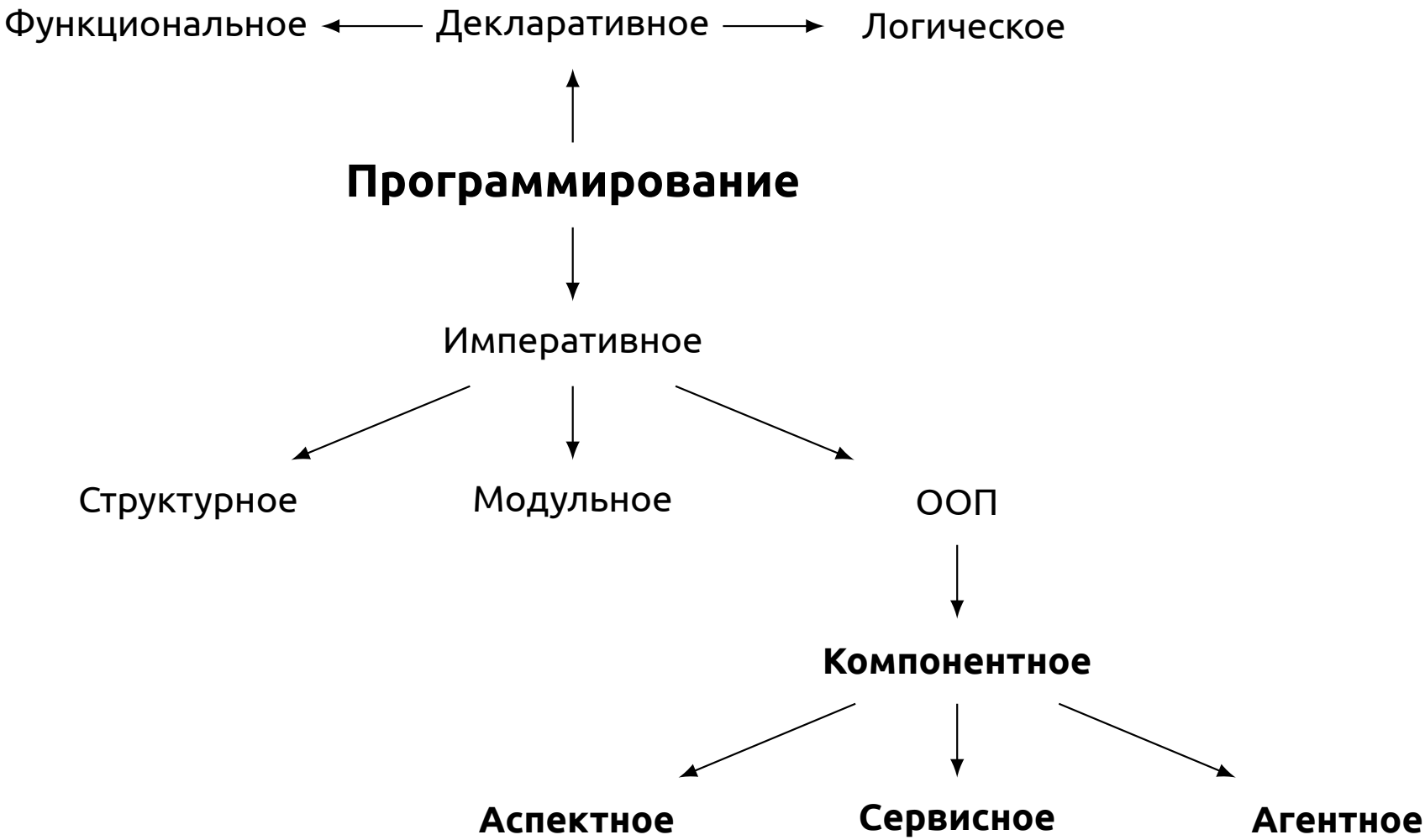
Парадигмы программирования (часть 2)

Алексей Островский

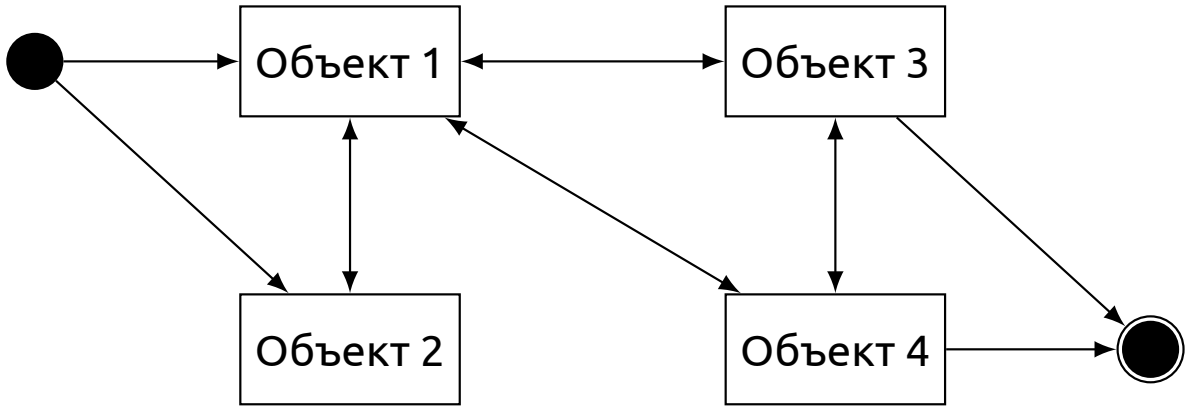
Физико-технический учебно-научный центр НАН Украины

5 декабря 2014 г.

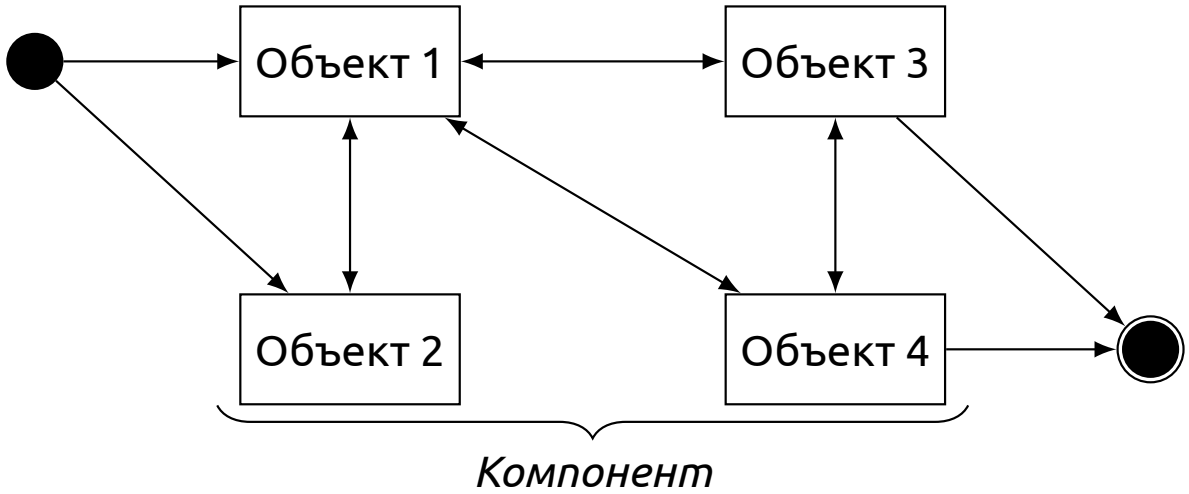
Парадигмы программирования



Причины появления компонентов



Причины появления компонентов



- **повторное использование** — объекты/классы тесно связаны между собой, что затрудняет использование в другой среде;
- **модульность** — связанность объектов препятствует выделению автономных модулей;
- **быстрая разработка** — затраты на конфигурацию объектов для новой системы чересчур высоки.

Компонентно-ориентированное программирование

Определение

Компонентно-ориентированное программирование (англ. *component-based software engineering*) — подход к разработке ПО, основанный на использовании слабо связанных (англ. *loosely coupled*) независимых программных компонентов в единой программной системе.

КОП — развитие ООП с акцентом на повторное использование кода (software reuse).

Определение

Компонент — самостоятельный программный продукт, который соответствует определенной компонентной модели и может сочетаться с другими компонентами без модификации кода согласно заданному стандарту.

Особенности парадигмы

- Разделение интерфейса и реализации — интерфейс полностью определяет функциональность компонента; его реализация может прозрачно измениться, в т. ч. во время исполнения программы.
- Использование стандартов — спецификация определений интерфейсов и формата взаимодействия компонентов. Компонент может быть написан на любом ЯП при соблюдении стандартов.
- Промежуточный слой (англ. *middleware*), реализующий взаимодействие между компонентами.
- Процесс разработки, оптимизированный для работы с компонентами повторного использования.

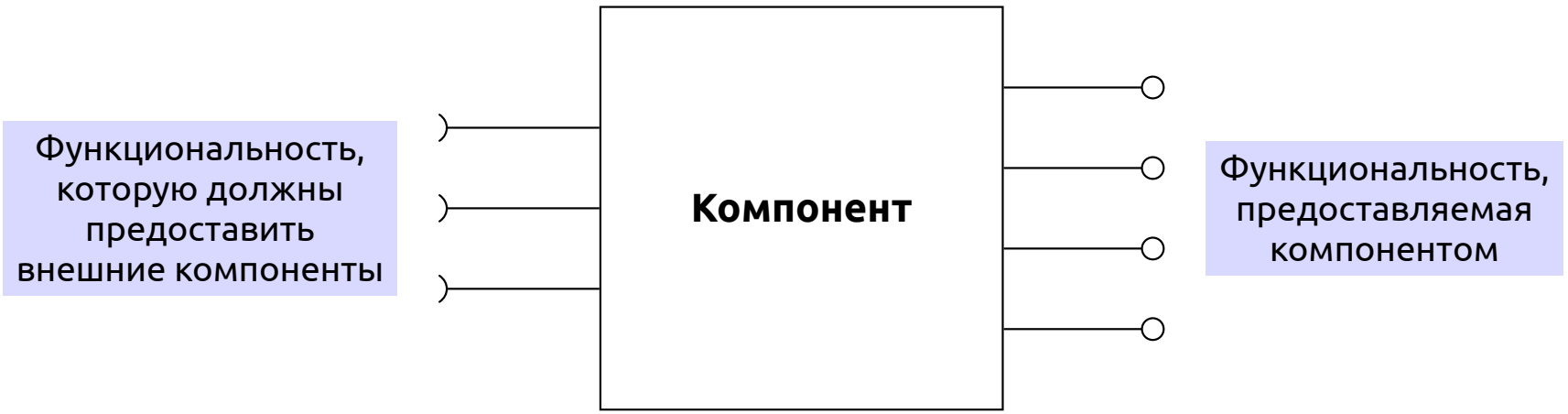
Характеристики компонентов

- **Стандартизация** — соответствие компонентной модели, в которой находится компонент. Модель определяет способ задания интерфейса, метаданных, документации компонента; инструменты для композиции и развертывания компонентов.
- **Независимость** — все зависимости от других компонентов должны быть явно заданы в интерфейсе.
- **Использование в составе системы** — все внешние взаимодействия должны производиться через задекларированные интерфейсы; компонент должен предоставить информацию о своих методах и свойствах.
- **Способность к развертыванию** — компонент должен предоставлять имплементацию, которая позволяет развернуть его в определенной среде с помощью стандартных средств.
- **Документация** — интерфейс компонента должен быть документирован для упрощения его повторного использования.

Отличия компонентов от объектов

- Объект выражает одно понятие предметной области; компонент может соответствовать нескольким связанным понятиям.
- Объекты тесно связаны между собой; компоненты связаны слабо, все зависимости должны быть задекларированы в интерфейсе компонента.
- Для объектов интерфейс и имплементация тесно связаны между собой (напр., определяются одним ЯП); для компонентов интерфейс и имплементация разграничены.
- Компоненты не имеют наблюдаемого состояния, все экземпляры компонента неразличимы (выполняется не всегда).

Интерфейсы компонентов



UML-диаграмма компонента

Сервис — развитие компонента, в котором отсутствуют зависимости от внешних компонентов.

Компонентная модель

Определение

Компонентная модель — совокупность стандартов, касающихся реализации, документирования и развертывания компонентов.

Аспекты компонентной модели:

- способ задания интерфейса (операции над компонентом, их параметры, генерируемые исключения);
- использование компонентов (способ идентификации, конфигурирование, …);
- развертывание (организация бинарного кода, необходимого для запуска компонента, напр., используемых сторонних библиотек).

Примеры компонентных моделей:

- Java EE / Enterprise Java Beans;
- Microsoft COM, Microsoft .NET;
- CORBA.

Разработка с компонентами

Типы процессов разработки с использованием компонентов:

- Разработка *для* повторного использования (англ. *development for reuse*) — разработка программных компонентов, которые могут использоваться в различных контекстах с минимальными затратами.

  Составляющие: обобщение функциональности компонента, минимизация зависимостей, стандартизация обработки исключительных ситуаций, сохранение в репозиторий, …

- Разработка *с* повторным использованием (англ. *development with reuse*) — разработка программных систем, использующих готовые компоненты и сервисы.

  Составляющие: выработка требований, подбор, тестирование и объединение компонентов в единую систему.

Аспектно-ориентированное программирование

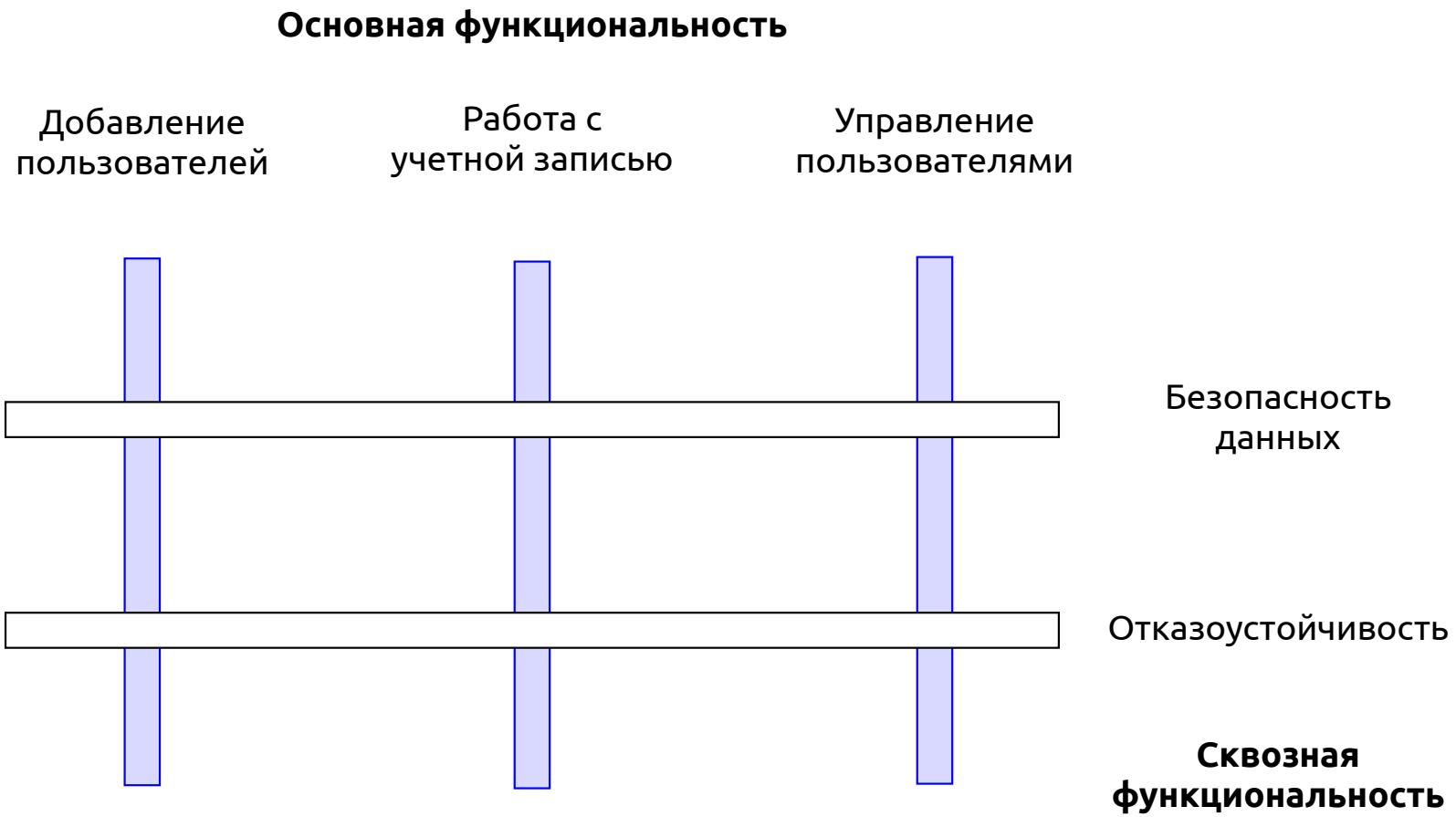
Определение

Аспектно-ориентированное программирование — парадигма программирования, основанная на обеспечении модульности при помощи выделения *сквозной функциональности* в отдельные модули.

Принцип разделения ответственности (англ. *separation of concerns*):
Каждый элемент компьютерной программы (в ООП — объекты, методы объектов) должен решать строго одну задачу; функции различных элементов не должны перекрываться.

Аспектно-ориентированные среды: AspectJ (на основе Java), AspectC++.

Сквозная функциональность



Сквозная функциональность

Определение

Сквозная функциональность (англ. *cross-cutting concern*) — это функциональность программы, которую невозможно полностью выделить в отдельные сущности (классы, методы).

Следствия сквозной функциональности:

- дублирование кода;
- сильные зависимости между элементами системы.

Примеры сквозной функциональности:

- ведение лога;
- аутентификация;
- синхронизация;
- кэширование;
- обработка транзакций.

Терминология АОП

Аспект — модуль, реализующий сквозную функциональность; содержит определение *срезов* и связанных с ними *советов*.

Совет (англ. *advice*) — код, реализующий сквозную функциональность. Совет может выполняться до, после или вместо основного кода.

Точка соединения (англ. *join point*) — точка в основной программе, в которой применяется *совет*.

Срез (англ. *pointcut*) — набор точек соединения, определяющий область применения совета.

Внедрение (англ. *weaving*) — изменение основного кода для добавления функциональности аспекта.

Модель точек соединения

Модель точек соединения (англ. *join point model*) — определение возможных мест для внедрения советов.

Точки соединения в AspectJ:

- вызов методов и конструкторов;
- инициализация классов или объектов;
- доступ или изменение полей объекта;
- обработка исключительных ситуаций.

Контекст выполнения: класс (объект `this` / класс, где находится код), пакет.

Примеры точек соединения

Точки соединения в AspectJ:

- `call(void show())`

  Вызов метода `show()` в произвольном классе.

- `call(* Point.set*())`

  Вызов метода с названием, начинающимся на `set`, в классе `Point`.

- `within(com.example.*) && call(*.new(int))`

  Вызов конструктора, принимающего один аргумент типа **`int`**, в классах из пакета `com.example`.

- `handler(ArrayOutOfBoundsException)`

  Обработка исключения `ArrayOutOfBoundsException`.

- `call(protected !static * *(..))`

  Вызов любого защищенного нестатического метода.

Пример

Основной код:

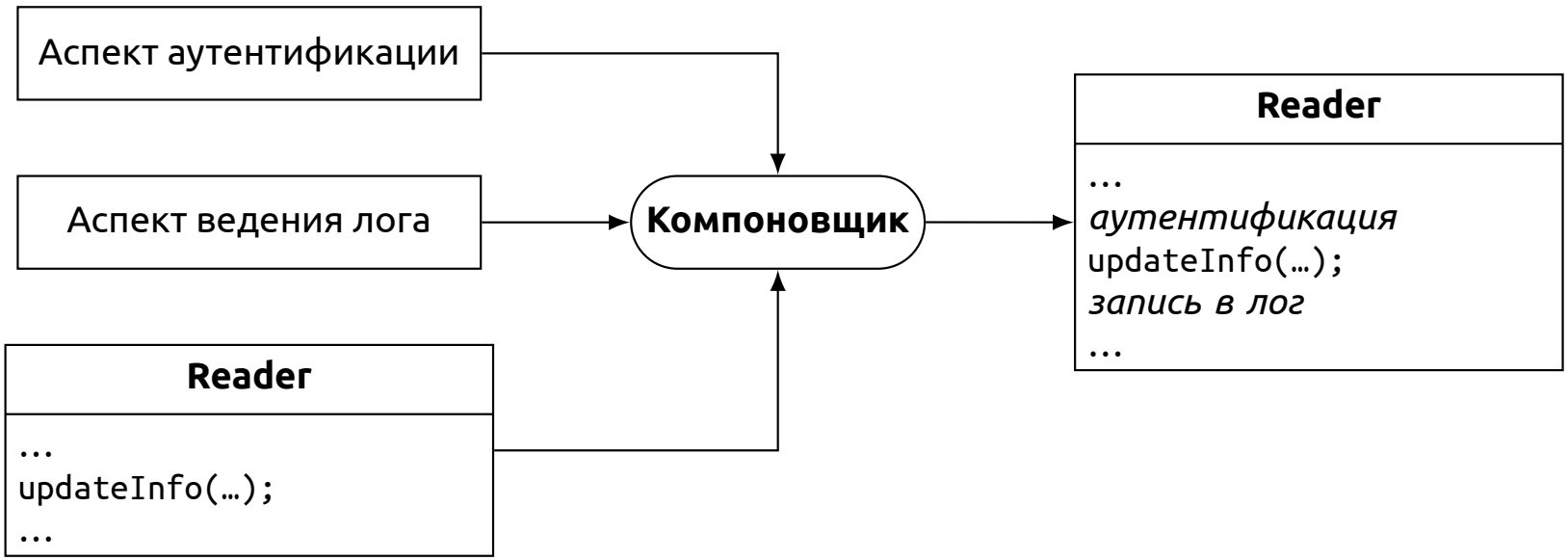
```
1  public class Point {
2      private double x;
3      private double y;
4
5      public void setX(double x) {
6          this.x = x;
7      }
8
9      public void setY(double y) {
10         this.y = y;
11     }
12 }
```

Пример

Код аспекта:

```
1  aspect LoggingAspect {
2      /* У аспектов могут быть переменные. */
3      Logger logger = //…
4
5      /* Определение среза: вызов метода Point.setX или Point.setY. */
6      pointcut setter(): target(Point)
7              && (call(void setX(double))
8              || call(void setY(double)));
9
10     /* Совет: занести в лог запись после выполнения метода. */
11     /* В переменные p и val извлекаются объект, вызвавший метод, и аргумент метода. */
12     after(Point p, double val): setter() && target(p) && args(val) {
13         logger.info("Set value " + val + " for point " + p);
14     }
15 }
```

Внедрение аспектов



Внедрение аспектов в AspectJ

Внедрение кода аспектов:

- на этапе компиляции (как фрагментов исходного кода);
- на этапе линковки (как скомпилированных фрагментов) — **основной способ**;
- на этапе выполнения (за счет рефлексии).

Сервис-ориентированная архитектура

Определение

Сервис-ориентированная архитектура (англ. *service-oriented architecture*) — парадигма программирования, в которой для обеспечения модульности применяются распределенные слабо связанные компоненты (*сервисы*), взаимодействующие с помощью стандартизованных протоколов.

Характеристики сервисов:

- модульность — сервис представляет логически связанные функции в определенной предметной области с заданными входами и выходами;
- автономность — отсутствие наблюдаемых для пользователей зависимостей;
- сокрытие реализации — рассматривается как «черный ящик».

Веб-сервисы

Определение

Веб-сервис — сервис, идентифицирующийся по адресу URL и взаимодействующий по Интернету с помощью высокоуровневых протоколов на основе HTTP, TCP/IP.

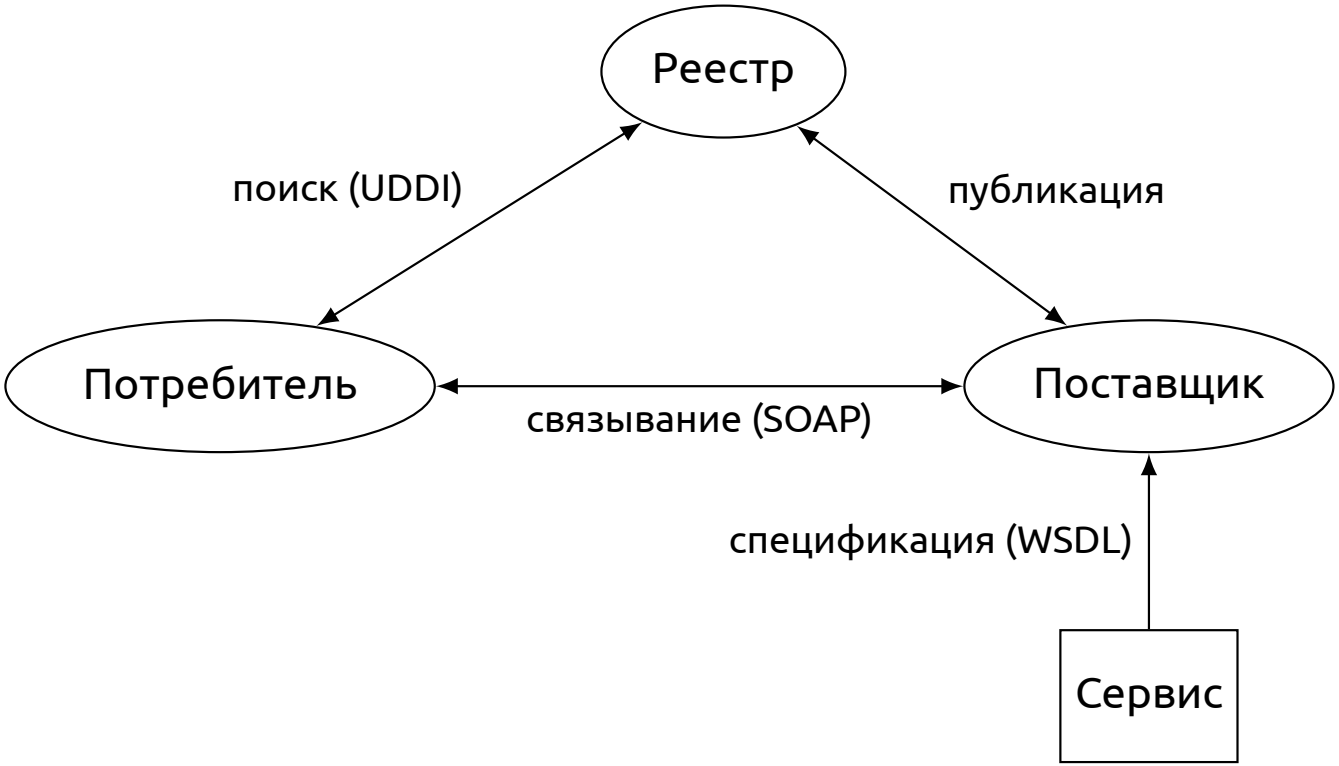


Схема взаимодействия с веб-сервисом

Спецификация веб-сервисов

Содержимое спецификации:

- Операции, предоставляемые сервисом ($\simeq$ методы в ООП), соответствующие входные и возвращаемые данные;
- формат сообщений для взаимодействия с сервисом;
- (необязательно) типы данных, используемые в сообщениях;
- определение конкретных протоколов доступа к операциям (с помощью SOAP или других методов).

Язык описания: WSDL (web service definition language) — на основе XML.

Использование веб-сервисов

Протокол: SOAP (simple object access protocol) — на основе XML.

Сообщение сервису:

- заголовок сообщения — нефункциональные характеристики запроса (приоритетность, время обработки, …);
- тело сообщения — список операций веб-сервиса и соответствующих параметров.

Ответное сообщение:

- тело сообщения — список с результатами выполнения операций;
- отказы — информация об отказах при проведении операций.

Разработка веб-сервисов

Цели разработки:

- минимизация количества обращений к сервису;
- скрытие состояния сервиса (хранение состояния — задача клиента; состояние может передаваться в сообщениях).

Этапы разработки:

1. определение функциональности;
2. описание операций и сообщений;
3. имплементация;
4. тестирование;
5. развертывание.

Разработка веб-сервисов

Средства автоматизации:

- Инструменты для создания WSDL-описания на основе классов.
- Серверы приложений — веб-серверы, обеспечивающие автоматическое развертывание и выполнение кода веб-сервиса, а также разбор / формирование SOAP-сообщений и обмен данными со средой, где выполняется сервис.
- Инструменты для автоматической генерации кода на основе WSDL для доступа к сервису как к локальному объекту.
- Средства для композиции сервисов (Business Process Execution Language, BPEL).

Альтернативы веб-сервисам

Недостатки веб-сервисов на основе SOAP:

- «тяжеловесность» используемых протоколов SOAP и WSDL;
- необходимость в комплексных средствах поддержки (сервере приложений и т. п.);
- отсутствие привязок типов данных к языкам программирования.

Альтернативные реализации СОА:

- REST;
- CORBA;
- DCOM;
- Java RMI;
- Apache Thrift.

Выводы

1. ООП в чистом виде обладает некоторыми недостатками, затрудняющими повторное использование кода. Для улучшения повторного использования используются парадигмы на основе ООП, в частности компонентно-ориентированное, аспектное и сервисное программирование.
2. Компонент — самостоятельный программный продукт, реализующий логически замкнутый набор функций системы. В отличие от объектов и классов, интерфейс и реализация компонентов всегда разграничены, что упрощает многоязыковую и мультиплатформенную разработку.
3. Аспект — модуль, объединяющий сквозную функциональность (напр., ведение лога, обработка исключений). Использование аспектов позволяет избежать дублирования кода и избыточных связей между элементами системы.
4. Сервис — развитие идеи компонента для распределенных приложений. Для коммуникации с сервисами используются сетевые протоколы.

Материалы

Sommerville, Ian

Software Engineering.

Pearson, 2011. — 790 p.

Лавріщева К. М.

Програмна інженерія (підручник).

К., 2008. — 319 с.

Heineman, G. T.; Councill, W. T.

Component-Based Software Engineering: Putting the Pieces Together.

(компоненты и сервисы)

Jacobson, I.; Ng P.-W.

Aspect-Oriented Software Development with Use Cases.

(аспекты)

Спасибо за внимание!