**Билет 5**

**Компактность:** желательно, чтобы длина функции не превышала 20 строк.

**Блоки и отступы**: функции не должны содержать вложенных структур, так как это приводит к их увеличению. Максимальный уровень отступов в функции не должен превышать одного-двух.

**Правило одной операции**: (ФУНКЦИЯ ДОЛЖНА ВЫПОЛНЯТЬ ТОЛЬКО ОДНУ ОПЕРАЦИЮ И ВЫПОЛНЯТЬ ЕЕ ХОРОШО). Если функция выполняет только те действия, которые находятся на одном уровне под **объявленным именем** функции, то эта функция выполняет **одну операцию**. (чтобы определить, что функция выполняет более одной операции, попробуйте извлечь из нее другую функцию, которая бы не являлась простой переформулировкой реализации. Если функция разделена на секции (объявления, инициализация, отбор), это очевидный признак того, что функция выполняет более одной операции.)

**Используйте содержательные имена**: не бойтесь использовать длинные имена. Длинное содержательное имя лучше короткого невразумительного. Выберите схему, которая позволяет легко прочитать слова в имени функции, а затем составьте из этих слов имя, которое описывает назначение функции. Используйте в именах функций те же словосочетания, глаголы и существительные, которые используются в ваших модулях. Принцип Уорда: «Вы работаете с чистым кодом, если каждая функция в основном делает то, что вы от нее ожидали».

**Аргументы функций**: В идеальном случае количество аргументов функции равно нулю (нуль-арная функция). Далее следуют функции с одним аргументом (унарные) и с двумя аргументами (бинарные). Функций с тремя аргументами (тернарных) следует по возможности избегать. Необходимость функций с большим количеством аргументов (полиарных) должна быть подкреплена очень вескими доводами — и все равно такие функции лучше не использовать.

**Аргументы-флаги**: Аргументы-флаги уродливы. Передача логического значения функции — воистину ужасная привычка. Она немедленно усложняет сигнатуру метода, громко провозглашая, что функция выполняет более одной операции. При истинном значении флага выполняется одна операция, а при ложном — другая! Такие функции следует разбивать на несколько.

**Объекты как аргументы**: Если функция должна получать более двух или трех аргументов, весьма вероятно, что некоторые из этих аргументов стоит упаковать в отдельном классе.

**Списки аргументов:** Иногда функция должна получать переменное количество аргументов. Если все переменные аргументы считаются равноправными, то их совокупность эквивалентна одному аргументу типа array).

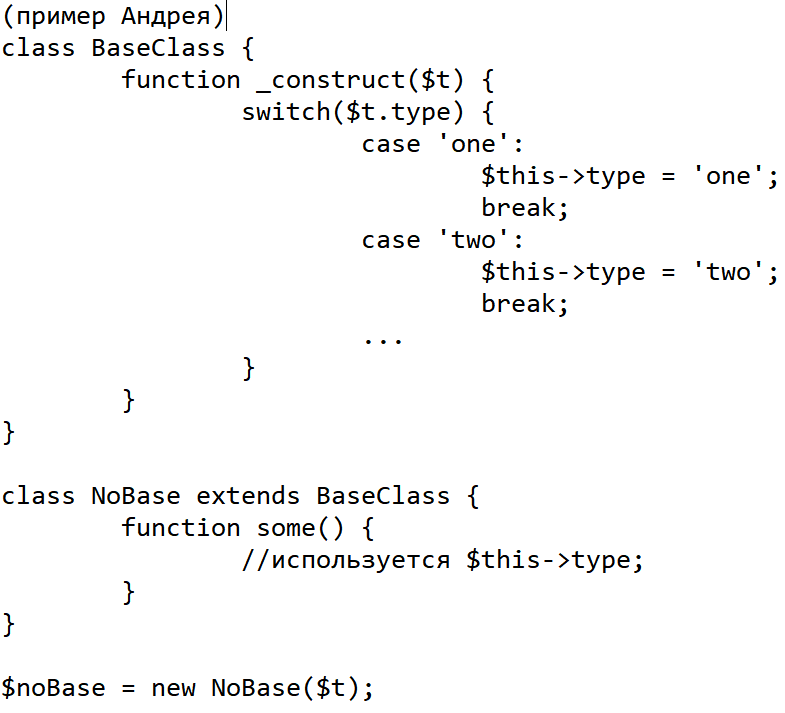
**Не повторяйтесь:** дублирование создает проблемы, потому что оно увеличивает объем кода, а при изменении алгоритма вам придется вносить изменения сразу в четырех местах. Также вчетверо возрастает вероятность ошибки. Объектно-ориентированные языки помогают сконцентрировать в базовых классах код, который в других обстоятельствах мог бы дублироваться в разных местах.

**Разделение команд и запросов:** Функция должна что-то делать или отвечать на какой-то вопрос, но не одновременно. Либо функция изменяет состояние объекта, либо возвращает информацию об этом объекте. Совмещение двух операций часто создает путаницу.

**Один уровень абстракции на функцию:** чтобы убедиться в том, что функция выполняет «только одну операцию», необходимо проверить, что все команды функции находятся на одном уровне абстракции. Смешение уровней абстракции внутри функции всегда создает путаницу. Читатель не всегда понимает, является ли некоторое выражение важной концепцией или второстепенной подробностью. Что еще хуже, при их смешении функция постепенно начинает обрастать все большим количеством второстепенных подробностей.

**Чтение кода сверху вниз**: код должен читаться как рассказ — сверху вниз. За каждой функцией должны следовать функции следующего уровня абстракции. Это позволяет читать код, последовательно спускаясь по уровням абстракции в ходе чтения списка функций.

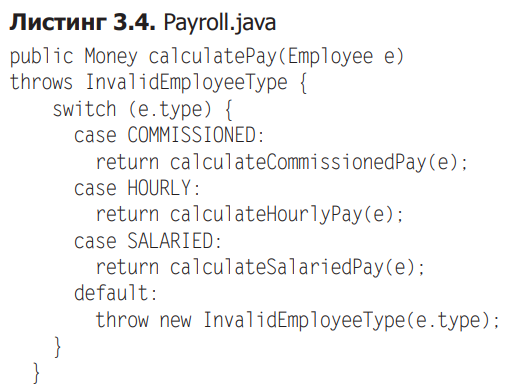
**Команды switch**: общее правило в отношении команд switch гласит, что эти команды допустимы, если они встречаются в программе **однократно**, используются для создания **полиморфных** объектов и скрываются за отношениями наследования, чтобы оставаться невидимыми для остальных частей системы (см. пример Андрея).



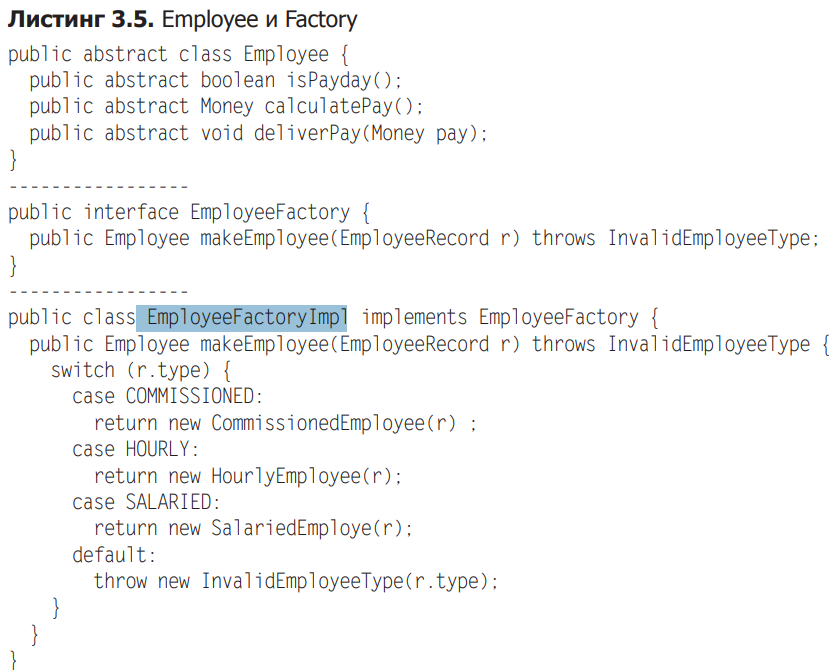
Конечно, правил без исключений не бывает и в некоторых ситуациях приходится нарушать одно или несколько условий этого правила.

*Дополнение(сложновато)*: написать **компактную** команду switch довольно сложно. Команды switch всегда выполняют N операций. К сожалению, обойтись без команд switch удается не всегда, но по крайней мере мы можем позаботиться о том, чтобы эти команды были скрыты в **низкоуровневом классе** и **не дублировались** в коде. И конечно, в этом нам может помочь полиморфизм.

В листинге 3.4 представлена всего одна операция, зависящая от типа работника



Эта функция имеет ряд недостатков. Во-первых, она велика, а при добавлении новых типов работников она будет разрастаться. Во-вторых, она совершенно очевидно выполняет более одной операции. Но самый серьезный недостаток заключается в том, что программа может содержать неограниченное количество других функций с аналогичной структурой, например: **isPayday**(Employee e, Date date) или **deliverPay**(Employee e, Money pay). Решение проблемы (листинг 3.5) заключается в том, чтобы похоронить команду switch в фундаменте АБСТРАКТНОЙ ФАБРИКИ (**EmployeeFactoryImpl**) и никому ее не показывать. Фабрика использует команду switch для создания соответствующих экземпляров потомков Employee, а вызовы функций calculatePay, isPayDay, deliverPay и т. д. проходят полиморфную передачу через интерфейс Employee.



**Стандартные унарные формы: с**уществует два очень распространенных случая вызова функции с одним аргументом. Первая — проверка некоторого условия, связанного с аргументом, как в вызове boolean fileExists("MyFile"). Вторая — обработка аргумента, его преобразование и возвращение. Например, вызов InputStream fileOpen("MyFile") преобразует имя файла в формате String в возвращаемое значение InputStream. Выбирайте имена, которые четко отражают различия, и всегда используйте две формы в логически непротиворечивом контексте.

(Еще есть **событие**. В этой форме имеется входной аргумент, а выходного аргумента нет. Предполагается, что программа интерпретирует вызов функции как событие и использует аргумент для изменения состояния системы, например, void passwordAttemptFailedNtimes(int attempts). Будьте внимательны при использовании данной формы. Читателю должно быть предельно ясно, что перед ним именно событие. Тщательно выбирайте имена и контексты.)

**Бинарные функции**: Функцию с двумя аргументами понять сложнее, чем унарную функцию. Например, вызов writeField(name) выглядит более доступно, чем writeField(outputStream, name). Конечно, в некоторых ситуациях форма с двумя аргументами оказывается уместной. Например, вызов Point p = new Point(0,0); абсолютно разумен. Точка в декартовом пространстве естественным образом создается с двумя аргументами. Бинарные функции не являются абсолютным злом, но стоит воспользоваться всеми доступными средствами для их преобразования в унарные.

**Тернарные функции**: разобраться в функции с тремя аргументами значительно сложнее, чем в бинарной функции. Проблемы соблюдения порядка аргументов, приостановки чтения и игнорирования увеличиваются более чем вдвое. Я рекомендую очень хорошо подумать, прежде чем создавать тернарную функцию.

**Глаголы и ключевые слова:** Выбор хорошего имени для функции способен в значительной мере объяснить смысл функции, а также порядок и смысл ее аргументов. В унарных функциях сама функция и ее аргумент должны образовывать естественную пару «глагол/ существительное». Например, write(name).

**Избавьтесь от побочных эффектов:** Ваша функция обещает делать что-то одно, но делает что-то другое, скрытое от пользователя. Иногда она вносит неожиданные изменения в переменные своего класса — скажем, присваивает им значения параметров, переданных функции, или глобальных переменных системы. В любом случае такая функция является коварной и вредоносной ложью, которая часто приводит к созданию противоестественных временных привязок и других зависимостей.

**Выходные аргументы:** Аргументы естественным образом интерпретируются как **входные** данные функции. Иногда аргументы оказываются **выходными**, а не входными(пояснение Андрея т.е. не агрументы используются для “вставки” куда-то, а что-то “вставляют” в них). В общем случае выходных аргументов следует избегать. Если ваша функция должна изменять чье-то состояние, пусть она изменяет состояние своего объекта-владельца. Пример: report.appendFooter();

**Используйте исключения вместо возвращения кодов ошибок:** Возвращение кодов ошибок функциями-командами является неочевидным нарушением принципа разделения команд и запросов. Они приводят к созданию структур слишком глубокой вложенности. При возвращении кода ошибки возникает проблема: вызывающая сторона должна немедленно отреагировать на ошибку.

**Изолируйте блоки try/catch:** Блоки try/catch выглядят весьма уродливо. Они запутывают структуру кода и смешивают обработку ошибок с нормальной обработкой. По этой причине тела блоков try и catch рекомендуется выделять в отдельные функции.

**Обработка ошибок как одна операция:** Функции должны выполнять одну операцию. Обработка ошибок — это одна операция. Значит, функция, обрабатывающая ошибки, ничего другого делать не должна. Отсюда следует, что если в функции присутствует ключевое слово try, то оно должно быть первым словом в функции, а после блоков catch/finally ничего другого быть не должно.

**Магнит зависимостей:** Возвращение кода ошибки обычно подразумевает, что в программе имеется некий класс или перечисление, в котором определяются все коды ошибок. Подобные классы называются *магнитами зависимостей*; они должны импортироваться и использоваться многими другими классами. Если вместо кодов ошибок использовать исключения, то новые исключения определяются производными от класса исключения.

**Структурное программирование:** функция должна содержать только одну команду return, в циклах не должны использоваться команды break или continue, а команды goto не должны использоваться никогда и ни при каких условиях. Но если ваши функции остаются очень компактными, редкие вкрапления множественных return, команд break и continue не принесут вреда, а иногда даже повышают выразительность по сравнению с классической реализацией с одной точкой входа и одной точкой выхода.