Domain-Driven Design

Юрий Литвинов
yurii.litvinov@gmail.com

9

Domain-Driven Design

Domain-Driven Design — модная нынче методология проектирования, использующая предметную область как основу архитектуры системы

- Архитектура приложения строится вокруг **Модели предметной области**
- Модель определяет **Единый язык**, на котором общаются и разработчики, и эксперты, описывая естественными фразами то, что происходит и в программе, и в реальности
- Модель — это не только диаграммы, это ещё (и прежде всего) код, и устное общение

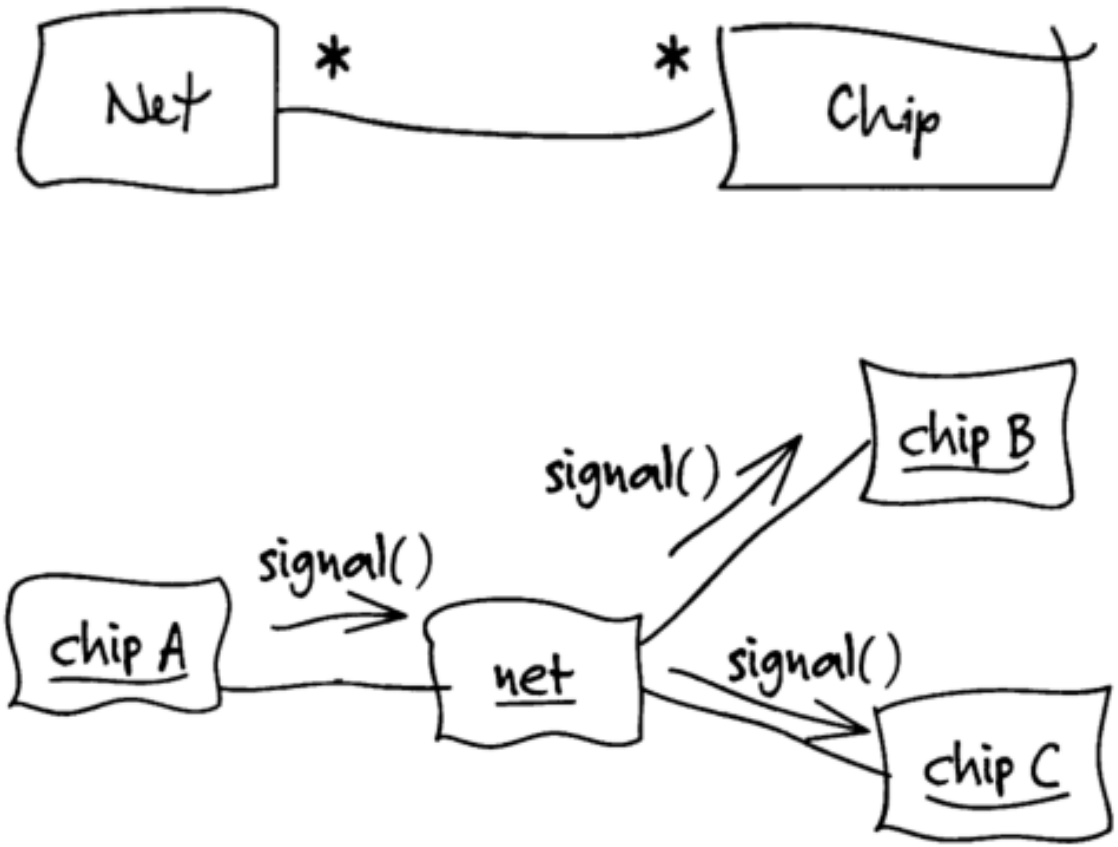
Причём тут UML — DDD даёт ответ на вопрос “откуда брать эти все классы” и позволяет целенаправленно уточнять и улучшать модель. Особенно полезно, когда предметная область не очень знакома (как будет в домашке).

Книжка

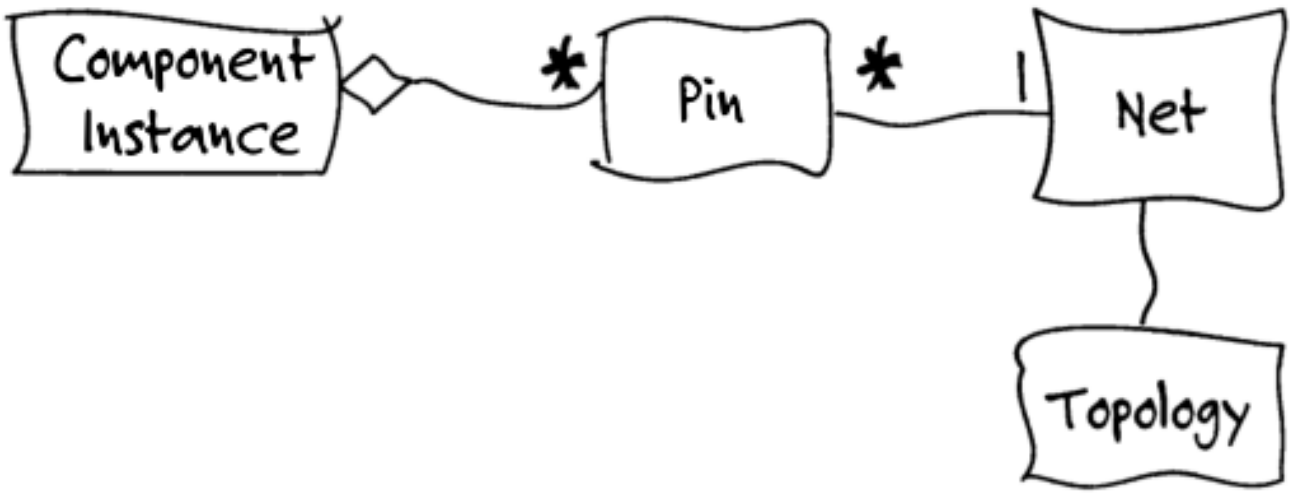
Эрик Эванс, “Предметно-ориентированное проектирование. Структуризация сложных программных систем”. М., “Вильямс”, 2010, 448 стр.

Domain-Driven Design, анализ

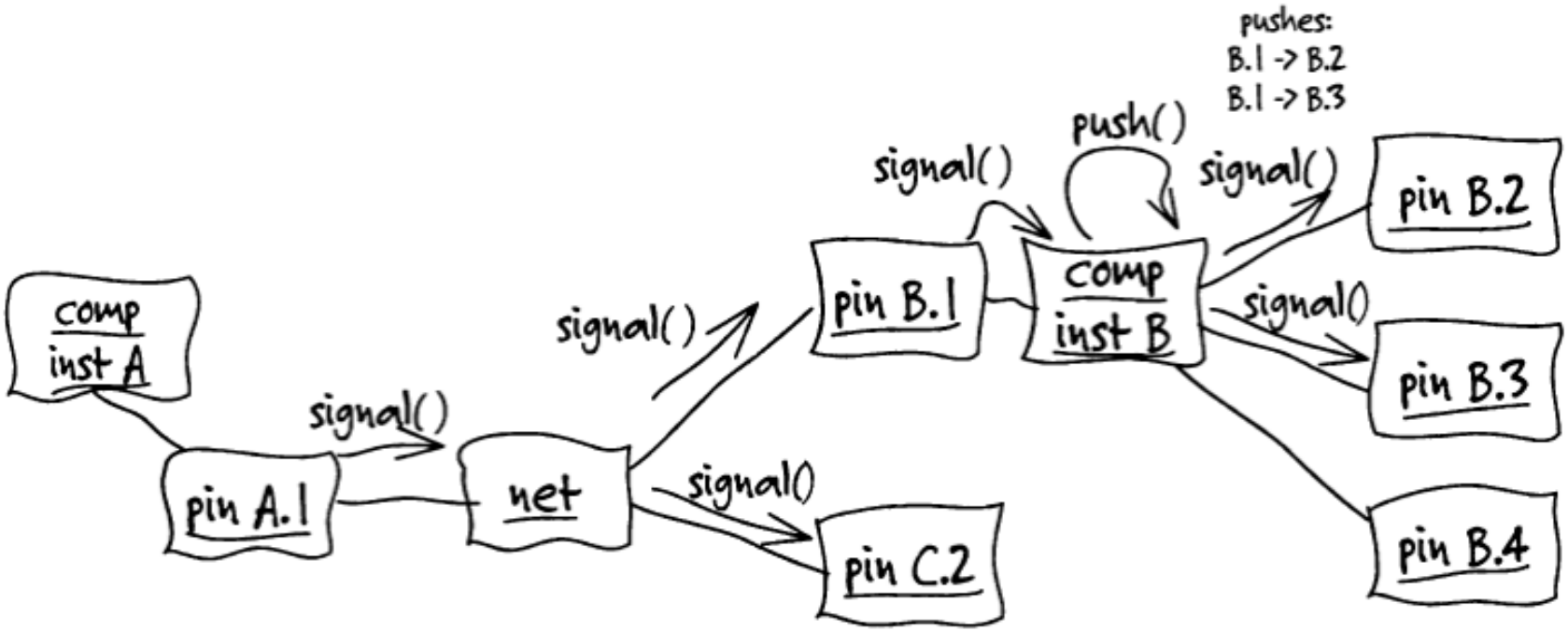
Пример: печатные платы



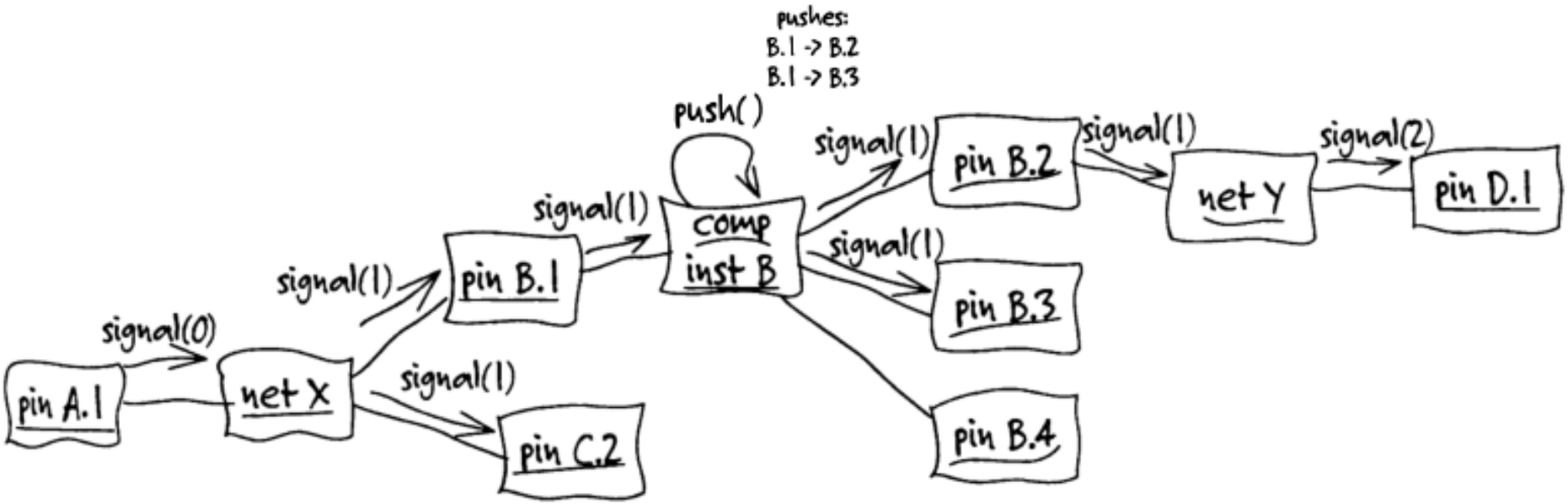
Печатные платы, топология



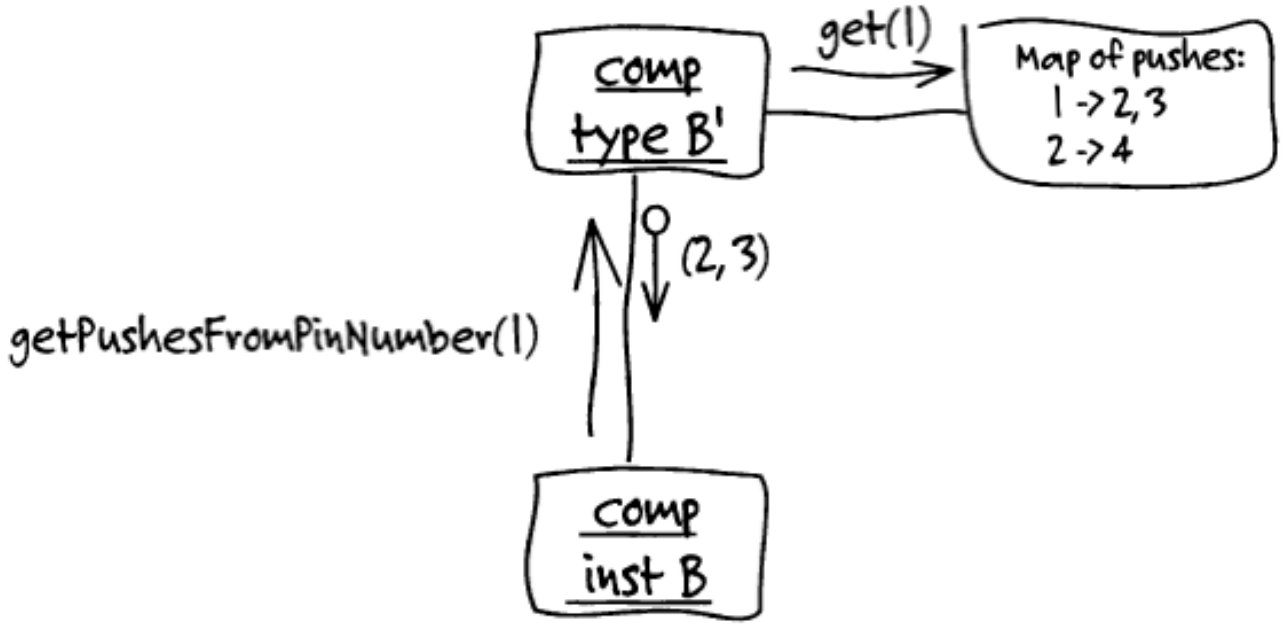
Печатные платы, сигналы



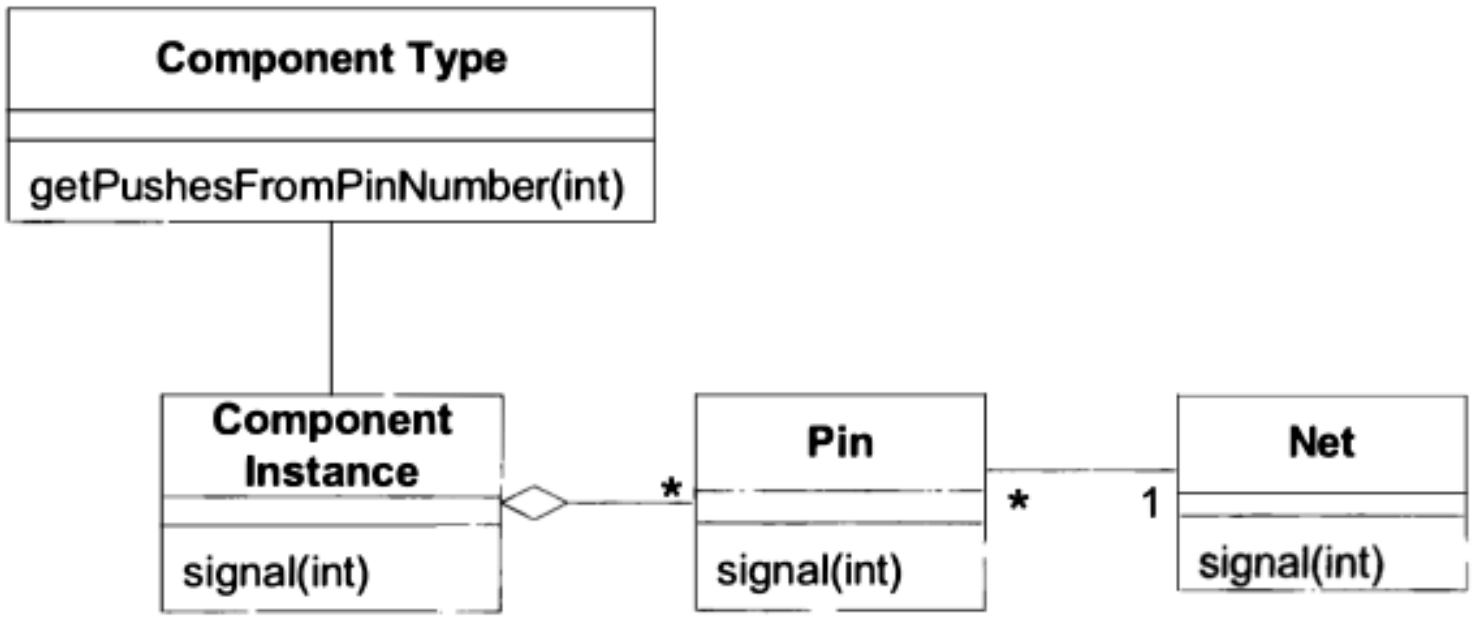
Печатные платы, прозванивание



Печатные платы, типы



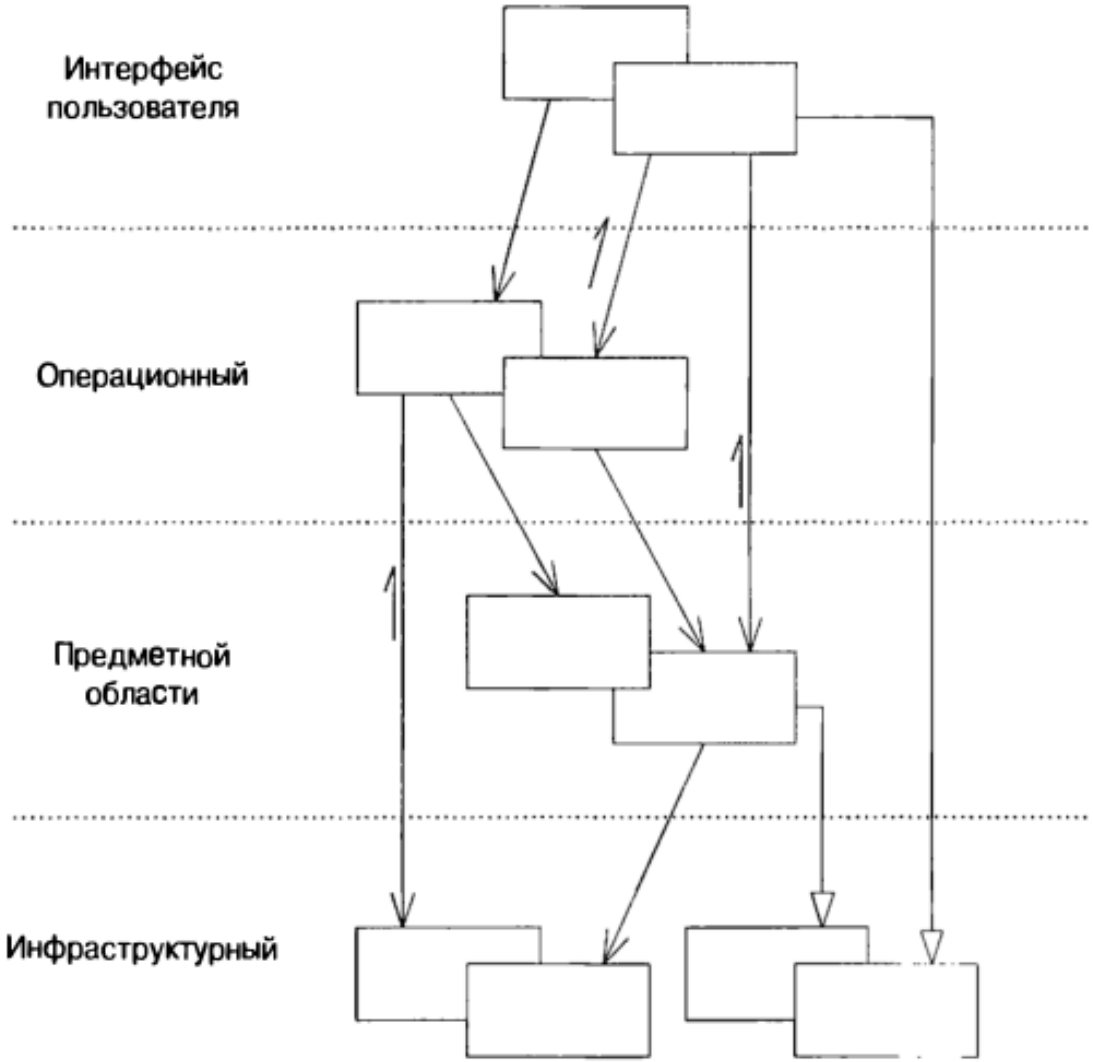
Печатные платы, модель



Выводы: правила игры

- Детали реализации не участвуют в модели
  - “База данных? Какая база данных?”
- Должно быть можно общаться, пользуясь только именами классов и методов
- Не нужные для текущей задачи сущности предметной области не должны быть в модели
- Могут быть скрытые сущности, которые следует выделить явно
  - при этом объяснив экспертам их роль в реальной жизни и послушав их мнение
  - например, различные ограничения могут стать отдельными классами
- Диаграммы объектов могут быть очень полезны

Изоляция предметной области



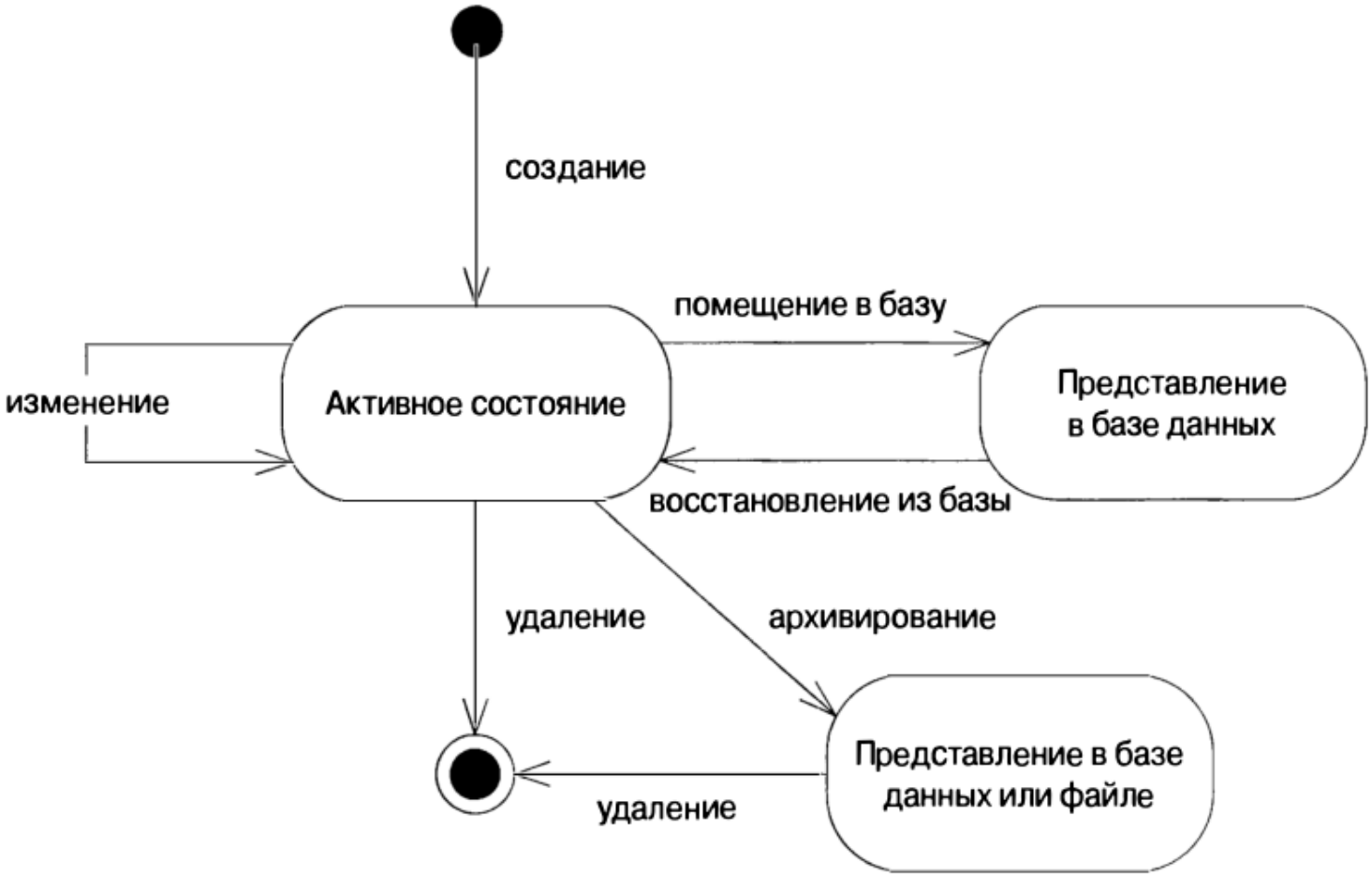
Антипаттерн “Умный GUI”

- А давайте всю бизнес-логику писать прямо в обработчиках на форме
- Код GUI напрямую работает с БД
- Делает невозможным проектирование по модели
- Не всегда плохо
  - Применимы средства быстрой разработки приложений
  - Прирост производительности на начальных этапах
  - Легко приделывать новые фичи и переписывать старые
- Не всегда хорошо
  - Очень сложно переиспользование
  - Сложно реализовать сложное поведение (зато легко простое)
  - Сложно интегрироваться

Основные структурные элементы модели

- **Сущность (Entity)** — объект, обладающий собственной идентичностью
  - Нужна операция идентификации
  - Нужен способ поддержания идентичности
- **Объект-значение (Value object)** — объект, полностью определяемый своими атрибутами
  - “Лучше”, чем сущность
  - Как правило, немутабельны
  - Могут быть разделяемыми
- **Служба (Service)** — объект, представляющий операцию
  - Как правило, не имеет собственного состояния
  - Операции нет естественного места в других классах модели
- **Модуль (Module)** — смысловые части модели

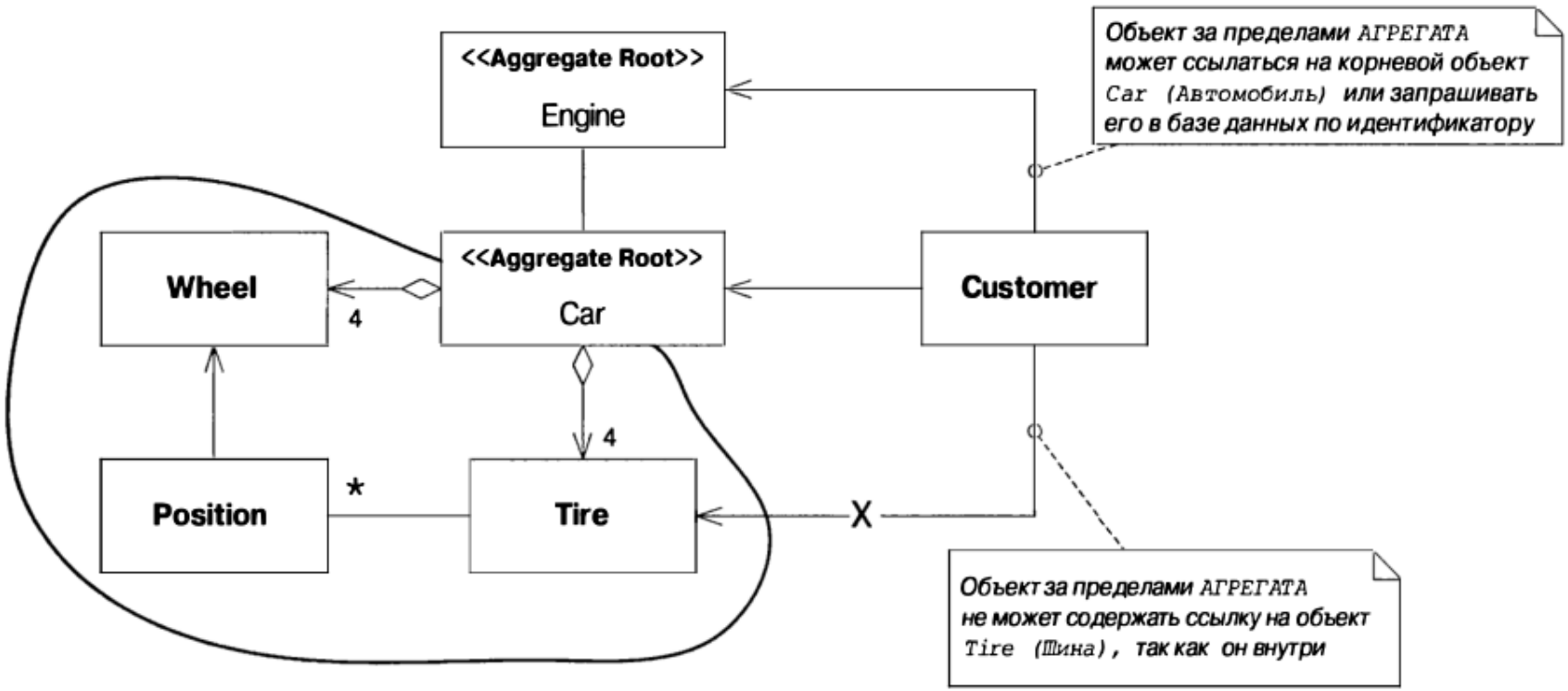
Жизненный цикл объекта



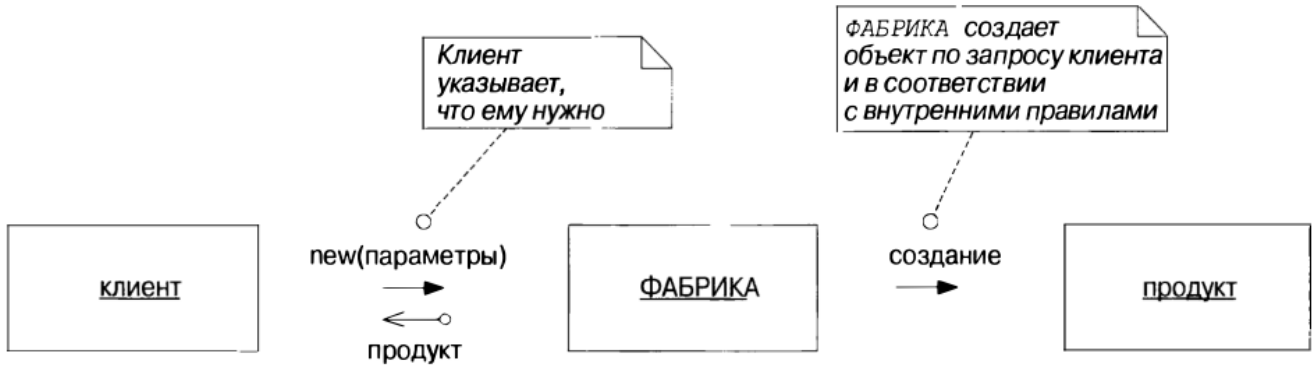
Агрегаты

- **Агрегат** — изолированный кусок модели, имеющий **корень** и **границу**
- Корень — глобально идентичный объект-сущность
- Остальные объекты в агрегате идентичны локально
- Извне агрегата можно хранить ссылку только на корень
  - Отдавать временную ссылку можно
- Корень отвечает за поддержание инвариантов всего агрегата

Агрегат, пример



Фабрика

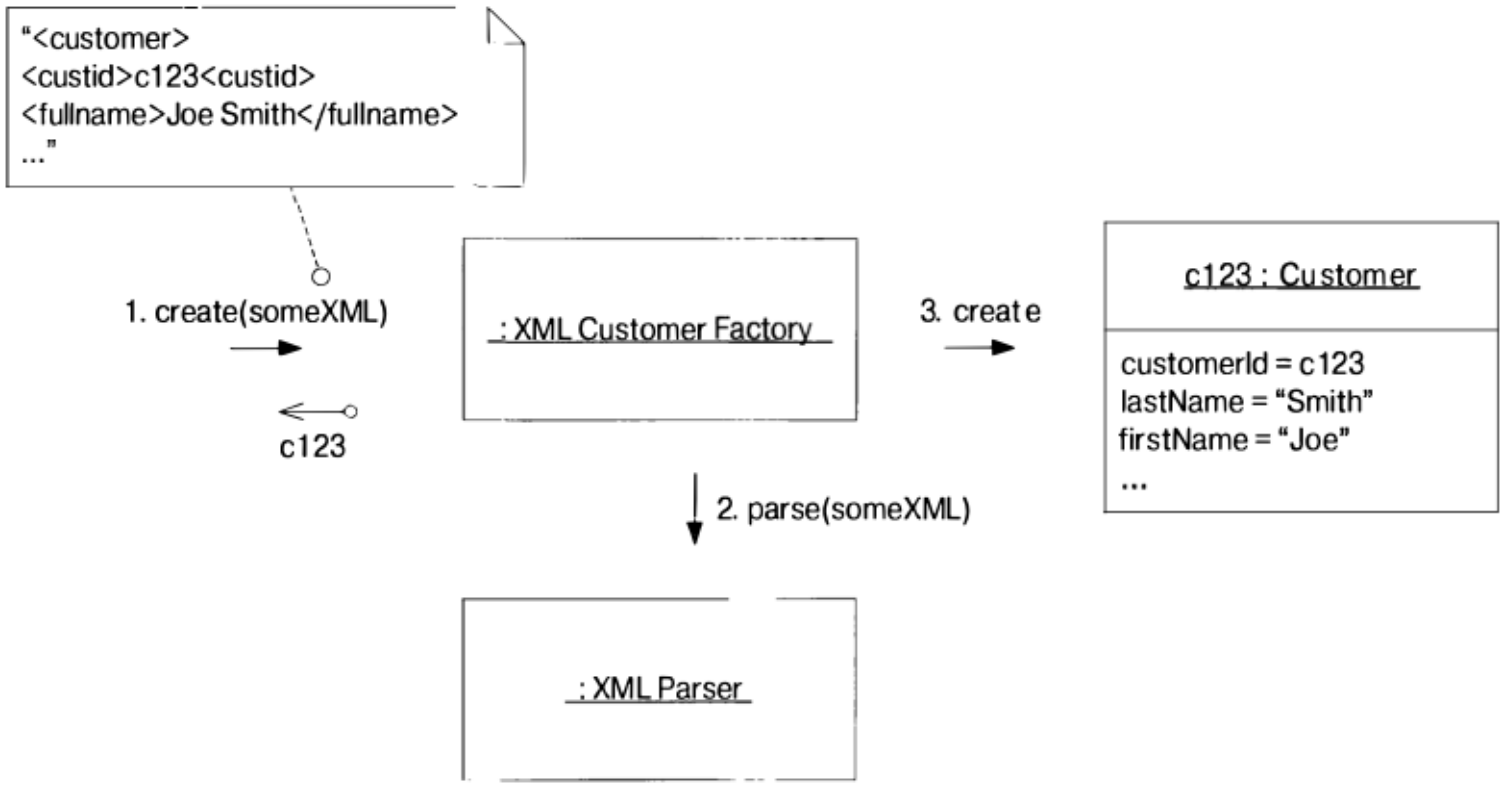


Фабрика служит для создания объектов или агрегатов

- Скрывает внутреннее устройство конструируемого объекта
  - Операция создания “атомарна” и обеспечивает инварианты
- Изолирует сложную операцию создания
- Как правило, не имеет бизнес-смысла, но является частью модели
- Реализуется аж несколькими разными паттернами

Пример

Фабрика, использующаяся для восстановления объекта



Хранилище (Repository)

Репозиторий хранит объекты и предоставляет к ним доступ

- Может инкапсулировать запросы к БД
- Может использовать фабрики
- Может обладать развитым интерфейсом запросов

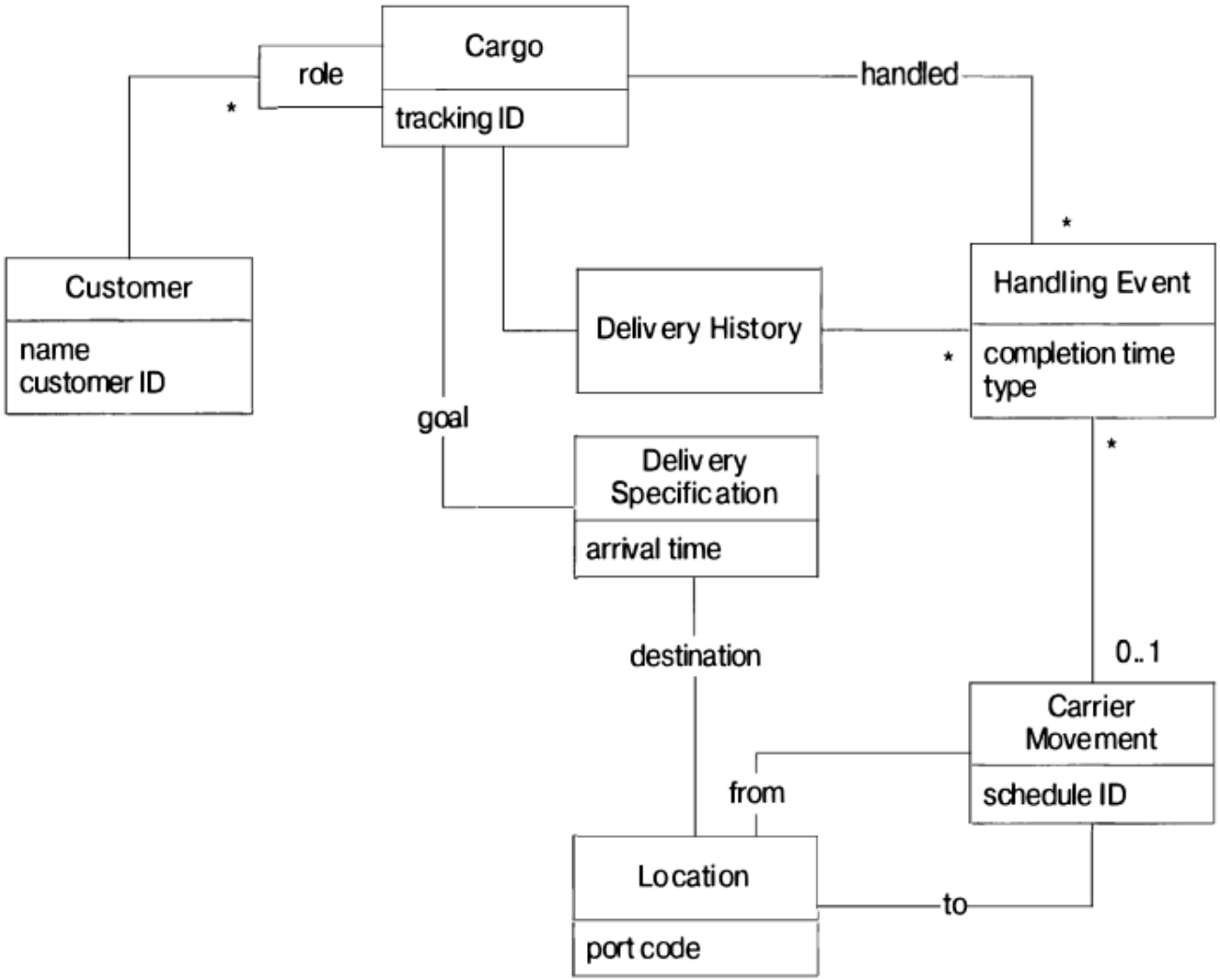
Пример, система грузоперевозок

Требования:

1. Отслеживать ключевые манипуляции с грузом клиента
2. Оформлять заказ заранее
3. Автоматически высылать клиенту счет-фактуру по достижении грузом некоторого операционного пункта маршрута

- В работе с Грузом (Cargo) участвует несколько Клиентов (Customers), каждый из которых играет свою роль (Role)
- Должна задаваться (be specified) цель (goal) доставки груза
- Цель (goal) доставки груза достигается в результате последовательности Переездов (Carrier Movement), которые удовлетворяют Заданию (Specification)

Модель


Cargo
tracking ID
role
handled
*
*
Customer
name
customer ID
Delivery History
Handling Event
completion time
type
*
*
goal
Delivery
Specification
arrival time
*
destination
0..1
Carrier
Movement
schedule ID
from
Location
port code
to

Уровень приложения

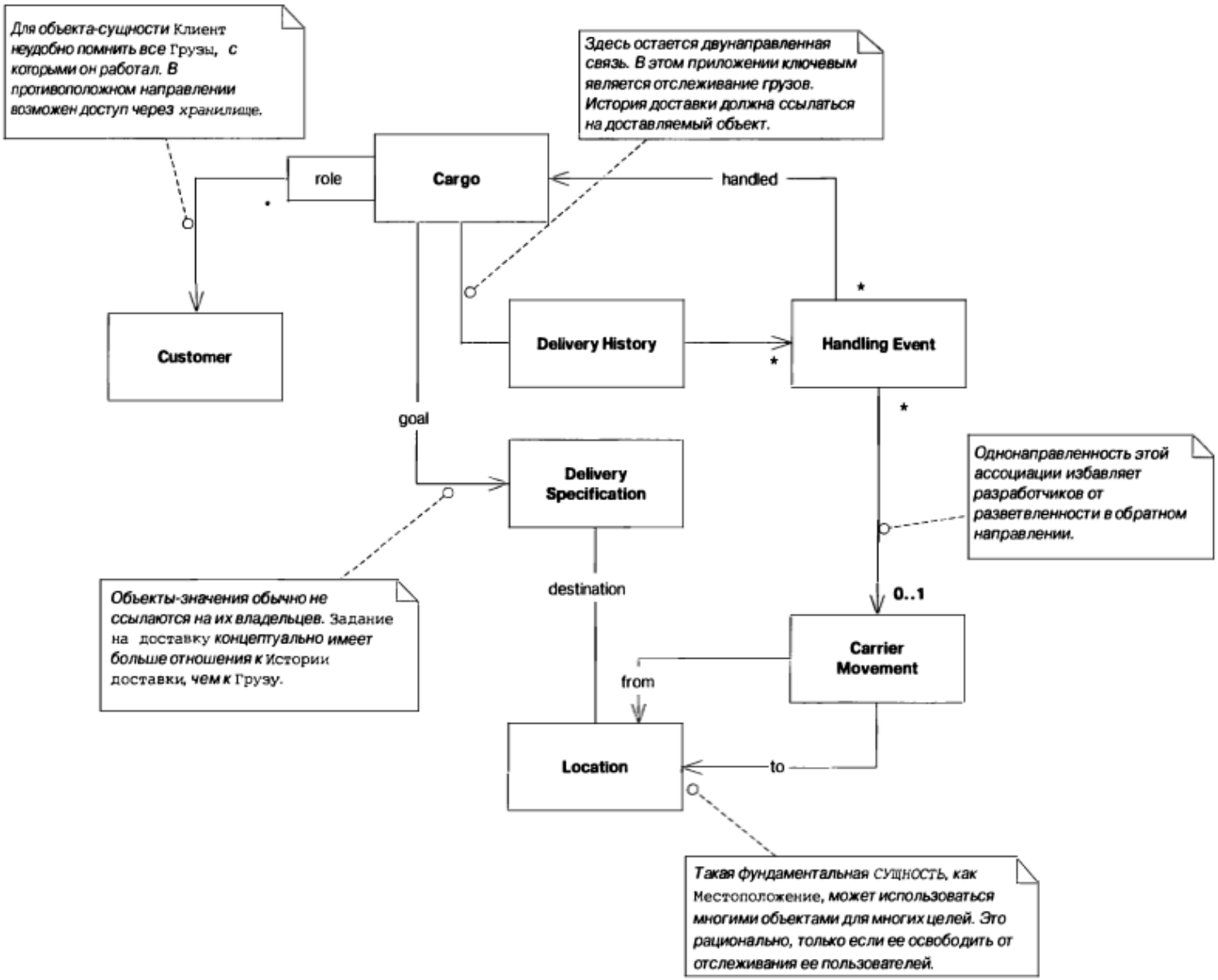
Применим уровневую архитектуру и выделим операции уровня приложения:

- Маршрутный запрос (Tracking Query) — манипуляции с конкретным грузом
- Служба резервирования (Booking Application) — позволяет заказать доставку нового груза
- Служба регистрации событий (Incident Logging Application) — регистрирует действия с грузом (связана с маршрутным запросом)

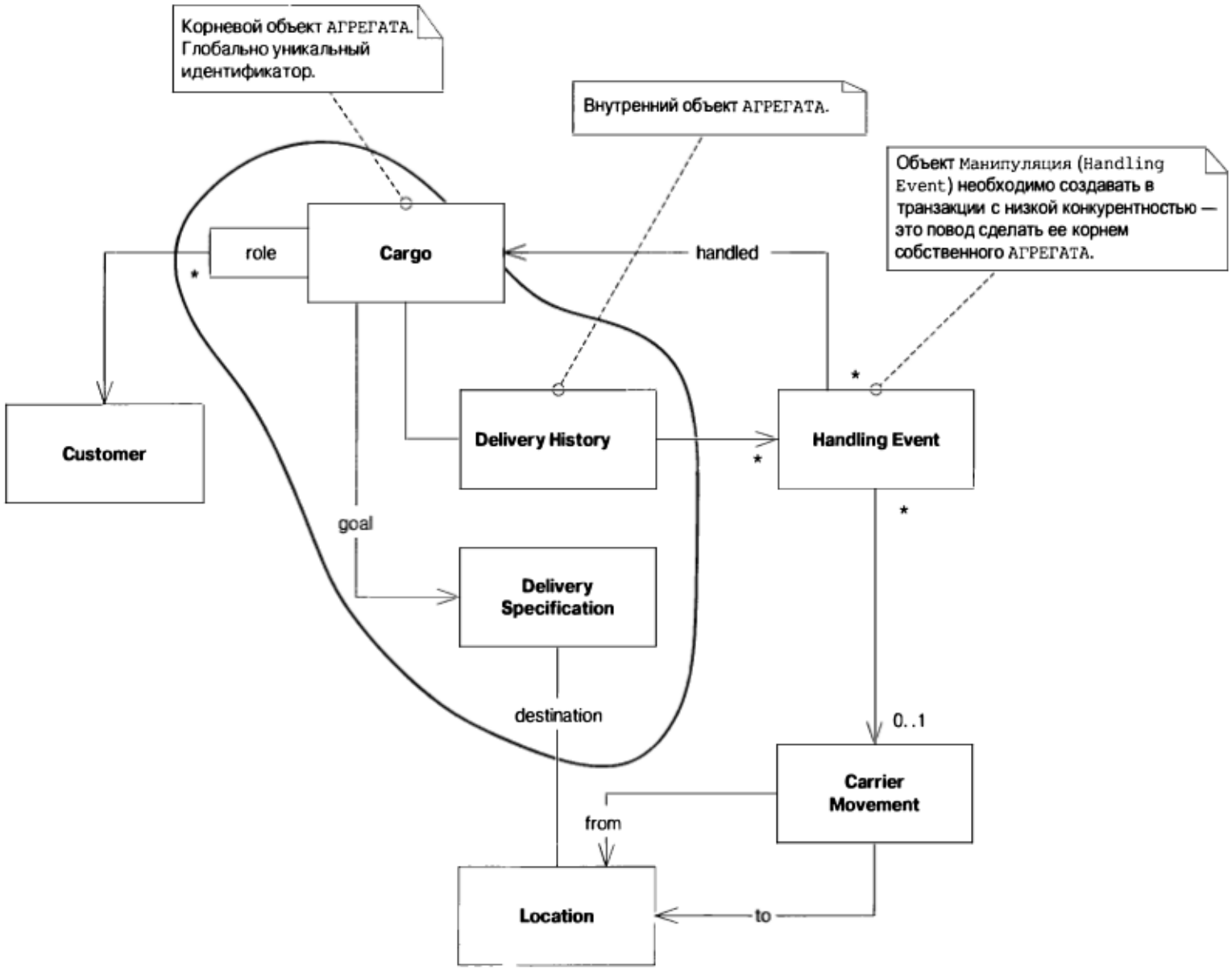
Сущности или значения?

- **Клиент (Customer)** — сущность
- **Груз (Cargo)** — сущность
- **Манипуляция (Handling Event)** и **Переезд (Carrier Movement)** — сущности
- **Местоположение (Location)** — сущность
- **История доставки (Delivery History)** — сущность, локально идентична в пределах агрегата “Груз”
- **Задание на доставку (Delivery Specification)** — значение
- Всё остальное — значения

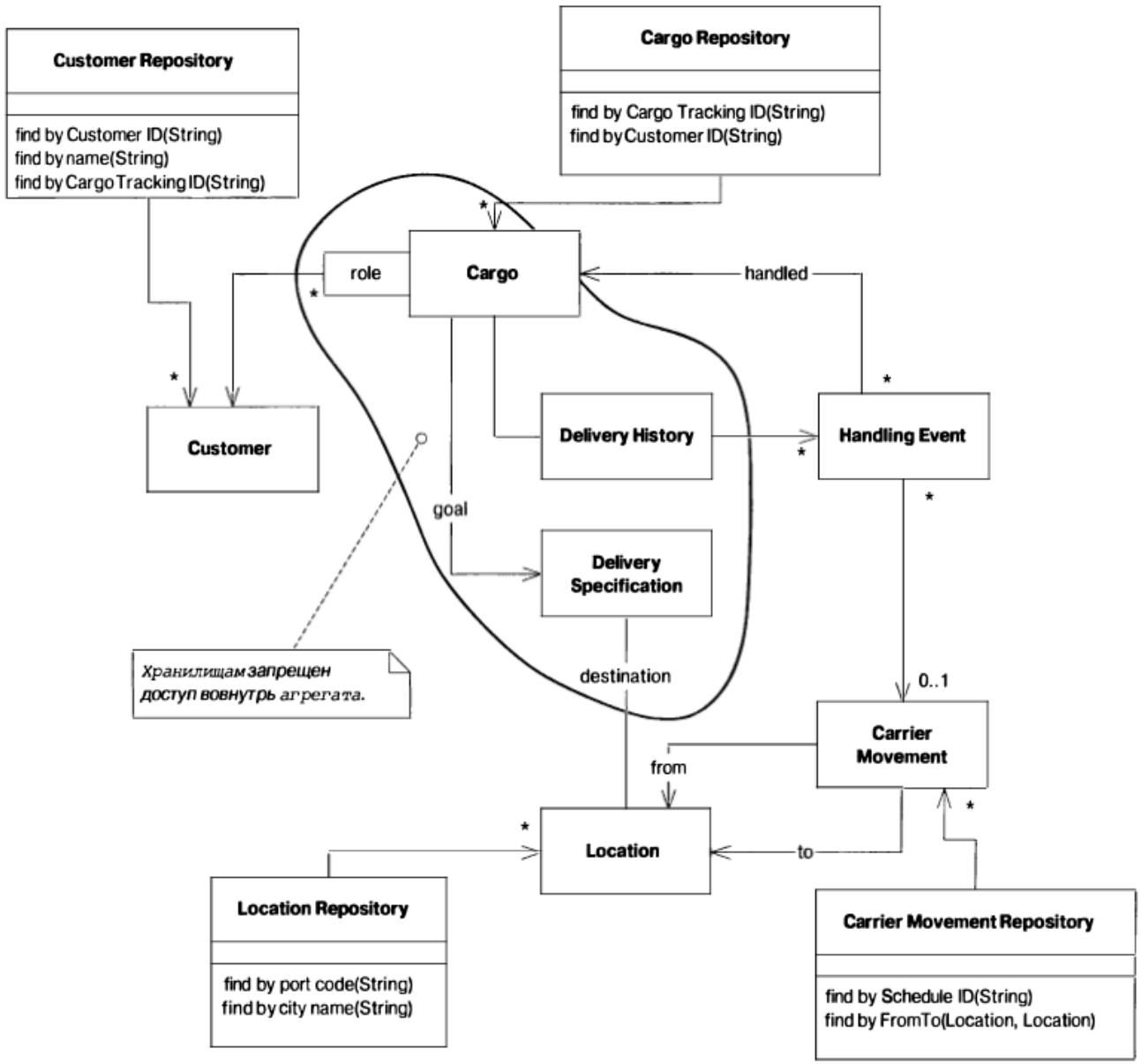
Направленность ассоциаций



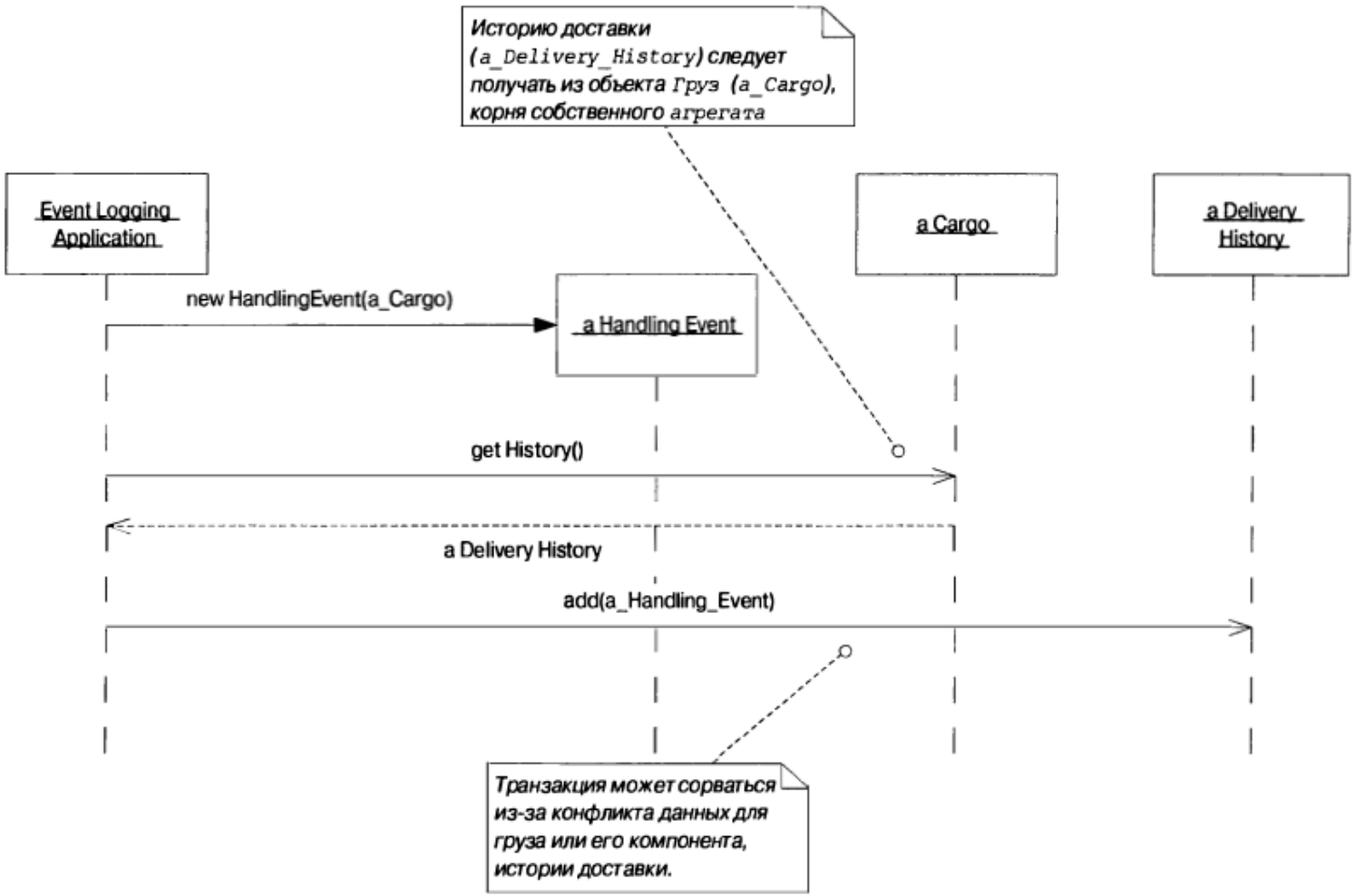
Границы агрегатов



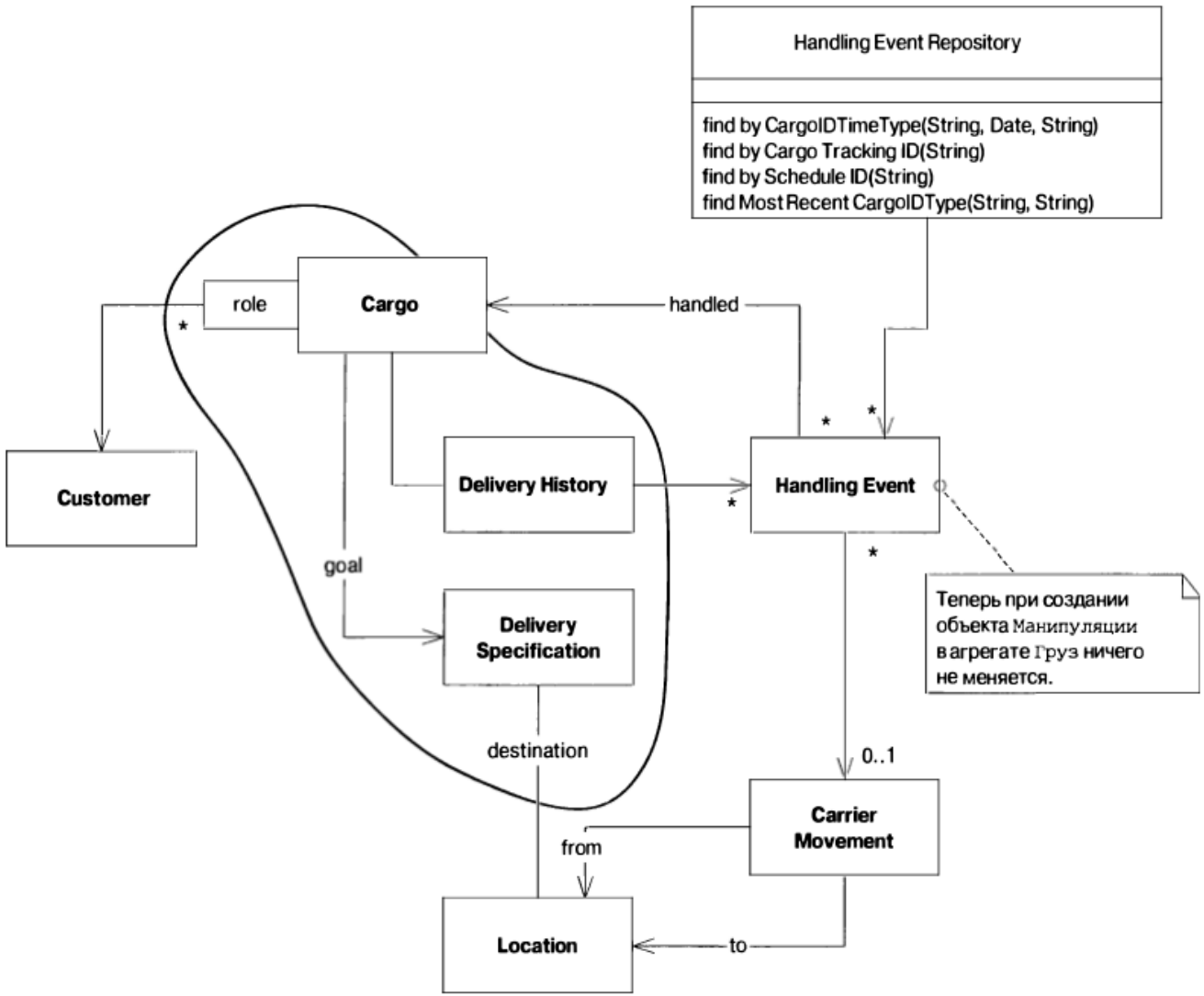
Хранилища


Customer Repository
find by Customer ID(String)
find by name(String)
find by Cargo Tracking ID(String)
Cargo Repository
find by Cargo Tracking ID(String)
find by Customer ID(String)
role
Cargo
handled
Customer
Delivery History
Handling Event
goal
Delivery Specification
Хранилищам запрещен доступ вовнутрь агрегата.
destination
0..1
Carrier Movement
from
Location
to
Location Repository
find by port code(String)
find by city name(String)
Carrier Movement Repository
find by Schedule ID(String)
find by FromTo(Location, Location)

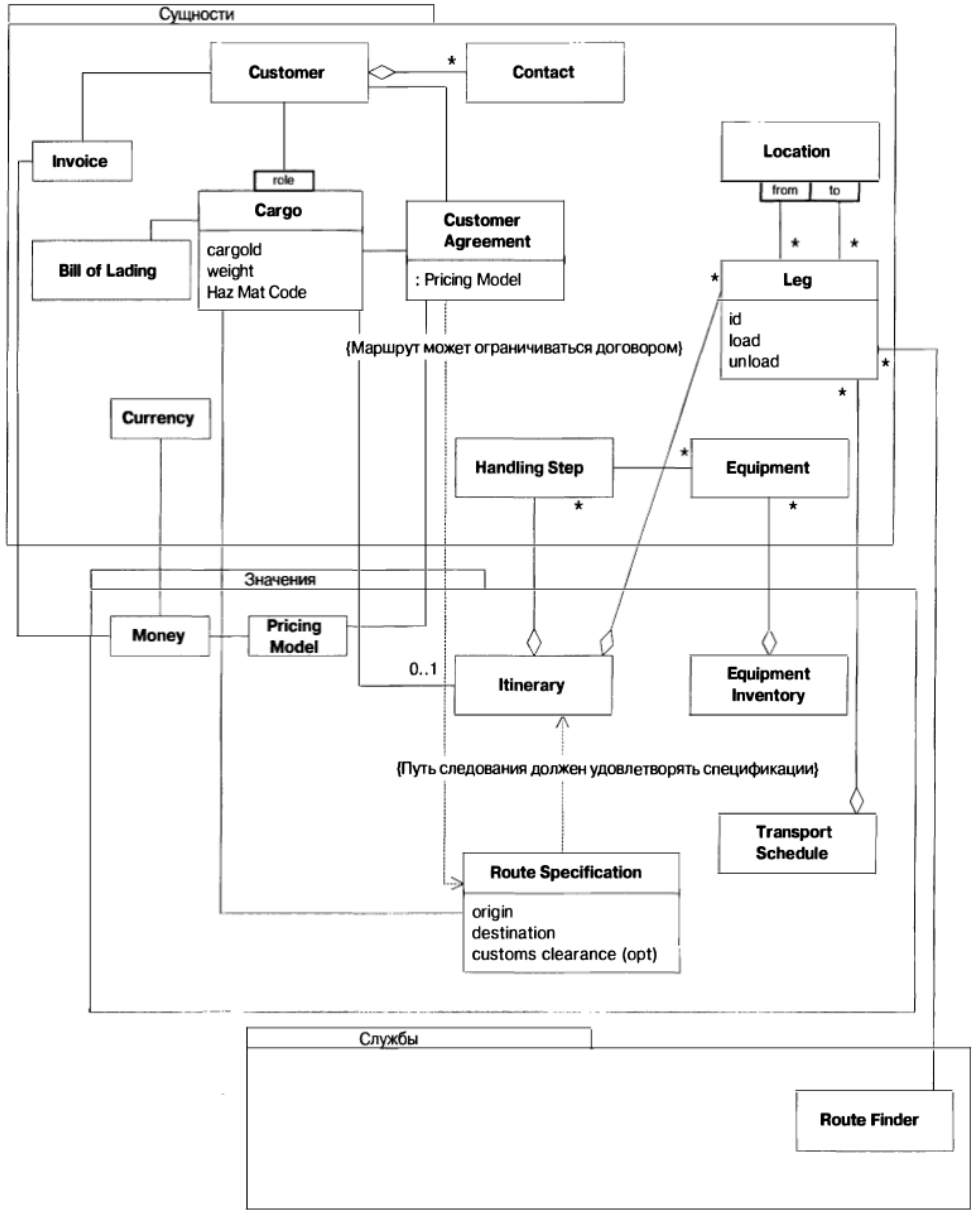
Тестовый сценарий, добавление события



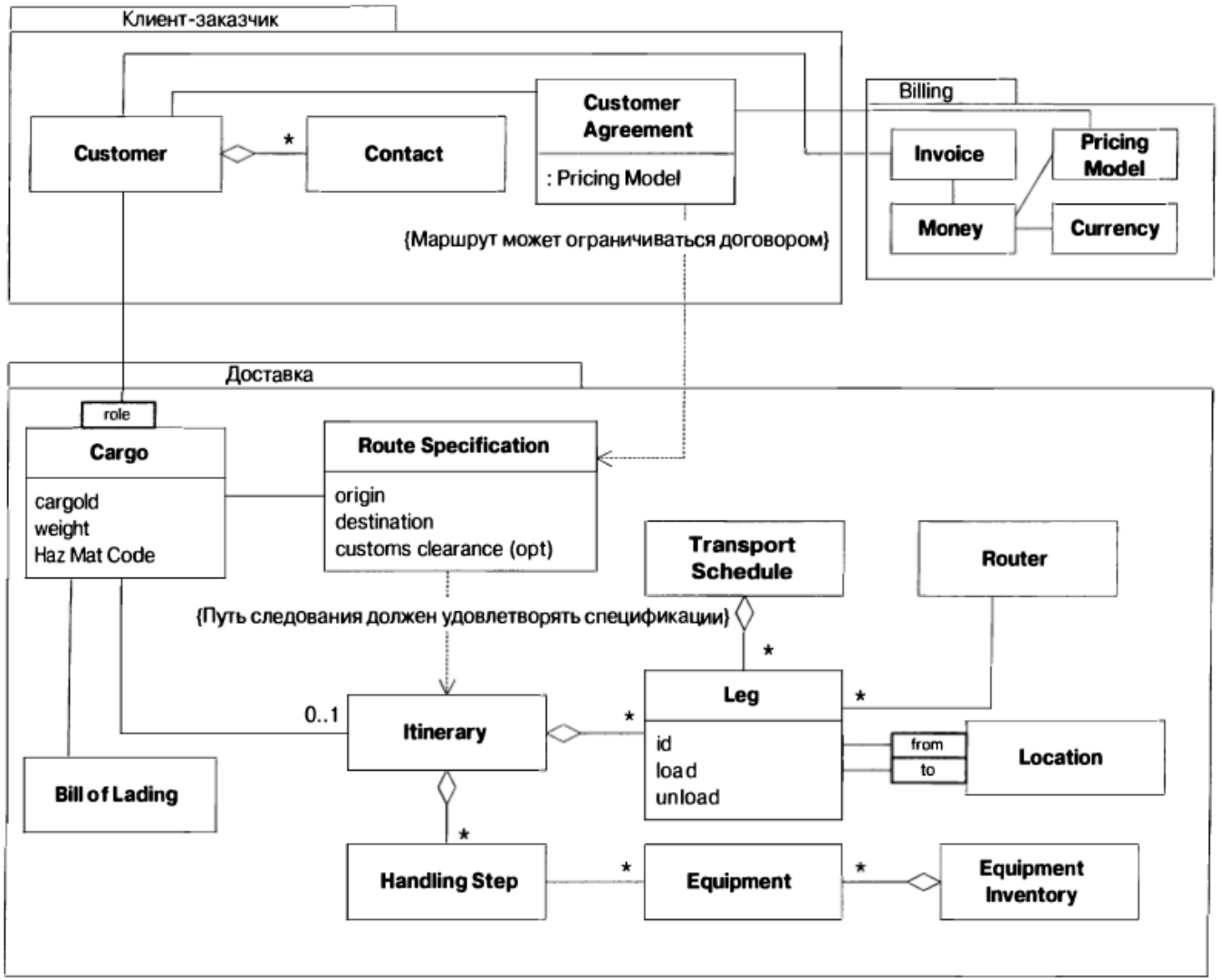
Рефакторинг, не хранить события явно



Разбиение по модулям, плохо

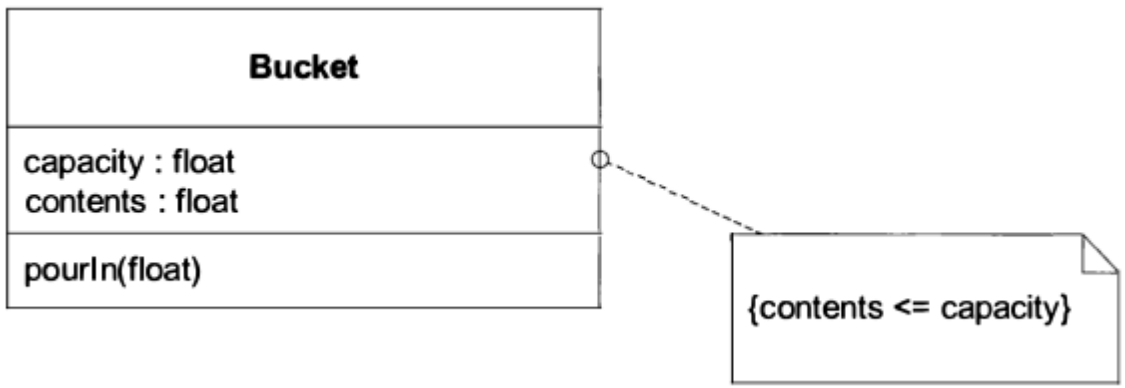


Разбиение по модулям, хорошо



Моделирование ограничений

Простой пример



Код, до

```
class Bucket {
    private float capacity;
    private float contents;

    public void pourIn(float addedVolume) {
        if (contents + addedVolume > capacity) {
            contents = capacity;
        } else {
            contents = contents + addedVolume;
    }
}
```

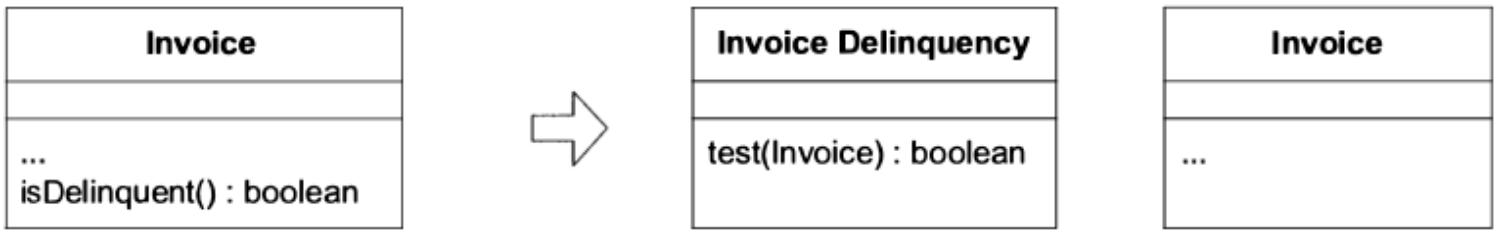
Код, после

```
class Bucket {
    private float capacity;
    private float contents;

    public void pourIn(float addedVolume) {
        float volumePresent = contents + addedVolume;
        contents = constrainedToCapacity(volumePresent);
    }

    private float constrainedToCapacity(float volumePlacedIn) {
        if (volumePlacedIn > capacity) return capacity;
        return volumePlacedIn;
    }
}
```

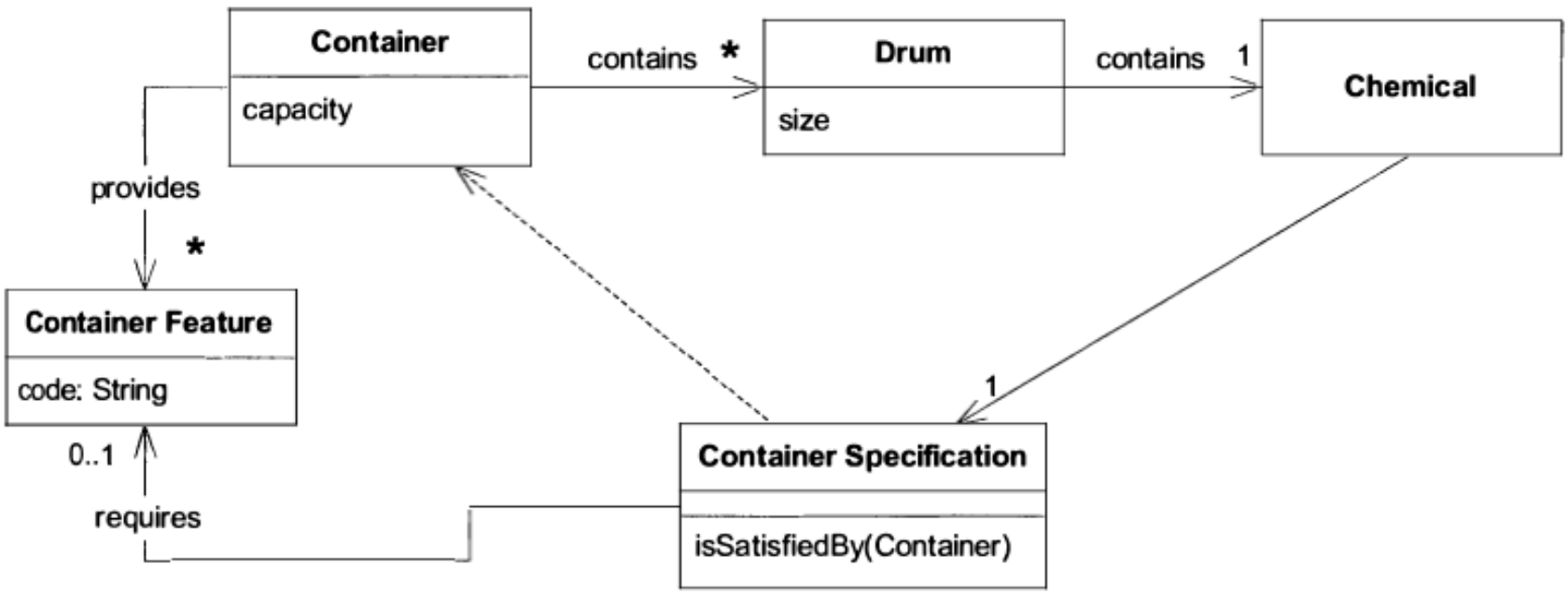
Паттерн “Спецификация”



Спецификация инкапсулирует ограничение в отдельном объекте

- Предикат
- Может быть использована для выборки или конструирования объектов

Пример: склад химикатов



Код, спецификация

```
public class ContainerSpecification {
    private ContainerFeature requiredFeature;

    public ContainerSpecification(ContainerFeature required) {
        requiredFeature = required;
    }

    boolean isSatisfiedBy(Container aContainer) {
        return aContainer.getFeatures().contains(requiredFeature);
    }
}
```

Код, контейнер

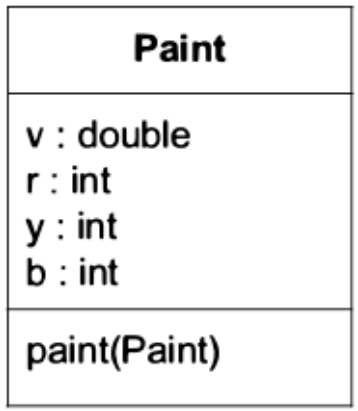
```
boolean isSafelyPacked() {
    Iterator it = contents.iterator();
    while (it.hasNext()) {
        Drum drum = (Drum) it.next() ;
        if (!drum.containerSpecification().isSatisfiedBy(this))
            return false ;
    }
    return true;
}
```

Приёмы обеспечения гибкости архитектуры

- Говорящие интерфейсы
- Функции без побочных эффектов
- Assertions
- Концептуальные контуры
- Изолированные классы
- Замкнутые операции

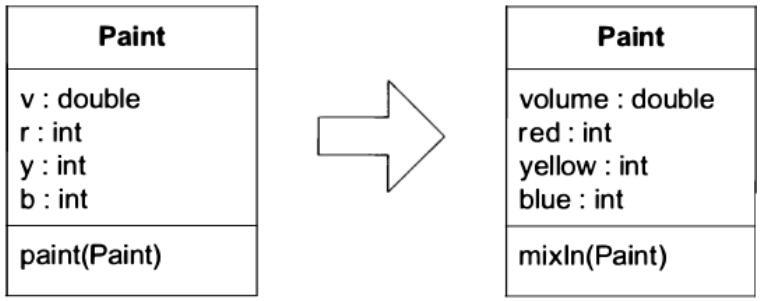
Пример рефакторинга, смешивание красок

Начальное состояние




```
public void paint (Paint paint } {
   v = v + paint.getV();  // После смешивания объем суммируется
// Опущено много строк сложного расчета смешивания цветов,
// который заканчивается присваиванием новых значений
// компонентов r (красного), b (синего) и y (желтого).
}
```

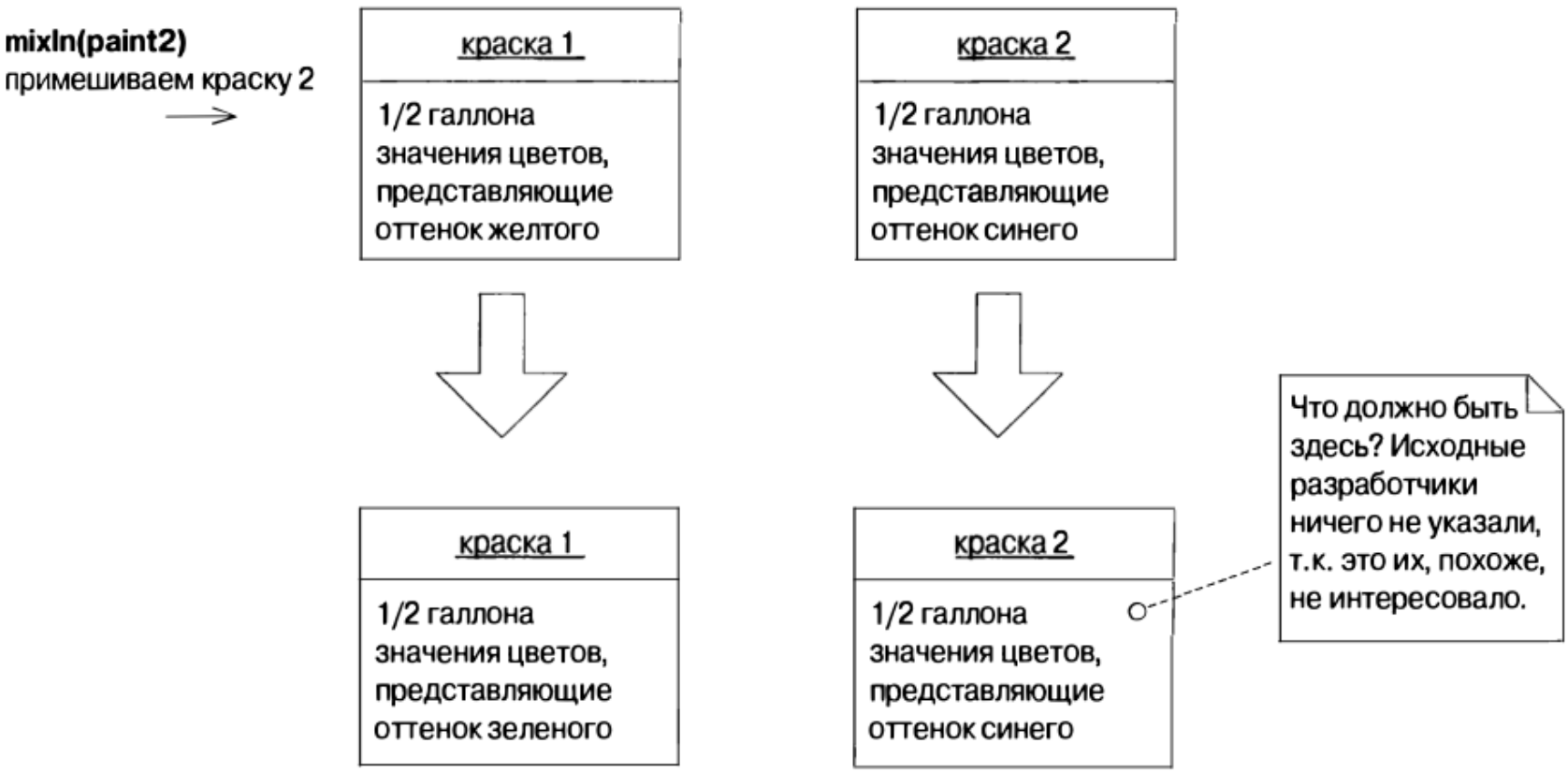
Шаг 1: говорящий интерфейс




```
public void testPaint() {
    // Начинаем с чистой желтой краски объемом = 100
    Paint ourPaint = new Paint(100.0, 0, 50, 0);
    // Берем чистую синюю краску объемом = 100
    Paint blue = new Paint(100.0, 0, 0, 50);
    // Примешиваем синюю краску к желтой
    ourPaint.mixIn(blue);
    // Должно получиться 200.0 единиц зеленой краски
    assertEquals(200.0, ourPaint.getVolume(), 0.01);
    assertEquals(25, ourPaint.getBlue());
    assertEquals(25, ourPaint.getYellow());
    assertEquals(0, ourPaint.getRed());
}
```

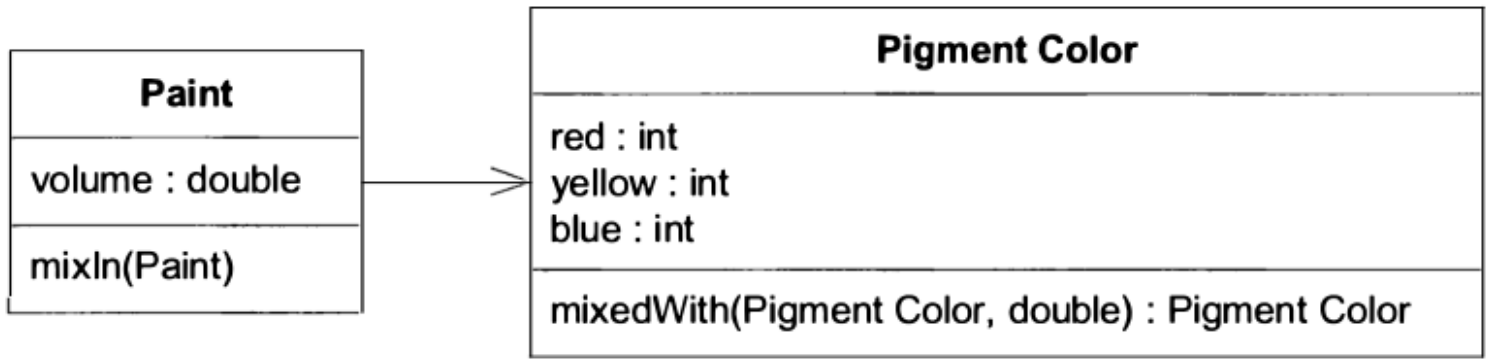
Шаг 2: функции без побочных эффектов (1)

Проблема



Шаг 2: функции без побочных эффектов (2)

Идея рефакторинга



Шаг 2: функции без побочных эффектов (3)

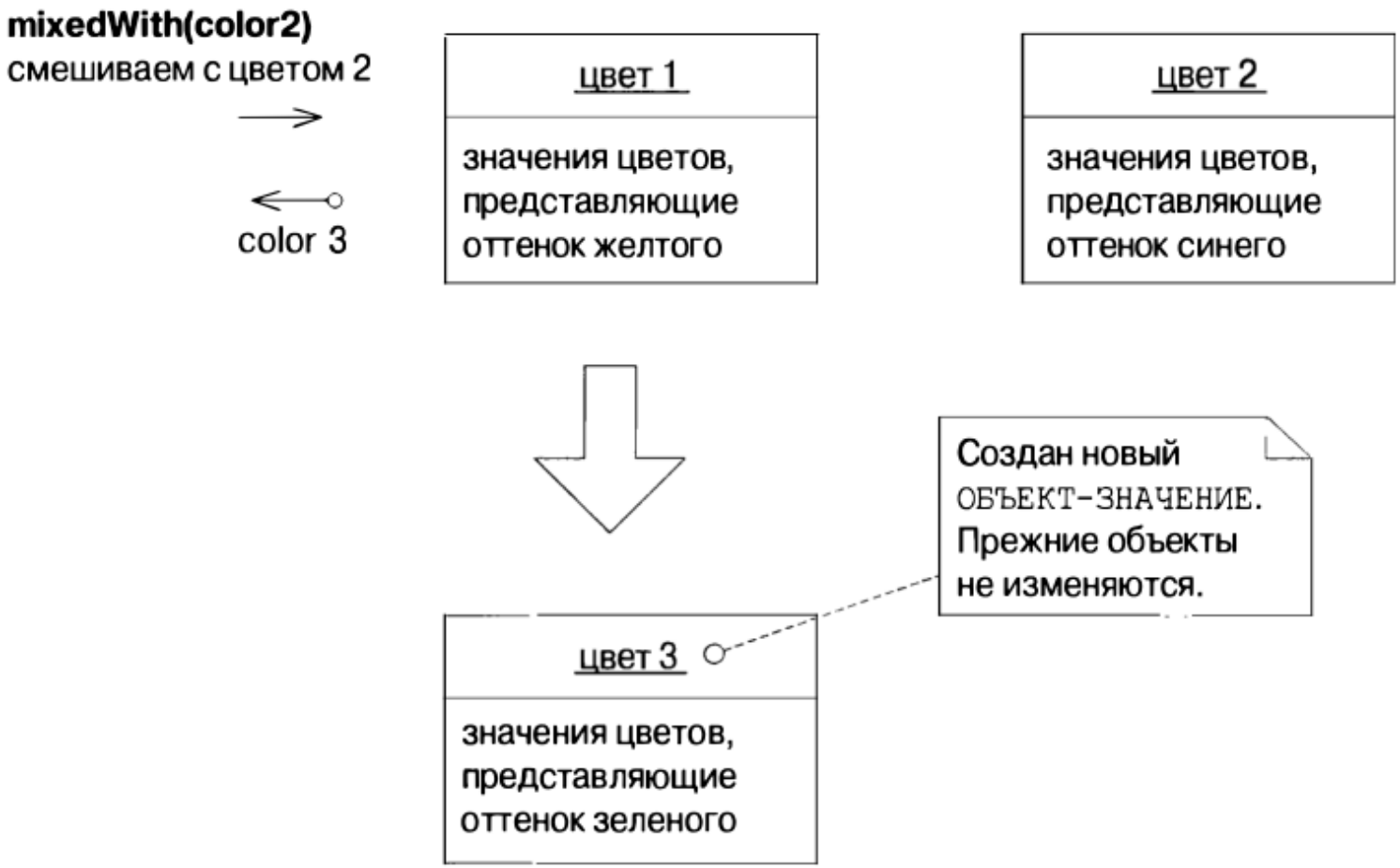
Рефакторинг

```
public class PigmentColor {
    public PigmentColor mixedWith(PigmentColor other, double ratio) {
        // Много строк сложного расчета смешивания цветов.
        // в результате создается новый объект PigmentColor
        // с новыми пропорциями красного, синего и желтого.
    }
}

public class Paint {
    public void mixIn(Paint other) {
        volume = volume + other.getVolume();
        double ratio = other.getVolume() / volume;
        pigmentColor = pigmentColor.mixedWith(other.pigmentColor(), ratio);
    }
}
```

Шаг 2: функции без побочных эффектов (4)

Результат



Шаг 3: assertions (1)

Инварианты, как они есть

Постусловие для mixIn():

После pl.mixIn(p2):
- pl.volume увеличивается на объем p2.volume
- p2.volume не изменяется

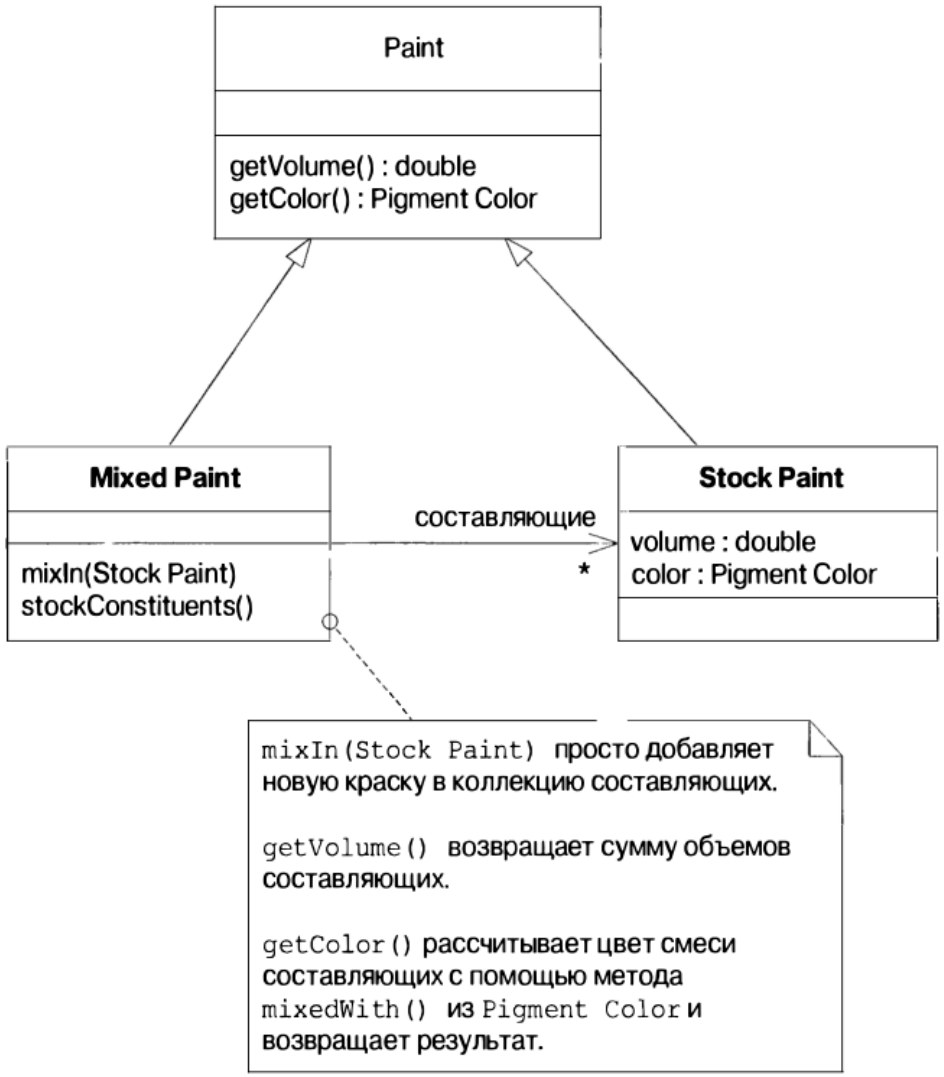
И инвариант:

Общий объем краски не должен измениться от смешивания

???

Шаг 3: assertions (2)

Рефакторинг



Замкнутость операций

Спецификации