

Лекция 11: Domain-Driven Design

Юрий Литвинов
yurii.litvinov@gmail.com

22.11.2019г

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

DDD даёт ответ на вопрос “откуда брать эти все классы” и позволяет целенаправленно уточнять и улучшать архитектуру системы. Особенно полезно, когда предметная область не очень знакома.

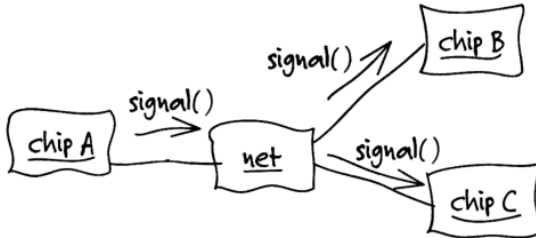
Книжка

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

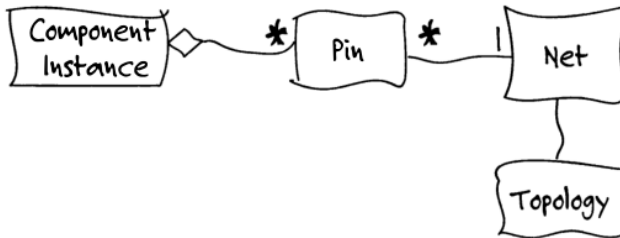


Domain-Driven Design, анализ

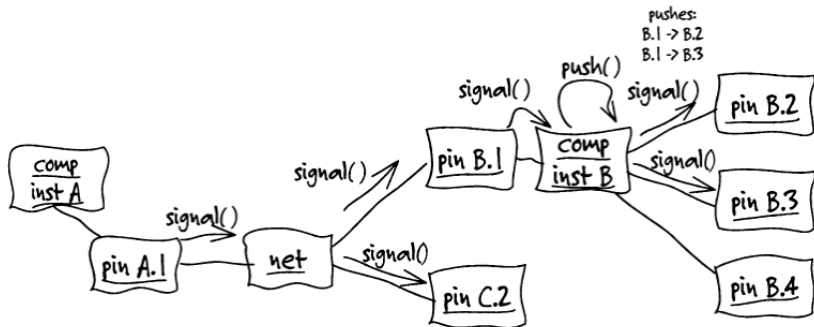
Пример: печатные платы



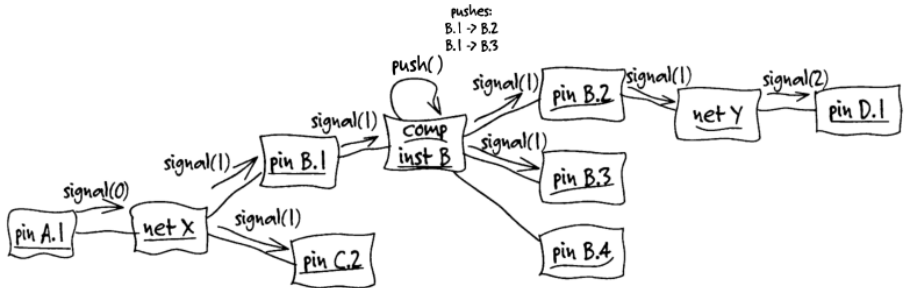
Печатные платы, топология



Печатные платы, сигналы



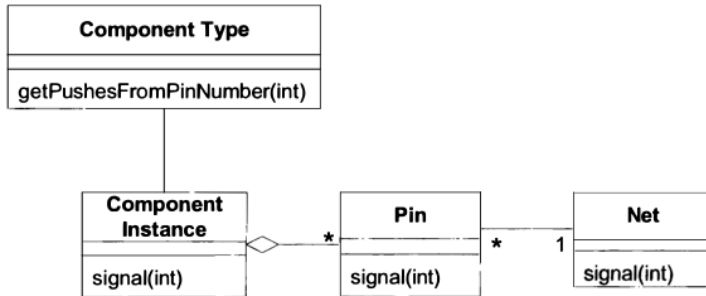
Печатные платы, прозванивание



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

Единый язык

- ▶ У программистов и специалистов предметной области свой профессиональный жаргон
- ▶ Свои жаргоны появляются даже среди групп разработчиков в одном проекте
- ▶ Необходимость перевода размывает смысл понятий
- ▶ “Еретики” используют понятия в разных смыслах
- ▶ Единый язык — понятия из модели (классы, методы), паттерны, элементы “высокоуровневой” структуры системы (которая не отражается в коде)
- ▶ Изменения в языке — рефакторинг кода
- ▶ Языков в проекте может быть много

Без единого языка

Cargo
cargoid origin destination customs clearance (opt) weight Haz Mat Code

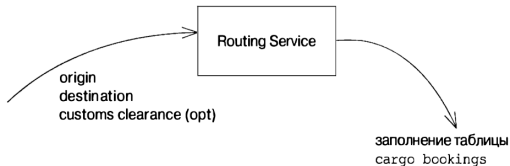
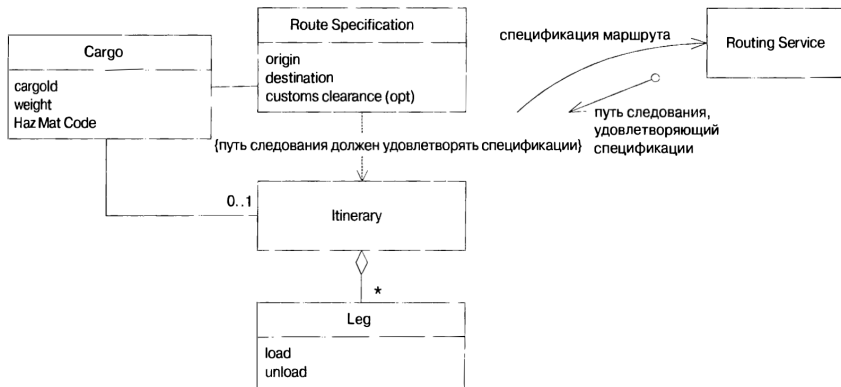


Таблица БД: cargo_bookings

Cargo_ID	Transport	Load	Unload

С единым языком



“Моделирование вслух”

*Если передать в **Маршрутизатор** пункт отправки, пункт назначения, время прибытия, то он найдет нужные остановки в пути следования груза, а потом, ну... запишет их в базу данных.*

*Пункт отправки, пункт назначения и все такое... все это идет в **Маршрутизатор**, а оттуда получаем **Маршрут**, в котором записано все, что нужно.*

***Маршрутизатор** находит **Маршрут**, удовлетворяющий **Спецификации маршрута**.*

Модель и реализация

- ▶ Модель, не соответствующая коду, бесполезна
- ▶ Код, созданный без модели, скорее всего, работает неправильно
 - ▶ “Разрушительный рефакторинг”
 - ▶ Нельзя разделять моделировщиков и программистов
- ▶ Модель в DDD выполняет роль и модели анализа, и модели проектирования одновременно
 - ▶ Это требует баланса между техническими деталями и адекватностью выражения предметной области
 - ▶ Часто требуется несколько итераций рефакторинга
- ▶ Язык программирования должен поддерживать парадигму модели
- ▶ Модель, привязанная к реализации, хороша и для пользователя

Изоляция предметной области



Изоляция предметной области, соображения

- ▶ Модель предметной области должна быть отделена от остальной программы
- ▶ Классы модели умеют делать только “суть”
- ▶ Сборка всего воедино и общее управление процессом — на операционный уровень
 - ▶ Бизнес-регламенты — на уровне модели предметной области
- ▶ Все технические вещи — на инфраструктурный уровень
 - ▶ Работа с БД
 - ▶ Middleware, сетевые коммуникации
 - ▶ Утилиты
 - ▶ Абстрактные базовые классы
- ▶ Observer или вариации MVC для связей “снизу вверх”

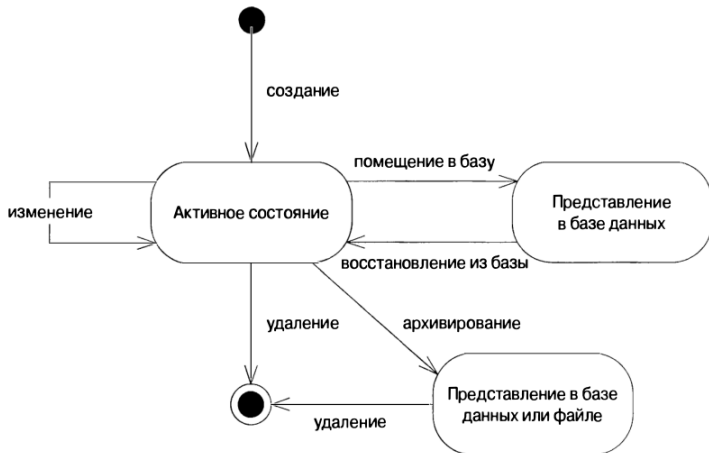
Антипаттерн “Умный GUI”

- ▶ 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

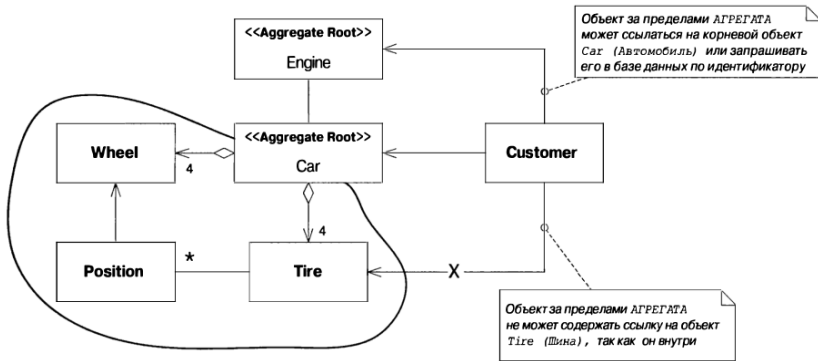
Жизненный цикл объекта



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, используемая для восстановления объекта



Хранилище (Repository)



Репозиторий хранит объекты и предоставляет к ним доступ

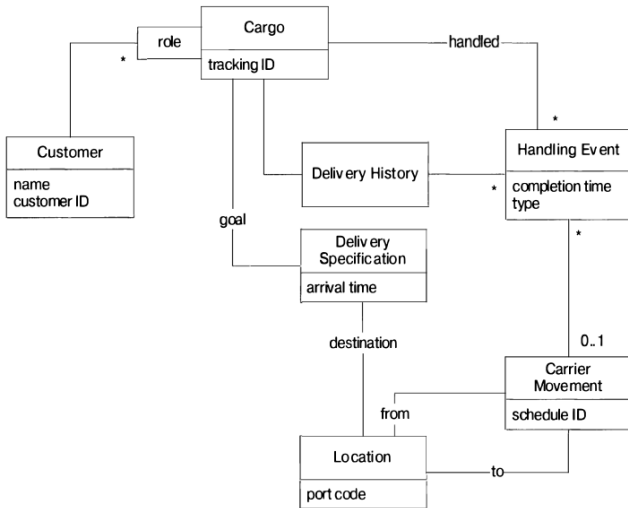
- ▶ Может инкапсулировать запросы к БД
- ▶ Может использовать фабрики
- ▶ Может обладать развитым интерфейсом запросов

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



Уровень приложения

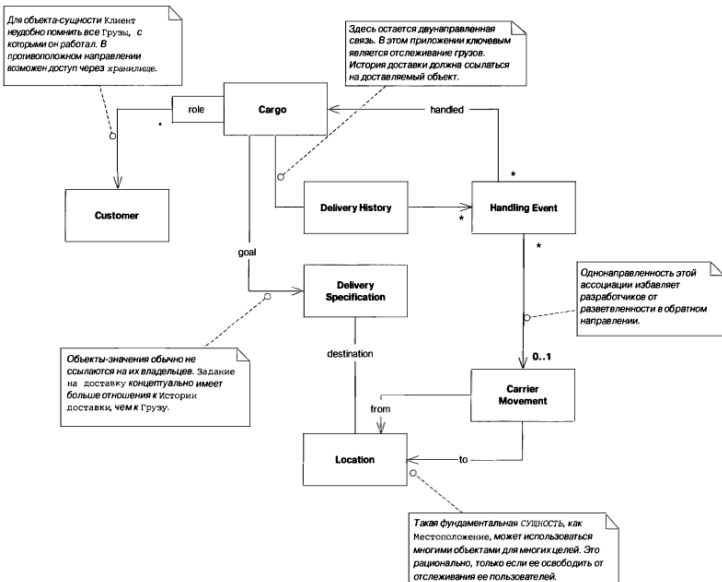
Применим уровневую архитектуру и выделим операции уровня приложения:

- ▶ 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)

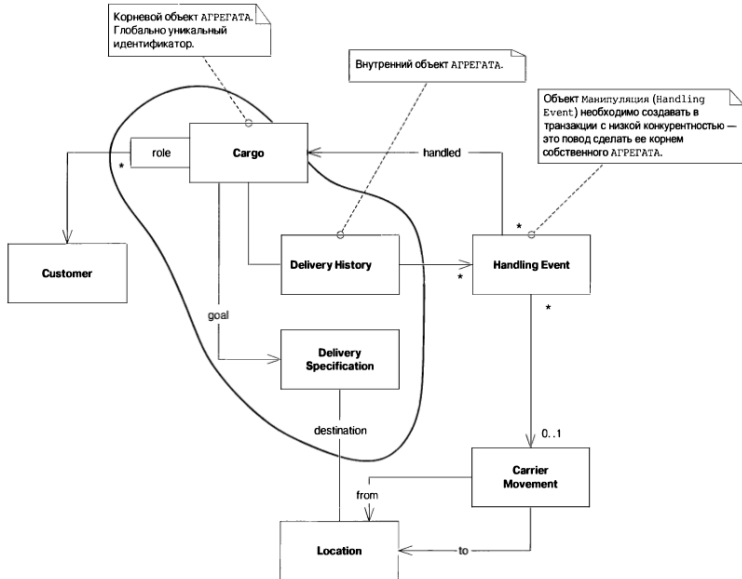
Сущности или значения?

- ▶ **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

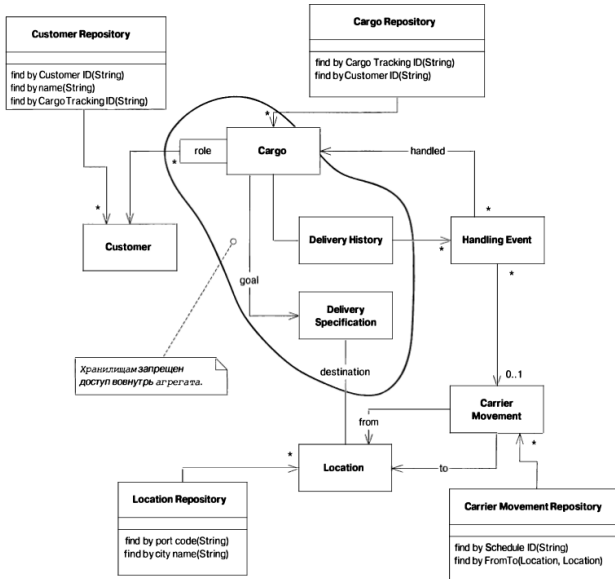
Направленность ассоциаций



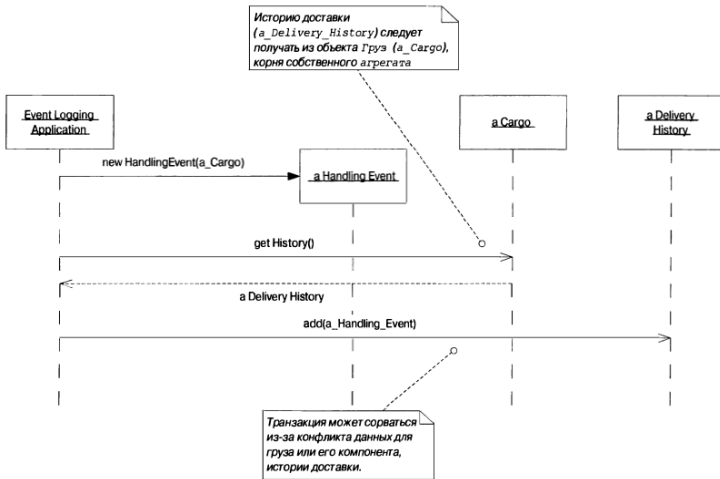
Границы агрегатов



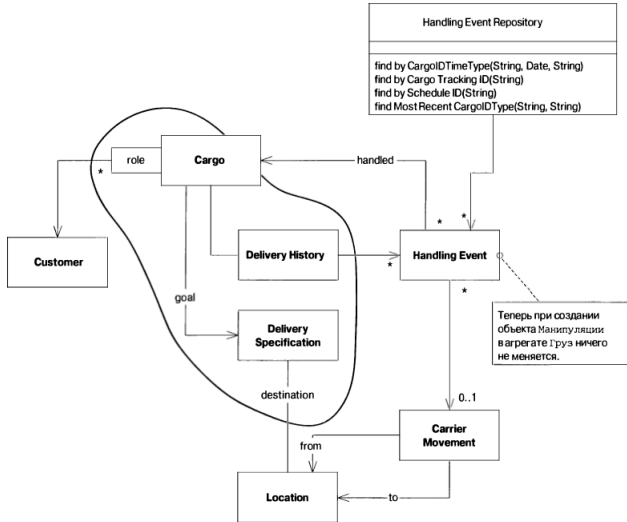
Хранилища



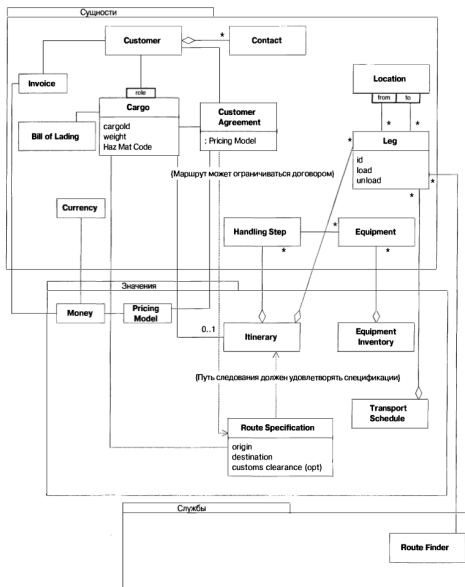
Тестовый сценарий, добавление события



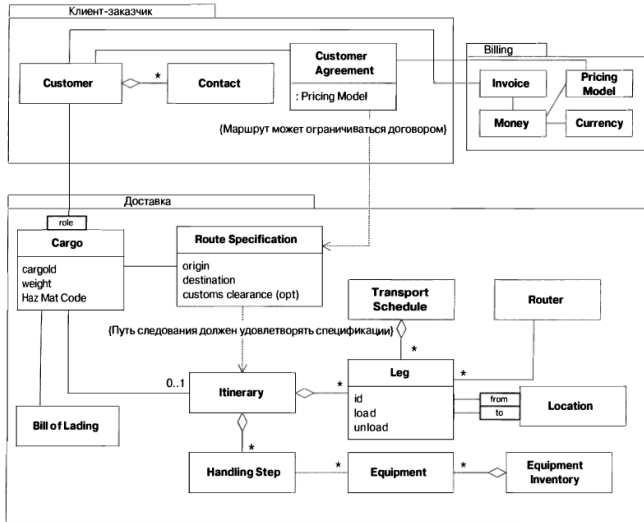
Рефакторинг, не хранить события явно



Разбиение по модулям, плохо

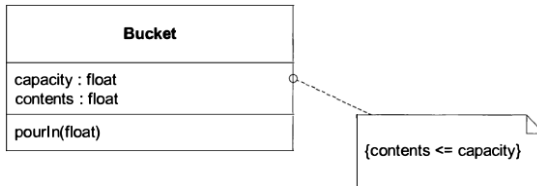


Разбиение по модулям, хорошо



Моделирование ограничений

Простой пример



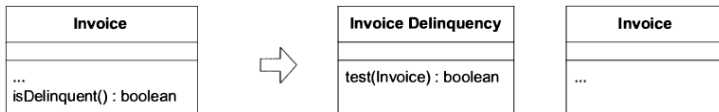
Код, до

```
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;  
    }  
}
```

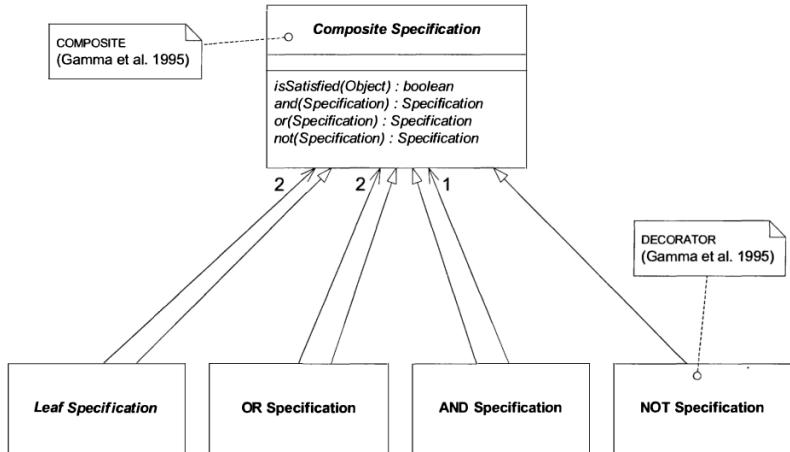
Паттерн “Спецификация”



Спецификация инкапсулирует ограничение в отдельном объекте

- ▶ Предикат
- ▶ Может быть использована для выборки или конструирования объектов

Композитные спецификации



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;  
}
```

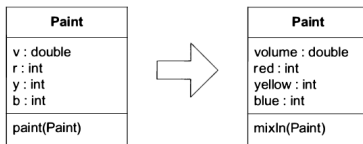
Пример рефакторинга, смешивание красок

Начальное состояние

Paint
<code>v : double</code> <code>r : int</code> <code>y : int</code> <code>b : int</code>
<code>paint(Paint)</code>

```
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



```

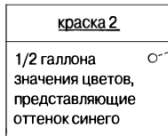
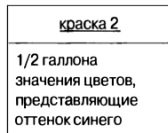
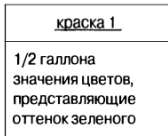
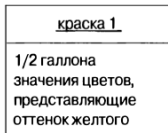
public void testPaint() {
    // Начинаем с чистой желтой краски объемом = 100
    Paint ourPaint = new Paint(100.0, 0, 50, 0);
    // Берем чистую синюю краску объемом = 100
    Paint blue = new Paint(100.0, 0, 0, 50);
    // Примешиваем синюю краску к желтой
    ourPaint.mixIn(blue);
    // Должно 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

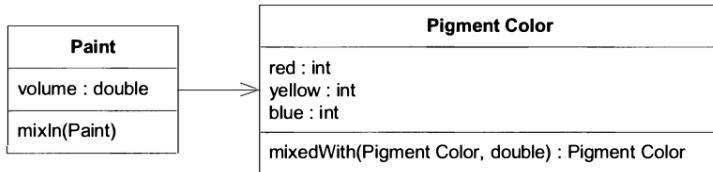
`mixIn(paint2)`
примешиваем краску 2



Что должно быть
здесь? Исходные
разработчики
ничего не указали,
т.к. это их, похоже,
не интересовало.

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

mixedWith(color2)

смешиваем с цветом 2



color 3

<u>цвет 1</u>
значения цветов, представляющие оттенок желтого

<u>цвет 2</u>
значения цветов, представляющие оттенок синего



<u>цвет 3</u>
значения цветов, представляющие оттенок зеленого

Создан новый
ОБЪЕКТ-ЗНАЧЕНИЕ.
Прежние объекты
не изменяются.

Шаг 3: assertions (1)

Инварианты, как они есть

Постусловие для `mixIn()`:

После `p1.mixIn(p2)`:

`p1.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

