Практическая работа № 11

Инспектирование программных систем.

Цели работы:

* 1. Закрепить теоретические знания по инспектирование компонентов программного обеспечения на предмет соответствия стандартам кодирования.
  2. Получить навыки производить инспектирование компонентов программного обеспечения.

|  |  |
| --- | --- |
| **Наименование правила** | **Описание правила** |
| Правило 1. Следуйте стандартам оформления кода | У каждого языка программирования есть свой стандарт оформления кода, который говорит, как надо делать отступы, где ставить пробелы и скобки, как называть объекты, как комментировать код и т.д. |
| Правило 2. Давайте наглядные имена | Имена классов и типов должны быть существительными. Название метода должно содержать глагол. Если метод определяет, является ли какая-то информация об объекте истинной или ложной, его имя должно начинаться с «is». Методы, которые возвращают свойства объектов, должны начинаться с «get», а устанавливающие значения свойств — «set». |
| Правило 3. Комментируйте и документируйте | Начинайте каждый метод и процедуру с описания в комментарии того, что данный метод или процедура делает, параметров, возвращаемого значения и возможных ошибок и исключений. Опишите в комментариях роль каждого файла и класса, содержимое каждого поля класса и основные шаги сложного кода. Пишите комментарии по мере разработки кода. |
| Правило 4. Не повторяйтесь | Никогда не копируйте и не вставляйте код. Вместо этого выделите общую часть в метод или класс (или макрос, если нужно), и используйте его с соответствующими параметрами. Избегайте использования похожих данных и кусков кода. Также используйте следующие техники: Создание справочников API из комментариев, используя Javadoc и Doxygen. Автоматическая генерация Unit-тестов на основе аннотаций или соглашений об именовании. Генерация PDF и HTML из одного размеченного источника. Получение структуры классов из базы данных (или наоборот). |
| Правило 5. Проверяйте на ошибки и реагируйте на них | Методы могут возвращать признаки ошибки или генерировать исключения. Обрабатывайте их. Не полагайтесь на то, что диск никогда не заполнится, ваш конфигурационный файл всегда будет на месте, ваше приложение будет запущено со всеми нужными правами, запросы на выделение памяти всегда будут успешно исполнены, или что соединение никогда не оборвётся. Да, хорошую обработку ошибок тяжело написать, и она делает код длиннее и труднее для чтения. Но игнорирование ошибок просто заметает проблему под ковёр, где ничего не подозревающий пользователь однажды её обнаружит. |
| Правило 6. Разделяйте код на короткие, обособленные части | Каждый метод, функция или блок кода должен умещаться в обычном экранном окне (25-50 строк). Если получилось длиннее, разделите на более короткие куски. Даже внутри метода разделяйте длинный код на блоки, суть которых вы можете описать в комментарии в начале каждого блока. Более того, каждый класс, модуль, файл или процесс должен выполнять определённый род задач. Если часть кода выполняет совершенно разнородные задачи, то разделите его соответственно. |
| Правило 7. Используйте API фреймворков и сторонние библиотеки | Изучите, какие функции доступны с помощью API вашего фреймворка. а также что могут делать развитые сторонние библиотеки. Если библиотеки поддерживаются вашим системным менеджером пакетов, то они скорее всего окажутся хорошим выбором. Используйте код, удерживающий от желания изобретать колесо (при том бесполезной квадратной формы). |
| Правило 8. Не переусердствуйте с проектированием | Проектируйте только то, что актуально сейчас. Ваш код можно делать довольно обобщённым, чтобы он поддерживал дальнейшее развитие, но только в том случае, если он не становится от этого слишком сложным. Не создавайте параметризованные классы, фабрики, глубокие иерархии и скрытые интерфейсы для решения проблем, которых даже не существует — вы не можете угадать, что случится завтра. С другой стороны, когда структура кода не подходит под задачу, не стесняйтесь рефакторить его. |
| Правило 9. Будьте последовательны | Делайте одинаковые вещи одинаковым образом. Если вы разрабатываете метод, функциональность которого похожа на функциональность уже существующего, то используйте похожее имя, похожий порядок параметров и схожую структура тела. То же самое относится и к классам. Создавайте похожие поля и методы, делайте им похожие интерфейсы, и сопоставляйте новые имена уже существующим в похожих классах. Ваш код должен соответствовать соглашениям вашего фреймворка. Например, хорошей практикой является делать диапазоны полуоткрытыми: закрытыми (включающими) слева (в начале диапазона) и открытыми (исключающими) справа (в конце). Если для конкретного случая нет соглашений, то сделайте выбор и фанатично придерживайтесь его. |
| Правило 10. Избегайте проблем с безопасностью | Современный код редко работает изолированно. У него есть неизбежный риск стать мишенью атак. Они необязательно должны приходить из интернета; атака может происходить через входные данные вашего приложения. В зависимости от вашего языка программирования и предметной области, вам возможно стоит побеспокоиться о переполнении буфера, кросс-сайтовых сценариях, SQL-инъекциях и прочих подобных проблемах. Изучите эти проблемы, и избегайте их в коде. |
| Правило 11. Используйте эффективные структуры данных и алгоритмы | Простой код часто легче сопровождать, чем такой же, но изменённый ради эффективности. К счастью, вы можете совмещать сопровождаемость и эффективность, используя структуры данных и алгоритмы, которые даёт ваш фреймворк. Используйте map, set, vector и алгоритмы, которые работают с ними. Благодаря этому ваш код станет чище, быстрее, более масштабируемым и более экономным с памятью. Например, если вы сохраните тысячу значений в отсортированном множестве, то операция пересечения найдёт общие элементы с другим множеством за такое же число операций, а не за миллион сравнений. |
| Правило 12. Используйте Unit-тесты | Сложность современного ПО делает его установку дороже, а тестирование труднее. Продуктивным подходом будет сопровождение каждого куска кода тестами, которые проверяют корректность его работы. Этот подход упрощает отладку, т.к. он позволяет обнаружить ошибки раньше. Unit-тестирование необходимо, когда вы программируете на языках с динамической типизацией, как Python и JavaScript, потому что они отлавливают любые ошибки только на этапе исполнения, в то время как языки со статической типизацией наподобие Java, C# и C++ могут поймать часть из них во время компиляции. Unit-тестирование также позволяет рефакторить код уверенно. Вы можете использовать XUnit для упрощения написания тестов и автоматизации их запуска. |
| Правило 13. Сохраняйте код портируемым | Если у вас нет особой причины, не используйте функциональность, доступную только на определённой платформе. Не полагайтесь на то, что определённые типы данных (как integer, указатели и временные метки) будут иметь конкретную длину (например, 32 бита), потому что этот параметр отличается на разных платформах. Храните сообщения программы отдельно от кода и на зашивайте параметры, соответствующие определённой культуре (например, разделители дробной и целой части или формат даты). Соглашения нужны для того, чтобы код мог запускаться в разных странах, так что сделайте локализацию настолько безболезненной, насколько это возможно. |
| Правило 14. Делайте свой код собираемым | Простая команда должна собирать ваш код в форму, готовую к распространению. Команда должна позволять вам быстро выполнять сборку и запускать необходимые тесты. Для достижения этой цели используйте средства автоматической сборки наподобие Make, Apache Maven, или Ant. В идеале, вы должны установить интеграционную систему, которая будет проверять, собирать и тестировать ваш код при любом изменении. |
| Правило 15. Размещайте всё в системе контроля версий | Все ваши элементы — код, документация, исходники инструментов, сборочные скрипты, тестовые данные — должны быть в системе контроля версий. Git и GitHub делают эту задачу дешёвой и беспроблемной. Но вам также доступны и многие другие мощные инструменты и сервисы. Вы должны быть способны собрать и протестировать вашу программу на сконфигурированной системе, просто скачав её с репозитория. |
| Правило 16. Следуйте принципу единственной ответственности (Single Responsibility Principle) | Один класс должен решать только какую-то одну задачу. Он может иметь несколько методов, но они должны использоваться лишь для решения общей задачи. Все методы и свойства должны служить одной цели. Если класс имеет несколько назначений, его нужно разделить на отдельные классы. |
| Правило 17. Следуйте принцип открытости/закрытости (Open-closed Principle) | Программные сущности (классы, модули, функции и прочее) должны быть расширяемыми без изменения своего содержимого. Если строго соблюдать этот принцип, то можно регулировать поведение кода без изменения самого исходника. |
| Правило 18. Следуйте принципу разделения интерфейса | Нельзя заставлять клиента реализовать интерфейс, которым он не пользуется. Это означает, что нужно разбивать интерфейсы на более мелкие, лучше удовлетворяющие конкретным потребностям клиентов. Как и в случае с принципом единственной ответственности, цель принципа разделения интерфейса заключается в минимизации побочных эффектов и повторов за счёт разделения ПО на независимые части. |
| Правило 19. Следуйте принципу инверсии зависимостей (Dependency Inversion Principle) | Высокоуровневые модули не должны зависеть от низкоуровневых. Оба вида модулей должны зависеть от абстракций. Абстракции не должны зависеть от подробностей. Подробности должны зависеть от абстракций. Применяя этот принцип, одни модули можно легко заменять другими, всего лишь меняя модуль зависимости, и тогда никакие перемены в низкоуровневом модуле не повлияют на высокоуровневый. |
| Правило 20. Оборачивайте примитивные типы (продолжение) | Оборачивая примитивные типы в классы, мы инкапсулируем (скрываем) тип. Если позже в ходе рефакторинга мы захотим изменить примитивные типы, это можно будет сделать в одном месте. А ещё такой код легче воспринимать, ведь по сигнатуре объекта-обёртки сразу ясно, что передавать методу в качестве параметров. А ещё такой код легче воспринимать, ведь по сигнатуре объекта-обёртки сразу ясно, что передавать методу в качестве параметров. |
| Правило 21. Не больше двух переменных экземпляра на класс | Один класс должен иметь дело с одним состоянием, максимум с двумя. Так что, если в классе более двух переменных экземпляра, возможно, он нарушает принцип единственной ответственности. |
| Правило 22. Соблюдайте принцип делегирования | Не делайте всё самостоятельно, делегируйте это в соответствующий класс. Классическим примером этого принципа являются методы equals() и hashCode() в ООП языках. Если нужно сравнить два объекта, это действие поручается соответствующему классу вместо клиентского. Основным преимуществом этого принципа является отсутствие дублирования кода и довольно простое изменение поведения. Этот принцип относится также к делегированию событий (событие делегируется соответствующему обработчику). |
| Правило 23. Программирование для интерфейса | Этот принцип подразумевает, что следует по возможности программировать для интерфейса, а не для его реализации. Это даст вам гибкий код, который может работать с любой новой реализацией интерфейса. Другими словами, нужно использовать тип интерфейса для переменных, возвращаемых типов или типа аргумента метода. Например, использовать для хранения объекта суперкласс, а не подкласс. |
| Правило 24. Инкапсулируйте изменяющийся код | Сервисы стремительно развиваются. Выход в релиз проекта подразумевает постоянные изменения кода и его поддержку. Отсюда следует — инкапсулируйте код, который с большой вероятностью будет изменён в будущем. Преимущество этого принципа ООП заключается в том, что инкапсулированный код легко тестировать и поддерживать. Воспользуйтесь алгоритмом, по которому переменные и методы по умолчанию имеют спецификатор private. Затем шаг за шагом увеличиваете доступ при необходимости (с private на protected, с protected на public). Одним из вариантов инкапсуляции является Фабричный метод. Он инкапсулирует код создания объекта и обеспечивает гибкость для последующего создания новых объектов без влияния на существующий код. |
| Правило 25. Используйте композицию вместо наследования | Существует два основных способа повторного использования кода: наследование и композиция. Оба они имеют свои преимущества и недостатки, но, как правило, предпочтение рекомендуется отдавать последнему, если это возможно. Обусловлено это тем, что композиция гибче наследования. Композиция позволяет изменять поведение класса прямо во время выполнения через установку его свойств. Реализуя интерфейсы, вы, таким образом, используете полиморфизм, который обеспечивает более гибкую реализацию. |
| Правило 26. Используйте принцип подстановки Барбары Лисков (LSP) | Согласно этому принципу подтипы должны быть заменяемыми для супертипа. Другими словами, методы или функции, работающие с суперклассом, должны иметь возможность без проблем работать также и с его подклассами. |

Задание на практическую работу:

1. Получите от преподавателя исходный текст для проведения инспектирования

users/views.py

from django.shortcuts import render, redirect, get\_object\_or\_404

from django.contrib import messages

from django.http import JsonResponse, HttpResponse

from django.contrib.auth.decorators import login\_required

from .forms import UserRegistrationForm, ProfileUpdateForm

from .models import User, Profile

import datetime

import os

def user\_register(request):

if request.method == 'POST':

form = UserRegistrationForm(request.POST)

if form.is\_valid():

form.save()

messages.success(request, 'Регистрация успешно завершена!')

return redirect('login')

else:

form = UserRegistrationForm()

return render(request, 'users/register.html', {'form': form})

@login\_required

def global\_var\_usage(request):

global users\_list

users\_list = User.objects.all()

return render(request, 'users/user\_list.html', {'users': users\_list})

@login\_required

def lst\_usr(request):

lst = User.objects.prefetch\_related('profile').all()

return render(request, 'users/user\_list.html', {'lst': lst})

def get\_user\_details(request, user\_id):

user = User.objects.get(id=user\_id)

return render(request, 'users/user\_detail.html', {'user': user})

@login\_required

def incorrect\_redirect(request):

messages.info(request, "You are being redirected")

return redirect('/unknown\_path/')

@login\_required

def mixed\_responsibility\_view(request):

if request.method == 'POST':

user = User.objects.create(username='test\_user')

print("User created")

else:

users = User.objects.all()

print("Fetched Users")

return redirect('home')

@login\_required

def hard\_dependency\_view(request):

class HardDependency:

def execute(self):

return "Hardcoded dependency executed"

dep = HardDependency()

return HttpResponse(dep.execute())

@login\_required

def duplicate\_user\_list(request):

users = User.objects.all()

return render(request, 'users/user\_list.html', {'users': users})

@login\_required

def duplicate\_user\_list2(request):

users = User.objects.all()

return render(request, 'users/user\_list.html', {'users': users})

@login\_required

def user\_email\_list(request):

users = User.objects.only('email')

for user in users:

print(user)

def search\_user(request):

username = request.GET.get('username')

users = User.objects.raw(f"SELECT \* FROM users\_user WHERE username = '{username}'")

return render(request, 'users/search\_results.html', {'users': users}

@login\_required

def ui\_mixed\_with\_logic(request):

if request.method == 'POST':

user = User.objects.create(username='ui\_logic\_user')

print("User created in view directly")

return render(request, 'users/register.html')

def error\_handling\_example(request):

try:

user = User.objects.get(id=999)

except:

print("Error occurred")

@login\_required

def hardcoded\_template(request):

return render(request, 'hardcoded\_template.html')

@login\_required

def long\_view\_function(request):

users = User.objects.all()

for i in range(100):

print(i)

for user in users:

print(user.username)

for j in range(200):

print(j)

return HttpResponse("Long function executed")

@login\_required

def secret\_key\_view(request):

api\_key = "SECRET\_KEY\_123"

return HttpResponse(f"API Key: {api\_key}")

@login\_required

def inefficient\_db\_queries(request):

users = User.objects.all()

user\_count = users.count()

for user in users:

print(user.username)

return render(request, 'users/user\_list.html', {'user\_count': user\_count})

1. Используя правила найдите дефекты исходного текста, указав способ исправления. Рекомендуются использовать табличный способ представления результаты инспекции

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование правила | Исходный текст | Дефект | Предложенный вариант исправления |
| (желтый) Правило 1. Следуйте стандартам оформления кода | def user\_register(request): if request.method == 'POST': form = UserRegistrationForm(request.POST) | Неправильные отступы и форматирование блока кода. | python<br>def user\_register(request):<br> if request.method == 'POST':<br> form = UserRegistrationForm(request.POST) |
| (зеленый) Правило 2. Давайте наглядные имена | def lst\_usr(request): | Имя функции lst\_usr неинформативно. | def list\_users(request): |
| (синий) Правило 3. Комментируйте и документируйте | def risky\_operation(): try: 1 / 0 except: pass | Отсутствие комментариев и обработки исключений. | python<br>def risky\_operation():<br> """Handle division by zero error."""<br> try:<br> 1 / 0<br> except ZeroDivisionError:<br> print("Cannot divide by zero") |
| (голубой) Правило 4. Не повторяйтесь | def duplicate\_user\_list(request): и def duplicate\_user\_list2(request): | Дублирование функциональности в двух методах. | Один метод user\_list должен быть универсальным для отображения пользователей. |
| (оранжевый) Правило 5. Проверяйте на ошибки и реагируйте на них | def get\_user\_details(request, user\_id): user = User.objects.get(id=user\_id) | Отсутствие обработки исключений при доступе к пользователю по ID. | python<br>def get\_user\_details(request, user\_id):<br> try:<br> user = User.objects.get(id=user\_id)<br> except User.DoesNotExist:<br> return HttpResponse("User not found", status=404) |
| (серый) Правило 6. Разделяйте код на короткие, обособленные части | def mixed\_responsibility\_view(request): | Метод смешивает логику создания и получения пользователей. | Разделить на два метода: create\_user() и get\_users() |
| (малиновый) Правило 7. Используйте API фреймворков и сторонние библиотеки | def get\_current\_time(): return datetime.datetime.now().isoformat() | Использование стандартного datetime вместо Django timezone. | python<br>from django.utils import timezone<br>def get\_current\_time():<br> return timezone.now().isoformat() |
| (бирюзовый) Правило 9. Будьте последовательны | def lst\_usr(request): и def user\_list(request): | Несогласованные имена функций. | Привести к одному стандарту: def user\_list(request): |
| (красный) Правило 10. Избегайте проблем с безопасностью | api\_key = "SECRET\_KEY\_123" | Хранение чувствительных данных в коде. | Сохранить ключ в переменных окружения или настройках Django. |
| (сиреневый) Правило 11. Используйте эффективные структуры данных и алгоритмы | for user in User.objects.all(): print(user.username) | Неоптимальный перебор данных из базы данных. | python<br>users = User.objects.values\_list('username', flat=True)<br>for username in users:<br> print(username) |
| (темно-серый) Правило 12. Используйте Unit-тесты | def critical\_function(): return "Important logic, but no tests" | Отсутствие тестов для метода. | Написать модульный тест, проверяющий корректность выполнения метода. |
| (коричневый) Правило 13. Сохраняйте код портируемым | path = os.path.join("C:\\", "system32") | Платформозависимый путь. | Использовать относительные пути или settings.BASE\_DIR. |
| (темно-синий) Правило 15. Размещайте всё в системе контроля версий | Отсутствие README.md и .gitignore | Добавить файлы .gitignore и README.md, описывающие проект и исключающие ненужные файлы. |  |
| (темно-красный) Правило 16. Следуйте принципу единственной ответственности | def mixed\_responsibility\_view(request): | Метод выполняет несколько несвязанных задач. | Разделить метод на отдельные задачи: create\_user() и list\_users() |
| (пурпурный) Правило 19. Следуйте принципу инверсии зависимостей | class HardDependency: def execute(self): return "Hardcoded" | Жесткая зависимость от конкретной реализации. | Внедрить зависимость через интерфейс или абстракцию. |
| (фиолетовый) Правило 20. Оборачивайте примитивные типы | def hardcoded\_secret(): api\_key = "SECRET\_KEY\_123" | Примитивный тип используется для чувствительных данных. | Использовать объект SecretKey с инкапсуляцией. |

Участки кода выделяются и отмечаются меткой или номером соответствующего требования. Если участок кода соответствует требованиям, то необходимо отметить этот факт либо цветом выделения, либо соответствующим текстовым примечанием. Если участок кода имеет проблемы, этот факт должен быть отражен либо цветом выделения, либо ссылкой на соответствующий пункт списка замечаний в бланке инспекции.

1. Исправленный код

from django.shortcuts import render, redirect, get\_object\_or\_404

from django.contrib import messages

from django.http import JsonResponse, HttpResponse

from django.contrib.auth.decorators import login\_required

from .forms import UserRegistrationForm, ProfileUpdateForm

from .models import User, Profile

from django.utils import timezone

import os

def user\_register(request):

if request.method == 'POST':

form = UserRegistrationForm(request.POST)

if form.is\_valid():

form.save()

messages.success(request, 'Регистрация успешно завершена!')

return redirect('login')

else:

form = UserRegistrationForm()

return render(request, 'users/register.html', {'form': form})

@login\_required

def list\_users(request):

users = User.objects.prefetch\_related('profile').all()

return render(request, 'users/user\_list.html', {'users': users})

def get\_user\_details(request, user\_id):

try:

user = User.objects.get(id=user\_id)

except User.DoesNotExist:

return HttpResponse("User not found", status=404)

return render(request, 'users/user\_detail.html', {'user': user})

@login\_required

def user\_list(request):

users = User.objects.all()

return render(request, 'users/user\_list.html', {'users': users})

@login\_required

def correct\_redirect(request):

messages.info(request, "You are being redirected")

return redirect('home')

@login\_required

def create\_user(request):

if request.method == 'POST':

User.objects.create(username='test\_user')

messages.success(request, 'User created successfully!')

return redirect('home')

@login\_required

def fetch\_users(request):

users = User.objects.all()

return render(request, 'users/user\_list.html', {'users': users})

class Dependency:

def execute(self):

return "Dependency executed successfully"

@login\_required

def dependency\_view(request):

dependency = Dependency()

return HttpResponse(dependency.execute())

@login\_required

def email\_list(request):

emails = User.objects.values\_list('email', flat=True)

for email in emails:

print(email)

return HttpResponse("Emails printed in console")

def search\_user(request):

username = request.GET.get('username')

users = User.objects.filter(username=username)

return render(request, 'users/search\_results.html', {'users': users})

@login\_required

def create\_user\_ui(request):

if request.method == 'POST':

User.objects.create(username='ui\_logic\_user')

messages.success(request, 'User created via UI!')

return render(request, 'users/register.html')

def error\_handling\_example(request):

try:

user = User.objects.get(id=999)

except User.DoesNotExist:

return HttpResponse("User does not exist", status=404)

@login\_required

def user\_template(request):

return render(request, 'users/user\_template.html')

@login\_required

def short\_function(request):

users = User.objects.values\_list('username', flat=True)

for username in users[:10]:

print(username)

return HttpResponse("Function executed successfully")

@login\_required

def secure\_view(request):

api\_key = os.environ.get('API\_KEY', 'default\_key')

return HttpResponse(f"API Key: {api\_key}")

@login\_required

def efficient\_db\_queries(request):

users = User.objects.values('username')

user\_count = users.count()

return render(request, 'users/user\_list.html', {'user\_count': user\_count})

class DependencyInjection:

def \_\_init\_\_(self, dependency):

self.dependency = dependency

def run(self):

return self.dependency.execute()

@login\_required

def dependency\_injection\_view(request):

dep = Dependency()

injection = DependencyInjection(dep)

return HttpResponse(injection.run())

Вывод:

1.1. Закрепил теоретические знания по инспектирование компонентов программного обеспечения на предмет соответствия стандартам кодирования.

1.2. Получил навыки производить инспектирование компонентов программного обеспечения.