* Основная идея функционального программирования - возможность создавать и управлять функциями, включая создание функций в runtime.
* Функции стали еще 1 сущностью, которой может управлять разработчик (ооп - только data).
* Java 8 моментально не стала функциональным языком, но начала поддерживать функциональное программирование поверх ООП.

**1. Сhain of responsibility**

GOF - настройка цепочки, создание объекта каждого хендлера.

Lambda - создаем Stream из функций, выполняем и ищем первую, которая вернула значение.

**2. Команда.**

Lambda и есть команда, функциональный подход - построен на командах. Нужно использовать

стандартные интерфейсы Predicate, Consumer, Function

Сможем вычистить много кода используя лямбда выражения или ссылки на методы

**3. Декоратор** - заворачиваем объект в несколько оберток для получения новых свойств.

GOF - Вложенные конструкторы

Lambda - стрим функций(каждая функция передает управление следующей)

**4. Интерпритатор.**

Синтаксический анализ, разбор текста на команды и их выполнение.

GOF - много церемоний, сложно добавить еще 1 инструкцию для интерпретатора

Lambda - карта функций, просто дополняется.

**5. Observer** - событийная модель

Суть в том, что у нас есть объект, который слушает и реагирует на события, которые произошли в других объектах.

GOF - класс для каждого обработчика, много церемоний

лямбда - если обзервер простой - очень упрощается реализация

**6. Стратегия**

Способ изменить алгоритм поведения программы исходя из runtime принятого решения.

lambda - меньше церемоний, только логика, которая реализует бизнес задачу.

Передавать функцию/лямбду со стратегией

**7. Фабричный метод**

Конструктор можно использовать как лямбду.

Можно уйти от отдельного класса Фабрики, использовать интерфейс Function.

Не подходит для абстрактной фабрики, она не меняется.

**8. Template method**

Если у нас есть похожие методы со своей спецификой - использовать паттерн.

Если разобрался в основном методе - с остальными разобраться просто.

Абстрактные методы - кастомизированые части алгоритма

lambda - передаем вызов метода как функцию. Меньше кода

**9.Visitor**

GOF - контроль типов, но много кода

Lambda - менше кода, но нет контроля типов

**10. Builder**

2 типа: Runtime, Storing, примеры.

Как можно улучшить в java 8: Передать функцию в билдер и там решать как создавать объекты.

**11. Адаптор**

НЕ нужно создавать отдельный клас для адаптеров с 1 методом. Весь адаптер - лямбда с другим методом

**12. Прокси**

Использовать AOP, не писать руками!!!

**13. Итератор**

GOF - от iterator до for

Лямбда - антипатерн, использовать стримы

**14. Singleton - antipattern**

Использовать средства фреймворка/контейнера

**15. Java 7 arm vs Around method pattern**

ARM - легко забыть и не написать try block, если не написал - никто не подскажет, код скомпилируется

AMP - приватный конструктор не даст создать ресурс и не закрыть его

Хорошо работает когда у нас не сложные методы. В противном случае - может ухудшиться читаемость кода. Золотое правило внедрения лямбд - чем **больше кода в лямбде - тем хуже читается код.**

**Optionals**

Использовать map/flatMap для выполнения операции на стороне данных, без распаковки

**Пирамида смерти**

Как избавится: написать/переиспользовать функцию. В данном случае flatMap в Optionals

Изменить тип входящих данные, которые исключают вероятность возникновения ошибки

**Передача управления (IOC)**

Если что-то случиться – не звони, я сам позвоню

Обработка исключительных ситуаций определяется вызывающей функцией.

Либо – подготовить данные таким образом, чтобы там не было «проблеманых» данных

**offtop** Optional.get() deprecation discussion

Brian Goetz: I think calling this method "get()" was our biggest API mistake in Java 8.

if (opt.isPresent()) {

doSomething(opt.get());

} else {

doSomethingElse();

}

In many of these cases, the code could be refactored to use other Optional

methods such as filter(), map(), or ifPresent().

In any case this reinforces the contention that use of get() leads to poor code.

For this changeset, in just about all cases I've simply replaced the call to

get() with a call to getWhenPresent(). In a couple cases I replaced the stream calls

.filter(Optional::isPresent).map(Optional::get)

with

.flatMap(Optional::stream)