Стандарт на кодирование ПО

Используемый стек технологий

Для реализации требований, предъявляемых к ПО, будут использоваться следующие технологии:

1. Go - язык программирования, используемый для разработки бэкенд сервисов.
2. Python - язык программирования, используемый для написания алгоритмов машинного обучения.
3. Swift - язык программирования, используемый для написания клиентского приложения.

Go

Для реализации требований, предъявляемых к серверной части разрабатываемой системы, выбран язык программирования Golang с использованием официального компилятора (https://go.dev/doc/install). Синтаксис, поведение конструкций языка, поведение данных, а также побочные эффекты языка отражены в официальной документации (https://go.dev/doc/).

Форма представления исходного кода

Стандарт на форму представления исходного кода информационной системы соответствует стандартам встроенного в `golang` форматировщика `gofmt` (https://pkg.go.dev/cmd/gofmt) и стандартным правилам синтаксического анализатора `golangci-lint` (https://golangci-lint.run/).

В основе форматировщика `gofmt` лежат правила, созданные разработчиками языка Go (https://go.dev/doc/effective_go).

Данные утилиты совместно с фреймворком `pre-commit` позволяет следить за соответствием исходного кода стандартам форматирования, выводить несоответствия, устранять их в полуавтоматическом или автоматическом режимах, а также запрещать фиксировать изменения кода, несоответствующие стандарту.

Стиль написания исходного кода

Группировать похожие объявления

Go поддерживает группировку похожих объявлений (`import` , `const` , `type` , `var`). При этом стоит группировать только схожие по смыслу объявления.

```
type Operation int

const (
  Add Operation = iota + 1
  Subtract
  Multiply
)

const ENV_VAR = "MY_ENV"
```

Порядок подключения зависимостей

Импортируемые пакеты должны быть разделены на две группы:

- Стандартная библиотека
- Все остальные пакеты

```
import (
  "fmt"
  "os"

  "go.uber.org/atomic"
  "golang.org/x/sync/errgroup"
)
```

Названия пакетов

При наименовании пакетов руководствуйтесь следующими принципами:

- Название должно состоять только из символов нижнего регистра. Использовать заглавные буквы и подчеркивания запрещено.
- Название при котором для большинства вызовов нет необходимости использовать именованный импорт.
- Коротко и ясно. Помните, что к имени пакета необходимо обращаться при каждом вызове.
- В единственном числе. Например, `net/url` вместо `net/urls` .
- Не "common", "util", "shared", или "lib". Это плохие и неинформативные названия.

Названия функций

Мы следуем соглашению сообщества Go о наименовании функций MixedCaps for function names. Исключением являются функции тестов, которые могут содержать нижнее подчеркивание с целью объединения родственных тест-кейсов, например `TestMyFunction_WhatIsBeingTested`.

Группировка и упорядочивание функций

- Функции должны быть отсортированы в порядке приблизительного вызова.
- Функции в файле должны быть сгруппированы по получателю.

Таким образом, экспортируемые функции должны располагаться в файле первыми, сразу после объявления `struct`, `const` и `var`.

Методы `newXYZ()` / `NewXYZ()` должны располагаться после определения типов, но до остальных методов получателя.

Поскольку функции сгруппированы по получателю, утилитарные функции должны располагаться в конце файла.

```
type something struct{ ... }

func newSomething() *something {
    return &something{}
}

func (s *something) Cost() {
  return calcCost(s.weights)
}

func (s *something) Stop() {...}

func calcCost(n []int) int {...}
```

Используйте префикс _ для глобальных неэкспортируемых переменных

Используйте префикс `_` для верхнеуровневых переменных `var` и констант `const`, для явного обозначения глобальных переменных.

Встраивание в структуры

Встраиваемые типы (такие, как мьютексы) следует определять в самом начале списка структуры, также необходимо разделять встраиваемые поля от обычных переносом строки.

```
type Client struct {
  http.Client

  version int
}
```

Используйте названия полей при инициализации структур

```
k := User{
    FirstName: "John",
    LastName: "Doe",
    Admin: true,
}
```

Уменьшайте область видимости переменных

Где возможно, уменьшайте область видимости переменных, только если это не ведет к увеличению вложенности.

```
if err := ioutil.WriteFile(name, data, 0644); err != nil {
 return err
}
```

Инициализация ссылок на структуры

Используйте `&T{}` вместо `new(T)` при инициализации ссылок на структуры, так как таким методом вы можете сразу инициализировать значения структуры.

```
sval := T{Name: "foo"}

sptr := &T{Name: "bar"}
```

Используемые линтеры

Приведем список используемых линтеров с их кратким описанием:

- asciicheck - проверяет исходный код на наличие non-ASCII символов;

- dogsled - проверяет операторы присвоения на излишне большое количество пустых присвоений;
- errcheck - проверяет исходный код на наличие необработанных ошибок;
- exhaustive - проверяет исходный код на полноту switch конструкций для enum;
- gocognit - проверяет функции исходного кода на когнитивную сложность;
- gocyclo - проверяет функции исходного кода на цикломатическую сложность;
- gofmt - проверяет был ли код форматирован согласно правилам gofmt;
- goimports - проверяет форматирование зависимостей;
- gosimple - производит упрощение кода;
- govet - производит анализ исходного кода на наличие возможных ошибок;
- ineffassign - проверяет исходный код на наличие неиспользуемых переменных;
- lll - проверяет исходных код на слишком длинные строки;
- misspell - проверяет код на наличие опечаток;
- nestif - проверяет код на излишнюю вложенность условных операторов;
- staticcheck - проводит статический анализ кода;
- typecheck - проводит типизированный анализ кода;
- unconvert - проводит анализ исходного кода на ненужные приведения типов;
- unused - проверяет исходный код на наличие неиспользуемых переменных, констант, типов и так далее;
- whitespace - проверяет строки исходного кода на открывающие и завершающие пробелы;
- durationcheck - проверяет исходный код на возможные ошибки при перемножение временных типов;
- forcetypeassert - проверяет исходный код на наличие принудительного приведения типов;
- importas - проверяет исходный код на правильные алиасы для зависимостей;
- predeclared - проверяет исходный код на переопределение одного из объявленных идентификаторов;
- tagliatelle - проверяет структурные теги в исходном коде;
- godot - проверяет комментарии в исходном коде;
- wsl - добавляет пустые строки в исходный код для улучшения читаемости;
- gocritic - предоставляет диагностику, которая проверяет наличие ошибок, проблем с производительностью и стилем;
  расширяемый без перекомпиляции с помощью динамических правил;
  динамические правила написаны декларативно с использованием шаблонов AST, фильтров, сообщения отчета и необязательного предложения;
- prealloc - проверяет исходный код на места, где можно использовать выделение памяти;
- stylecheck - проверяет исходный код на следование стилю;
- nlreturn - проверяет исходный код на отсутствие новой строки перед return;
- unparam - проверяет исходный код на наличие неиспользуемых параметров в функциях;

Комментарии

При помощи комментариев в исходном коде осуществляется документирование кода. Для этого используется пакет `godoc` (https://pkg.go.dev/golang.org/x/tools/cmd/godoc). Согласно рекомендациям по написанию документирующих комментариев (https://tip.golang.org/doc/comment), каждая экспортируемая часть кода (функция, класс, интерфейс, пакет и т.д.) должна иметь свой документирующий комментарий. Размещая такие комментарии в коде и обращаясь к `godoc` можно получить веб-приложение, предоставляющее всю информацию об экспортируемых методах в виде веб-страниц.

Средствами комментирования также обеспечивается трассируемость исходного кода, написанного на Go. В документирующем комментарии, в первой строке, указывается версия документа с требованиями и идентификатор требования, которое реализуется в данной части кода. Формат такого комментария: <Name>_<YYYYMMdd_hhmm#num>.

```
// Requirements_20221010_13:30#0-2-BB-00005
// Returns the square root of an example
func (e Example) Sqrt() Example {
        return Example(math.Sqrt(float64(e)))
}
```

Таким образом обеспечивается связь исходного кода и требований, которые в нем реализуются.

Модульные тесты

Стандарт написания модульных тестов описан в стандартном пакете `testing` (https://pkg.go.dev/testing). Тесты для каждой функции группируются согласно подходу "Table Driven Unit Test".

Организация проекта

В качестве макета организации проекта используется (https://github.com/golang-standards/project-layout), совмещенный с подходами Clean Architecture.

- директории Go:
  - /cmd - основные приложения проекта;
  - /internal - внутренний код приложений и библиотек;
    - /entity - сущности бизнес-логики;
    - /interactor - интеракторы, реализующие бизнес-логику;
    - /repository - репозитории, осуществляющие работу с данными;

    - /pkg - код библиотек, пригодных для использования в сторонних приложениях;
    - /vendor - зависимости приложений;
- директории приложений-сервисов:
    - /api - спецификации OpenAPI/Swagger, файлы JSON schema, файлы определения протоколов;
- директории web-приложений
    - /web - специальные компоненты для веб-приложений: статические веб-ресурсы, серверные шаблоны и одностраничные приложения;
- распространенные директории:
    - /configs - шаблоны файлов конфигураций и файлы настроек по-умолчанию;
    - /init - файлы конфигураций для процессов инициализации системы (systemd, upstart, sysv) и менеджеров процессов (runit, supervisord);
    - /scripts - скрипты для сборки, установки, анализа и прочих операций над проектом;
    - /build - сборка и непрерывная интеграция (Continuous Integration, CI);
    - /deployments - шаблоны и файлы конфигураций систем оркестраций IaaS, PaaS, операционных систем и контейнеров (docker-compose, kubernetes/helm, mesos, terraform, bosh);
    - /test - дополнительные внешние приложения и данные для тестирования;
- другие директории
    - /docs - документы пользователей и дизайна (в дополнение к автоматической документации godoc);
    - /tools - инструменты поддержки проекта;
    - /examples - примеры приложений и/или библиотек;
    - /third_party - внешние вспомогательные инструменты, ответвления кода и другие сторонние утилиты (например, Swagger UI);
    - /githooks - git hooks;
    - /assets - другие ресурсы, необходимые для работы (картинки, логотипы и т.д.);

Python

Для реализации требования, предъявляемых к серверной части разрабатываемой системы, отвечающей за нейронные сети, выбран язык программирования Python.

Форма представления исходного кода

Стандарт на форму представления исходного кода, написанного на Python, соответствует стандарту написания кода, предложенному разработчиками языка Python, а именно `PEP 8` (https://pep8.org/).

В качестве синтаксических анализаторов используются линтеры `flake8` (https://flake8.pycqa.org/en/latest/) и `pylint` (https://pylint.pycqa.org/en/latest/).

Стиль написания исходного кода

Имена пакетов и модулей

Модули должны иметь короткие, полностью строчные имена. Подчеркивания могут быть использованы в названии модуля, если это улучшает удобочитаемость. Пакеты Python также должны иметь короткие имена со строчными буквами, хотя использование подчеркиваний не рекомендуется.

Когда модуль расширения, написанный на C или C++, имеет сопутствующий модуль Python, который обеспечивает интерфейс более высокого уровня (например, более объектно-ориентированный), модуль C/C++ имеет начальное подчеркивание (например, _socket).

Имена классов

В именах классов обычно должно использоваться CapWords.

Имена типовых переменных

Имена переменных типа (стандарт `PEP 484`), обычно должны содержать заглавные слова, и быть короткими: `T` , `AnyStr` , `Num` . Рекомендуется добавлять суффиксы `_co` или `_contra` к переменным, используемым для объявления ковариантного или контравариантного поведения соответственно:

```
from typing import TypeVar

VT_co = TypeVar('VT_co', covariant=True)
KT_contra = TypeVar('KT_contra', contravariant=True)
```

Имена функций и переменных

Названия функций должны быть строчными, а слова должны быть разделены подчеркиванием по мере необходимости для улучшения удобочитаемости.

Имена переменных соответствуют тому же соглашению, что и имена функций.

`mixedCase` разрешен только в контекстах, где это уже преобладающий стиль (например `threading.py`), чтобы сохранить обратную совместимость.

Аргументы функции и метода

Всегда используйте `self` в качестве первого аргумента для методов экземпляра.

Всегда используйте `cls` в качестве первого аргумента для методов класса.

Если имя аргумента функции противоречит зарезервированному ключевому слову, обычно лучше добавить один завершающий символ подчеркивания, а не использовать аббревиатуру или искажение правописания (`class_` лучше, чем `clss`).

Имена методов и переменные экземпляра

Используйте правила именования функций: строчные буквы со словами, разделенными подчеркиванием, по мере необходимости для улучшения удобочитаемости.

Используйте одно начальное подчеркивание только для непубличных методов и переменных экземпляра.

Чтобы избежать конфликтов имен с подклассами, используйте два начальных символа подчеркивания для вызова правил искажения имен в Python.

Подключение зависимостей

- Импорты должны быть в отдельных строках.
- Импорты всегда помещаются в начало файла, сразу после любых комментариев модуля и строк документации, а также перед глобальными значениями модуля и константами.
  Импорты должны быть сгруппированы в следующем порядке:
  1. Импорт стандартной библиотеки.
  2. Связанный импорт третьей стороной.
  3. Импорт, специфичный для локального приложения/ библиотеки.
  4. Вы должны поместить пустую строку между каждой группой импорта.
- Рекомендуется использовать абсолютный импорт, поскольку он обычно более удобочитаем и, как правило, лучше работает (или, по крайней мере, выдает лучшие сообщения об ошибках).
- Следует избегать импорта wildcard-ами (`from <module> import *`), поскольку они делают неясным, какие имена присутствуют в пространстве имен, сбивая с толку как читателей, так и многие автоматизированные инструменты.

Комментарии

Комментарии используются для документирования. Данный процесс в `PEP 257` , который, так же как `PEP 8` , был разработан и предложен создателями языка Python. Данное соглашение описывает правила написания docstring - строковых переменных, которые идут сразу за объявлениями модулей, функций, классов, методов. С использованием генератора документации `pydoc` (https://docs.python.org/3/library/pydoc.html) можно получить документа в любом формате.

Средствами комментирования также обеспечивается трассируемость исходного кода, написанного на Python. В документирующем комментарии, в первой строке, указывается версия документа с требованиями и идентификатор требования, которое реализуется в данной части кода. Формат такого комментария: <Name>_<YYYYMMdd_hhmm#num>.

```
def Example():
    """
    Requirements_20221010_13:30#0-2-BB-00005
    Prints string.
    """
    print "Example string"
```

Модульные тесты

Стандарт на написание модульных тестов описан в стандартном пакете `unittest` (https://docs.python.org/3/library/unittest.html).

Организация проекта

В качестве макета будем использовать следующую структуру директорий и файлов:

- /data - директории с данными;
  - /external
  - /interim
  - /processed
  - /raw
- /src - исходный код проекта;
  - /data - скрипты для работы с данными;
  - /features - скрипты для преобразования данных в features;
  - /models - скрипты для тренировки и использования обученных моделей;
  - /visualization - скрипты для визуализации
- другие директории

- /reports - отчеты по моделям;
- /references - разъяснительные материалы к данным и моделям;

Swift

Для реализации требования, предъявляемых к клиентской части разрабатываемой системы выбран язык программирования Swift, разработанный компанией Apple, для написания приложения для своих платформ.

Форма представления исходного кода

Стандарт написания кода на Swift описан в официальном гайде по стилю (https://google.github.io/swift/).

В качестве синтаксического анализатора используется линтер `SwiftLint` (https://github.com/realm/SwiftLint)

Стиль написания исходного кода

Инициализаторы

Для наглядности аргументы инициализатора, которые непосредственно соответствуют сохраненному свойству, имеют то же имя, что и свойство. Явное `self.` используется во время присвоения, чтобы устранить их неоднозначность.

```
public struct Person {
  public let name: String
  public let phoneNumber: String

  public init(name: String, phoneNumber: String) {
    self.name = name
    self.phoneNumber = phoneNumber
  }
}
```

Статические свойства и свойства класса

Статические свойства и свойства класса, которые возвращают экземпляры объявляемого типа, не имеют суффикса в имени типа.

```
public class URLSession {
  public class var shared: URLSession {          // GOOD.
    // ...
  }
}

public class URLSession {
  public class var sharedSession: URLSession {   // AVOID.
    // ...
  }
}
```

Глобальные константы

Как и другие переменные, глобальные константы объявляются в `lowerCamelCase`. Венгерские обозначения, такие как ведущие `g` или `k`, не используются.

```
let secondsPerMinute = 60
```

Методы делегирования

Методы в протоколах делегирования и подобных делегатам протоколах (таких как источники данных) называются с использованием лингвистического синтаксиса, описанного ниже, который вдохновлен протоколами Cocoa.

Все методы принимают исходный объект делегата в качестве первого аргумента.

Для методов, которые принимают исходный объект делегата в качестве единственного аргумента:

- Если метод возвращает значение `Void` (например, те, которые используются для уведомления делегата о том, что произошло событие), то базовым именем метода является тип источника делегата, за которым следует указательная глагольная фраза, описывающая событие. Аргумент не помечен.

```
func scrollViewDidBeginScrolling(_ scrollView: UIScrollView)
```

- Если метод возвращает `Bool` (например, те, которые делают утверждение о самом исходном объекте делегата), то имя метода - это тип источника делегата, за которым следует указательная или условная глагольная фраза, описывающая утверждение. Аргумент не помечен.

```
func scrollViewShouldScrollToTop(_ scrollView: UIScrollView) -> Bool
```

- Если метод возвращает какое-либо другое значение (например, запрашивающее информацию о свойстве исходного объекта делегата), то базовым именем метода является существительное, описывающее запрашиваемое свойство. Аргумент помечается предлогом или фразой с завершающим предлогом, который соответствующим образом объединяет словосочетание существительного и исходный объект делегата.

```
func numberOfSections(in scrollView: UIScrollView) -> Int
```

Для методов, которые принимают дополнительные аргументы после исходного объекта делегата, базовое имя метода само по себе является исходным типом делегата, а первый аргумент не помечен. Затем:

- Если метод возвращает значение `Void`, второй аргумент помечается указательной глагольной фразой, описывающей событие, для которого аргумент является прямым объектом или предложным объектом, а любые другие аргументы (если они присутствуют) предоставляют дополнительный контекст.

```
func tableView(
  _ tableView: UITableView,
  willDisplayCell cell: UITableViewCell,
  forRowAt indexPath: IndexPath)
```

- Если метод возвращает `Bool`, второй аргумент помечается индикативной или условной глагольной фразой, которая описывает возвращаемое значение в терминах аргумента, а любые другие аргументы (если они присутствуют) предоставляют дополнительный контекст.

```
func tableView(
  _ tableView: UITableView,
  shouldSpringLoadRowAt indexPath: IndexPath,
  with context: UISpringLoadedInteractionContext
) -> Bool
```

- Если метод возвращает какое-либо другое значение, второй аргумент помечается словосочетанием существительного и завершающим предлогом, который описывает возвращаемое значение в терминах аргумента, а любые другие аргументы (если они присутствуют) предоставляют дополнительный контекст.

```
func tableView(
  _ tableView: UITableView,
  heightForRowAt indexPath: IndexPath
) -> CGFloat
```

Подключение зависимостей

Импорты - это первые маркеры (не считая комментариев) в исходном файле. Они сгруппированы в группы, упорядочены лексикографически, и имеют ровно одну пустую строку между каждой группой:

- Импорт модуля/подмодуля не тестируется.
- Импорт отдельных деклараций (class, enum, func, struct, var).
- Модули, импортированные с помощью @testable (присутствуют только в тестовых источниках).

```
import CoreLocation
import MyThirdPartyModule
import SpriteKit
import UIKit

import func Darwin.C.isatty

@testable import MyModuleUnderTest
```

Комментарии

Комментарии используются для документирования, в качестве основного инструмента для генерации документации используется решение от Apple - `DocC` (https://developer.apple.com/documentation/docc). Данный инструмент использует особый синтаксис - модифицированную версию языка разметки Markdown, для генерации документации.

Соглашения написания документации при помощи Markdown описаны в официальном гайде от Apple (https://developer.apple.com/documentation/xcode/formatting-your-documentation-content).

Средствами комментирования также обеспечивается трассируемость исходного кода, написанного на Swift. В документирующем комментарии, в первой строке, указывается версия документа с требованиями и идентификатор требования, которое реализуется в данной части кода. Формат такого комментария: <Name>_<YYYYMMdd_hhmm#num>.

```
// Requirements_20221010_13:30#0-2-BB-00005
// Returns string.
func example() -> String {
    return "Example string"
}
```

Организация проекта

В качестве макета будем использовать макет, основанный на принципах MVVM и Clean Architecture.

- директории приложения:
  - /Application - файлы конфигурации;
  - /Data - исходный код для работы с данными (репозитории, gateways и т.д.);
  - /Domain - исходный код, реализующий бизнес-логику:
    - /Entity - сущности бизнес-логики;
    - /Interactor - интеракторы, реализующие бизнес-логику;
  - /Presentation - исходный код, реализующий представления;
- директория тестов;
- директория тестов интерфейса;