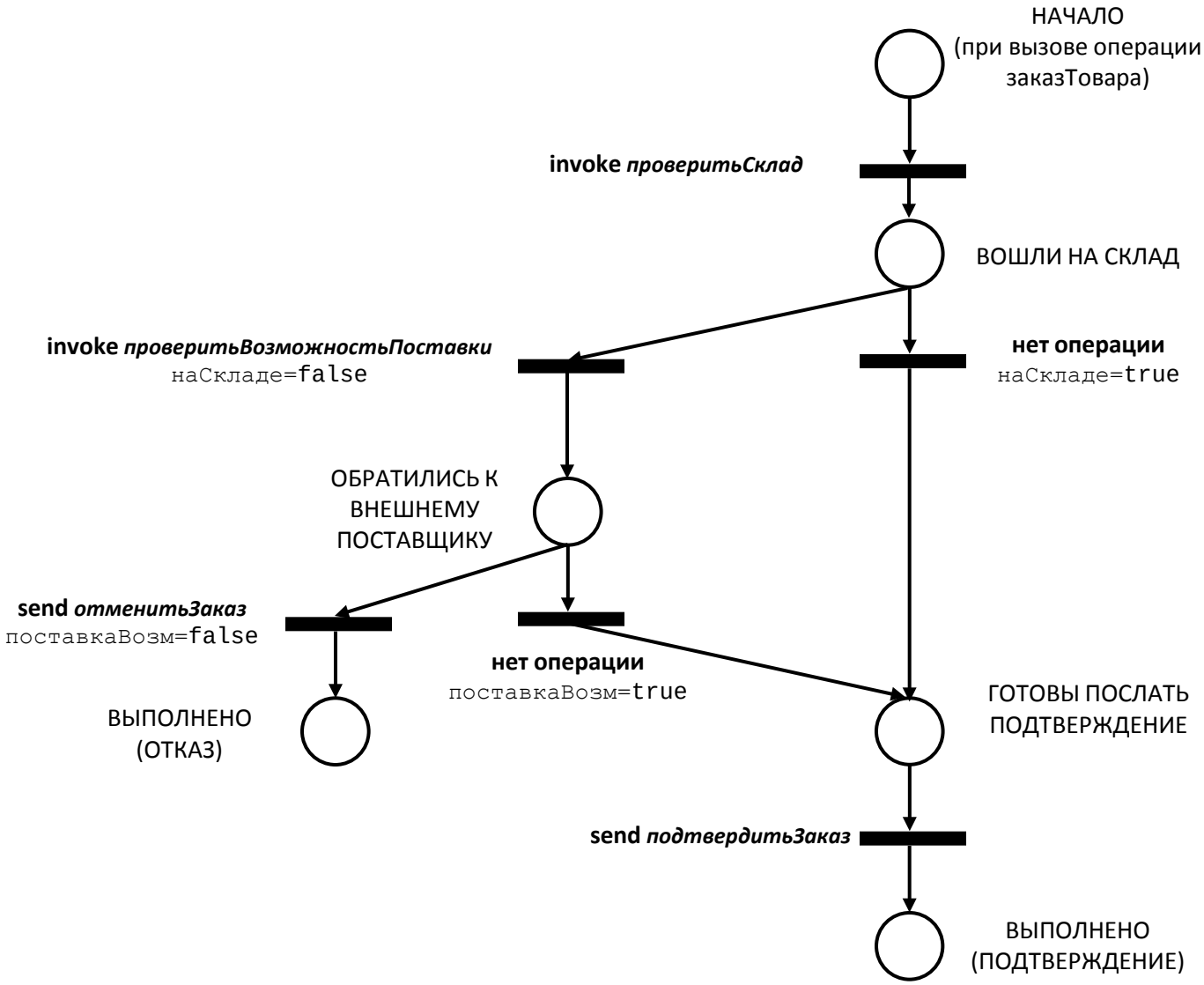


Пунктирными линиями отмечены отношения между (внутренними) активностями и (внешними) протокольными сообщениями

Описание процесса с помощью диаграммы состояний



Описание процесса с помощью сети Петри



Описание процесса с помощью π-исчисления и с помощью правил

A=**receive**ЗаказТовара, **invoke**ПроверитьСклад
B=[поставкаВозм=`false`]**send**ОтменитьЗаказ+[поставкаВозм=`true`]**send**ПодтвердитьЗаказ
C=**invoke**ПроверитьВозможностьПоставки.B

Закупка=A.(([наСкладе=`false`]C) + ([наСкладе=`true`]**send**ПодтвердитьЗаказ))

ON **receive** заказТовара
IF `true`
THEN **invoke** проверитьСклад;

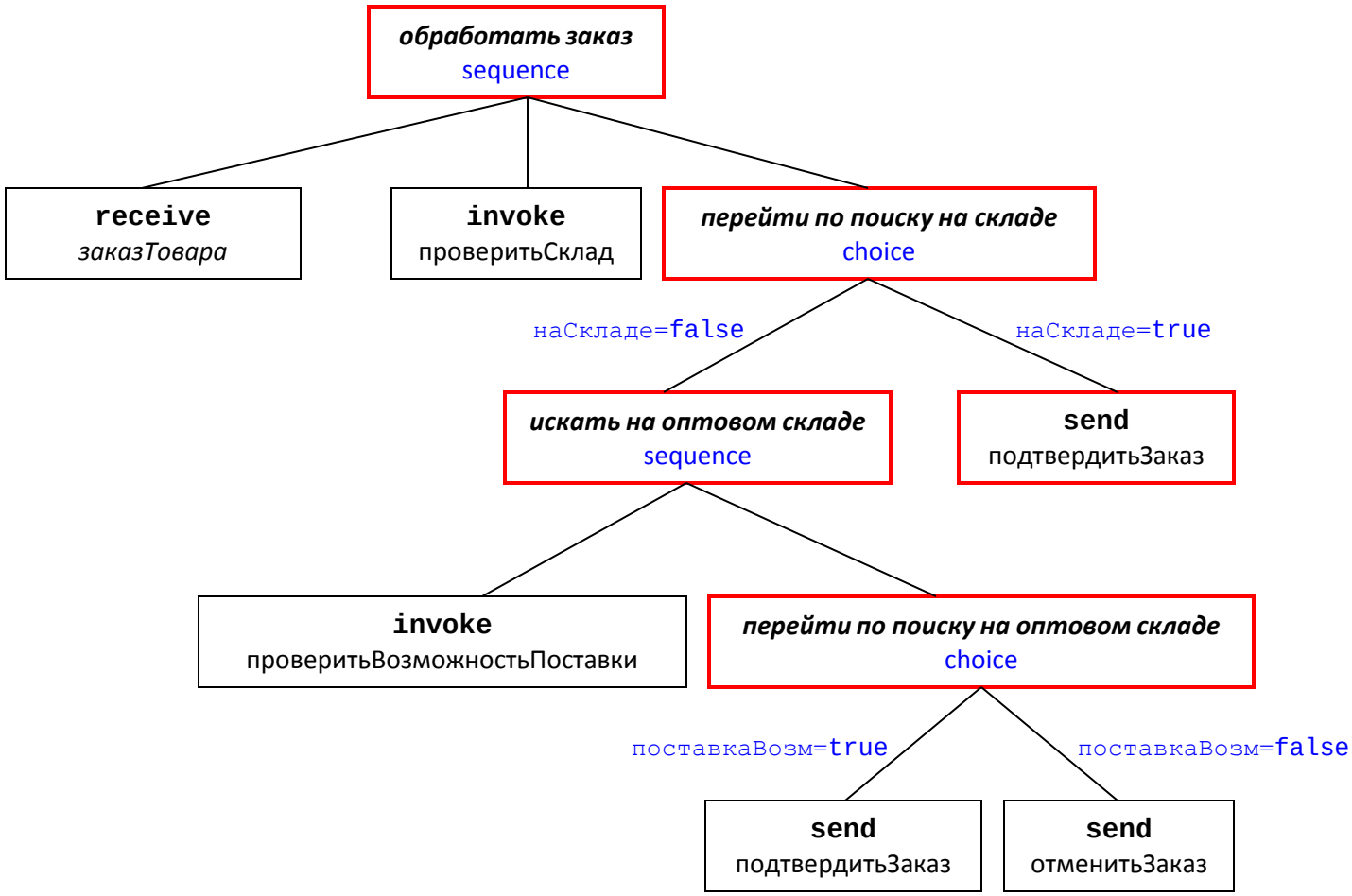
ON **complete**(проверитьСклад)
IF (наСкладе==`true`)
THEN **send** подтвердитьЗаказ;

ON **complete**(проверитьВозможностьПоставки)
IF (поставкаВозм==`false`)
THEN **send** отменитьЗаказ;

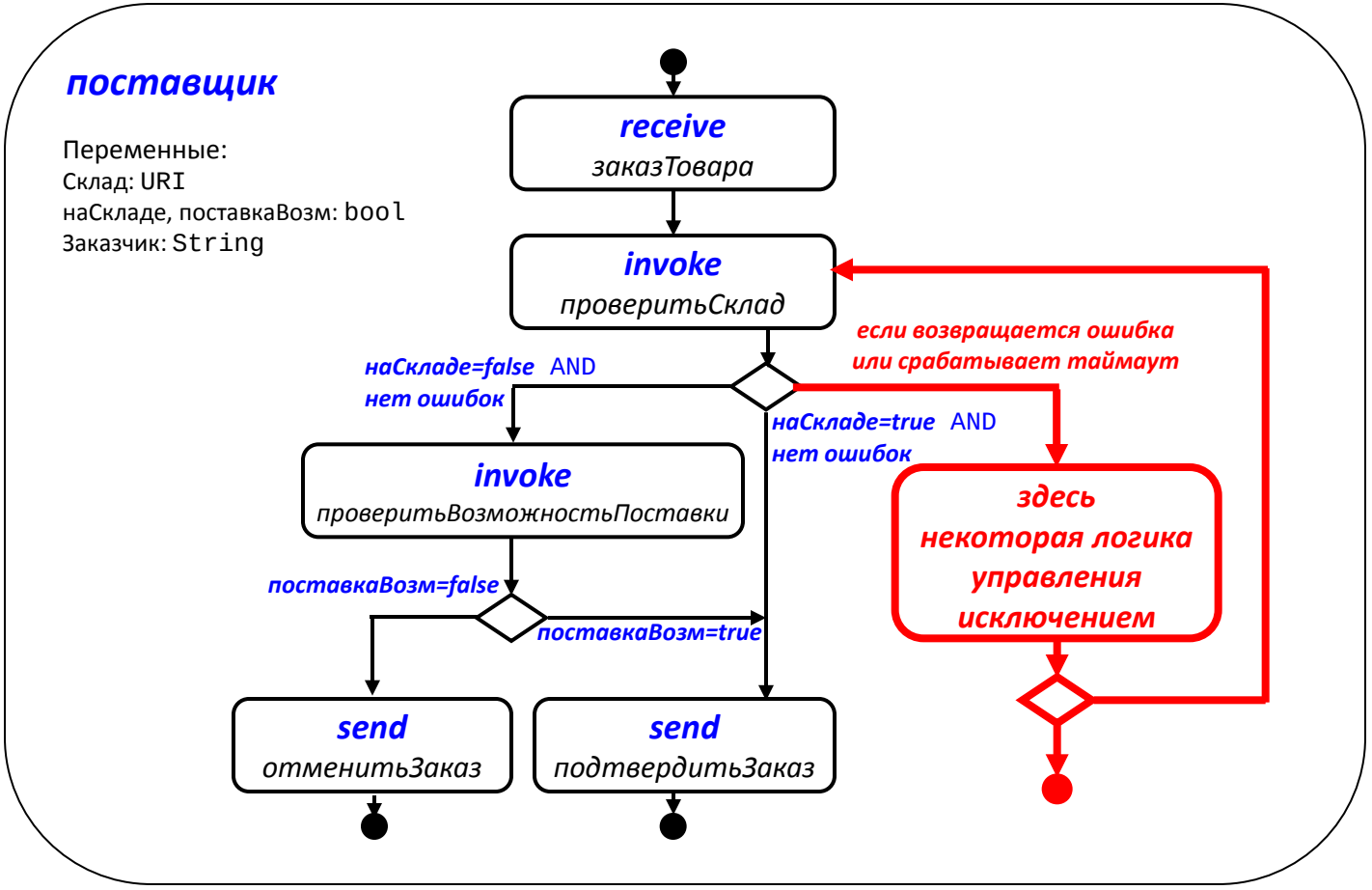
ON **complete**(проверитьСклад)
IF (наСкладе==`false`)
THEN **invoke** проверитьВозможностьПоставки;

ON **complete**(проверитьВозможностьПоставки)
IF (поставкаВозм==`true`)
THEN **send** подтвердитьЗаказ;

Описание процесса с помощью иерархии активностей



Ошибки могут обрабатываться аналогично тому, как это делается во многих языках программирования, не имеющих специальной поддержки



Выделена часть потока, управляющая исключением

"Попытка-перехват-возбуждение" (try-catch-throw)

- Логика управления исключением ассоциируется с активностью или с группой активностей.
- Если некоторое логическое условие выполнено, выполняется и часть программы, управляющая исключением.
- Преимущества:
  1. Обычная логика чётко отделена от логики исключений, что помогает структурировать композицию, облегчить её проектирование и сопровождение.
  2. Если композиция структурирована в соответствии с деревом декомпозиции, аналогично можно организовать и определения исключений.
  3. Оказывается возможным описать стратегию продолжения, то есть указать, что произойдет с вызовом композиции и, в частности, с "исключительной" активностью, после завершения обработки исключения.

Подходы, основанные на правилах (событие-условие-действие/event-condition-action)

- ***Событие*** (например, отказ заказчика) облекается в форму сообщений, посылаемых клиентом композитной службе, или в форму таймаутов.
- ***Условие*** – логическое выражение над сообщением, которое проверяет, действительно ли событие относится к исключительной ситуации, которую надо обработать.
- ***Действие*** вызывает операцию или прерывает транзакцию.
- Преимущества и недостатки:
  - Подходы, основанные на правилах, чётко разделяют нормальное и исключительное поведение процесса. Эти подходы хороши, если число правил невелико.
  - В системе появляется ещё один язык, который системе надо интерпретировать, а разработчику – изучать. Трудно анализировать и понимать совместное поведение крупного набора правил и (возможно неожиданное и нежелательное) взаимодействие между правилами и потоками внутри набора.