ДИСЦИПЛИНА	Программирование корпоративных систем
ИНСТИТУТ	Институт перспективных технологий и индустриального программирования
КАФЕДРА	Кафедра индустриального программирования
ВИД УЧЕБНОГО МАТЕРИАЛА	Практические задание
ПРЕПОДАВАТЕЛЬ	Адышкин Сергей Сергеевич
СЕМЕСТР	5 семестр, 2025-2026 гг.

Практическое занятие №14

Тестирование и оптимизация мобильного приложения.

Исправление ошибок (Flutter)

Цели занятия

- Освоить базовые виды тестирования во Flutter: unit-, widget- и integration-tests.
- Настроить линтинг и статический анализ, повысить качество кода.
- Научиться профилировать производительность (FPS, jank, память, пропуски кадров) через DevTools и Performance Overlay.
- Применить практики оптимизации: уменьшение количества перестроений, работа со списками, изоляция тяжёлых вычислений, оптимизация изображений.
- Настроить сбор аварий (Crashlytics/Sentry — опционально).
- Отработать цикл «поиск дефекта → воспроизведение → минимальный пример → исправление → тест/регресс».

Теоретическая часть

Виды тестов во Flutter

- **Unit tests** — проверяют чистую логику (валидаторы, форматтеры, репозитории без платформенной зависимости).
- **Widget tests** — рендеринг и поведение виджетов в изолированной среде (pump, pumpWidget, pumpAndSettle).
- **Integration tests** — end-to-end сценарии на реальном устройстве/эмуляторе (запуск приложения, навигация, ввод).

Инструменты:

- `flutter_test` (из коробки), `integration_test` (официальный пакет).
- Покрытие: `flutter test --coverage` → `coverage/lcov.info`.

Линтинг и статический анализ

- Подключите `flutter_lints` и/или расширенный набор правил.
- Запуск анализа: `flutter analyze`.
- Цель: ранний захват проблем стиля/безопасности (nullable, dispose, неправильные ключи списков и т.д.).
- Линты особенно важны при работе со списками и состоянием, где ранее мы уделяли внимание производительности и корректной идентичности элементов (Keys).

Профилирование и метрики производительности

- **Режимы сборки**: `debug` (удобно для разработки), `profile` (точные замеры производительности), `release`.
- **Performance Overlay**:

`WidgetsApp(showPerformanceOverlay: true)` или через Flutter Inspector.

- **DevTools → Performance/CPU/Memory**:
  - Timeline Events (build/layout/paint), пропуски кадров (jank).
  - Аллокации памяти и GC-паузы.
- **Анализ размера сборки**: `flutter build apk --release --analyze-size` (откроет отчёт для DevTools).

Практики оптимизации UI и списков

- Используйте `const` конструкторы для неизменяемых поддеревьев.
- Следите за **границами перестроения**: выносите тяжёлые части в отдельные виджеты.
- Для больших коллекций — `ListView.builder`/`GridView.builder`; фиксируйте `itemExtent/prototypeItem`, не злоупотребляйте `shrinkWrap`.
- Стабильные `Key` (обычно `ValueKey(id)`) для строк списков, **особенно** при анимациях/удалении.
- Изображения: правильные размеры, кэширование, `FadeInImage`, `precacheImage`.
- Тяжёлые вычисления и парсинг — в отдельный изолят/`compute`.
- Для локальной БД — пагинация/индексы; для облака — серверная сортировка/фильтрация. (Мы уже прорабатывали эти аспекты в ПЗ про списки и БД.)
-

Обработка ошибок и логирование

- Глобальные перехватчики:
  - `FlutterError.onError = ...` (ошибки Flutter),
  - `runZonedGuarded(..., (e, s) { ... })` (async).
- Пользовательские экраны падения: `ErrorWidget.builder = ...`.
- Crash reporting (опционально): Firebase Crashlytics / Sentry.

Практический цикл исправления дефекта

1. **Воспроизвести**: точные шаги, устройство/ОС, версия приложения.
2. **Сузить** до минимального примера.
3. **Пишем тест**, воспроизводящий баг (по возможности).
4. **Исправляем** и запускаем тесты/анализ/профилирование.
5. **Регресс**: проверяем смежные сценарии и показатели производительности.

Практическая часть

Исходник: возьмите одно из ваших учебных приложений по курсу (например, *Simple Notes* или *Notes SQLite* — из ПЗ №5/10) и используйте его как базу для тестирования и оптимизации.

Шаг 0. Подготовка

- Обновите зависимости: `flutter pub upgrade`.
- Включите линты в `pubspec.yaml`:

```
dev_dependencies:
  flutter_lints: ^4.0.0
```

В `analysis_options.yaml`:

```
include: package:flutter_lints/flutter.yaml
```

- Запустите: `flutter analyze` — зафиксируйте исходные предупреждения (в отчёт).

Шаг 1. Unit-tests (логика)

- Создайте `test/validators_test.dart` (или для вашей бизнес-логики).
- Напишите 3–5 тестов, покрывающих граничные случаи (пустые строки, длинные значения, неверные форматы и т.п.).
- Запустите: `flutter test` и `flutter test --coverage`.
- Зафиксируйте проценты покрытия (добавьте скрин/цифры в отчёт).

Шаг 2. Widget-tests (UI-поведение)

- Создайте `test/notes_page_test.dart`.
- Примеры проверок:
  - Рендер заголовка экрана.
  - Тап по `FloatingActionButton` открывает диалог/форму.
  - Ввод текста и нажатие «Сохранить» добавляет элемент (с заглушками репозитория, если надо).
- Используйте `pumpWidget`, `find.text`, `find.byIcon`, `tester.tap`, `pumpAndSettle`.

Шаг 3. (Опционально) Integration-test

- Подключите `integration_test` и создайте сценарий E2E: открытие приложения, добавление, редактирование, удаление заметки.
- Запуск на эмуляторе/устройстве:
    - Android: `flutter test integration_test` или через `flutter drive` (в зависимости от шаблона).

Шаг 4. Профилирование

- Соберите профильную сборку: `flutter run --profile`.
- Откройте **DevTools** → вкладки **Performance** и **Memory**.
- Включите **Performance Overlay**, воспроизведите типичный пользовательский сценарий (скролл длинного списка, добавление/редактирование).
- Зафиксируйте: средний FPS, пики времени кадра, число пропущенных кадров, рост памяти. Скриншоты/заметки — в отчёт.

Шаг 5. Оптимизация

Примените минимум **5** из пунктов ниже (с пояснением «что было/что стало»):

- Заменить `ListView(children: ...)` на `ListView.builder` в больших списках; добавить `itemExtent`/`prototypeItem`.
- Добавить стабильные `Key` для элементов списка/`Dismissible`.
- Разбить большие виджеты на подвиджеты, пометить неизменяемые поддеревья `const`.
- Вынести тяжёлый парсинг/сериализацию в `compute`/изолят.
- Оптимизировать изображения (уменьшить исходный размер, использовать `FadeInImage`/кэширование, `precacheImage`).
- Для локальной БД (если используете SQLite из ПЗ №10): добавить индекс по полю сортировки, внедрить пагинацию (`LIMIT/OFFSET`).
- Для облачных БД (Firebase/Supabase из ПЗ №8/9): сортировка/фильтрация на стороне сервиса, корректные правила безопасности (в продакшене).

Повторно прогоните **Шаг 4** и сравните метрики «до/после».

Шаг 6. Анализ размера приложения

- `flutter build apk --release --analyze-size`
- Откройте отчёт в DevTools и найдите крупные вкладчики (packages/assets).
- Примените минимум **2** меры снижения размера:
  - Tree-shake иконок (`flutter_launcher_icons`/`cupertino_icons` по необходимости).
  - `--split-per-abi` для APK: `flutter build apk --release --split-per-abi`.
  - Удаление неиспользуемых ассетов/шрифтов.
- Зафиксируйте сравнение: размер до/после.

Шаг 7. Обработка ошибок

- Добавьте глобальные перехватчики (`FlutterError.onError`, `runZonedGuarded`).
- Переопределите `ErrorWidget.builder` на дружелюбный экран.
- (Опционально) Подключите Crashlytics/Sentry и убедитесь, что неудачные сценарии логируются (скрин/описание в отчёт).

Контрольные точки

1. Проект собирается, линтер настроен, `flutter analyze` без ошибок (или с сокращённым числом предупреждений vs baseline).
2. Написано ≥5 unit/widget тестов; прогон `flutter test` успешен; собраны метрики покрытия.
3. Получены профильные метрики (DevTools/Overlay) **до** оптимизаций.
4. Внесены оптимизации (≥5 пунктов), метрики **после** зафиксированы и улучшены.
5. Выполнен анализ размера, приняты меры снижения, показано «до/после».

6. Реализован базовый перехват ошибок + пользовательский экран ошибки.

Что сдаём (контрольные задания)

- Скрин `flutter analyze` (до/после — можно 2 скрина).
- Скрин/лог `flutter test` и строка покрытия (coverage %).
- 2–3 скрина из DevTools/Overlay: **до** оптимизаций и **после** (с краткими комментариями).
- Табличку «Оптимизация → Зачем → Как реализовано → Эффект (цифрами/наблюдениями)».
- Скрин отчёта **Analyze Size** и список предпринятых мер; фактический размер APK/AAB до/после.
- 1 скрин пользовательского экрана ошибки + фрагмент кода перехватчика.

Формат отчётности и детализация шагов соответствуют практикам курса (см. контрольные задания и отчётность в ПЗ №3–10).

Требования к отчёту

- Объём: 3–5 страниц + скриншоты контрольных этапов.
- Структура:

1. Цели, 2) Среда/версии, 3) Тестирование (unit/widget/integration + coverage),
2. Профилирование «до», 5) Перечень оптимизаций (с пояснениями и кодом),
3. Профилирование «после», 7) Анализ размера «до/после», 8) Обработка ошибок, 9) Выводы.

- По желанию — ссылка на репозиторий с веткой `feature/pz14-testing-optimization`.

Контрольные вопросы

1. Чем отличаются unit-, widget- и integration-tests? Когда какой вид выбирать?
2. Какие приёмы помогают снизить количество перестроений виджетов?
3. Зачем спискам стабильные `Key` и какой тип ключа предпочтителен?
4. Когда уместно выносить задачи в изолят/`compute`? Какие симптомы подскажут, что это нужно?
5. Какие показатели в DevTools вы смотрите в первую очередь для оценки производительности?
6. Что даёт `--analyze-size` и какие типичные источники «раздувания» сборки?
7. Чем чревато использование `shrinkWrap` и когда его можно включать без вреда?
8. Какие различия между локальной оптимизацией (SQLite) и облачной (Firebase/Supabase)? Примеры мер «там и там».