
Яндекс

Яндекс

SECR

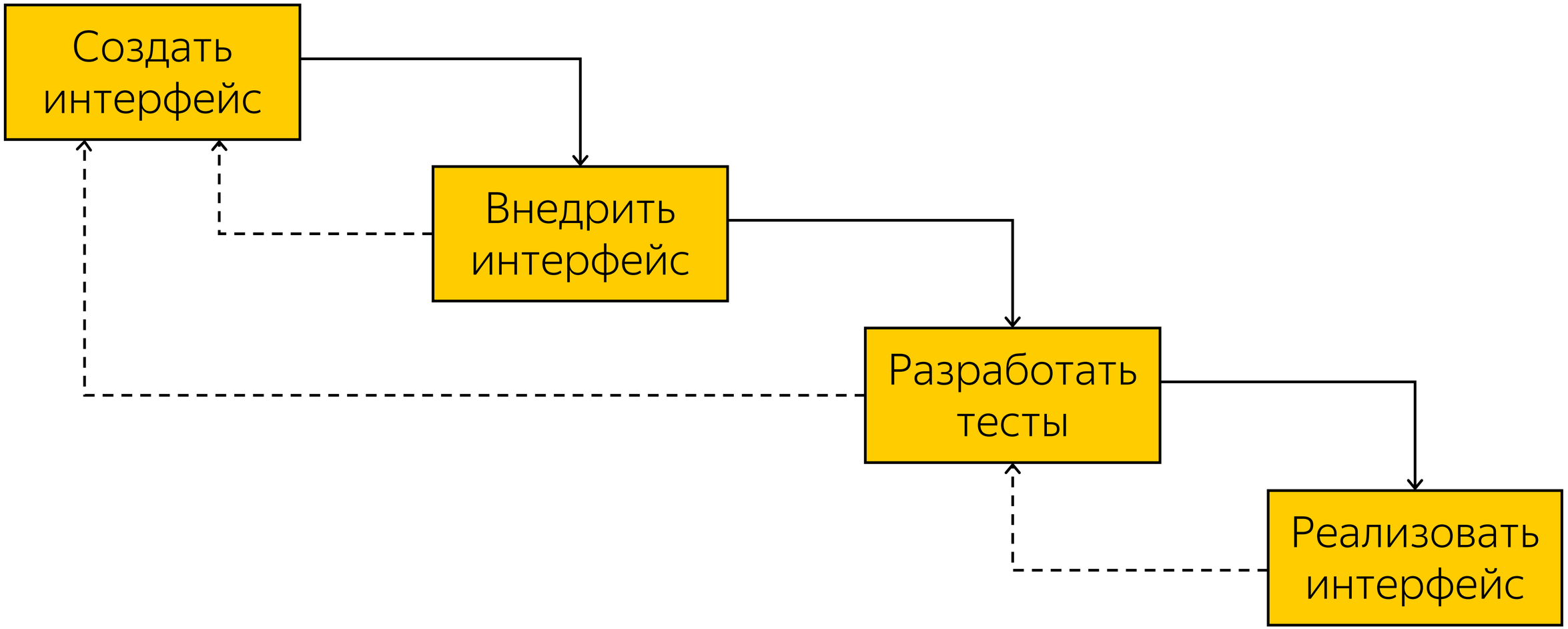
Техника безопасности при работе с кодом большого проекта

Илья Шишков, старший разработчик

Область применения техники

- Добавление новой функциональности
- Рефакторинг

Изменение кода в четыре шага



Пример

HTTP-сервер комментариев пользователей

Действие	Запрос	Ответ
Добавить пользователя	POST /add_user	200 OK userId (int)
Добавить комментарий	POST /add_comment u=(userId)&c=(текст)	200 OK
Получить комментарии	GET /user_comments?u=(userId)	200 OK Comment1 Comment2 ... CommentN

Исходное состояние кода

```
struct HttpRequest {
    std::string method, path, body;
    std::map<std::string, std::string> get_params;
};

class CommentServer {
private:
    std::vector<std::vector<std::string>> comments_;
public:
    void ServeRequest(const HttpRequest& req, std::ostream& os);
};
```

```
void CommentServer::ServeRequest(const HttpRequest& req, std::ostream& os) {
    if (req.method == "POST") {
        if (req.path == "/add_user") {
            comments_.emplace_back();
            auto response = std::to_string(comments_.size() - 1);
            os << "HTTP/1.1 200 OK\r\n"
                << "Content-Length: " << response.size() << "\r\n"
                << "\r\n"
                << response;
        } else if (req.path == "/add_comment") {
            auto[user_id, comment] = ParseIdAndComment(req.body);
            comments_[user_id].push_back(comment);
            os << "HTTP/1.1 200 OK\r\n\r\n";
        } else {
            os << "HTTP/1.1 404 Not found\r\n\r\n";
        }
```

Выполним рефакторинг

- Избавимся от необходимости каждый раз хардкодить HTTP-ответы
- Напишем класс HttpResponse, который будет
  - инкапсулировать строковое представление HTTP-ответов
  - предоставлять удобный интерфейс для их создания
- Воспользуемся нашими четырьмя шагами

Шаг первый – создадим интерфейс

Мы создаём интерфейс без реализации!

```
class HttpResponse {
public:
    explicit HttpResponse(int code);
    void SetContent(std::string value);
    friend std::ostream& operator << (std::ostream& os, const HttpResponse& resp);
};
```

```
void CommentServer::ServeRequest(const HttpRequest& req, std::ostream& os) {
    if (req.method == "POST") {
        if (req.path == "/add_user") {
            comments_.emplace_back();
            auto response = std::to_string(comments_.size() - 1);
            os << "HTTP/1.1 200 OK\r\n"
               << "Content-Length: " << response.size() << "\r\n"
               << "\r\n"
               << response;
        } else if (req.path == "/add_comment") {
            auto[user_id, comment] = ParseIdAndComment(req.body);
            comments_[user_id].push_back(comment);
            os << HttpResponse(200);
        } else {
            os << HttpResponse(404);
        }
```

Шаг первый – создадим интерфейс

```
enum class HttpCode {
    OK = 200,
    NotFound = 404,
};

class HttpResponse {
public:
    explicit HttpResponse(HttpCode code);
    HttpResponse& SetContent(std::string value);
    friend std::ostream& operator << (std::ostream& os, const HttpResponse& resp);
};
```

Второй шаг – внедряем интерфейс

```
void CommentServer::ServeRequest(const HttpRequest& req, std::ostream& os) {
    if (req.method == "POST") {
        if (req.path == "/add_user") {
            comments_.emplace_back();
            os << HttpResponse(HttpCode::OK).SetContent(std::to_string(comments_.size() - 1));
            HttpResponse resp(200);
            resp.SetContent(std::to_string(comments_.size() - 1));
            os << resp;
        } else if (req.path == "/add_comment") {
            auto [user_id, comment] = ParseIdAndComment(req.body);
            comments_[user_id].push_back(comment);
            os << HttpResponse(HttpCode::OK);
        } else {
            os << HttpResponse(HttpCode::NotFound);
        }
```

Второй шаг – внедряем интерфейс

- Внедрение позволяет понять, насколько интерфейс подходит для нашей задачи
- Мы можем сразу его исправить
- Откладывая реализацию, мы экономим время
- В итоге мы получаем наилучший интерфейс именно для нашей задачи

Третий шаг – пишем юнит-тесты

- Покроем наш интерфейс юнит-тестами
- Мы всё ещё его не реализовали!

```
class HttpResponse {
public:
    explicit HttpResponse(HttpCode code);
    HttpResponse& SetContent(std::string value);
    friend std::ostream& operator << (std::ostream& os, const HttpResponse& resp);
};
```

```
void TestConstruction() {
    std::ostringstream os;
    os << HttpResponse(HttpCode::OK);
    assert(os.str() == "HTTP/1.1 200 OK\r\n\r\n");
    os.str("");
    os << HttpResponse(HttpCode::NotFound);
    assert(os.str() == "HTTP/1.1 404 Not found\r\n\r\n");
}

void TestSetContent() {
    std::ostringstream os, expected;
    const std::string content = "Hello, SECR 2017!";
    os << HttpResponse(HttpCode::OK).SetContent(content);
    expected << "HTTP/1.1 200 OK\r\n“ << "Content-Length: " << content.size() << "\r\n"
             << "\r\n“ << content;
    assert(os.str() == expected.str());
}
```

Третий шаг – пишем юнит-тесты

Разработка юнит-тестов до реализации позволяет

- заранее продумать все крайние случаи
- и учесть их во время реализации
- убедиться, что новые тесты корректно встроены в систему тестирования (они должны падать)

Третий шаг – пишем юнит-тесты

- Тесты пустой реализации интерфейса должны падать
- Это позволяет убедиться, что они встроены в систему тестирования

```
$ ./build-project && ./run-unit-tests
251 tests – OK

$ ./build-project && ./run-unit-tests
251 tests – OK
2 tests – Fail
```

Четвёртый шаг – реализация интерфейса

На данный момент у нас есть

- интерфейс именно для нашей задачи
- юнит-тесты – способ контроля корректности его реализации

Сразу после реализации

- мы сможем быстро поймать и исправить большинство багов
- чем меньше время между допущением и обнаружением ошибки, тем проще её исправить

Четвёртый шаг – реализация интерфейса

```
class HttpResponse {
public:
    explicit HttpResponse(HttpCode code);
    HttpResponse& SetContent(std::string value);
    friend std::ostream& operator << (std::ostream& os, const HttpResponse& resp);
private:
    HttpCode code_;
    std::string content_;
};

HttpResponse::HttpResponse(HttpCode code) : code_(code) {
}
```

```
HttpResponse& HttpResponse::SetContent(std::string value) {
    content_ = std::move(value);   return *this;
}
std::ostream& operator << (std::ostream& os, const HttpResponse& resp) {
    const std::string_view crlf("\r\n");
    os << "HTTP/1.1 " << static_cast<int>(resp.code_) << ' ';
    switch (resp.code_) {
        case HttpCode::OK:          os << "OK";          break;
        case HttpCode::NotFound:  os << "Not found"; break;
    }
    os << crlf;
    if (!resp.content_.empty()) {
        os << "Content-Length: " << resp.content_.size() << crlf;
    }
    return os << crlf << resp.content_;
}
```

Четвёртый шаг – реализация интерфейса

- Реализовав интерфейс, запускаем юнит-тесты
- Убеждаемся, что они проходят

```
$ ./build-project && ./run-unit-tests
253 tests – OK
```

Итоги

- Мы получили корректный код
- Корректно встроенный в систему
- Мы минимизировали риски
  - возникновения багов
  - необходимости переделывать свою работу
- Сэкономили уйму времени на отладке и переписывании кода

Спасибо!

Илья Шишков

Старший разработчик компании Яндекс

 ishfb@yandex-team.ru

 ishfb

 telegram: ishfb

 github: ishfb