**АНАЛИЗ ТРЕБОВАНИЙ**

Одни из методик определения обьектов — проанализировать требования. Например:

**user story**

nouns

Я, как **онлайн-покупатель**, хочу добавить **товар** в **корзину**, что бы я мог купить его

Имена существительные — будут 3 обьекта системы

verbs

Я, как онлайн-покупатель, хочу **добавить** товар в корзину, что бы я мог **купить** его

Глаголы в данном случае отвечают операциям корзины

Так же требования могут помочь определить отношения между обьектами, в данном случае обычно 1 покупатель ассоциируется с 1 корзиной, корзина может содержать множество единиц товара

**Категории обьектов проекта**

**Обьекты сущностей** —отвечают обьектам реального мира, например юзер, или дом

**Обьект контроля** — обьект, который координирует другие обьекты, в тоже время оставляя их слабо связанными. Отичный паттерн — Mediator

**Обьект границы** — граничат между системами. Например обьект, который запрашивает информацию в нете; или обьект, который выводит информацию для юзера или забирает его инпут;

**Концептуальное проектирование** определить основные обьекты задачи

**Техническое проектирование** (более углубленный, после концептуального) — определяет детали обьектов, включая их свойства и поведение

**Основные принципы проектирования:**

- гибкость

- реюзабельность

- поддерживаемость

**Концептуальное проектирование — CRC**

**CRC** — карточка Class, Responsibility, Collaborators

Class — сам класс, пишется вверху каты

Resposibility — обязанности класса, пишутся под классом справа

Collaborators — другие классы, с которыми класс взаимодействует что бы выполнять свои обязанности

Пример: банковский автомат. Приходит юзер, вставляет карточку, автомат просит ввести ПИН-код, после чего юзер может на выбор: снять деньги, посмотреть баланс или …

|  |  |
| --- | --- |
| Пользователь | |
| Resposibilities  - вставить карточку  - выбрать операцию | Collaborators  - Банковский автомат |

|  |  |
| --- | --- |
| Банковский автомат | |
| Resposibilities  - аутентифицировать юзера  - отобразить опции выбора  - снятие и запихивание наличных  - проверка баланса | Collaborators  - Пользователь |

**Абстрактный тип данных**

Тип данных, который не встроен в язык и определяется програмистом. Это группировка связанной информации

**Цель обьектно-ориентированного программирования:**

* Сделать абстрактный тип данных более легким к написанию
* Построить систему вокруг абстрактных типов данных, называемых классами
* Представить возможность для абстрактных типов данных расширять друг друга

**ЧЕТЫРЕ ПРИНЦИПА ПРОЕКТИРОВАНИЯ**

**Абстракция (abstraction)** — идея упрощения концепта в проблемной области. Она разбивает задачу на упрощенное описание , которое игнорирует несущественные детали и подчеркивает основные, необходимые для концепции

**Rule of least astonishment.** Абстракция должна следовать правилу наименьшего удивления. Это правило предполагает, что основные атрибуты и поведение должны быть описаны без сюрпризов и без каких-либо определений, выходящих за рамки необходимых. Это отсекает нерелевантные характеристики как часть абстракции и помогает убедиться, что абстракция имеет смысл для целей концепции.

Абстракция содержит **атрибуты** и **поведение:**

Абстракция собаки, атрибуты:

- имя

- порода

- возраст

- цвет

Абстракция собаки, поведение:

- есть

- спать

- бегать

- прыгать

**Инкапсуляция (encapsulation)**

* Данные и функции, которые ими управляют обьединяются в 1 обьект
* Данные и функции обьекта могут быть открытыми и доступными из других классов
* данные и функции могут быть ограничены для использования только внутри обьекта

Инкапсуляция создает **абстрактный барьер**, который не позволяет ее разрушать. **Черный ящик** — клиент не знает, что происходит внутри класса, он пользуется лишь его интерфейсом и получает на выходе данные

**Декомпозиция (decomposition)**

Декомпозиция разделяет большую вещь составляющие, как и наоборот

Составляющие могут быть фиксированными или динамическими (изменяющимися)

Время жизни целого объекта и составляющих может отличаться

Так же может быть вложенность составлящих или они могут находится одновременно в нескольких составляющих

В декомпозиции присутствует 3 типа отношений:

* ассоциация (осведомленность) — обьект использует другой обьект. Один обьект получает ссылку на другого, но не управляет временем его жизни.

Пример — посетитель и отель. Посетитель может взаимодействовать с отелем, а отель может взаимодействовать с множеством посетителей. Loose partnership — свободное партнерство, никак не связаны между собой

* аггрегация (делегирование) — один обьект является частью другого. Пример — авиаперевозчик пользуется услугами команды в качестве специалистов по обслуживанию самолета. Weak partnership — слабо связаны между собой
* композиция (аггрегация) — целое не может существовать без частей, и когда целое уничтожается, части также уничтожаются. Получить доступ к частям можно только через целое. Пример — дом с комнатой. Комната не может существовать без дома. Strong partnership

**Обобщение** (generalization) — выделение из методов общих операций и вынос их в отдельные функции. Помогает уменьшить избыточность. Перекликается с принципом DRY.

В понятиях классов — вывести общий родительский класс. Например класс собаки и кота можно наследовать от класса ЖИВОТНОЕ

**COUPLING AND COHESION (сопряжение и связность)**

**Coupling** — насколько модуль сопряжен с другими модулями (внешняя связь с другими модулями)

Tightly coupled — все классы связаны между собой не через интерфейсы, а через включения и прочее, реюзабельность минимальна

Loosely coupled — у модуля четко определенный интерфейс, слабое сопряжение с другими модулями

**Degree**

Количество соединений между одним модулем и другими модулями. Degree нужно держать небольшим, как и coupling. Например, модулю понадобится подключить к другому модулю через несколько параметров или узкому (narrow) интерфейсу, degree будет небольшим, а сопряжение свободным (loose coupling)

**Ease**

Насколько очевидны соединения между одним модулем и другими (например неявная передача параметров или использование глобальных параметров). Как и сопряжение, соединения необходимо поддерживать простыми что бы не было необходимости знать реализацию остальных модулей

**Flexibility**

Показывает, насколько заменяемы другие модули для данного модуля.

Модули внутри модуля должны легко заменяться на другие

**Cohesion** — насколько сфокусирован модуль (внутренняя связь между классами) Ясность обязанностей модуля

High cohesion — модуль или класс имеет одну четкую цель

Low cohesion — модуль распылен, обязанности не определены четко

Разделение модулей для увеличения cohesion ведет к увеличению degree из-за того, что модулю приходится чаще общаться с другими модулями

И наоборот, соединение модулей в один большой уменьшает coupling, но ведет так же к уменьшению cohesion — цель модуля становится менее ясной

**Separation of** **concerns (разделение обязанностей)**

Concern — все, что касается обеспечения решения проблемы

Если класс Собаки может есть, должен быть класс Владельца, который бы кормил эту собаку.

**Information hiding**

Скрывать реализацию, показывать интерфейсы

**Conceptual integrity**

Команда должна следовать одним принципам

**KRUTCHEN`S 4+1 VIEW MODEL**

Для того, что бы полностью смоделировать поведение и разработку программной системы, нужны разные точки зрения. Эта модель предполагает:

**Logical view**

Фокусируется на функциональности и необходимых обьектах.

Разбирается, что система должна делать для удовлетворения потребностей клиентов с точки зрения функциональности, какие для этог нужны обьекты.

Контекст — это сервисы, которые должны быть предоставлены конечным юзерам. На этом этапе включается UML class and state diagrams

* Создание словаря проблемы в рамках системы
* обозначение всех классов, их атрубутов и поведения

**Process view**

Фокусируется на процессах, полученных как результат работы обьектов в *logical view*. Разбирается с точки зрения эффективности системы или взаимодействия подпроцессов во время исполнения программы. Система рассматривается в разрезе производительности, кокурирующих запросов и тд. На этом этапе включают UML sequence and activity diagram. Разбирается через призму атрибутов качества, таких как

* производительность системы
* доступность системы

**Development view**

Фокусируется на внедрении стандартов и соглашений, таких как иерархическая структура ПО. Выбранные программный язык имеет сильное значение на конечную структуру и следовательно, привносит свои ограничения. Это распространяется на данные прожект менеджмента, такие как планирование, бюджет и задания

* Языки программирования
* библиотеки
* тулзы

**Physical view**

Фокусируется на физических компонентах системы и их взаимодействию — количество серверов, что на них будет находится и тд.

Тут строится deplyment UML diagram

**Scenarios**

Сценарии описывают варианты использования, которые требовались конечными пользователями. Сценарии предоставляют контекст, чтобы помочь детализировать четыре *views*.

Каждый сценарий является скриптом, в котором описана последовательность взаимодействий между обьектами и процессами. Это вкоючает ключевые обьекты, определенные в logical view, процессы, описанные в process view, иерархию и разные ветки, описанные в physical view. Сценарии их обьеденяют для цельноый картины

Далеко не все системы нужно описывать через призму данной модели, часто некоторые view можно исключить. Например, если logical и development view чень похожи, они могут быть описаны вместе